

User Manual for EDOLAB: Running, Configuring, and Extending the Platform

MAI PENG*, School of Automation, China University of Geosciences, Wuhan, Hubei Key Laboratory of Advanced Control and Intelligent Automation for Complex Systems, and Engineering Research Center of Intelligent Technology for Geo-Exploration, Ministry of Education, China (email: pengmai1998@gmail.com)

DELARAM YAZDANI, Liverpool Logistics, Offshore and Marine (LOOM) Research Institute, Faculty of Health, Innovation, Technology and Science, Liverpool John Moores University, United Kingdom (email: delaram.yazdani@yahoo.com)

DANIAL YAZDANI*, School of Computing Technologies, RMIT University, Australia (email: danial.yazdani@gmail.com)

ZENENG SHE, School of Computer Science and Technology, Harbin Institute of Technology, China (email: 20s151103@stu.hit.edu.cn)

WENJIAN LUO, Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies, Institute of Cyberspace Security, School of Computer Science and Technology, Harbin Institute of Technology, China (email: luowenjian@hit.edu.cn)

CHANGHE LI, School of Artificial Intelligence, Anhui University of Science & Technology, State Key Laboratory of Digital Intelligent Technology for Unmanned Coal Mining, Anhui University of Science & Technology, China (email: changhe.lw@gmail.com)

JUERGEN BRANKE, Warwick Business School, University of Warwick, United Kingdom (email: Juergen.Branke@wbs.ac.uk)

TRUNG THANH NGUYEN, The Liverpool Logistics, Offshore and Marine (LOOM) Research Institute, Faculty of Health, Innovation, Technology and Science, Liverpool John Moores University, United Kingdom (email: T.T.Nguyen@ljmu.ac.uk)

AMIR H. GANDOMI, Faculty of Engineering & Information Technology, University of Technology Sydney, Australia and University Research and Innovation Center (EKIK), Obuda University, Hungary (email: Gandomi@uts.edu.au)

SHENGXIANG YANG, Digital Future Institute, School of Computer Science and Informatics, De Montfort University, United Kingdom (email: syang@dmu.ac.uk)

YAOCHU JIN, Department of Artificial Intelligence, School of Engineering, Westlake University, China (email: jinyaochu@westlake.edu.cn)

XIN YAO[†], School of Data Science, Lingnan University, China (email: xinyao@ln.edu.hk)

*For assistance with EDOLAB's code or any related inquiries, please feel free to reach out to Danial Yazdani at danial.yazdani@gmail.com or Mai Peng at pengmai1998@gmail.com.

Abstract— This user manual provides a comprehensive guide for running, configuring, and extending EDOLAB, an open-source MATLAB platform for evolutionary dynamic optimization algorithms (EDOAs). The platform includes two key applications: the *Education* application, which visualizes algorithm behavior over time, and the *Experimentation* application, which is designed for conducting experiments and comparing algorithms. EDOLAB supports both graphical (GUI) and Code (script-based/non-GUI) modes, offering flexibility to users with different preferences, goals, and levels of expertise. Additionally, instructions are provided for running EDOLAB in GNU Octave, a free and open-source alternative to MATLAB. To begin, clone the EDOLAB repository from GitHub: [<https://github.com/EvoMindLab/EDOLAB>]. Below is the list of sections covered in this manual:

CONTENTS

Contents	2
M-I Architecture	2
M-II Running	6
M-II.1 Running EDOLAB in GUI Mode	6
M-II.2 Running EDOLAB in Code Mode	13
M-III Extension	14
M-III.1 Adding a Benchmark Generator	14
M-III.2 Adding a Performance Indicator	15
M-III.3 Adding an EDOA	15
M-IV Using EDOLAB in Octave	16
M-IV.1 Structure of the Octave Version	16
M-IV.2 Compatibility Notes and Required Adjustments	16
References	17

M-I Architecture

EDOLAB is a function-based software implemented in MATLAB. It supports two modes of operation: *GUI Mode* and *Code Mode*. The GUI Mode is programmatically developed using MATLAB's object-oriented programming features and provides a user-friendly interface for configuring and managing experiments. In contrast, the Code Mode enables users to interact with EDOLAB through direct scripting, offering flexibility for advanced usage, such as manipulating the internal components and procedures of algorithms.

The root directory of EDOLAB includes the following:

- Two .m files: (1) `GUIMode.m` — the entry point for launching EDOLAB's graphical interface, providing interactive features for configuring and executing experiments, and (2) `CodeMode.m` — a script-based interface that allows users to run EDOLAB entirely through code.
- Six folders:
 - `Algorithm` — This folder contains several sub-folders, each corresponding to an EDOA listed in Table 1. Each EDOA sub-folder typically includes the following .m files:
 - (1) `main_EDOA.m` — the main function that invokes and coordinates the EDOA's internal components,

- (2) `SubPopulationGenerator_EDOA.m` — a function responsible for generating sub-populations used by the optimization component,
 - (3) `IterativeComponents_EDOA.m` — a function that implements the EDOA components executed at each iteration or under specific conditions,
 - (4) `ChangeReaction_EDOA.m` — a function that handles change reaction mechanisms following environmental changes, and
 - (5) `getAlgConfigurableParameters_EDOA.m` — a configuration interface function that defines and manages user-modifiable parameters of the EDOA. This function is invoked by `getAlgConfigurableParameters.m` in the `Algorithm` folder to provide a unified parameter control interface.
- **Benchmark** — This folder contains a sub-folder for each benchmark generator included in EDOLAB. Each benchmark sub-folder includes three `.m` files:
- (1) `BenchmarkGenerator_Benchmark.m` — responsible for initializing the benchmark and generating the sequence of environments,
 - (2) `fitness_Benchmark.m` — defines the baseline function of the benchmark used to compute fitness values, and
 - (3) `getProConfigurableParameters_Benchmark.m` — provides a configuration interface for customizing benchmark parameters.
- These functions are invoked by four core `.m` files located in the main `Benchmark` folder:
- (1) `BenchmarkGenerator.m` — serves as the entry point for initializing the selected benchmark and generating its environments,
 - (2) `fitness.m` — handles function evaluations by calling the related benchmark's fitness function, manages benchmark state, counters, and gathers performance-related data,
 - (3) `EnvironmentVisualization.m` — provides visualization of the environment's landscape in the Education application, and
 - (4) `getProConfigurableParameters.m` — offers a unified interface for retrieving and modifying benchmark parameters at runtime.
- **Indicators** — This folder contains a JSON file that defines the available performance indicators in EDOLAB:
- (1) `indicators.json` — specifies all supported performance indicators, including their names, types, and visualization options, using a simple and extensible JSON structure.
- **Results** — This folder contains three sub-folders where EDOLAB stores experiment outputs:
- (1) **BatchRecords**: Stores results from multiple experiments executed using the GUI's batch mode. Each file (saved as a `.mat` file) contains several experiment outcomes, including individual results and their corresponding settings.
 - (2) **StatisticalAnalysis**: Contains the results of statistical analyses performed via the GUI. These are saved as Excel files and include the applied statistical test, experiment configurations, and computed statistical indicators.
 - (3) **TaskDetailResults**: Stores detailed outputs for each experiment run (from both GUI and Code Mode), saved as Excel files. These include per-run performance statistics, experimental settings, and output values.

- Utility — This folder contains various utility functions that support EDOLAB's operations. It includes the Output sub-folder, which provides functions for generating output files and figures used in experiment reporting and visualization.
- OctaveVersion — This folder provides a self-contained, Octave-compatible version of EDOLAB designed to operate in code mode only. Users who wish to run EDOLAB in Octave should use the files in this folder directly by executing `OctaveCodeMode.m`. This implementation is fully independent from the main MATLAB version and has been specifically adapted for the Octave environment.

Regardless of whether EDOLAB is executed via the graphical interface (`GUIMode.m`) or the script-based interface (`CodeMode.m`), the core process for running EDOAs remains fundamentally the same. Both modes invoke the same set of core functions to execute optimization tasks. Understanding this shared execution flow is important for users who wish to extend the platform by adding new algorithms, benchmarks, or performance indicators. Figure M-1 presents a high-level sequence diagram that illustrates how EDOLAB operates internally when executing an EDOA.

After generating the sequence of environments, the initial sub-population(s) or individuals are generated by the sub-population/individual generation function (for example, `SubPopulationGenerator_AmQSO.m`).

First, the user sets up an experiment using either the GUI or the script-based interface (`CodeMode.m`) and initiates the run. In both modes, EDOLAB first invokes the function `getAlgConfigurableParameters.m` to retrieve the list of configurable parameters for the selected EDOA. In Code Mode, these parameters are specified directly within `getAlgConfigurableParameters.m` and loaded by `CodeMode.m` before being passed to the main algorithm function (e.g., `main_AmQSO.m`). In GUI Mode, the same function is called to identify available parameters, which are then presented via interactive user interface controls. Once the user configures the parameters and starts the task, the GUI reads these values and passes them to the main function for execution.

At the start of the main algorithm function, the benchmark generator function (`BenchmarkGenerator.m`) is called. As with algorithm parameters, benchmark parameters are first defined in `getProConfigurableParameters_Benchmark.m`. In Code Mode, these parameters are manually edited in the script before being passed to `BenchmarkGenerator.m` for initialization. In GUI Mode, the parameters are automatically extracted from `getProConfigurableParameters_Benchmark.m` and displayed through interface controls for user customization. Once the experiment begins, the GUI reads the selected values and passes them to `BenchmarkGenerator.m` to initialize the benchmark and generate the sequence of environments.

In EDOLAB's experimentation application, identical random streams are used to initialize benchmark instances and environmental changes in `BenchmarkGenerator.m`. As a result, with the same parameter settings, all algorithms are evaluated on the same sequence of problem instances, ensuring fair comparisons. Without controlling random streams, variations in the generated landscapes could introduce unintended differences in problem difficulty [Yazdani et al. 2021b]. To prevent this, EDOLAB enforces random seed and stream control mechanisms.

After generating the sequence of environments, the initial sub-populations or individuals are created using the corresponding sub-population generation function (e.g., `SubPopulationGenerator_AmQSO.m`). EDOLAB uses a separate shuffled random seed for initializing the algorithm.

At the end of each iteration, if the environment has changed, the change reaction components are called (for example, `ChangeReaction_AmQSO.m`). The main loop of the EDOA continues until the number of function evaluations (FE) reaches its predefined maximum value (FE_{max}). This procedure is repeated for the specified number of runs ($RunNumber$). Afterward, the results are processed, including the calculation of performance indicators. The results,

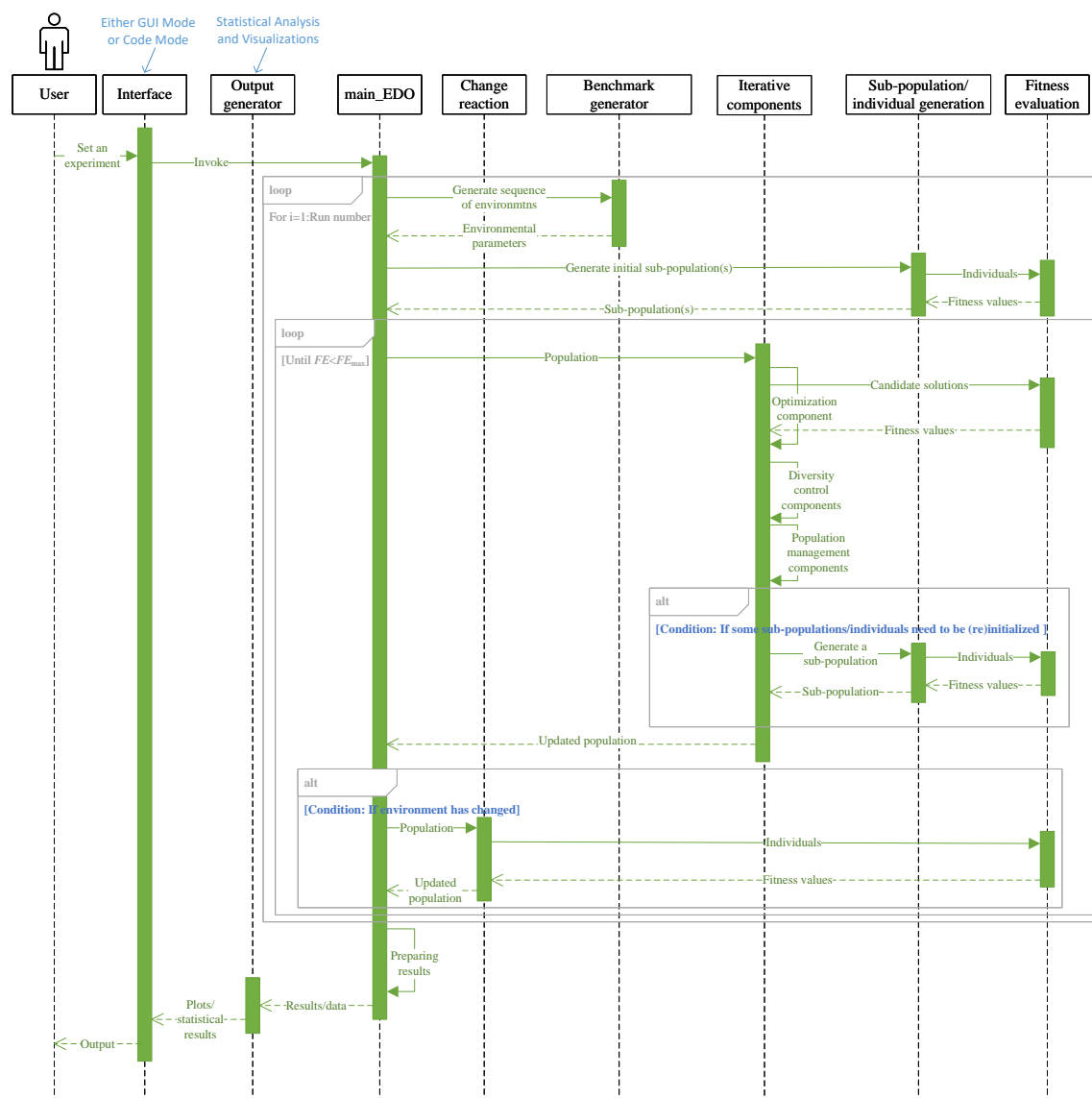


Fig. M-1. A high-level sequence diagram of running an EDOA in EDOLAB.

along with any collected data, are then sent to output generator functions responsible for creating output plots, tables, and files. Finally, the output tables and figures are returned to the interface. In the GUI, users can collect results from multiple experiments and perform further statistical analysis to compare the performance of different algorithms.

Afterward, the main loop of the EDOA is executed. In each iteration, the iterative components of the EDOA—such as the optimizer (e.g., PSO or DE), diversity control, and population management [Yazdani et al. 2020]—are executed by calling the corresponding function (e.g., `IterativeComponents_AmQSO.m`). In many EDOAs with adaptive sub-population numbers and/or population sizes, new individuals or sub-populations are generated when certain

conditions are met [Yazdani et al. 2021a]. Additionally, some diversity and population management components may require the reinitialization of specific sub-populations or individuals. When such cases arise, the sub-population or individual generation function is invoked to perform the initialization, and the updated population is then returned to the main EDOA function.

As in many real-world scenarios where the algorithm is explicitly informed about changes—such as through sensors, the arrival of new jobs, or shifts in task priorities—EDOLAB adopts a controlled simulation setup in which the algorithm is notified of environmental changes via a dedicated flag. Once a change occurs, the flag is triggered, and the algorithm immediately invokes the change reaction components (e.g., `ChangeReaction_AmQS0.m`), regardless of whether it is in the middle of updating a sub-population or performing any other operation. This mechanism ensures consistent and fair comparisons by eliminating timing biases that could otherwise arise from delayed or inconsistent triggering of change reaction mechanisms.

The main loop continues until the number of function evaluations (FE) reaches the predefined maximum (FE_{\max}). This procedure is repeated for the specified number of runs (`RunNumber`). Once all runs are completed, the results are processed, including the calculation of performance indicators. The collected data is then passed to the output generation functions responsible for creating plots, summary tables, statistical analysis, and export files.

M-II Running

As noted earlier, EDOLAB supports two operational modes: GUI Mode and Code Mode. In this section, we describe how to use the platform through both modes, highlighting their respective features and usage scenarios.

M-II.1 Running EDOLAB in GUI Mode

The GUI is implemented using MATLAB's object-oriented programming framework and can be launched by running `GUIMode.m` from EDOLAB's root directory. A minimum MATLAB version of *R2020b* is required to use the GUI. For users with older versions, the Code Mode via `CodeMode.m` remains fully functional and provides access to all core features (see Section M-II.2). The GUI also supports parallel execution using either a process pool or a thread pool, which requires the MATLAB Parallel Computing Toolbox to be installed. To enable thread pool support, MATLAB *R2024a* or later is required; otherwise, only process pool execution is available.

EDOLAB's GUI includes two main applications—*Experimentation* and *Education*—which are described in the following subsections.

M-II.1.1 Experimentation Application. The Experimentation application is designed for conducting and managing optimization experiments. Figure M-2 shows its interface, which is organized into four main parts:

- (1) *Task Settings* — configure the parameters of experimental tasks,
- (2) *Pending Tasks* — view and manage experiments queued for execution,
- (3) *Completed Tasks* — access results of previously run experiments, and
- (4) *Statistical Analysis* — perform significance tests and visualize performance comparisons.

Each of these components is described in detail in the following paragraphs.

Task Settings. This panel provides users with powerful customization capabilities for defining experimental tasks. Upon selecting an algorithm (EDOA) or a benchmark generator, the GUI automatically updates to display the relevant

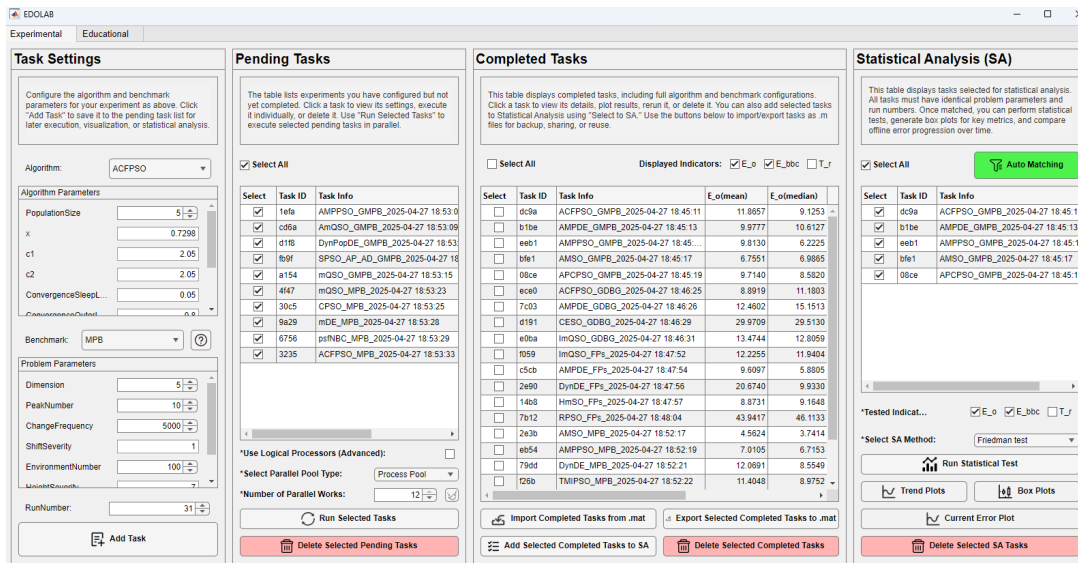


Fig. M-2. The Experimentation Application in EDOLAB, providing a graphical interface for configuring and running experiments.

configurable parameters for the selected component. Once the configuration is complete, the user can click the “Add Task” button to queue the task in the pending task list.

Pending Tasks. This panel displays the list of experimental tasks that have been configured but not yet executed. By left-clicking on a task, users can view its detailed configuration, including the selected algorithm, benchmark parameters, run number, and other relevant settings. Each task is assigned a unique 4-character ID that remains associated with it throughout its lifecycle, from configuration to execution and post-processing. From the task detail view, users can choose to either execute or delete the selected task. Once all desired tasks have been added, execution can be initiated by clicking the “Run Selected Tasks” button.

EDOLAB supports the concurrent execution of multiple tasks using either multithreading or multiprocessing. Within the Pending Tasks panel, users can configure parallelization settings, including:

- *Use logical processors* — Enabling this option may increase the number of available workers but could introduce performance trade-offs depending on system architecture.
- *Select parallel pool type* — Users can choose between the default *Processes pool*, which provides memory isolation, and the *Threads pool*, which offers faster startup and shared memory. Note that the Threads pool is available only in MATLAB R2024a and later. Furthermore, the Free Peaks (FPs) benchmark is incompatible with the Threads pool due to its reliance on MEX-based C++ interfaces.
- *Number of parallel workers* — This setting defines the maximum number of tasks that can run simultaneously, allowing users to balance performance and system resource usage.

During execution, the Pending Tasks table dynamically reflects the status and progress of each task. Upon completion, tasks are automatically moved to the *Completed Tasks* panel. Users can cancel tasks at any point during execution. If no tasks are currently running, users can also choose to shut down the active thread or process pool. Shutting down a process pool can help free system memory, but restarting it later will incur additional overhead.

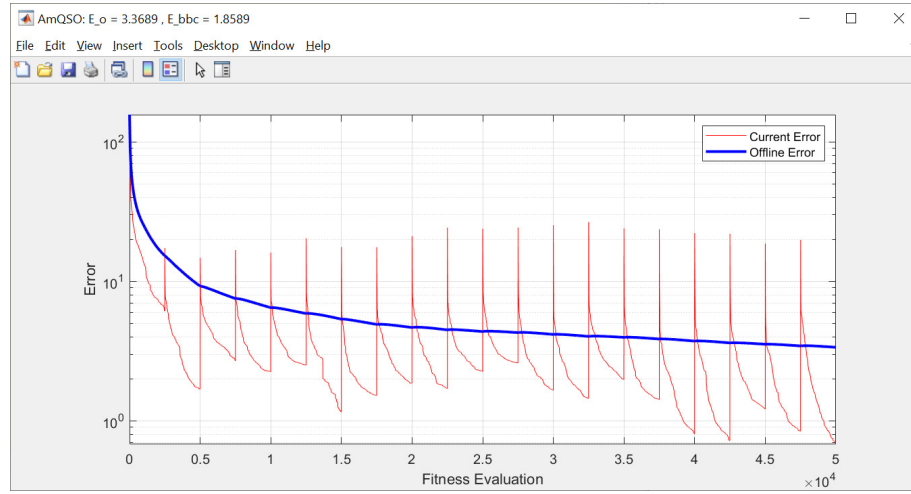


Fig. M-3. Current and Offline Error Plot from a Completed Experiment in EDOLAB. This figure shows the average convergence behavior of an algorithm, where results are averaged over all runs. The Current Error reflects the algorithm's ability to re-converge after each environmental change. The plot is accessible via the Completed Task Panel in GUI Mode and can also be generated in Code Mode.

Completed Tasks. This panel displays all completed experimental tasks. Clicking on any task reveals detailed execution information, including the selected algorithm, benchmark settings, run number, and more. Within the task detail view, EDOLAB provides the following five operations for each completed task:

- **Plot Results:** Displays trend plots of average current error and offline error (E_o) over time across all runs for the selected task (see Figure M-3).
- **Save Results:** Saves the detailed results of the selected task to the /Results/Task Detail Results folder. Files are named using the format EDOA_Benchmark_DateTime.xlsx.
- **Select to SA:** Adds the selected task to the Statistical Analysis (SA) panel for further evaluation. This allows statistical comparison of multiple algorithms on benchmark instances with identical settings and run numbers.
- **Move to Pending:** Moves the selected task back to the Pending Tasks panel for re-execution.
- **Delete Task:** Permanently deletes the selected task.

The Completed Tasks panel also supports batch operations:

- **Export Selected Completed Tasks to .mat:** Exports selected tasks and their results to a .mat file for later use. Note that completed tasks are not saved automatically when the GUI window is closed, so saving important tasks is recommended. Exported files are saved in the /Results/Batch Records folder and follow the naming format: CompletedTasks_DateTime.mat.
- **Import Completed Tasks from .mat:** Imports previously saved .mat files. Tasks with duplicate IDs are skipped to avoid duplication.
- **Add Selected Completed Tasks to SA:** Adds the selected tasks in batch to the Statistical Analysis panel. All tasks must have identical benchmark settings and run numbers to ensure valid comparisons.
- **Delete Selected Completed Tasks:** Permanently deletes selected tasks from the Completed Tasks panel.

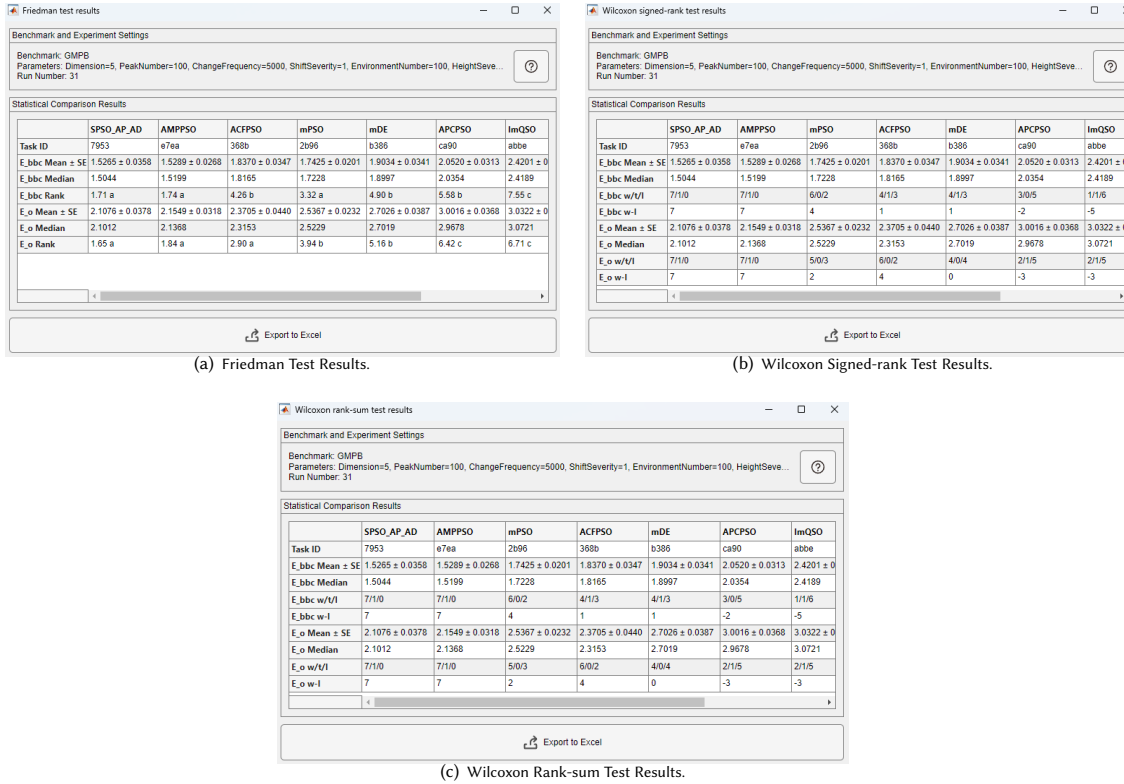


Fig. M-4. Tabular Summary of Statistical Analysis Results in EDOLAB. This figure showcases the output tables generated in the Statistical Analysis Panel of GUI Mode. The tables summarize statistical comparisons across multiple algorithms.

In addition, EDOLAB provides checkboxes for selecting specific performance indicators to display in the panel. This allows users to focus on relevant metrics, and the system supports user-defined indicators for customized analysis.

Statistical Analysis. This panel allows users to perform in-depth comparisons of algorithm performance across multiple completed experimental tasks added to the Statistical Analysis (SA) list. Once tasks are selected, EDOLAB provides two main analytical tools: *Statistical Tests* and *Result Visualization*.

- **Statistical Tests** — The current version of EDOLAB supports three widely used non-parametric hypothesis testing methods for evaluating the significance of performance differences across selected indicators and algorithm configurations:
 - **Friedman test:** Detects differences in performance across multiple algorithms on the same set of tasks. It ranks algorithms per task and tests whether the average ranks differ significantly. If significant differences ($p < 0.05$) are detected, the *Nemenyi test* performs pairwise comparisons to identify which algorithm pairs differ. Algorithms sharing the same letter (e.g., “a”, “b”) in the ranking column are not significantly different at the 95% confidence level.

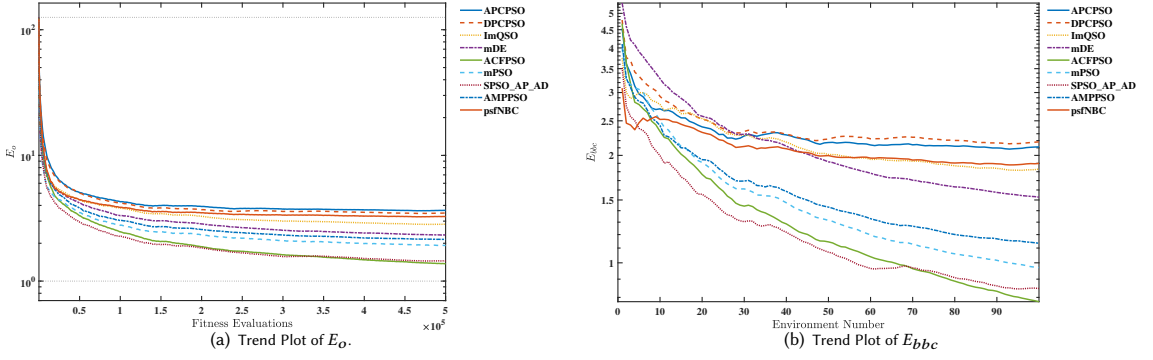


Fig. M-5. This figure visually compares the performance of algorithms using trend plots of E_o over all fitness evaluations and E_{bbc} over all environments. Each plot shows the average results across all runs for each algorithm. These visualizations are generated through the Statistical Analysis Panel in GUI Mode.

- *Wilcoxon signed-rank test*: This non-parametric test compares the performance of two algorithms under the same experimental conditions. For each selected performance indicator, EDOLAB computes and displays the number of wins, ties, and losses of each algorithm against the other comparison algorithms.
- *Wilcoxon rank-sum test* (Mann–Whitney U test): This test compares the performance of two independent algorithms across selected indicators. EDOLAB summarizes the results in terms of wins, ties, and losses across all selected tasks.

Users can select one of the available methods for their analysis. As shown in Figure M-4, EDOLAB generates a tabular summary for each statistical test, including the Friedman test, Wilcoxon signed-rank test, and Wilcoxon rank-sum test. Each table includes algorithm rankings or win–tie–loss summaries, depending on the method used. All result tables also include experiment metadata such as benchmark configuration, parameter settings, and the number of runs. The results can be exported as Excel files, which are saved in the /Results/Statistical Analysis directory using the naming convention SA_AnalysisMethod_DateTime.xlsx.

- **Result Visualization** — This feature generates graphical representations of selected task outcomes to support intuitive performance analysis. It includes three visualization components:
 - *Trend Plots* — These plots show the evolution of selected performance indicators over time (e.g., E_o versus fitness evaluations or E_{bbc} versus environment number; see Figure M-5). User-defined indicators are also supported, as described in Section M-III.2.
 - *Box Plots* — Notched box plots visualize the distribution of final performance indicator values across tasks (see Figure M-6). Like trend plots, box plots also support user-defined indicators.
 - *Current Error Plots* — These plots display the current error of the best-so-far solution within the current environment, averaged across all runs (see Figure M-7). Unlike other trend plots, they specifically highlight the real-time convergence behavior of algorithms within each environment. Current error plots offer valuable insight into an algorithm’s convergence dynamics, including how quickly it responds to environmental changes and how effectively it converges during each environment.

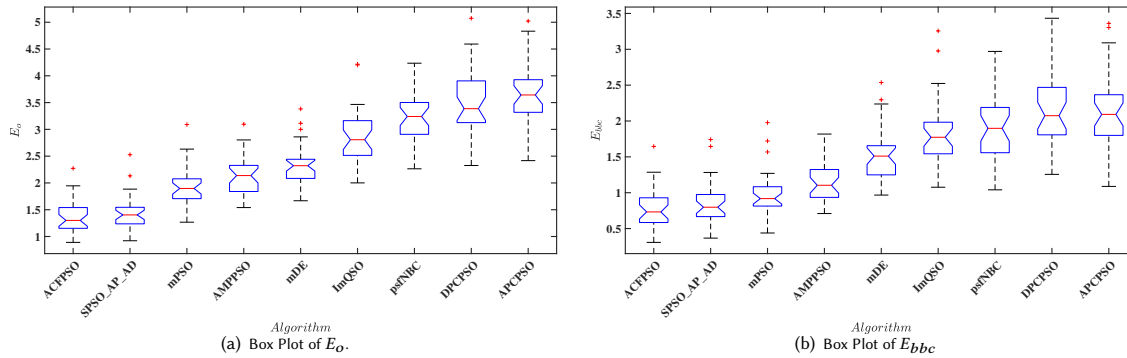


Fig. M-6. Box plot comparison of algorithm performance using E_o and E_{bbc} metrics. These visualizations are generated through the Statistical Analysis Panel in GUI Mode and illustrate performance variations across multiple runs for a set of comparison algorithms.

Only tasks with identical benchmark configurations and number of runs can be included in the SA list. To simplify task selection, the Auto Matching button searches the *Completed Tasks* panel for experiments that match the configuration of the task(s) already in the SA list and adds them automatically. A Delete SA Tasks button is also provided to remove selected tasks from the SA list, giving users full control over their comparative evaluations.

The overall workflow of the *experimentation application* proceeds as follows: users begin by configuring one or more experimental tasks in the *Task Settings* panel, where they can customize algorithm and benchmark parameters. These tasks are then queued in the *Pending Tasks* panel and executed—either sequentially or in parallel—based on the configured parallelization settings. Upon completion, results are automatically transferred to the *Completed Tasks* panel, where users can inspect detailed outputs, generate individual plots, and export results. Selected tasks with matching benchmark settings and run numbers can be added to the *Statistical Analysis* panel for further performance evaluation through statistical tests, rankings, and result visualizations.

M-II.1.2 Education Application. The education application allows users to visually observe the current environment, environmental changes, and the behavior of individuals over time. Figure M-8 shows the interface of EDOLAB's education application. On the left side of the interface, users can configure experimental parameters in a manner similar to the experimentation application. However, this application only supports two-dimensional problem instances, as its primary purpose is to visualize the search space and the dynamics of optimization.

Once the experiment is configured and the RUN button is pressed, EDOLAB archives the environmental settings and the positions of individuals over time. The time required for a run depends on the selected algorithm, benchmark configuration, and available hardware resources. Upon completion, the archived data is rendered in the interface to enable visual exploration.

Using this data, the education application generates a video that shows how the environment and individuals evolve over time. The interface includes video player-style controls that allow users to navigate through generations by entering a specific value or using a progress bar, and to adjust the playback speed. As illustrated in Figure M-8, the core visualization is a two-dimensional contour plot of the environment. Promising regions are shown with contour levels; the centers of visible promising regions (i.e., those not occluded by larger regions) are marked with blue circles,

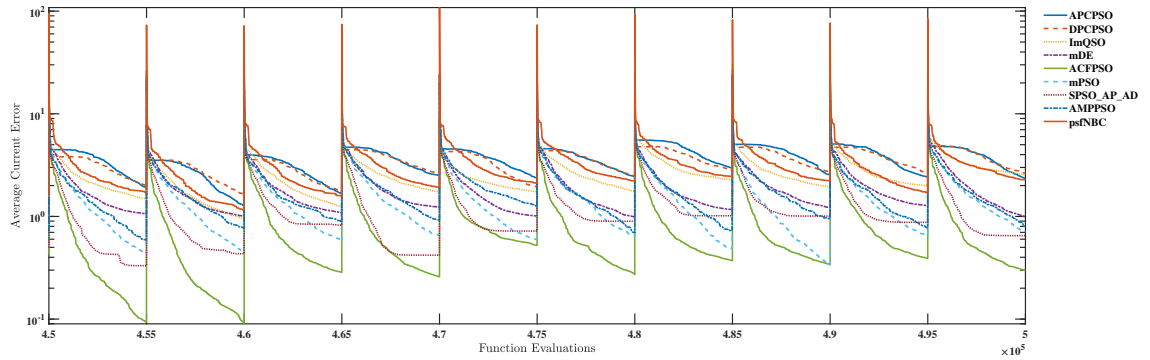


Fig. M-7. Current Error Plot generated using the Statistical Analysis panel in the GUI Mode of EDOLAB. This figure illustrates the convergence behavior of multiple algorithms on a dynamic optimization problem. The *current error* represents the error of the best-so-far solution within the current environment and is plotted over fitness evaluations, averaged across all runs. The sharp spikes visible in the plot correspond to moments when environmental changes occur, causing a sudden increase in error values. Between two spikes, the environment remains stationary, and the plot reveals how rapidly and effectively each algorithm re-converges. Due to the dense and fluctuating nature of these plots, it is advisable to visualize only a small portion of the environments for clarity. In this example, we display 10 environments with a change frequency of 5000 fitness evaluations.

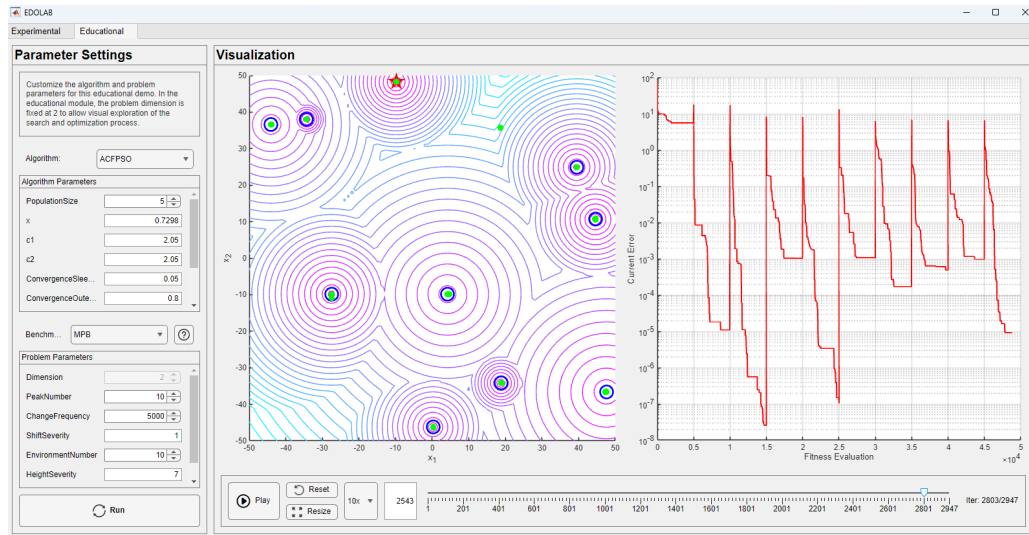


Fig. M-8. The Education Application in GUI Mode. Users can select an algorithm and a benchmark, configure their parameters, and visualize the environment's contour plot, its changes over time, the behavior of individuals, and the current error. This application provides intuitive insights into algorithmic behavior and environmental dynamics. In the GUI Mode, it operates like a video player with control buttons to navigate through time steps. The Education Application is also accessible via Code Mode, though without the video control interface.

the global optimum is indicated by a red pentagram, and individuals are displayed as green filled circles. Individual positions are updated each iteration, and the landscape visualization is refreshed after every environmental change. Additionally, a real-time plot of current error and a counter showing the environment number are presented to offer additional insights.

By examining the movement of individuals, the evolving landscape, the environment number, and current error values, users can analyze an algorithm's exploratory and exploitative behavior, as well as its responsiveness to changes. This visualization helps in assessing the effectiveness of specific algorithmic components such as mutual exclusion in promising regions [Blackwell and Branke 2006], sub-population generation strategies [Blackwell et al. 2008], diversity mechanisms, and change reaction behaviors.

Unlike the experimentation application, which uses identical random seeds to generate consistent problem instances across different algorithms for fair comparison, the education application assigns a new random seed for each run. This means that each execution produces a different sequence of environments, allowing users to observe how the algorithm responds to diverse dynamic scenarios.

M-II.2 Running EDOLAB in Code Mode

While EDOLAB offers a comprehensive and user-friendly graphical interface, it also supports a *Code Mode* that allows users to interact with the platform directly through scripting. This mode is intended for users who wish to: (1) modify or debug the internal implementation of algorithms and benchmark components, (2) experiment with custom logic or structural changes, or (3) operate EDOLAB on systems that do not support the GUI (e.g., MATLAB versions older than R2020b). In contrast to the GUI, which supports features such as batch task configuration, parallel execution, automatic statistical analysis, and rich visualizations, the Code Mode provides greater flexibility for inspecting, modifying, and testing individual components of the platform.

To enhance readability, understanding, and ease of navigation within EDOLAB's source code, we have:

- divided the code into *sections* using the `%%` command, with each section having a descriptive header. These sections group related lines of code, such as those implementing components (e.g., exclusion [Blackwell and Branke 2006]), initializing EDOA parameters, or preparing output values;
- assigned meaningful and descriptive names to all structures, parameters, and functions in EDOLAB;
- added informative comments throughout the code to assist users in understanding and customizing the platform.

To run EDOLAB in Code Mode, users begin by editing the `CodeMode.m` script in the root directory. This script allows users to specify the desired EDOA and benchmark generator by setting the `AlgorithmName` and `BenchmarkName` variables (e.g., `AlgorithmName = 'mQSO'`, `BenchmarkName = 'GMPB'`).

Parameter settings for each algorithm and benchmark can be modified through their corresponding configuration interfaces:

- `getAlgConfigurableParameters_EDOA.m` — contains default and configurable parameters of the selected algorithm;
- `getProConfigurableParameters_Benchmark.m` — defines the adjustable parameters of the selected benchmark.

Users can update the value field of each parameter structure to change its default value before the experiment begins.

EDOLAB's Code Mode supports both the *Experimentation* and *Education* applications. To activate the Education Application (e.g., contour visualization of 2D environments), set `VisualizationOverOptimization = 1`. To run experiments using the Experimentation Application, set `VisualizationOverOptimization = 0`.

In Experimentation Mode, users can enable output generation features by modifying the following flags:

- `OutputFigure = 1` — generates a figure showing the offline and current error over time;

- `GeneratingExcelFile = 1` — exports an Excel file with experiment statistics to the `/Results` folder.

Once configurations are complete, executing `CodeMode.m` will launch the experiment. Progress updates, including current run and environment numbers, will be displayed in the MATLAB Command Window, and summary results will be printed upon completion.

Note that although Code Mode is supported in EDOLAB, the GUI Mode is the recommended interface for most users. It provides complete access to all parameters, advanced experimental workflows, parallel execution, statistical testing, and advanced visualizations, all within a streamlined user experience.

M-III Extension

EDOLAB is an open-source platform designed with extensibility in mind. This section provides guidance on how users can add new benchmark generators, performance indicators, and EDOAs to the system.

M-III.1 Adding a Benchmark Generator

To add a new benchmark, suppose called ABC, users should begin by creating a subfolder named ABC within the Benchmark directory. Within this folder, three functions must be implemented: `getProConfigurableParameters_ABC.m`, `BenchmarkGenerator_ABC.m`, and `fitness_ABC.m`.

`getProConfigurableParameters_ABC.m` defines the set of user-configurable parameters for the benchmark. These parameters should be structured using a standardized format that specifies default values, allowable ranges, data types, and descriptions. The supported data types are:

- `numeric` — continuous real numbers,
- `integer` — discrete integers,
- `option` — categorical choices from a predefined list.

An example definition of a parameter is shown below:

```
ConfigurableParameters.ExampleParameter = struct( ...
    'value', 5, ...
    'type', 'integer', ...
    'range', [1, 100], ...
    'description', 'Example of an integer-type parameter.');
```

`BenchmarkGenerator_ABC.m` uses the defined parameters to generate all environmental states of the benchmark. It should output a structure named `Problem`, which includes the environmental dynamics and required control settings. This function should *not* redefine configurable parameters; instead, it should rely entirely on the input from `getProConfigurableParameters_ABC.m`.

The final required function, `fitness_ABC.m`, implements the benchmark's baseline fitness function. Both `BenchmarkGenerator_ABC.m` and `fitness_ABC.m` must conform to the input/output conventions used by existing benchmarks in EDOLAB.

Once these three functions are added, no further modifications are necessary. The new benchmark will automatically appear in the GUI's benchmark list and will also be accessible via `CodeMode.m`.

M-III.2 Adding a Performance Indicator

In dynamic optimization problems (DOPs), performance indicators are typically computed over time—at the end of each environment [Trojanowski and Michalewicz 1999], after every function evaluation [Branke and Schmeck 2003], or upon deploying solutions within each environment [Yazdani 2018].

In EDOLAB, users can add new performance indicators through a simple and extensible JSON interface. Only minimal edits are required: users define the indicator in a JSON file and update the `fitness.m` script accordingly; all other integration steps are handled automatically by EDOLAB.

To add a new performance indicator, follow these steps:

- **Edit the JSON file**

Open `Indicators/indicators.json` and add a new entry for your indicator. The `type` field must be specified as one of the following: “FE based”, “Environment based”, or “None”. Example:

```
"MyNewIndicator": {
  "type": "FE based"           % or "Environment based" / "None"
}
```

- **Implement the indicator in `fitness.m`**

Update the `fitness.m` function to compute and store the new indicator at the appropriate location. Note that for FE based and Environment based indicators, you must populate the `trend` field; for None type indicators, you must populate the `final` field.

- FE based indicators (updated at every fitness evaluation):

```
Problem.Indicators.MyNewIndicator.trend(Problem.FE) = <your calculation>;
```

- Environment based indicators (updated at the end of each environment):

```
Problem.Indicators.MyNewIndicator.trend(Problem.Environmentcounter) = <your calculation>;
```

- None type indicators (updated only at the final fitness evaluation, storing a single final value):

```
Problem.Indicators.MyNewIndicator.final = <your calculation>;
```

After updating the JSON file and modifying `fitness.m`, EDOLAB will automatically recognize and integrate the new performance indicator into both the statistical analysis panel and the visualization components of the GUI. The new indicator will also be fully available when running experiments using `CodeMode.m`.

M-III.3 Adding an EDOA

Adding a new EDOA to EDOLAB requires minimal modifications to ensure compatibility with the platform. Follow these steps to add a new EDOA:

- Create a sub-folder inside the `Algorithm` folder, named after the new EDOA. Add all relevant functions for the EDOA into this sub-folder. The required five files are listed in Section M-I.
- Ensure that the new EDOA can be invoked by `CodeMode.m`. The inputs and outputs of the EDOA’s main function must be consistent with those expected by `CodeMode.m`.
- Define the EDOA’s configurable parameters in `getAlgConfigurableParameters_EDOA.m` using the standardized format, including fields for value, type, range, and description. These parameters will be available for both GUI-based and code-based configurations.
- In the main function, use `BenchmarkGenerator.m` to generate the problem instance.

- To enable support for the education application, include the code for collecting visualization data during optimization. This code can be found in the %%Visualization for education module section of the existing EDOAs.
- Use `fitness.m` for solution evaluation.
- Inject the user-defined parameters, specified in `getAlgConfigurableParameters_EDOA.m` and configured via the GUI or code mode, into the optimizer before initialization. Within the optimizer, define any additional internal parameters or data structures necessary for runtime statistics, performance indicators, and other outputs.
- At the end of the EDOA's main function, prepare the output structure following the template used in existing EDOAs to ensure compatibility with EDOLAB's reporting and analysis modules.
- Name the main function of the new EDOA as `main_EDOA.m` to enable automatic integration with EDOLAB.

For example, if the new EDOA is named XYZ, the sub-folder should be named XYZ, and the main function should be named `main_XYZ.m`. Once completed, the new EDOA will automatically appear in the list of available algorithms in GUI panels. Additionally, it can be run programmatically by setting `AlgorithmName = 'XYZ'` in `CodeMode.m`.

M-IV Using EDOLAB in Octave

EDOLAB was originally developed in MATLAB, but a standalone version has been created to support execution in Octave, an open-source alternative that offers high compatibility with MATLAB. This allows researchers to access EDOLAB's key functionalities without a MATLAB license.

M-IV.1 Structure of the Octave Version

The `OctaveVersion` folder provides a self-contained implementation of EDOLAB that is specifically adapted for compatibility with Octave. It operates exclusively in code mode and does not include a GUI interface. To run EDOLAB in Octave, users should execute the `OctaveCodeMode.m` script located in this folder.

The folder mirrors the structure of the main MATLAB version and includes:

- **Algorithm** — Contains sub-folders for each supported EDOA.
- **Benchmark** — Includes Octave-adapted benchmark definitions and generators.
- **Indicators** — Defines available performance indicators.
- **Utility** — Provides general-purpose helper functions and output routines.
- **Results** — Stores experiment outputs.

The Octave version is fully independent from the MATLAB version.

M-IV.2 Compatibility Notes and Required Adjustments

Several adjustments were made to ensure compatibility with Octave, considering its differences from MATLAB:

- **Required Packages** — The statistics and io packages must be installed and loaded before running experiments.
- **Excel Output** —
 - Functions relying on `xlswrite` have been simplified to avoid ActiveX dependencies and work with Octave's I/O system.
- **Plotting Limitations** —
 - Advanced plotting functions (e.g., `append`, `parfor`) have been removed or replaced where necessary.

- *String Handling* —
 - Octave does not support double-quote strings (""); cell arrays of character vectors are used instead.
- *Free Peaks Benchmark* —
 - Since Octave cannot use MATLAB's .mexw64 files, we have manually compiled the KDTree C++ source using:

```
mkoctfile -v --mex ConstructKDTree.cpp
```

References

- Tim Blackwell and Juergen Branke. 2006. Multiswarms, exclusion, and anti-convergence in dynamic environments. *IEEE Transactions on Evolutionary Computation* 10, 4 (2006), 459–472.
- Tim Blackwell, Juergen Branke, and Xiaodong Li. 2008. Particle swarms for dynamic optimization problems. In *Swarm Intelligence: Introduction and Applications*, Christian Blum and Daniel Merkle (Eds.). Springer Lecture Notes in Computer Science, 193–217.
- Juergen Branke and Hartmut Schmeck. 2003. Designing Evolutionary Algorithms for Dynamic Optimization Problems. In *Advances in Evolutionary Computing*, A. Ghosh and S. Tsutsui (Eds.). Springer Natural Computing Series, 239–262.
- Krzysztof Trojanowski and Zbigniew Michalewicz. 1999. Searching for optima in non-stationary environments. In *Congress on Evolutionary Computation*, Vol. 3. 1843–1850.
- Danial Yazdani. 2018. *Particle swarm optimization for dynamically changing environments with particular focus on scalability and switching cost*. Ph. D. Dissertation. Liverpool John Moores University, Liverpool, UK.
- Danial Yazdani, Ran Cheng, Cheng He, and Juergen Branke. 2020. Adaptive control of subpopulations in evolutionary dynamic optimization. *IEEE Transactions on Cybernetics* 52, 7 (2020), 6476–6489.
- Danial Yazdani, Ran Cheng, Donya Yazdani, Jürgen Branke, Yaochu Jin, and Xin Yao. 2021a. A Survey of Evolutionary Continuous Dynamic Optimization Over Two Decades – Part A. *IEEE Transactions on Evolutionary Computation* 25, 4 (2021), 609–629.
- Danial Yazdani, Ran Cheng, Donya Yazdani, Jürgen Branke, Yaochu Jin, and Xin Yao. 2021b. A Survey of Evolutionary Continuous Dynamic Optimization Over Two Decades – Part B. *IEEE Transactions on Evolutionary Computation* 25, 4 (2021), 630–650.