Single and Multi-Objective Optimization using Computational Intelligence Techniques: A Desktop Application

Tomás Líbano Monteiro
tomas.libano.monteiro@tecnico.ulisboa.pt

Nuno Horta
nuno.horta@tecnico.ulisboa.pt

***Abstract*—With the increase in computational power, computational intelligence techniques have been progressively adopted to solve single and multi-objective optimization problems. A series of general optimization frameworks were developed to try to establish a comprehensive set of tools to facilitate research in this area. Some graphical user interfaces are offered, but they often lack important characteristics to be effectively useful, such as extensibility and customization, or are behind a paywall. Furthermore, as Python becomes one of the most used programming languages in the world, it would be of interest to have a Python-based graphical interface, where user code could be easily integrated, fostering faster development in the optimization research area.**

This work aims to tackle this issue by developing a Python-based desktop application on top of an existing state-of-the-art optimization framework, pymoo. It brings together more than 100 problems and 14 algorithms for single and multi-objective optimization, as well as 11 performance indicators and 9 termination criteria. More than 40 operators of various types are provided, allowing for great customization of the algorithms. Optimization processes can be set in motion with a click of a button, visualized through appropriate techniques, and the results saved. The open-source nature of the project and the application's balance between ease of use and customization make it a valuable resource for both novice and experienced users in the field of optimization.

***Index Terms*—Computational Intelligence, Single-Objective Optimization, Multi-Objective Optimization, Pymoo, Desktop Application, Graphical User Interface, Python.**

I. Introduction

Optimization is now an intrinsic part of our lives. From the search for the shortest route between home and the supermarket using GPS, to the goal of building the most satisfying and reliable smartphone from the user's perspective at the smallest cost, or the most effective way to schedule flights for airlines, optimization shapes the very fabric of our daily routines.

What were for many centuries problems solved through human reason and intuition are now replaced or aided by algorithms of intelligent computation. With the rise of computational power, it is now possible to solve not only single-objective optimization problems (SOOPs), but also multi and many-objective problems (MOOPs and MaOOPs). In the later cases, instead of a single best solution, a set of solutions that correspond to the best trade-offs between objectives emerges, called the optimal Pareto set. Evolutionary Algorithms (EA) are frequently used [1, 2] to achieve or approximate this optimal Pareto set, due to their population-based methods.

Over the years, several frameworks have been developed to assist in the implementation of various algorithms, as well as in the definition of a series of test problems, performance measures, and visualization techniques. This is the case for frameworks such as MOEA [3], written in Java [4], PlatEMO [5], in MATLAB [6], pymoo [7] and DEAP [8], written in Python [9].

Some of the frameworks already include a Graphical User Interface (GUI), such as PlatEMO and MOEA. However, as Python becomes one of the most widely used programming languages among users around the world [10], there is a lack of a GUI that could not only facilitate the process of studying and discovering new algorithms in this language, but also be an entry point for several users with lower programming skills or know-how in optimization.

Furthermore, existing GUIs are often limited in what they allow the user to do, as they are not being developed in a way to harness the fullness of the framework in the backbone of the application. This highlights the need for the work proposed in this thesis.

The objective is to create a desktop application for optimization algorithms. It should have as a backbone an existing optimization framework to be seamlessly integrated. Then it should build a graphical interface on top of it, so it fulfills the needs of what is not yet available on the market. The application should meet the following requirements:

1) **Cost-free.**
2) **Python-based.**
3) **Test multiple state-of-the-art algorithms on various benchmarking problems.**
4) **Adequate visualization techniques.**
5) **Easily extensible.**
6) **Customizable and easy to use.**

The document details the fundamental concepts and related work in Section II. Then it lays out the architecture of the application in Section III, explaining the design choices to meet the requirements. Section IV explores the ways in which the app is useful for single and multi-objective optimization research. The concluding remarks are presented in Section V.

II. Background and Related Work

A. Fundamental Concepts

1) Single and Multi-Objective Optimization: Generally, a single-objective optimization problem can be defined as follows:

$$\begin{array}{lll} \min & f(x) & \\ \text{s.t.} & g_j(x) \leq 0 & j = 1, \ldots, J \\ & h_k(x) = 0 & k = 1, \ldots, K \\ & x_i^L \leq x_i \leq x_i^U & i = 1, \ldots, N \\ & x \in \Omega, & \end{array} \tag{1}$$

where $f(x)$ represents the objective function, x_i represents the i-th variable to be optimized, x_i^L and x_i^U its lower and upper bound, g_j the j-th inequality constraint and h_k the k-th equality constraint. The objective function f is supposed to be minimized by satisfying all the equality and inequality constraints.

Various characteristics of the SOOP should be accessed to decide the best algorithm to solve it. This includes the number and type of variables, uncertainty in the objective function, or constraints.

In contrast to SOO, multi-objective optimization involves finding the optimal solution for multiple, conflicting objective functions, altering (1) with the following.

$$f(x) \longrightarrow f_m(x),\ m = 1, .., M, \text{with } M > 1. \tag{2}$$

MOOPs can be more complex than SOOPs because often there is not a single optimal solution that is the best for all objectives. Instead, there may be a set of solutions that are Pareto-optimal, meaning that they are not dominated by any other individual. An individual dominates the other if it is better in every objective function.

The difficulty now consists not only in the convergence to the true Pareto front but also in maintaining diversity among solutions.

When the number of objectives increases, turning the problem into a MaOOP, more difficulties arise, such as a large portion of the population being non-dominated, evaluating diversity and performance becoming computationally expensive, and the difficulty to recombine solutions.

2) Evolutionary Algorithms: EAs differ from traditional gradient searches in two fundamental ways: instead of a single point-to-point search, the algorithm uses a population of candidate solutions. This population evolves through a specific process, introducing randomness during the search. This makes them robust and able to handle large search spaces. Multiple alternative trade-offs can be generated in a single optimization run, making them a suitable match for MOOPs. They follow these general steps:

1) Set initial population.
2) Create new individuals.
3) Choose the next generation.

Using natural processes as inspiration, many EAs were proposed, such as the standard Genetic Algorithm (GA) [11] which uses operators such as survival, selection, crossover and mutation to evolve the population into the next generation. GA versions tailored for specific problems have been popularized, such as Ant Colony Optimization (ACO) [12], which mimics the behavior of ants to solve combinatorial problems.

The rise in computational power raised interest in more complex MOOPs, with the creation of the tailored GA for MOOPs, NSGA-II, using the concept of pareto rank survival and crowding distance to ensure good convergence and diversity among solutions. NSGA-III upgrade it to face MaOOPs, using a set of user-defined reference directions to focus the search in specific areas.

B. Optimization Frameworks

As the aim of the work is to build a general application for optimization, it is of interest to explore the most established general optimization frameworks available to know what is the best match to be the application backbone.

In Java [4], jMetal [13] and MOEA [3] stand as the most versatile and highly customizable frameworks. PlatEMO [5], developed in Matlab [6] includes a powerful GUI and multiple state-of-the-art EAs to be tested in multiple problems. PyGMO [14], DEAP [8] and pymoo [7] stand as the most established Python frameworks. The first focus on parallelization, and the second is an evolutionary computation framework for rapid prototyping and testing of ideas. The latter provides state-of-the-art single and multi-objective optimization algorithms, as well as tools for visualization, decision-making, and parallelization. The summary is presented in Table I.

TABLE I
Optimization Frameworks.

Name	License	First Release	Lang.	Visual.	GUI
MOEA	GNU LGPL	2011	Java	✓	✓
jMetal	MIT	2015	Java	✓	✗
PlatEMO	MIT	2016	Matlab	✓	✓
DEAP	LGPL-3.0	2012	Python	✗	✗
PyGMO	GPL-3.0	2013	Python	✗	✗
pymoo	Apache 2.0	2020	Python	✓	✗

C. Graphical User Interfaces

To understand what is already available in terms of GUIs for optimization, it is important to look at existing general frameworks that already provide one. In the case of the Java-based MOEA framework, a simple GUI is provided to illustrate its potential. Alongside several tutorial pre-programmed examples that show the MOEA capabilities at solving different problems, a Diagnostic Tool is provided. This tool allows the user to run one algorithm on one problem at a time. In addition, a window is displayed to analyze and compare the results.

In the case of PlatEMO, the developers make use of the capabilities of the MATLAB software, which is able to incorporate the visual aspects with the code itself. Starting from the MATLAB editor, the GUI is divided into three modules.

The Test Module lets the user test one algorithm on a problem with specified parameter settings.

The Application Module allows the user to create his own problem and solve it through the available algorithms. The Experiment Module allows the evaluation of multiple algorithms in multiple problems, creating a table to observe the statistical results of the run. Table II provides a summary of the key characteristics between the available GUIs and the proposed solution.

TABLE II
CHARACTERISTICS OF AVAILABLE OPTIMIZATION FRAMEWORKS WITH A GUI AND PROPOSED SOLUTION.

Name	Free	Python Based	Runs mult. algo. at once	Tests diff. Operators	User Code Integrat.
MOEA	✓	✗	✗	✗	✗
PlatEMO	✗	✗	✓	✗	✓
Prop. Sol.	✓	✓	✓	✓	✓

It is now possible to understand where the proposed solution can provide benefits. Besides being Python-based, it is supposed to allow the user to quickly test an Algorithm through multiple operators, and easily integrate self-coded options to the App. Furthermore, it should be available cost-free and should not have resistance to academic, personal, or commercial use.

III. ARCHITECTURE

The optimization framework chosen to be the backbone of the application is pymoo [7]. Under the permissive Apache License that meets **requirement 1** of being cost-free, it stands out as one of the most comprehensive frameworks. It offers state-of-the-art optimization algorithms and benchmark problems, meeting **requirements 2 and 3**. In addition, its many visualization techniques will help with **requirement 4**. The GUI is built using the PyQt library.

The architecture has two main modules, frontend and backend, detailed in the following sections. Fig. 1 shows the main components of each module and the links to the pymoo framework.

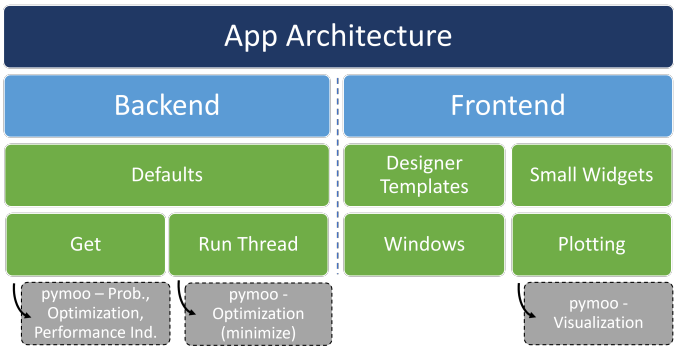


Fig. 1. Overview of application architecture.

A. Frontend

The frontend module is responsible for presenting the interface and interacting with users. It has the code for the Main and Edit Windows of the application. The static graphical components of these windows are made through the Designer Tool offered in PyQt [15] and stored in the *designer_templates* folder. The connections and logic are added later in the code. The following sections describe the role of each window.

1) Main Window: Shown at the top of Fig. 2, this is the window where the app naturally opens. To tackle not only single but also multi-objective optimization, it has two pages (highlighted in the dotted green box), one for each type of optimization. The pages contain the same widget class, *MainTabsWidget*, instantiated once for SOO and the other for MOO.

It has two fixed tabs: Run Options Tab, the one shown in Fig. 2, and History Tab. The first is responsible for collecting the user options for the run: the problems, algorithms, performance indicators, termination criteria, and number of seeds. These options are refered to as Run Options, and the optimization process through all of them a Run.

These Run Options can be saved into a *.pickle* file as a Python dictionary, where each key corresponds to a column header and the associated value is a list with the IDs in the column. This file can later be loaded into the app.

After pressing the "Run" button (hidden behind the Edit Window in the Figure), it tries to instantiate a *RunThread* object looping through the columns: Problems→Algorithms→Perf. Indicators and Terminations.

If all objects are instantiated for the chosen options, a *HistoryFrame* widget is created in the History Tab, which shows the progress and allows the user to cancel the Run if needed. When finished, a Run Tab is created to analyze the Run results, displaying a Table and Plot sections, as shown in Fig. 3. Similarly to Run Options, the Run can be saved in a *.pickle* file and later loaded back into the app.

The "Plot" button instantiates the plot class which corresponds to the selected ID in the first combo box of the Plot Section. This class must be a child from the *QPlot* class, coded in *plotting.py*. Already four child classes, based on pymoo's implementation, are available to meet **requirement 4**: *QProgress* (Fig. 4); *QFitnessLandscape* (Fig. 5); *QParetoSets* (Fig. 6); *QPCP* (Parallel Coordinates Plot).

There is still the need to give the user an option to not only change the parameters that will be used in each problem, algorithm, performance metric, termination criterion and plots, but also create new ones. This is an essential part of **requirement 6**, making the optimization process as customizable as possible through the app without losing its ease of use. For this, the Edit Window is devised.

2) Edit Window: Shown at the bottom of Fig. 2, this class is instantiated twice, one for SOOPs and the other for MOOPs. It needs a nested dictionary for the *parameters* argument, with the following structure at each level.

1) Each first-level key corresponds to a tab that will be created with the respective table of class options.

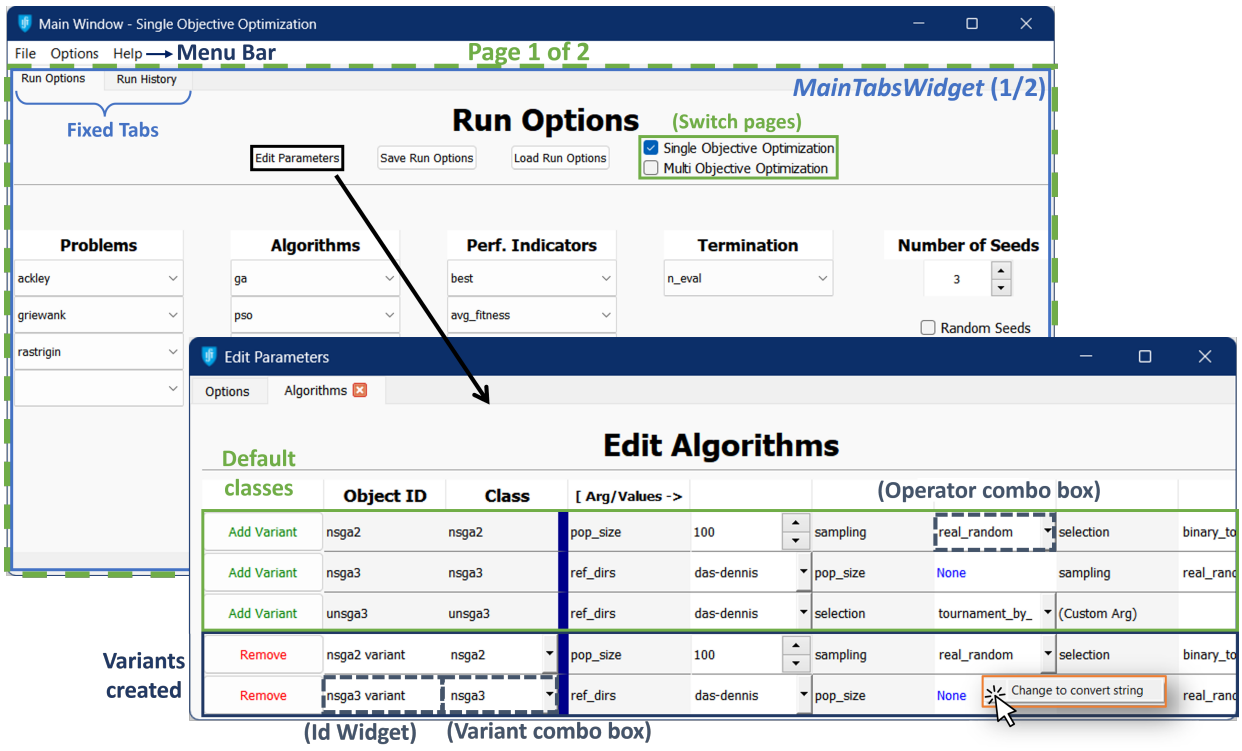


Fig. 2. Main and Edit Windows.

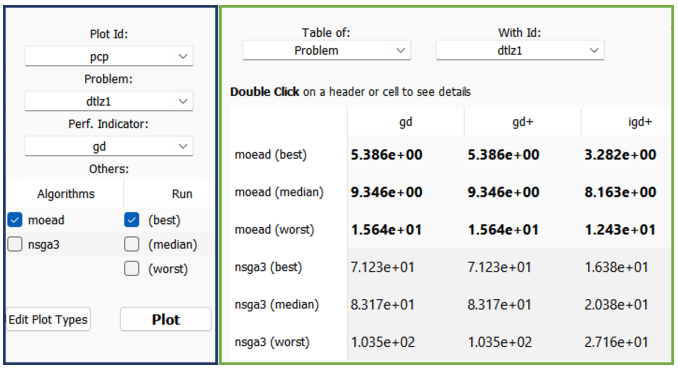


Fig. 3. Run Tab Plot (blue) and Table (Green) sections.

2) The second-level keys correspond to the IDs for each row of the respective table.
3) In the third level, the key-value pairs represent the arguments and default values for the specific class.

The first-level keys include the plot types and all the Run Options classes: problems, algorithms, performance indicators, and terminations. Furthermore, all GA operators (sampling, survival, selection, crossover, and mutation) also have their dedicated tab, as well as other components often used in MOO, such as decomposition and reference directions.

As seen in Fig. 2, variants can be added (marked with the blue box) to create different initializations of the same default class (marked with the green box). The ID widget guarantees that there are no repeated IDs.

The default values for the initialization arguments of the classes are represented by different widgets coded in *small_widgets.py*, depending on their type. Float and integer types are represented with the *SpinBox* widget; Bolean types with *MyCheckBox*; Strings with *MyTextEdit*; Operators with *MyComboBox*, which allows choosing between the respective operator available IDs, shown in the combo box options.

A signal, a PyQt event used for communication between objects, with the name *itemsSignal*, is emitted from the ID widgets whenever an ID is changed. The signal is propagated by the Edit Window through its own signals *runOptionsUpdates* and *operatorUpdates*.

These signals are respectively connected to the combo boxes from the "Run Options" tab of the Main Window (with the exception of the "Plot Type" combo box which is present in the *RunTab*), or the operator combo boxes (such as the one highlighted in a dotted gray box) among the *EditTabs*, depending on their respective class.

When the signal is received, the items are updated to include the new ID. This guarantees that the items in the combo boxes are always up-to-date, preventing errors trying to get unexistent IDs from a table. As with the Run Options, the Parameters can be saved and loaded through a conversion to and from a *.pickle* file.

To help meet **requirement 6**, another feature was added to the application, allowing greater customization of the parameters. The *MyTextEdit* widget allows the user to change between a "plain string" or a "convert string", saving it as the

Boolean attribute *convert* (pop-up highlighted in orange at the bottom of Fig. 2).

The first option will assume the value of the argument to be just the sequence of characters, while the second one will try to substitute the keywords present in Table III for their respective values and evaluate the mathematical expression if possible.

TABLE III
KEYWORDS FOR MYTEXTEDIT SET TO CONVERT THE STRING AND THEIR SUBSTITUTION.

Keyword	Substitution	Can be used in		
		Algo.	Term.	PI
None	python's "None" null value	✓	✓	✓
n_var	number of variables in the problem to be optimized	✓	✓	✓
n_obj	number of objectives in the problem to be optimized	✓	✓	✓
prob_id	id of the problem	✓	✓	✓
prob_object	problem object	✓	✓	✓
prob_pf	PF of the problem	✓	✓	✓
algo_id	id of the algorithm		✓	✓
algo_object	algorithm object		✓	✓
term_id	id of the termination			✓
term_object	termination object			✓

B. Backend

The backend handles the core functionality of the application, such as data processing and the actual optimization process. It is divided into three files - *get.py*, *defaults.py*, and *run.py*, detailed in the following sections.

1) Get: This file contains a *get* function for each type of class to be instantiated through the app - *get_algorithm*, *get_problem*, etc. As shown in Code Snippet 1), they contain a dictionary for the SOO options and another for MOO options. The dictionaries link a unique string to a pymoo class of the respective type, and will try to instantiate it with the given arguments.

Through these functions, the class strings presented in the app can be used to choose the class of the object to be instantiated. The values of the arguments used in the initialization are converted from the widgets of the respective row of the Edit Window.

2) Defaults: The *Defaults* class, defined in the file, goes through the options provided by the *get* functions of problems, algorithms, terminations, and operators in the *get.py* file when instantiated. It stores their arguments with the default values in the *parameters* dictionary, one of the class attributes.

It starts with the available operators and then checks the algorithms. If one of the algorithms' arguments key matches an operator key, it substitutes it with the correspondent ID. Otherwise, if a non-operator argument requires an object to be instantiated, it is not presented in the app.

3) Run: In the *run.py* file, the *RunThread* class is defined. It is a child of the threading class *QThread*, so the optimization running in the background does not interfere with the graphical interface of the app. In addition, it makes it possible to perform multiple optimizations, since different *RunThread* objects are created when the user clicks the "Run" button, which is important to meet **requirement 3**.

The optimization is started for every problem-algorithm combination and with all different seeds using the *run* method. At the beginning of each single run (problem-algorithm-seed combination), a signal is emitted (*progressUpdate* method) with a string and a percentage to inform the optimization progress or if an error occurs. The *RunThread* class stores the results in its *data* attribute, with the structure shown in Table IV.

TABLE IV
STRUCTURE OF DATA.

prob.	algo.	seed	n_eval	n_gen	igd	gd	
dtlz1	moead	2	100	5	8.83	9.73	[...]
[...]							

Additionally, a set of feasible solutions is stored in the Python dictionary best_gen, also a *RunThread* attribute, for visualization purposes. If the optimization concerns SOOPs, it will store the search and objective space values of the population in the last generation, in the 'X' and 'F' entries of the dictionary, respectively. If it concerns MOOPs, the dictionary will store the search and objective space values referring to the best Pareto set found throughout the optimization process.

C. Other Aspects

1) User Code Integration: As stated in **requirement 5**, the application should be easily extensible. For this goal, it is essential that user-programmed classes can be seamlessly integrated into it.

Having coded a child class from the respective pymoo parent, the user only has to locate the type of class (problem, algorithm, termination, mutation, crossover, selection, sampling, decomposition, or reference directions), import it, and add it to the respective *get* function. Code Snippet 1) shows the process of adding a user coded *MyAlgorithm* class into the single-objective optimization part of the app.

Code Snippet 1 Adding an algorithm into the app.

```
def get_algorithm(name, *args, **kwargs):
    from pymoo[...] import NSGA2
    [...]
    from my_classes import MyAlgorithm

    ALGORITHMS_SINGLE = {
        'ga': GA,
        [...]
        'my_algorithm': MyAlgorithm}

    ALGORITHMS_MULTI = {
        ('nsga2', NSGA2),
        [...]}
    return [...]
```

The algorithm is then going to appear in the app, with the default values represented by the respective widget.

2) Robustness: As the desktop application influence is only on the local machine, the main task of the app's architecture regarding robustness is to prevent what can be a well-intended miss-use of the application to crash it.

This is achieved throughout the app using Python's *try/except* statements, carefully located at the app's most sensible points where user input could break it: 1. In the optimization a problem through an algorithm; 2. In the instantiation of a class with the arguments from the Edit Window; 3. Trying to convert a convertible *MyTextEdit* argument; 4. Instantiating a plot for a specific problem and algorithm. All errors are handed gracefully, with the error message shown in a pop-up window so the user understands what needs to be altered.

In addition, the load points of the app (load Run Options, Parameters, or Run) are set with a verification process to add robustness. The dictionaries imported from the pickle files are checked to see if they have the expected structure (necessary keys, adequate number of levels, etc.) before being used to load the values into the app itself.

3) Scalability: In a desktop application, the work load has to be performed on the single local machine. The scalability is therefore limited by the machine's characteristics and performance. There are, however, some techniques used to maximize what the local machine can handle at a given time.

First, the pymoo framework already offers out-of-the-box parallelization of evaluations whenever possible. This is done through a vectorized evaluation using matrices where each row represents a solution. Therefore, a vectorized evaluation refers to a column that includes the variables of all solutions. By using vectors, the objective values of all solutions are calculated at once.

Furthermore, the already mentioned *RunThread* allows the user to start multiple optimization processes at once, being limited by the maximum number of *historyFrame* spaces, seven at a time.

IV. Results

The GitHub repository[1] contains the installation guide and all the code related to the experiments in the respective subfolder of the *thesis* folder. It is possible to verify the results by running the correspondent scripts or loading the saved pickle files of the respective Run into the app and reproduce it.

The objective of these experiments is not primarily to provide new insights or improvements to the single and multi-objective research domain, but rather to prove the methodology of such studies can be assisted by the use of the app.

A. App-Framework match

Pymoo [7], led by Julian Blank, has more than 37 contributors and an active community of developers. For this reason, if it is ensured that the use of the app produces exactly the same results as the direct use of the framework through a Python script, its reliability is reasonably guaranteed.

This is done through the *Test* and *FrameWorkTest* classes. The first automatically creates the app and simulates a "Run" button click for a series of different Run Options, saving the results in a file. The latter directly instantiates the same Run Options in pymoo and performs the optimization.

As there is no difference between the produced files for all the algorithms and problem classes in the app, it can be stated that the application has no conflicts with what would be expected through the direct use of pymoo.

B. Single-Objective Optimization

Regarding single-objective optimization, the PSO [16] algorithm is explored to see in what ways its performance is changed by the different initialization arguments. The study used as a reference is that of Trelea [17], which explores optimization in problems such as the Rosenbrock and Rastringin functions.

The metrics for evaluating performance are different from the general multi-objective study case. A certain fitness value is set as the goal, and the algorithm should obtain it in the least number of function evaluations possible. Table V provides the details of the problems to be optimized.

TABLE V
PROBLEM OPTIONS FOR PSO OPTIMIZATION.

Name	Range of x	Optimal F	Number of Variables	Goal for F
Rosenbrock	$[-30, 30]^n$	0	30	100
Rastringin	$[-5.12, 5.12]^n$	0	30	100

The algorithm is tested on two sets of fixed weights for its particle update equation, with it's velocity described by:

$$V_d^{(i)} = \omega V_d^{(i)} + c_1 r_1 \left(P_d^{(i)} - X_d^{(i)}\right) + c_2 r_2 \left(G_d^{(i)} - X_d^{(i)}\right). \tag{3}$$

Set 1 has w = 0.6, and both c_1 and c_2 equal to 1.7. Set 2 has w = 0.73, and both c_1 and c_2 equal to 1.5. Pymoo's adaptive implementation is set to False, so the weights remain the same through the optimization. The velocity rate is set to 0.9, and *perturbe_best* argument set to false. The runs are made with 20 different seeds. The population varies between 15, 30, and 60 elements.

As for the termination, a multi-criterion termination class is designed, which takes into consideration three aspects: if the goal of the respective function was met, if the best solution found is the same over the last 1000 function evaluations, and if the maximum number of 10000 generations was met. If any of these criteria are met, the run is terminated.

The "Run" button is then clicked, and after it finishes, the results shown in Table VI are presented. The column E-FE - Expected Function Evaluations, is the only one not showing directly through the app, as it is the result of post-processing calculation by dividing the Average FE (A-FE) by the Success Rate (SR) .

[1]https://github.com/tomaslibanomonteiro/Thesis-code.git

TABLE VI
PSO ALGORITHM PERFORMANCE.

Pop	Set	Rastrigin-30 A-FE	Rastrigin-30 SR	Rastrigin-30 E-FE	Rosenbrock-30 A-FE	Rosenbrock-30 SR	Rosenbrock-30 E-FE
15	1	**2,694**	0.25	10,776	**14,523**	0.30	48,408
	2	4,973	0.20	24,863	26,128	0.95	27,504
30	1	3,971	0.70	**5,672**	21,858	**1.00**	21,858
	2	6,774	0.50	13,548	14,790	**1.00**	**14,790**
60	1	6,578	**0.80**	8,222	18,138	**1.00**	18,138
	2	10,478	**0.80**	13,097	19,311	**1.00**	19,311

In terms of population size, the lowest population has all the best average function evaluation values, as expected. In contrast, the 60-member population has the highest success rate, as more elements can search the space more thoroughly. This can also be seen in Fig. 4, where the average fitness of the population is plotted by function evaluations. In the starting point, the PSO with the lowest population converges quickly, but in the last stages it often misses the goal by getting stuck in a close local minimum. On the other hand, the 60-member PSO has a slower start, but better search space exploration leads to an overtake at approximately 9000 function evaluations.

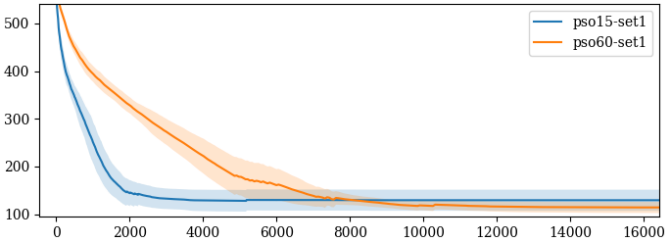


Fig. 4. Average population fitness on Rastrigin-30 by function evaluations.

A population of 30 members proves to achieve the best trade-off between exploitation and exploration, as noted in the original study. It performs better in both functions, regarding the expected function evaluations.

It is worth noting the difference between Set 1 and Set 2. As the first has a lower inertia weight value and a higher social weight value, its population should, on average, converge faster. This is verified as the first takes on average 18% less function evaluations to achieve the goal. Set 2 on the other hand has a success rate 23% higher. In general, these differences will cancel out, leading to approximately the same expected function evaluations. This means that there is no better set for all cases, but it depends on the specific function. Set 1 outperformed Set 1 on the Griewank and Rastrigin problems, while Set 2 performed better on Rosenbrock.

To assist in the visualization of what is happening in the optimization process, a Run on 2 variable problems with a population size of 3, 7 and 15 is done. By plotting the fitness landscape (the search space corresponds to the x and y axes and z to the fitness value) it is possible to see how the steep local minima on Rastringin can trap even large populations (Fig. 5(a)), explaining the lower success rates. On the other hand, through Fig. 5(b) it is possible to see that even in unimodal functions, a small population (3 elements in this case) can converge too quickly and fail to explore the search space to find the global minimum.

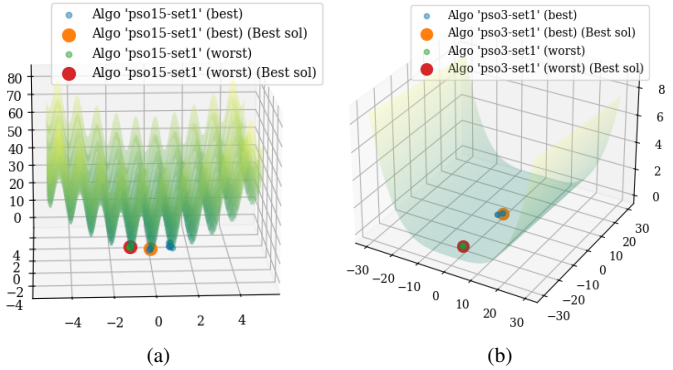


Fig. 5. Fitness Landscape of Rastrigin-2 (a) and Rosenbrock-2 (b).

C. Multi-Objective Optimization

To understand what the proper methodology is for multi-objective optimization research and how the app could facilitate it, a well-known study presented in the article introducing the NSGA-III algorithm, by Deb and Jain [18], is taken as a reference. It tests NSGA-III against MOEA/D [2] on a series of problems from the DTLZ test suite [19].

MOEA/D decomposes a multi-objective problem into single-objective optimization sub-problems. It then evolves a population of candidate solutions over multiple generations, combining solutions from neighboring sub-problems to optimize them cooperatively. In this study case, both the Tchebycheff and penalty-based boundary intersection (PBI) decomposition methods are explored [2]. This is done in the app by creating two MOEA/D variants with the decomposition operator chosen accordingly.

The reference directions are based on the Riesz s-Energy [20] method with 100 points. The rest of the algorithms parameters are set according to the default values in the pymoo framework.

The algorithms are tested in DTLZ2 and DTLZ3 problems with the number of objectives and variables varying, respectively, between 3-12, 5-14, and 8-17. The performance indicator chosen is the Inverted Generational Distance [21], as it can measure both the accuracy and diversity of the population. All problem-algorithm combinations are ran on 10 different seeds. Finally, the termination type used in the study is the number of generations, set to 500.

The “Run” button is then pressed and, when it finishes, the application displays the results showing the worst, median, and best values for each run, presented in Table VII. The best value of its type for a given problem is indicated in bold font.

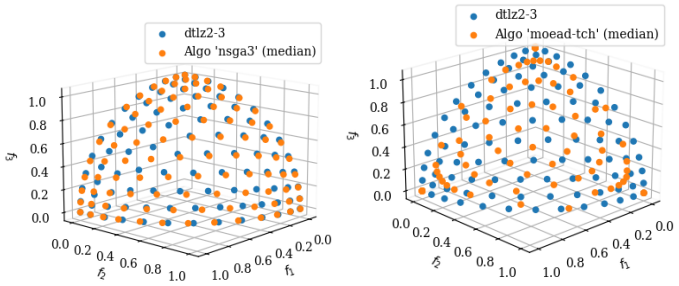
Looking at Table VII, it is clear that MOEA/D-TCH performs worse than the other two algorithms in both tests. MOEA/D-PBI performs better in DTLZ2 for all objectives. In

DTLZ3, the IGD values are closer, with NSGA-III performing slightly better than MOEA/D-PBI on 3 objectives, and the tendency inverting for 8 objectives.

TABLE VII
IGD METRIC FOR BEST, MEDIAN AND WORST RUNS.

	DTLZ2			DTLZ3		
Objectives:	3	5	8	3	5	8
MOEA/D (TCH)	7.54E-02	4.21E-01	6.94E-01	7.29E-02	5.19E+00	4.83E+00
	7.78E-02	4.42E-01	8.60E-01	7.60E-02	1.30E+01	5.16E+01
	7.86E-02	4.50E-01	8.67E-01	1.05E+00	8.94E+01	8.48E+01
MOEA/D (PBI)	**3.33E-04**	**3.06E-03**	**6.92E-03**	7.85E-02	**9.25E-02**	**7.31E-02**
	4.39E-04	**3.54E-03**	**7.56E-03**	2.71E-01	3.47E+00	**1.15E+00**
	5.48E-04	**4.54E-03**	**1.02E-02**	1.71E+00	2.19E+01	**2.78E+00**
NSGA-III	3.00E-02	1.24E-01	1.40E-01	**2.61E-02**	2.02E-01	2.11E-01
	3.01E-02	1.24E-01	1.41E-01	**3.17E-02**	**3.98E-01**	2.47E+00
	3.02E-02	1.24E-01	1.43E-01	**1.05E+00**	**1.85E+00**	6.13E+00

This corroborates what was found in the initial study done in the paper introducing NSGA-III [18]. Also, using the existing plotting options of the app, the obtained Pareto sets are plotted, shown in Fig. 6 for DTLZ2 with 3 objectives. As expected, NSGA-III is converging better than MOEA/D-TCH to the reference points.



(a) (b)

Fig. 6. Final Pareto sets obtained on 3 objective DTLZ2 for the median runs of NSGA-III (a), and MOEA/D-TCH (b).

In Fig. 7, the value of the IGD indicator in the MOEA / D-PBI and NSGA-III algorithms is plotted by function evaluations. The thick line indicates the average value and the shadows represent the standard deviation. The y axis is in the logarithmic scale, and thus the lower and upper limits of the standard deviation appear to be unequal. An observation similar to the original study can be made: MOEAD/PBI starts with a steeper progress, but is surpassed by the NSGA-III algorithm around the 10,000 function evaluations, maintaining a worst value from then on.

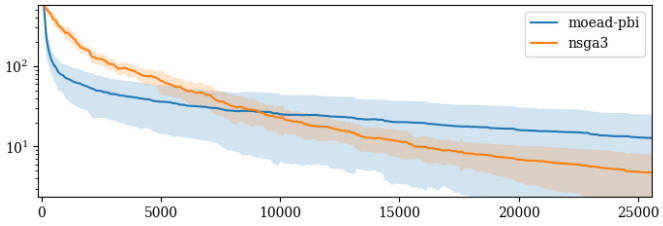
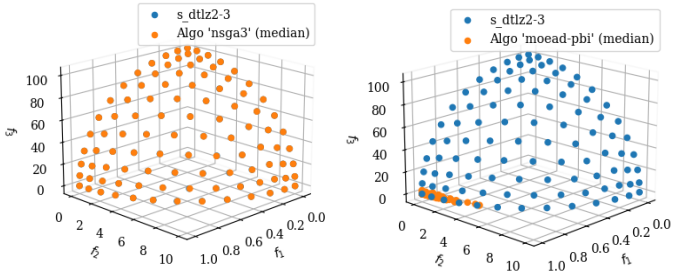


Fig. 7. IGD progress for MOEA/D-PBI and NSGA-III on 5 objective DTLZ3.

This corroborates what was found in the initial study done in the paper introducing NSGA-III [18]. Also, using the existing plotting options of the app, the obtained Pareto sets are plotted, shown in Fig. 6 for DTLZ2 with 3 objectives. As expected, NSGA-III is converging better than MOEA/D-TCH to the reference points.

Another aspect verified is the effectiveness of the normalization procedure inherent in NSGA-III. When performing the same optimization with a scaled DTLZ2, where the i-th objective is scaled by 10^i, the Pareto sets obtained clearly demonstrated the contrast between the algorithms' ability to balance the optimization of the different objectives (Fig. 8), with MOEA/D failing to maintain diversity among solutions.



(a) (b)

Fig. 8. Final Pareto sets obtained on 3 objective scaled DTLZ2 for the median runs of NSGA-III (a), and MOEA/D-PBI (b).

D. Worst-Case Scenario

This section shows how the application can be useful even if the optimization at hand is to be made through a specific algorithm, problem, and even visualization technique that is not already implemented in pymoo.

1) Problem: An adaptation of the renown Traveling Salesman Problem (TSP) is done. It is transformed into a multi-objective problem with mixed variable types, where the salesman not only has to choose the best order to visit every city once, but also the mode of transportation to minimize both cost and time of the total trip.

For each type of transportation, the cost and time matrices must be provided. Naturally, one mode of transport can make the trip shorter but more expensive or vice versa. The nadir point, later used as a reference point to calculate the hypervolume, is set as the worst value found in the time and cost matrices multiplied by the number of cities.

For the case study, three transportation methods are available: car, train, and plane. The car matrices are calculated on the basis of the distance value between the two cities. The train and plane time matrices are calculated by multiplying the car time matrix by $1/\text{factor}$, and the cost matrix by the factor. The factors for train and plane are set by default as 5 and 10, respectively. Lastly, the problem is inserted into the application by adding it to the *get_problem* function options.

2) Algorithms: There are two cases to be studied. The framework already provides operator classes for making a generic GA capable of handling a simple permutation problem.

These classes are then altered to fit the problem variable representation, and a simple flip mutation is added to guarantee the variability in the transportation methods.

The class *PermutationNSGA2*, is created with all these altered operators set as default values. The survival and selection processes are copied from the default NSGA-II methods.

Furthermore, a new algorithm (ACO-NSGA2) is created that mixes the NSGA2 and ACO methods to see if it obtains better results from the heuristic-based approach that ACO provides, tailored to graph problems. In this regard, the NSGA-II survival operator *RankAndCrowding*, is significantly extended in *RankAndCrowdingACO*. After the usual surviving procedure, the pheromone update is performed with the equation below.

$$\text{new ph}(i,j) = ev \cdot \text{ph}(i,j) + \sum_{k}^{m} \Delta\, \text{ph}(i,j)^k \qquad (4)$$

where $\text{new ph}(i,j)$ is the updated pheromone, ev is the pheromone evaporation coefficient, m is the number of ants and $\Delta\, \text{ph}(i,j)^k$ is the amount of pheromone deposited by k-th ant, typically given by a fixed value divided by the ant's fitness value. As the problem is multi-objective, a decomposition strategy is developed assuming as the fitness value the sum of the normalized objectives.

Each ant selects its next move based on a probability correlated with current pheromone trails and a heuristic function that calculates the direct distance between cities.

As time and cost values are strongly correlated with the heuristic value, it is expected that this algorithm will outperform *PermutationNSGA2*. However, it remains to be seen how the original crossover and mutation operators affect the search process. Two versions of the algorithm are to be tested, ACO-NSGA2 and ACO-NSGA2 (s) - simple, the latter maintaining only the transport flip part of the mutation and performing no crossover after the ACO procedure.

3) Visualization: A special *TSPplot* class is designed, to plot the grid with the coordinates of the cities and different colors for the different transportation methods, as seen in Fig. 11. The problems, set to 10, 20 and 60 cities, are optimized by the three detailed algorithms with a population of size 20, for 10 different seeds, and 4,000 function evaluations as the termination criteria.

4) Optimization Results: After the "Run" button is clicked and the Run finished, the results for the normalized hypervolume are presented in the app, shown in Table VIII.

TABLE VIII
AVERAGE NEGATIVE HYPERVOLUME VALUES.

	MixedTSP-10	MixedTSP-30	MixedTSP-60
ACO-NSGA2 (n)	-0.897	-0.928	**-0.941**
ACO-NSGA2 (s)	**-0.903**	**-0.931**	-0.939
Permutation-NSGA2	-0.894	-0.897	-0.869

As expected, both ACO-NSGA2 versions outperform Permutation-NSGA2. The heuristic-based method is shown to converge better, as it uses the problem specific knowledge (the cities coordinates) to better evolve the solutions. With regard to the two ACO-NSGA2 types, the simple performs better in MixedTSP-10 and 20, and the normal one in MixedTSP-60.

Fig. 9 presents the negative hypervolume values by function evaluations for MixedTSP-10, 20 and 60. In the 10 cities problem, the three algorithms manage to converge quickly, with ACO-NSGA2 (s) achieving the best results.

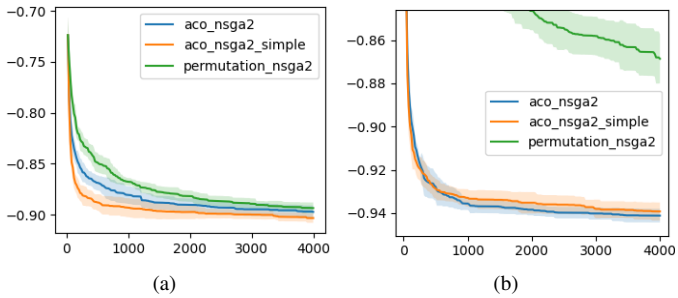


Fig. 9. Negative hypervolume value by function evaluations for MixedTSP-10 (a) and 60 (b).

With 60 cities, Permutation-NSGA2 unable to find a good solution for the termination set, and ACO-NSGA2 (s) outperforms its twin before 500 function evaluations. This suggests that with a simpler problem, the mutations and crossovers of ACO-NSGA2 actually stall the convergence process, but as they become more complex, they are helpful in not getting stuck in local minimums and converge to the true Pareto front. The Pareto sets obtained by the median runs of the algorithms are shown in Fig. 10, with the x and y axes corresponding to the time and cost values of each solution.

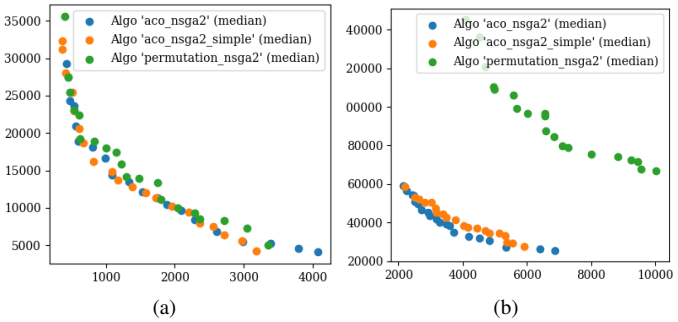
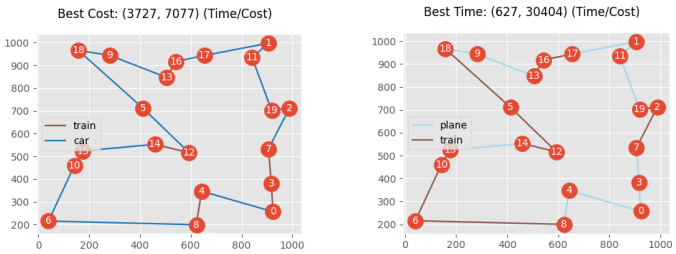


Fig. 10. Final Pareto sets obtained for MixedTSP-10 (a), and 60 (b).

They are also in accordance with what can be perceived with the hypervolume convergence Figures. In Fig. 10(a), the Pareto sets are closer to each other. In Fig. 10(b), it becomes clear that ACO-NSGA2 surpasses the (s) version.

To analyze in more detail the best solutions found by the algorithms on the 20 cities problem, the *TSPPlot* is used, resulting in Fig. 11.

Fig. 11(a) shows the path with the best cost. As expected, the most frequent transportation method is the car, the cheapest on average. However, there are still some connections by train. In contrast, Fig. 11(b) shows the path with the best time, where the most frequent transportation method is the plane, but multiple connections are also traveled by train. Therefore, it can only be concluded that the algorithm has yet to converge



(a) (b)

Fig. 11. Graph of the solutions obtained by ACO-NSGA2 (simple) with the best path cost (a), and path time (b).

to the Pareto optimal front extreme with the lowest cost. The same analysis holds for the time objective function.

It stems from this observation that in order to achieve these Pareto front extremes, it would be necessary to raise the number of function evaluations set as the termination criterion, or change the mutation transport flip part to privilege paths with the same transportation mode, in order to obtain a quicker convergence.

It is worth noting that both paths, while being the extremes of the Pareto front, have the same sequence of visited cities. This is because the order of the path generally minimizes both objectives, so they are not competing in that regard. This suggests that the algorithm converged to the best order and is simply finding the best transportation. However, it is important to note that this can not always be the case if the time and cost matrices are not calculated using the distance between cities directly.

V. Conclusions

In Section II, the fundamental concepts in single and multi-objective optimization were explored. In addition, the available framework options were analyzed and the offered GUIs detailed to understand where the proposed solution could bring value. In Section III, the design choices for the architecture were presented to ensure that all requirements are met.

In Section IV, the reliability of the application was ensured through a comparison with the direct use of pymoo. Furthermore, single and multi-objective studies done through the application demonstrate that it has the tools to properly assist and accelerate research in the area. Lastly, an example incorporating new problems, algorithms, and visualization options shows the versatility and customization capabilities of the app.

In summary, the work demonstrates the potential of integrating an existing optimization framework with a custom-built graphical interface to create a solution that fulfills the needs not yet met by the market. The application's open-source nature and its balance between ease of use and customization make it a valuable resource for both novice and experienced users in the field of optimization.

References

[1] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-II," *Trans. Evol. Comp*, vol. 6, pp. 182–197, Apr. 2002.

[2] Q. Zhang and H. Li, "Moea/d: A multiobjective evolutionary algorithm based on decomposition," *IEEE Transactions on Evolutionary Computation*, vol. 11, no. 6, pp. 712–731, 2007.

[3] D. Hadka, "Moea framework." http://moeaframework.org/, 2015. Accessed: November 22, 2023.

[4] J. Gosling, *The Java language specification*. Addison-Wesley Professional, 2000.

[5] Y. Tian, R. Cheng, X. Zhang, and Y. Jin, "PlatEMO: A MATLAB platform for evolutionary multi-objective optimization," *IEEE Computational Intelligence Magazine*, vol. 12, no. 4, pp. 73–87, 2017.

[6] D. J. Higham and N. J. Higham, *MATLAB guide*. SIAM, 2016.

[7] J. Blank and K. Deb, "pymoo: Multi-objective optimization in python," *IEEE Access*, vol. 8, pp. 89497–89509, 2020.

[8] F.-A. Fortin, F.-M. De Rainville, M. Gardner, M. Parizeau, and C. Gagné, "Deap: Evolutionary algorithms made easy," *Journal of Machine Learning Research, Machine Learning Open Source Software*, vol. 13, pp. 2171–2175, 07 2012.

[9] G. Rossum, "Python reference manual," 1995.

[10] K. Srinath, "Python–the fastest growing programming language," *International Research Journal of Engineering and Technology*, vol. 4, no. 12, pp. 354–357, 2017.

[11] J. H. Holland, *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.

[12] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, vol. 26, no. 1, pp. 29–41, 1996.

[13] J. J. Durillo and A. J. Nebro, "jmetal: A java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, no. 10, pp. 760–771, 2011.

[14] F. Biscani and D. Izzo, "A parallel global multiobjective framework for optimization: pygmo," *Journal of Open Source Software*, vol. 5, no. 53, p. 2338, 2020.

[15] M. Summerfield, *Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming (paperback)*. Pearson Education, 2007.

[16] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, pp. 1942–1948 vol.4, 1995.

[17] I. C. Trelea, "The particle swarm optimization algorithm: convergence analysis and parameter selection," *Information Processing Letters*, vol. 85, no. 6, pp. 317–325, 2003.

[18] K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 577–601, 2014.

[19] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler, "Scalable test problems for evolutionary multiobjective optimization," in *Evolutionary multiobjective optimization: theoretical advances and applications*, pp. 105–145, Springer, 2005.

[20] J. Blank, K. Deb, Y. Dhebar, S. Bandaru, and H. Seada, "Generating well-spaced points on a unit simplex for evolutionary many-objective optimization," *IEEE Transactions on Evolutionary Computation*, vol. 25, no. 1, pp. 48–60, 2021.

[21] H. Ishibuchi, H. Masuda, Y. Tanigaki, and Y. Nojima, "Modified distance calculation in generational distance and inverted generational distance," in *International conference on evolutionary multi-criterion optimization*, pp. 110–125, Springer, 2015.