

A Proposed Framework for the Analysis of the Performance of Newly Proposed Metaheuristics

by

Remi EHOUNOU

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE IN
PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE
OF MASTERS WITH THESIS IN SOFTWARE ENGINEERING
M. Sc. A.

MONTREAL, MARCH 21, 2022

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC



Rémi Ehounou, 2022



This Creative Commons licence allows readers to download this work and share it with others as long as the author is credited. The content of this work can't be modified in any way or used commercially

BOARD OF EXAMINERS (THESIS M.Sc.A.)
THIS THESIS HAS BEEN EVALUATED
BY THE FOLLOWING BOARD OF EXAMINERS

Professor Alain April, Thesis Supervisor
Department of Software Engineering and IT at École de technologie supérieure

Professor Ali Ouni, President of the Jury
Département of Software Engineering and IT at École de technologie supérieure

Professor François Coallier, Member of the jury
Département of Software Engineering and IT at École de technologie supérieure

THIS THESIS WAS PRESENTED AND DEFENDED
IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

MARCH 15, 2022

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ACKNOWLEDGMENT

Foremost, I would like to thank my research director, the professor Alain April for giving me the opportunity of making the transition from a master's in aerospace engineering into Software engineering with Thesis.

I would also like to thank Mr. Iannick Gagnon, my senior scientist for his assistance and insights throughout the process of this research. His advice and comments played a key role in the successful execution of this research.

Last but not least, I would like to thank my family especially my parents for their support and encouragement through this process. Their unrelenting support allowed me to remain focused on my task.

Cadriciel d'analyse de la performance des propositions de nouvelles métaheuristiques

Rémi Ehounou

RÉSUMÉ

L'évaluation des métaheuristiques joue un rôle important dans le développement de nouveaux algorithmes pour les problèmes d'optimisation. Cependant, plusieurs études dans le domaine critiquent le manque de rigueur méthodologique dans plusieurs études de cas. L'implémentation de cadres d'analyse améliorés est proposée comme une réponse adéquate à ce problème.

Cette recherche étudie la faisabilité de ce principe en implémentant les recommandations de bonnes pratiques méthodologiques comme l'intégration d'analyses statistiques comme des études de régression entre la classification d'un problème et la performance des métaheuristiques à le résoudre. L'importance d'utiliser des études contrôlées est reconnue comme étant nécessaire pour assurer que les résultats soient fiables et répétables.

Des méthodes de classification des problèmes de test en catégories basées sur des caractéristiques calculées de la librairie FLACCO (Feature-Based Landscape Analysis of Continuous and Constrained Optimization) native au langage R sont présentées. La sélection des paramètres d'évaluation, des instances de problèmes de test et des méthodes statistiques est aussi traité. Un cadriciel nommé Metaheuristics Design and Analysis Framework (MDAF), fut augmenté et testé avec une étude de cas présenté à la fin de ce mémoire.

Il en ressort que l'évaluation de la performance d'algorithmes d'optimisation permet de déterminer des paramètres efficaces pour les études d'optimisation sur des fonctions spécifiques.

Mots-clés : métaheuristiques, optimisation mathématique, optimisation stochastique, cadre d'analyse

A Proposed Framework for the Analysis of the Performance of Newly Proposed Metaheuristics

Rémi Ehounou

ABSTRACT

Metaheuristics benchmarking plays a key role in developing new algorithms for optimization problems. However, a number of published studies criticize the lack of reliable and repeatable experimentation in the analysis of many newly proposed metaheuristics. The enhancement of an analysis framework is studied and implemented as an adequate response to this issue.

This research presents a framework for the design and analysis of the performance of newly proposed metaheuristic algorithms named Metaheuristics Design and Analysis Framework (MDAF). The importance of well-constructed and controlled studies is recognized as a necessary step for the benchmarking results to be reliable and repeatable.

Methods such as the classification of problem instances into categories based on a representation calculated from the FLACCO (Feature-Based Landscape Analysis of Continuous and Constrained Optimization) library are discussed. The selection of benchmarking parameters, problem instances, and statistical methods are also presented.

It is observed from the analyses that the implementation of valid experimental methods is an effective strategy for Benchmarking the performance of optimization algorithms.

Keywords: metaheuristics, mathematical optimization, stochastic optimization, analytical framework

TABLE OF CONTENTS

	Page
INTRODUCTION	3
0.1. Problem Definition	6
0.2. Contribution	7
0.3. Research Objective	8
0.4. Future Work	9
CHAPTER 1 LITERATURE REVIEW	10
1.1. Choice of a Programming Paradigm	20
1.2. Conclusion	21
CHAPTER 2 RESEARCH PLANNING	23
2.1. Phase I — Definition	25
2.2. Phase II – Planning	25
2.3. Phase III – Operation	26
2.4. Phase IV — Interpretation	27
2.5. Conclusion	28
CHAPTER 3 DESIGN OF THE FRAMEWORK	29
3.1. Requirements tracking	38
3.2. Conclusion	39
CHAPTER 4 EXPLORATORY LANDSCAPE ANALYSIS	41
4.1. Conclusion	44
CHAPTER 5 IMPLEMENTATION OF THE METAHEURISTICS DESIGN AND ANALYSIS FRAMEWORK	45
5.1. Default Test Functions	48
5.2. PyPI Packaging	50
5.3. Experimental Planning	50
5.4. Automated Testing	51
5.5. Efficient computation	53
5.6. Conclusion	54
CHAPTER 6 CASE STUDY: BENCHMARKING THE PARTICLE SWARM ALGORITHM WITH DIFFERENT PARAMETERS TO DETERMINE THE EFFICIENT VALUES	55
6.1. Methodology	55

6.2.	Results.....	56
6.3.	Discussion.....	69
6.4.	Test Case Conclusion	75
CONCLUSION AND PERSPECTIVES		76
APPENDIX I. SIMULATED ANNEALING PSEUDO CODE		78
APPENDIX II. PACKAGE META DATA.....		79
APPENDIX III. MDAF CODE		80
APPENDIX IV. SAMPLE STRUCTURE OF PREPROGRAMMED TEST FUNCTIONS		94
APPENDIX V. AUTOMATED TESTS ALGORITHMS.....		95
APPENDIX VI. FLACCO Features Data.....		98
APPENDIX VII. PSO PERFORMANCE DATA		103
APPENDIX VIII. CASE STUDY CODE		118
APPENDIX IX. PSO ALGORITHM		121
APPENDIX X. PACKAGE INIT FILE.....		126
LIST OF REFERENCES		127

LIST OF TABLES

	Page
Table 1.1. Existing Metaheuristic Frameworks organized by capabilities	11
Table 2.1 Phase I – Definition of the Research	25
Table 2.2 Phase II - Planning Stage of the Research	26
Table 2.3 Phase III - Planning stage of the framework design and test.....	27
Table 2.4 Phase IV - Interpretation and results of the research	28
 Table-A VI-1 The gcm Feature Set Values	 98
Table-A VII-1 c1 Parameter Values Used for the Case Study	103
Table-A VII-2 c2 Parameter Values Used for the Case Study	105
Table-A VII-3 Average Number of calls Required to Optimize the Leon Test Function using PSO for each c1 and c2 Parameter variations	106
Table-A VII-4 Standard Errors for the Average Numbers of calls Statistics Required to Optimize the Leon Test Function using PSO for each c1 and c2 Parameter variations	107
Table-A VII-5 Average Number of calls Required to Optimize the Price2 Test Function using PSO for each c1 and c2 Parameter variations	108
Table-A VII-6 Standard Errors for the Average Numbers of calls Statistics Required to Optimize the Price2 Test Function using PSO for each c1 and c2 Parameter variations	109
Table-A VII-7 Average Number of calls Required to Optimize the Brown Test Function using PSO for each c1 and c2 Parameter variations	110
Table-A VII-8 Standard Errors for the Average Numbers of calls Statistics Required to Optimize the Brown Test Function using PSO for each c1 and c2 Parameter variations	111

II

Table-A VII-9 Average Number of calls Required to Optimize the Ackley2 Test Function using PSO for each c1 and c2 Parameter variations	112
Table-A VII-10 Standard Errors for the Average Numbers of calls Statistics Required to Optimize the Ackley2 Test Function using PSO for each c1 and c2 Parameter variations.....	113
Table-A VII-11 Average Number of calls Required to Optimize the Styblinski-Tang Test Function using PSO for each c1 and c2 Parameter variations.....	114
Table-A VII-12 Standard Errors for the Average Numbers of calls Statistics Required to Optimize the Styblinski-Tang Test Function using PSO for each c1 and c2 Parameter variations	115
Table-A VII-13 Average Number of calls Required to Optimize the Step Test Function using PSO for each c1 and c2 Parameter variations	116
Table-A VII-14 Standard Errors for the Average Numbers of calls Statistics Required to Optimize the Step Test Function using PSO for each c1 and c2 Parameter variations.....	117

LIST OF FIGURES

	Page
Figure 0.1 - Sample Objective Function Topology	4
Figure 0.2 Structure of the Proposed Framework.....	9
Figure 1.1 - Multimodal test function example: Six-Hump Camel Back.....	17
Figure 1.2 - Unimodal test function example: Trid	17
Figure 1.3 - Example of large basin in Rosenbrock's function	18
Figure 1.4 - Example of the Topology of a Benchmarking Test Function.....	19
Figure 2.1 - Work Breakdown Structure of this Research Activity.....	24
Figure 3.1 - A Diagram of the "4+1" View Model	30
Figure 3.2 - Legend of the Components in the Flowchart	30
Figure 3.3 - Main Logical View of the FrameWork.....	32
Figure 3.4 - The Preprocessing Module.....	34
Figure 3.5 - The Postprocessing Module	35
Figure 3.6 - The Experiment Running Process.....	36
Figure 3.7 - The Algorithm Module	37
Figure 3.8 - The Technical Debt Landscape.....	38
Figure 4.1 - 3D Barrier Tree Visualization by FLACCO	42
Figure 4.2 - Cell Mapping Visualization by FLACCO.....	43
Figure 4.3 - Information Content Plot by FLACCO.....	43
Figure 5.1 - Test Function Visualization produced by MDAF	47

Figure 5.2 - Radar Plot of Bukin2 and Bukin6 test functions' ela_meta feature set by MDAF	48
Figure 5.3 - Bukin6 Visualization.....	49
Figure 5.4 - Brown Function Visualization	50
Figure 5.5 - Snapshot of the Automated Tests Results in VS Code	52
Figure 5.6 - Automated Python Doctest Results.....	53
Figure 5.7 - Optimally Loaded Central Processing Unit (CPU).....	54
Figure 6.1 - Result of the Visualize2D function for the Step Test Function	57
Figure 6.2 - Result of the Visualize2D function for the Price2 Test Function	57
Figure 6.3 - Result of the Visualize2D function for the Leon Test Function	58
Figure 6.4 - Result of the Visualize2D function for the Brown Test Function	58
Figure 6.5 - Result of the Visualize2D function for the Ackley2 Test Function.....	59
Figure 6.6 - Result of the Visualize2D function for the Styblinski-Tang Test Function	60
Figure 6.7 - Result of the plotfuncs Function for gcm features	61
Figure 6.8 - Average Number of Function Calls for Solving the Leon Test Function vs c1 and c2 Parameter Values	63
Figure 6.9 - Standard Errors for the Number of Function Calls for Solving the Leon Test Function vs c1 and c2 Parameter Values	63
Figure 6.10 - Average Number of Function Calls for Solving the Price2 Test Function vs c1 and c2 Parameter Values	64
Figure 6.11 - Standard Errors for the Number of Function Calls for Solving the Price2 Test Function vs c1 and c2 Parameter Values	64
Figure 6.12 - Average Number of Function Calls for Solving the Brown Test Function vs c1 and c2 Parameter Values	65
Figure 6.13 - Standard Errors for the Number of Function Calls for Solving the Brown Test Function vs c1 and c2 Parameter Values	65

Figure 6.14 - Average Number of Function Calls for Solving the Ackley2 Test Function vs c1 and c2 Parameter Values	66
Figure 6.15 - Standard Errors for the Number of Function Calls for Solving the Ackley2 Test Function vs c1 and c2 Parameter Values.....	66
Figure 6.16 - Average Number of Function Calls for Solving the Styblinski-Tang Test Function vs c1 and c2 Parameter Values	67
Figure 6.17 - Standard Errors for the Number of Function Calls for Solving the Styblinski-Tang Test Function vs c1 and c2 Parameter Values	67
Figure 6.18 - Average Number of Function Calls for Solving the Step Test Function vs c1 and c2 Parameter Values	68
Figure 6.19 - Standard Errors for the Number of Function Calls for Solving the Step Test Function vs c1 and c2 Parameter Values	68
Figure 6.20 - PSO Average Number of Function Calls Heatmap for Sub Region of Interest for the Ackley2 Test Function	72
Figure 6.21 - Standard Error Heatmap for the Above Average Values of the Performance of PSO on Ackley2.....	72
Figure 6.22 - Comparison of the Representation and Performance of the Ackley2 and Step Test Functions	74

LIST OF ABBREVIATIONS AND ACRONYMS

ELA	Exploratory Landscape Analysis
EP	Experiment Plan
FBP	Flow Based Programming
FLACCO	Feature-Based Landscape Analysis of Continuous and Constrained Optimization
MDAF	Metaheuristics Design and Analysis Framework
MOF	Metaheuristics Optimization Framework
NaN	Not a Number
NFLT	No Free Lunch Theorem
OOP	Object Oriented Programming
PSO	Particle Swarm Optimization
SA	Simulated Annealing

LIST OF ALGORITHMS

	Page
Algorithm 5.1. PyTest Command	52
Algorithm-A I-1 Simulated Annealing Pseudo Code.....	78
Algorithm-A II-1 Package Meta Data for MDAF	79
Algorithm-A III-1 MDAF Algorithm Implementation.....	80
Algorithm-A IV-1 The Bukin2 Test Function Implementation.....	94
Algorithm-A V-1 The Unittest Automated Testing Script	95
Algorithm-A VIII-1 Case Study Python Script used with MDAF to Generate the Results.....	118
Algorithm-A IX-1 The PSO Algorithm.....	121
Algorithm-A X-1 MDAF Package Init File.....	126

INTRODUCTION

Several disciplines like computer science, engineering and economics require optimization problems to be solved. To accomplish this, different mathematical techniques and tools have been proposed. This research is interested in a special kind of optimization algorithm family called metaheuristics. More specifically, it is concerned with the evaluation and benchmarking of new metaheuristics proposals for various types of problems.

The field of metaheuristics optimization has been applied to a wide range of real-world problems with success. Such problems are often too complex to solve manually and/or too resource intensive for common methods such as those of linear programming (Sala & Müller, 2020). According to (Sörensen & Glover, 2013), metaheuristics define a high-level algorithmic framework that provides strategies to implement low-level heuristic optimization algorithms (Sörensen & Glover, 2013). Heuristics, from *heuriskein* in Greek which means “to find”, are algorithms that are designed using problem-specific information to find a good solution with relative ease (Bianchi et al., 2009). Therefore, metaheuristics can be seen as an abstract framework for heuristics. Metaheuristics research has grown into an established discipline, and it now contains a vast body of knowledge that can be applied to almost every field that involves optimization.

Constrained optimization problems are concerned with the maximization or minimization of objective functions whose input variables are constrained usually for system identification reasons. These constraints generally represent characteristics of the real-world problems being modeled and help ensure that the solutions found make physical sense. Objective functions model one or more variables of the studied systems that we wish to control. Constraints can also be applied on the runtime, although this type of constraint is considered to be of a different kind than that of the constraints on the objective function variables since they are problem

independent. Figure 0.1 shows a sample objective function topology. A typical goal is to find the position of the global minimum of the function. This can be accomplished by using the well-known gradient descent algorithm (Curry, 1944), but the prevalence of local minima almost guarantees that it will find a suboptimal solution unless the starting point is at or near the global optimum. It is therefore necessary to use a different approach, such as the use of stochastic parameters, to let the optimization escape from these regions (Luke, 2013). Optimization problems can also be categorized in different ways which determine the type of metaheuristics that will be most effective for solving it. One such distinction is made between separable and non-separable objective functions. The former is considered easier to solve due to the simple linear relationship between the components (Brownlee, 2007).

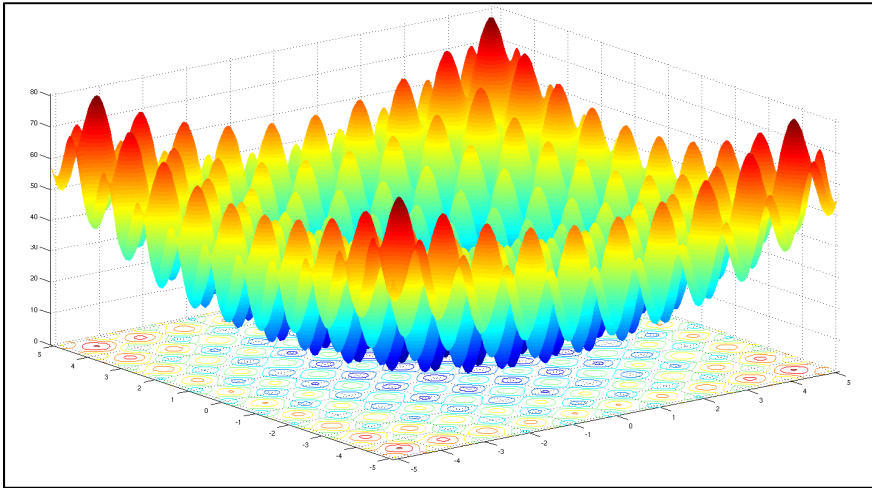


Figure 0.1 - Sample Objective Function Topology (Diego Andrés Alvarez Marín, 2010)

A canonical metaheuristic algorithm called Simulated Annealing (SA) follows a process similar to the annealing process in materials engineering (Luke, 2013). This mimicking of nature is quite common with recently developed metaheuristic algorithms since many natural processes tend to have optimizing properties (Bianchi et al., 2009). As the name indicates, this algorithm optimizes objective functions by modeling the physical process by which particles inside a material rearrange themselves to achieve thermal equilibrium. The version outlined by Luke is based on the principle of local search heuristics which explores the problem space by

comparing the best obtained value of the objective function to that of a neighbor. It then selects the new point if it is better than the current one, or randomly selecting a new point depending on the temperature parameter: the higher the temperature parameter the more random changes are (Bianchi et al., 2009). A control parameter which effectively represents the temperature of the annealing process determines how much randomness is included in the optimization process. This principle is presented in APPENDIX I, which outlines the simulated annealing process in pseudo code. The temperature parameter T is used in the calculation of the probability factor p . The calculated value is then compared to a randomly generated number to decide if it accepts the new position or not. It is this mechanism that makes SA a stochastic algorithm. It is also this mechanism that allows SA to sometimes move to non-improving positions. The so-called temperature of the algorithm is typically set to a high value (eg. $T=1000$) at the beginning of the process and then decreased gradually as time goes by (Luke, 2013). When the temperature is reduced to a small enough value, SA becomes a purely greedy algorithm which is an algorithm that consistently follows the locally optimal choice at each iteration. The stochastic component is useful at the start of the process to help reduce the chances of the algorithm getting trapped in a local extreme value.

As the field keeps advancing, algorithms are continuously being proposed so a systematic evaluation method is needed. To solve this issue, test problems are also being created to benchmark new metaheuristics to determine what types of problems they are good at solving. Benchmarking is defined as the determination of a metaheuristic's performance when applied to a specific type of problem using a combination of theoretical and empirical approaches (Sala & Müller, 2020). In the context of this research, a framework is developed to evaluate metaheuristics on standard test problems to determine their performance. For example, the Cross-Entropy Toolbox proposed in (Zdravko Botev et al., 2004) contains a diverse set of test functions for both constrained and unconstrained optimization that can be used to evaluate the performance of metaheuristics. This is discussed in more detail in Section 0. In addition to the test functions, principles and theories have been developed to address the challenges involved in benchmarking metaheuristic algorithms. For instance, the No Free Lunch Theorem (NFLT)

stipulates that no single algorithm is appropriate for all possible types of problems (Koppen et al., 2001) (Brownlee, 2007) (Sala & Müller, 2020).

0.1. Problem Definition

The field of metaheuristics is criticized for the lack of rigor in the current evaluation methods of newly proposed algorithms. The following points outline the main issues reported in the literature concerning current practices (Eiben & Jelasity, 2002):

- **Ad hoc selection of test functions:** New Algorithm proposals are often tested on a set of functions without the justification of a solid experimental design, which prevents a full understanding of their performance in different contexts;
- **Overgeneralization of obtained results:** The results of benchmark tests published for new algorithm proposals are often generalized beyond the specific functions on which they have been tested. Better definition and classification of the problems into categories is needed to improve generalizability;
- **Poor reproducibility:** Since the source code associated with the algorithm proposal is often not made available to the public, it is difficult or practically impossible to replicate/validate the claims and reported results;
- **Lack of clearly stated objectives:** Proposed algorithm Results/claims are sometimes interpreted without being related to the experimental objectives and expectations.

The common practice of what esteemed Operations Research professor John Hooker called ‘competitive testing’, in which algorithms are directly compared to each other by their direct runtime performance metrics (e.g., convergence time), is highly discouraged (Hooker, 1995). He describes them as non-scientific, citing two undesirable consequences when using this algorithm comparison technique, and proposes better methods for comparing algorithms. In fact, directly comparing the performance numbers of metaheuristics has led to a focus on speed which distracts researchers in the field from building well-designed and controlled experiments. The author also adds that machine speed and implementation-specific details of the various metaheuristics (e.g., programming language and style, architecture, etc.) have too

much of an impact on the runtime of algorithms for direct comparison to be scientifically meaningful.

Another issue in metaheuristics benchmarking is the classification of problems into classes as well as the selection of problem instances for testing. For the selection of problem instances, it is preferable to perform the study on as many classes of problems as possible to acquire a general understanding of the algorithm. When it comes to the classification of test problem instances, the structural properties as well as the methods of generation of the problems (i.e. real world model or artificially constructed test functions) are proposed as possible taxonomic criteria (Brownlee, 2007).

Finally, the optimizing tendency of a genetic algorithm is a characteristic that complex systems share. Therefore, carefully tuning such algorithms is an important preliminary step to testing and special attention also needs to be paid to the selection of performance metrics (Brownlee, 2007). These recommendations must be included in future benchmark frameworks to ensure their effectiveness at comparing metaheuristics.

In short, the benchmarking of metaheuristics requires careful consideration. In addition, recommendations for conducting this process exist in the literature such as the use of controlled experiments, the classification of problem instances into categories and the tuning of algorithms before benchmarking.

0.2. Contribution

This research is part of a larger research project on the design and analysis of metaheuristics. It aims to implement and automate some of the recommendations from (Gagnon, 2020) about the necessity of benchmarking new metaheuristics proposals. Recommendations from the literature which are described in Chapter 0 will also be implemented in this research. Figure 0.2 shows a structure for the project where the elements in green are the components being addressed by this research. This research proposes a framework for testing metaheuristics with modules for statistical analysis as well as data preprocessing to identify the characteristics of each experiment to be run through the framework. For example, the preprocessing module can

be used to set up a new study to benchmark a newly developed metaheuristic's performance on a specific family of problems. The framework will then be able run the tests before performing a statistical analysis. A post processing stage is also included for exporting the results data and performing some basic analysis. This framework will be implemented in the Python 3 programming language and deployed as a PyPI package. PyPI is the official repository of the Python programming language packages.

0.3. Research Objective

The main objective of this research is to propose an analytical framework that could help researchers increase the quality of their research. In order to experiment with a solution proposal, an initial version of a prototype software of the framework for metaheuristics performance benchmarking will be designed and implemented based on some of the recommendations made in (Gagnon, April & Abran, 2020). This experimental framework prototype could serve as a tool to perform more rigorous statistical analyses of newly proposed metaheuristic algorithms in the future. Before a software prototype can be designed and coded, Figure 0.2 shows a high-level view of the architecture of the proposed framework. The green boxes represent the components that will be addressed in this research. Note that an important characteristic of the framework's proposed software architecture is the ability for parallel computing which is described in more details in section 5.5 to improve performance. The following secondary objectives will also be addressed:

- A modular architecture to make it easy to change the algorithms being tested;
- The framework must be available on PyPI;
- The presence of functionality to address some of the literature recommendations: modular architecture for test instances and algorithms, implementation of parametric statistical methods, implementation of more precise runtime calculation technics to determine the computing loads demanded by the algorithms.

This research, therefore, proposes to automate the benchmarking process of new metaheuristics.

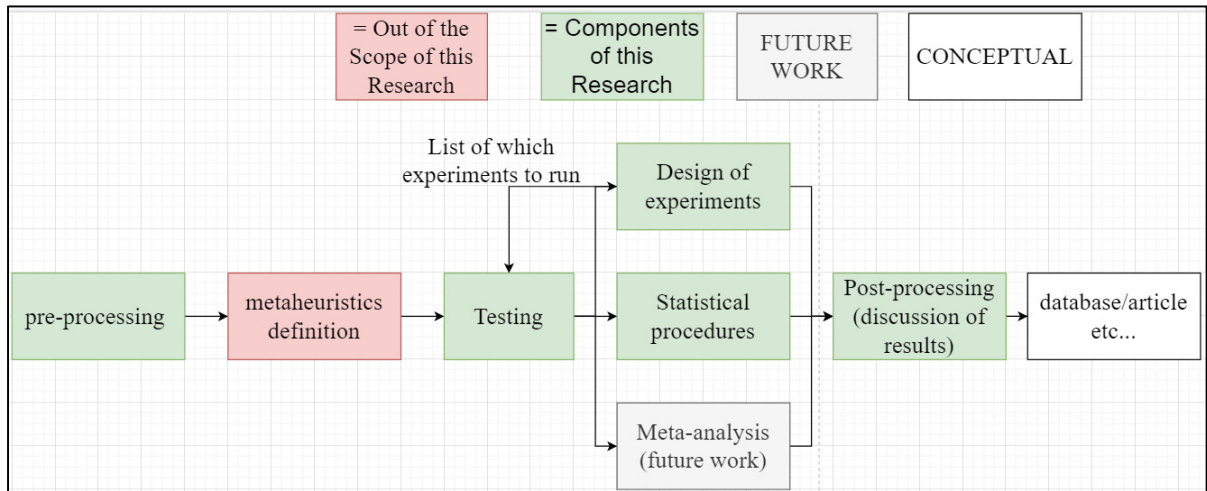


Figure 0.2 Structure of the Proposed Framework

0.4. Future Work

At the last step of this research, results will be assessed, and potential improvements will be identified. One improvement that can already be identified aims to adapt the proposed framework to include the possibility of performing meta-analysis by consolidating the results of multiple independent studies completed with the framework. These will not be addressed by the current research activities.

CHAPTER 1

LITERATURE REVIEW

This Chapter addresses the state of the art in benchmarking the performance of metaheuristics. An in-depth survey of the literature shows that this is still an unsolved problem in many ways. For example, there are no standardized sets of methods to accomplish this task (Jamil & Yang, 2013) (Brownlee, 2007). This has evidently motivated attempts to compile various test functions to facilitate the evaluation of newly proposed algorithms. It is relevant to highlight the difference between metaheuristic optimization frameworks (MOFs) and benchmarking frameworks. MOFs like the Opt4J (Lukasiewicz et al., 2011) (*Opt4J*, 2020) and EasyLocal++ (Di Gaspero & Schaerf, 2002) are used to design metaheuristics while benchmarking frameworks are used to test and compare the performance of existing metaheuristics. OptiBench by the company CIRG@UP, is an example of a benchmarking framework that is similar to what this research aims to accomplish (Peer et al., 2003). It comprises a library of standard problem instances and popular metaheuristics, an engine for data science, and a centralized results repository. Other frameworks similar to Opt4J include the following

Table 1.1. Existing Metaheuristic Frameworks organized by capabilities (Sean Luke et al., 2020) (Lukasiewicz et al., 2011) (*MOEA Framework, a Java Library for Multiobjective Evolutionary Algorithms*, n.d.)

Frameworks	MOF	Test Function Library
jMetal	Yes	Yes
EvA2	Yes	Yes
Watchmaker framework	Yes	Yes
ECJ27	Yes	Yes
JCLEC	Yes	No
MOEA framework	Yes	Yes
Paradiseo	Yes	No

It is recommended that benchmarking methods be more extensive than just the test functions provided with the MOF as is discussed in the paragraphs below. This research is focused on benchmarking frameworks and will be integrated with algorithms either generated by a MOF or taken from the literature like the simulated annealing algorithm (Gagnon et al., 2020).

Metaheuristics are used in almost all the domains of engineering (Gandomi & Yang, 2011) as well as in computer science and mathematics (Mendes et al., 2009) (*Web of Science Core Collections*, 2020). According to Web of Science, there were 681 publications with “metaheuristics” in their title in the past 5 years, and there is a growing number of citations each year, totaling 1509 citations since 2016 (*Citation Report*, 2020). Therefore, this field is growing and affects other disciplines, making it important that the algorithms being proposed are well understood to ensure their appropriate use and value. One of the most popular techniques to validate a new algorithm is to test it using common multi-modal mathematical functions whose global min/maxima are known and to compare the results (e.g., number of objective function evaluations, first hitting times, etc.) with those of other algorithms that already exist (Hooker, 1995). This validation method is criticized by (Hooker, 1995) and (Brownlee, 2007) for lacking rigor and for being too simplistic considering the complexity of the algorithms being evaluated. A more mathematically rigorous validation approach is suggested in which the new algorithms are evaluated on a series of benchmark problems and

the results are analyzed with statistical methods to better address their stochastic nature as well as ensuring their complexities (Brownlee, 2007) (Jamil & Yang, 2013). These test results may then carefully be extrapolated to other classes of problems based on how representative the sample problems used for the benchmark were (Sala & Müller, 2020). The authors also advocate for the substitution of real-life optimization problems with “computationally affordable representative benchmark problems” citing the No Free Lunch Theorem (NFLT) as a justification. This theorem is discussed in the following paragraphs. Possible problem instances are divided into classes based on their characteristics and the theory is that heuristic algorithms will have a similar performance when applied to problems of the same class. The following equation is a general mathematical description of typical optimization problems. This will be useful in standardizing the test problems in the framework as objects.

$$\begin{aligned} & \text{Given } f: R^n \rightarrow R \\ & \text{find } x^\phi \in R^n \text{ for which } f(x^\phi) \leq f(x), \text{ for all } x \in R^n \end{aligned} \tag{1}$$

Where $f(x)$ represents an objective function to be minimized or maximized depending on the specific problem being modeled.

The NFLT stipulates that all optimization algorithms have similar average performance when tested over all possible types of objective functions (Brownlee, 2007), which implies that the search for a single general-purpose algorithm is not a viable endeavor. More importantly, this theorem implies that the observed behavior of an algorithm on a specific problem class requires careful analysis when attempting to extrapolate to other problem classes. Therefore, optimization algorithms need to be matched to the specific problem classes they are suited for. This encourages the use of domain specific knowledge in optimization algorithms as a good practice since its ability to be applied to all possible optimization problems is not a meaningful characteristic of a heuristic. Even in the case of metaheuristic algorithms like particle swarm optimization (PSO) (Russell Eberhart et al., 2001), the use of domain-specific knowledge is recommended since they are designed for specific problem classes (Brownlee, 2007) (Jamil & Yang, 2013). Caution is advised when interpreting the NFLT since many do not apply it correctly due to the misunderstanding of its implications (Russell Eberhart et al., 2001). The theorem does not argue against the generalization of metaheuristics since most algorithms can

solve a wide range of problems, like machine learning algorithms (Moon et al., 2019) (Lu et al., 2020) (Zhang et al., 2020). However, it does warn that the performance of metaheuristics cannot be optimal for all types of problems.

The challenges involved in getting meaningful and publishable results must be discussed. They are divided into the following three main categories as per (Brownlee, 2007):

1. Parameter selection;
2. Problem instance selection;
3. Selection of statistical methods for the analysis and interpretation of results.

The selection of algorithmic parameters is a challenge because of their non-linear correlation with the performance of the algorithm. Many approaches are proposed for this issue such as self-adaptive parameters in which the parameters of the algorithm are encoded as binary strings, meta-algorithms which optimize the parameters of the algorithm in question, and sensitivity analysis which determines the sensitivity of the algorithm to changes in each parameter (Brownlee, 2007). Empirical selection by trial and error is also recommended as a good starting point, although deficiencies in this approach have been highlighted in (Francois & Lavergne, 2001). For example, they stipulate that seeking general rules for parametrization will lead to a lack of convergence and/or low efficiency. Many approaches are presented to address the selection of parameters: the Calibration and Relevance Estimation approach proposed in (Nannen, 2006) (Eiben & Jelasity, 2002); the steepest decent approach by (Coy et al., 2001); and the design of experiments (DOE) approach applied to metaheuristics research as in (Bartz, 2003). Finally, the use of Monte Carlo methods along with other statistical methods is presented for the intelligent sampling of the parameter space in (Birattari, 2002).

In addition to the selection of algorithmic parameters, a rigorous procedure for experimentation is important to ensure that the collected results will be statistically relevant. The following methods are proposed in (Brownlee, 2007):

1. Define the goals of the experiment;
2. Select measures of performance and factors to explore;
3. Design and execute the experiment;
4. Analyze the data and draw conclusions;
5. Report the experimental results.

The author also reminds the reader about the usual guidelines for scientific experimentation in general. He proposes that all important factors capable of influencing the results such as computer code and the runtime environment have to be reported, that the measures be taken precisely, that the results be compared with those of other methods, that all parameters be specified, and the importance of the use of statistical experiment design (Brownlee, 2007). He reminds the reader of these scientific principles because they are often lacking in the field of meta-heuristic benchmarking. Key issues emphasized in the algorithm benchmarking literature can be identified as the duplication of efforts by the various groups working in the field due to ineffective written communication, insufficient testing, occasional failure to test using state-of-the-art techniques, poor choices of parameters, conflicting results, and sometimes invalid statistical inference (Brownlee, 2007) (Peer et al., 2003).

The issues outlined above are important to the field and addressing them is the subject of this research. For example, as explained in (Brownlee, 2007), other similar fields of study, such as data science, have already passed the step of establishing standardized benchmarking methods and procedures. These methods and procedures act as standards for the field, increasing the trustworthiness of the results produced. This enables more effective collaboration between researchers and with industry. Scientifically, the establishment of standard benchmarking and testing methods is crucial as it is at the core of the scientific method itself. Standardizing metaheuristic performance benchmarking methods will make the results more reliable and easier to reproduce, thus eliminating the risks of duplicated efforts and providing robust grounds on which the field can build upon.

The next theme concerns the selection of the problem instances and classes that are used to perform the benchmarking tests. Many of these have been created by various actors in the

industry (Brownlee, 2007; Jamil & Yang, 2013). Examples of such problem instances are found in the GLOBAL library which is part of the cross-entropy toolbox (Zdravko Botev et al., 2004). This MATLAB toolbox is a collection of test problems with relevant data that can be of use to a researcher looking to benchmark a new heuristic. However, this collection of resources is challenging to implement into a study because it misses the statistical rigor alluded to in (Brownlee, 2007). This is also the case for most of the other resources available. All the components exist but there is a need to put them into a concise whole. Other resources for finding problem instances include GAMS World, which is a library of functions and test problems (*GLOBAL World - GLOBALLib*, n.d.). It was made to bridge the gap between academia and industry by providing a platform as well as resources to perform metaheuristic studies more easily. The “Cuter” testing environment was developed at the Polytechnique Montréal (Dominique Orban, 2002). It has now been superseded by a newer version named “CUTEst” which is an acronym for Constrained and Unconstrained Testing Environment with Safe Threads. This environment focuses on having a wide range of test functions numbering approximately 1150 in total (Gould et al., 2015). It is a mature software and is optimized to be able to run the tests efficiently. It could therefore play a role in this research for the testing phase of each analysis. It is important to note that this environment does not mention any experimental framework with statistical analysis which this research aims to address. The global optimization test problems' collection by (Abdel-Rahman Hedar, n.d.) is a collection of test problems divided in constrained and unconstrained groups. This collection is much less advanced than the “CUTEst” environment, but it is more versatile with the problems being formulated in mathematical form as well as MATLAB code compared to those of “CUTEst” which are only programmed in MATLAB. The collection of continuous global optimization test problems (*The COCONUT Benchmark*, n.d.) is a set of libraries containing test functions as well as some tools for performing basic analysis like calculating the standard unit time of an algorithm. The libraries of COCONUT use the AMPL modeling language, which is not ideal for this research since it plans to contribute to the Python library and should model its algorithms and functions using Python. Opt4J is a metaheuristics optimization framework that includes the following benchmark problem collections (Lukasiewicz et al., 2011) (*Opt4J*, 2020) :

- Knapsack;
- Zitzler–Deb–Thiele (ZDT) (Lim et al., 2015);
- Deb, Thiele, Laumanns and Zitzler (DTLZ) (Meneghini et al., 2020);
- WFG (Meneghini et al., 2020).

It is written in Java and its test functions can be used for this research. ECJ is also written in Java and contains benchmark test functions like Opt4J despite being focused on the algorithm design aspect of generating new metaheuristics (Scott & Luke, 2019; Sean Luke et al., 2020). It is mainly used for evolutionary algorithms. The examples given above are well known, but other such sets of test functions exist which are not necessarily well documented or recognized (Andrei, 2008; Auger & Hansen, 2005; *GEATbx - Genetic and Evolutionary Algorithms Toolbox in Matlab - Main Page*, n.d.; *Kaj Madsen - Head of Department•DTU Informatics*, n.d.; Mishra, 2006). For this research, these test functions will be useful as building blocks of the framework being implemented.

Despite the widespread use of benchmark functions, their simple use without a rigorous experiment design with statistical analysis is discouraged because they do not produce reproducible results due to the stochastic nature of the algorithms being tested (Brownlee, 2007). Brownlee advocates instead for more statistical analysis of the algorithm being tested on multiple test problems in a controlled setting, and a well-documented procedure is also advised to ensure the rigor of the experiment.

As proposed in (Jamil & Yang, 2013), benchmark functions can be classified in the following terms (Jamil & Yang, 2013):

- **Modality:** the number of peaks and valleys in the topology of the test function. This is relevant because the ambiguous peaks tend to trap the algorithm toward a local minimum (see Figure 3 and Figure 4).

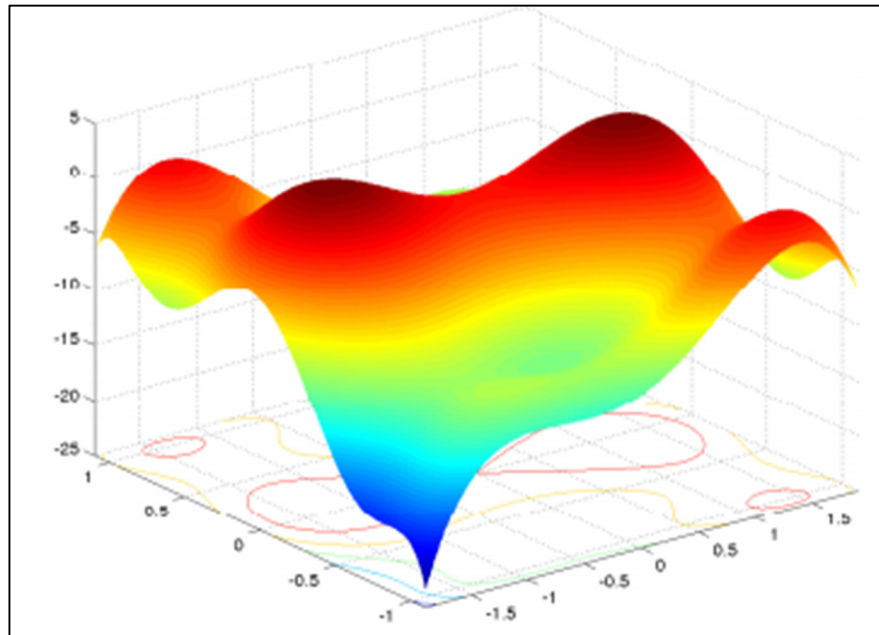


Figure 1.1 - Multimodal test function example: Six-Hump Camel Back (Li et al., 2013)

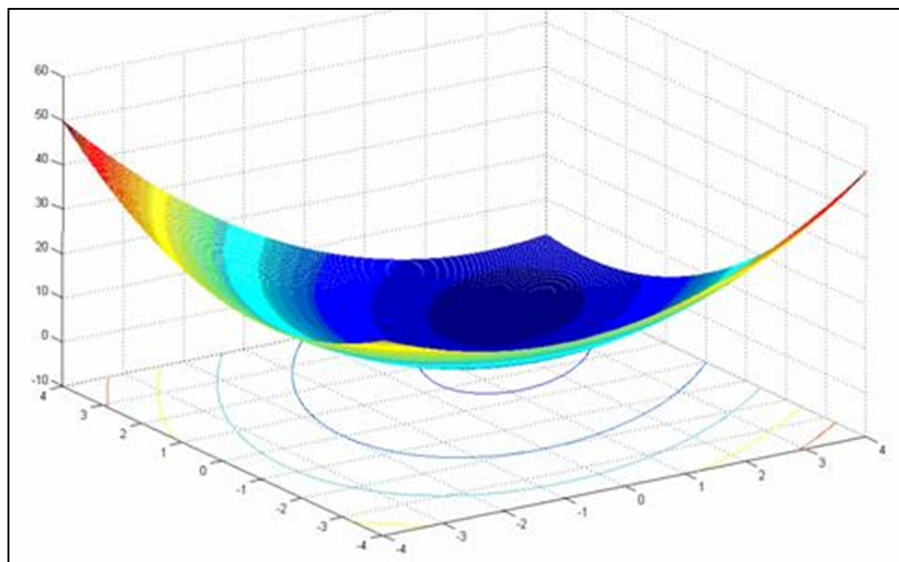


Figure 1.2 - Unimodal test function example: Trid (A. Hedar's, n.d.)

- **Basins:** They are defined as a steep decline surrounding a large area. They can hamper the optimization process if the algorithm falls into a basin that leads to a local minimum. The following figure shows a graphical representation of a basin.

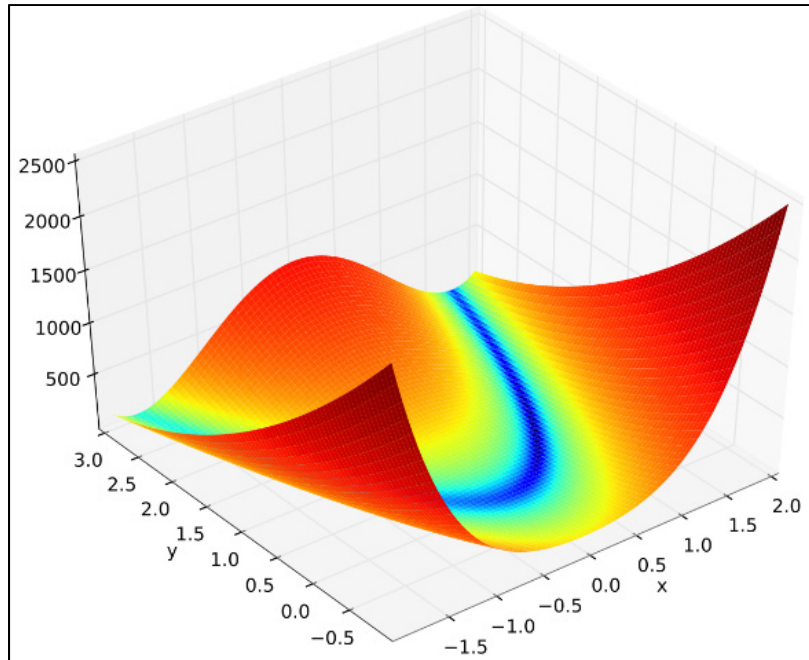


Figure 1.3 - Example of large basin in Rosenbrock's function

- **Valleys:** Like geographical valleys, they can slow down the process when the algorithm gets to the bottom of the valley because this type of region usually does not provide local information that leads to the global solution. Figure 1.4 shows two valleys running parallel to the coordinate axes.

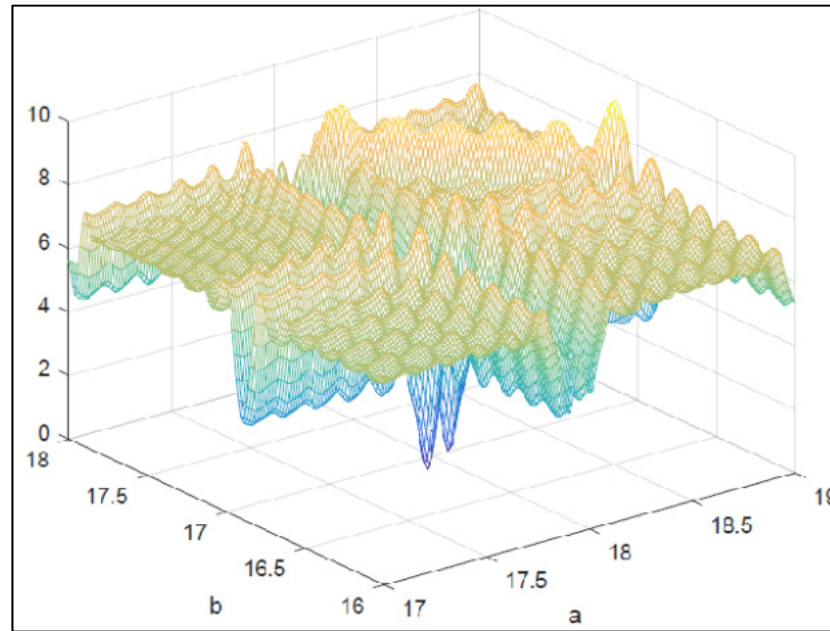


Figure 1.4 - Example of the Topology of a Benchmarking Test Function (Tsang, 2018)

- **Separability:** A measure of how difficult the test function is to solve. Separable functions are more linear than less separable ones. Equation 2.1 expresses the requirement for a function to be considered separable where x represents one of the components of the vector \bar{x} . Separable functions can, therefore, be optimized with respect to each of the input components separately while keeping the others constant. This greatly reduces the difficulty.

$$\frac{\partial f(\bar{x})}{\partial x_i} = g(x_i)h(\bar{x}) \quad (2)$$

- **Dimensionality:** The number of components of the input vector of the objective function. The difficulty of a problem generally scales with its dimensionality (Jamil and Yang, 2013).

1.1. Choice of a Programming Paradigm

To implement the framework, a programming paradigm must be selected. This section discusses object-oriented programming (OOP) and the flow-based programming (FBP) paradigms and their applicability to this research. They both have their strengths and weaknesses which make them more appropriate for modeling some programs compared to others.

OOP is a scheme in which the program is modeled as a collection of object types to be instantiated and which interact with one another via message passing (Grady Booch et al., 2007). Message passing happens when an object is passed as an argument to another object's methods or when an object's method directly accesses the attributes of another object for which it has access rights. This approach to programming offers the convenience of organizing the components of the solution into well-defined classes that can be interchanged or modified easily. For example, (Brownlee, 2007) proposed the creation of algorithm and problem classes to represent the components of an optimization scheme (Grady Booch et al., 2007). This makes the framework very modular, enabling the swapping of algorithms and test functions with relative ease when performing experiments. A drawback of OOP is the issue with encapsulation (Gamma et al., 1995). Encapsulation defines the accessibility of object data to the various methods of the program. A common approach is to set all the attributes of a specific class as private and creating getter as well as setter methods to access the needed ones. The problem is that this approach is not always followed correctly, and alternative encapsulation schemes are often poorly supported by even the most powerful object-oriented languages like C++ (Gamma et al., 1995). For this reason, an OOP architecture can be rendered unnecessarily complex if the design of the program is not carefully considered.

FBP is a scheme which is centered around the flow of information inside the program. In fact, this scheme organizes processes in chains with data going through them one after the other performing computation units (Grady Booch et al., 2007). This often requires the use of parallel computing (Grady Booch et al., 2007), which improves the performance of large programs compared to non-parallel computation programs by utilizing all the processing units available compared to single thread programs. Another advantage of this approach is that the computing

units can be interchanged readily to improve the program or try alternative units. Weaknesses of this programming scheme is the complexity of timing the various parallel processes involved in the program so that they communicate effectively. In fact, errors in these types of programs are not easy to detect and would allow the program to keep running before failing under the right conditions. A famous example of this is the reset issue of the pathfinder Mars rover (Durkin, 1997) (Bertrand Meyer, 1997). This means that special attention needs to be paid when correcting programs based on the FBP paradigm to make sure that no errors have been made in their implementation.

1.2. Conclusion

This chapter presented a review of the prior articles in the field of metaheuristic benchmarking. Common challenges in getting meaningful and publishable results are outlined and include issues of parameter selection, problem instance selection, and shortcomings in the application of solid statistical methods. The existing techniques and recommendations in the literature to improve the quality of the results obtained from metaheuristic benchmarking studies were then addressed. Some of these recommendations are to pay attention to the experiment design to make sure that it follows the guidelines of scientific control studies, that the test problems be divided into classes with similar characteristics, and that the goals of benchmarking experiments as well as the metrics to be measured be carefully considered. It is also recommended to report all factors capable of influencing the results of the experiment, like the runtime environment and the computer code. Possible programming paradigms for the implementation of the testing framework have also been presented: object-oriented and flow-oriented programming.

CHAPTER 2

RESEARCH PLANNING

This second Chapter of the thesis presents an overview of the proposed research using the Basili framework (Basili et al., 1986) (Bourque & Côté, 1991). This software engineering research planning framework is particularly relevant at this stage of the research because it enables a clear definition of the research subject as well as the activities that are necessary for its realization. It sets clear expectations for the activities and the objectives that will be included or excluded from the research as well as a list of activities.

The research question will be addressed by designing, implementing, and testing a Python framework for metaheuristics benchmarking. The work is divided into four phases that encapsulate the activities and evolution of the research project. Figure 2.1 shows a work breakdown structure for the research. It follows the structure of Basili's framework to provide a full view of the activities.

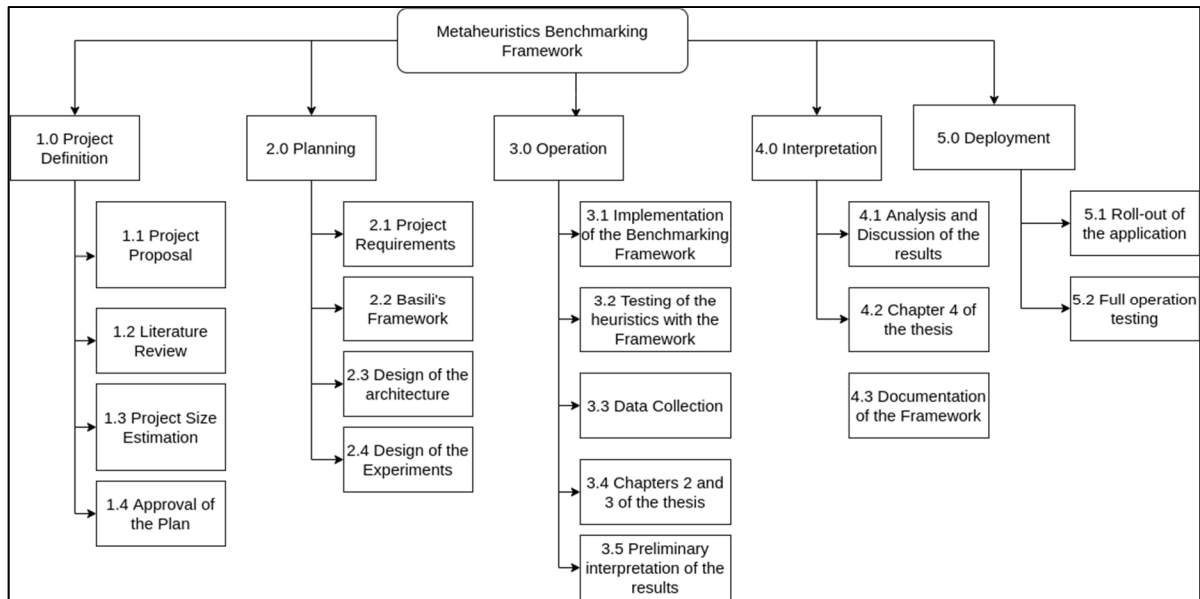


Figure 2.1 - Work Breakdown Structure of this Research Activity

2.1. Phase I — Definition

This first phase presents the definition and the audience for which the research is being conducted. The motivation, subject, objectives and users of the proposed framework are identified in the table below, as recommended in Basili's framework (Basili et al., 1986).

Table 2.1 Phase I – Definition of the Research

Motivation	Subject	Objectives	Users
<ul style="list-style-type: none"> Address the criticisms of the current benchmarking paradigm in metaheuristics research. with respect to benchmarking. 	<ul style="list-style-type: none"> The evaluation of the current methodological paradigm for metaheuristics benchmarking. 	<ul style="list-style-type: none"> Propose an analytical framework that helps researchers increase the quality of their work. 	<ul style="list-style-type: none"> Students and researchers involved or interested in mathematical optimization; Professionals involved in projects that include or would benefit from mathematical optimization.

2.2. Phase II – Planning

This second phase of the research will focus on a literature review of the field of metaheuristic benchmarking to describe the state of the art relating to analytical frameworks that validate the claims of a proposed metaheuristic. The deliverables are:

1. Compilation and classification of currently existing metaheuristic test functions as well as frameworks;
2. Compilation and organization of tools and procedures for quantitative analysis of metaheuristics (i.e., the analytical framework);
3. Comparison between practices in the literature and the proposed framework.

Table 2.2 Phase II - Planning Stage of the Research

Milestones	Inputs	Deliverables
<ul style="list-style-type: none"> • Literature review; • Analytical framework. 	<ul style="list-style-type: none"> • Research articles, books, etc. 	<ul style="list-style-type: none"> • Classification of existing benchmarking resources; • Introduction and Chapter 1 of the thesis;

2.3. Phase III – Operation

This third phase describes what will be accomplished by implementing the proposed framework, organizing the test functions within its structure, running the experiment following the analytical frameworks gathered in the previous phase, and performing a comparative analysis between the proposed framework and the observed practices from the literature.

The proposed framework will have elements of both object oriented and flow programming. A visual representation of its structure is shown in Figure 0.2. The metaheuristics definition step is outside the scope of this research. For testing a specific algorithm, the framework comes in the form of a Python software in which relevant problem instances are selected at the preprocessing stage. The algorithm is then run on the selected problem at the testing stage during which the results are collected and stored for analysis and post-processing.

Table 2.3 Phase III - Planning stage of the framework design and test

Preparation	Execution	Analysis
<ul style="list-style-type: none"> • Pilot study using standard algorithms; • Design of experiments: Classification, and selection of problem instances • Collection and classification of metaheuristics test functions; 	<ul style="list-style-type: none"> • Use of the framework to benchmark a representative set of proposed algorithms • Collection of experimental data issued from the application of the framework; 	<ul style="list-style-type: none"> • Analysis of the gathered experimental data with tools and procedures taken from the framework; • Chapters 2 and 3 of the thesis.

2.4. Phase IV — Interpretation

This last phase of the research plan describes the research steps where the results of the experimentation of the new framework proposal are assessed and interpreted. It allows for reviewing the initial research objectives, organizing and discussing the results, and identifying future research opportunities. Conclusions about the utility and usability of the proposed framework are also presented.

Table 2.4 Phase IV - Interpretation and results of the research

Interpretation context	Extrapolation of results	Future works
<ul style="list-style-type: none"> • Analysis of algorithms based on mathematical simplifications and statistical analysis of performance distributions on benchmark functions; • Analysis of originality based on patent law principles. 	<ul style="list-style-type: none"> • Generalization of results based on comparative studies; • Future metaheuristics can be compared with previous results using the framework to assess originality of contribution. 	<ul style="list-style-type: none"> • Application of framework on some of the metaheuristics that were not selected in this work; • Quantitative analysis of the trade-off between exploration and exploitation; • Automation of experimental design procedures for metaheuristics;

This section briefly returned on the initial goals of the proposed research. The overall planning proposed, based on Victor Basili's software engineering research method, helps in providing a well laid out project structure.

2.5. Conclusion

This chapter presented a work breakdown of this research following the framework of Basili. All the relevant activities have been outlined and organised in four phases: definition, Planning, Operation, and Interpretation. It was established that the objective of the project is to propose an analytical framework that helps researchers increase the quality of their work. Deliverables for the Planning stage like a literature review have been identified as well as the critical factors for the correct execution of the Operation and Interpretation stages.

CHAPTER 3

DESIGN OF THE FRAMEWORK

The proposed framework's requirements were specified using diagrams similar to the unified modelling language (UML). A set of logical diagrams are presented in the subsequent figures and have been conceived to serve as the requirement specifications for this project. Visual representations have been preferred to producing a written requirements document in the form of the system requirement specification format (SRS document). In addition, UML is widely used in the software engineering discipline and is conducive to better requirement engineering as there is less ambiguity between the parties and it allows for an easier requirements tracking processes (Borges & Mota, 2007). More specifically, the 4+1 views model introduced by (Kruchten, 1995) will be used. It is composed of four views dedicated to the functionalities of the software:

1. A use case view: This view outlines the workflows of the software as well as the resources involved
2. The process view defines the interactions between the various execution threads as well as tasks and how they are synchronized.
3. The logical view describes the processes of the algorithm as well as the data structures involved. Descriptions of the objects are also included
4. The realisation view organizes the components of the software in the development environment.
5. A view dedicated to the deployment and production environment of the software is also included in the model.

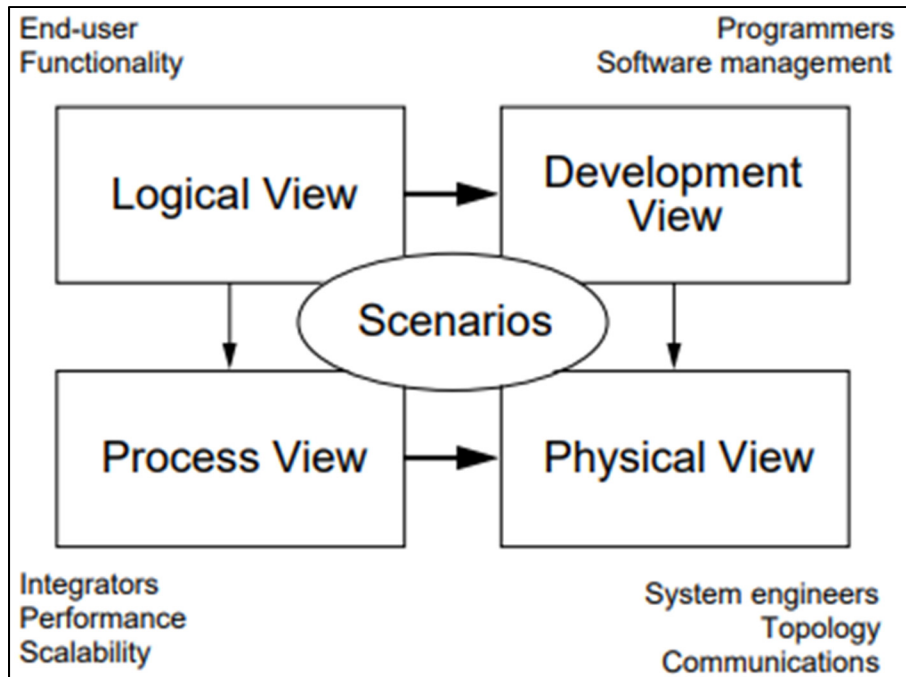


Figure 3.1 - A Diagram of the "4+1" View Model (Kruchten, 1995)

The following figures outline the design that was produced for this research on the framework for the analysis of the performance of proposed metaheuristics.

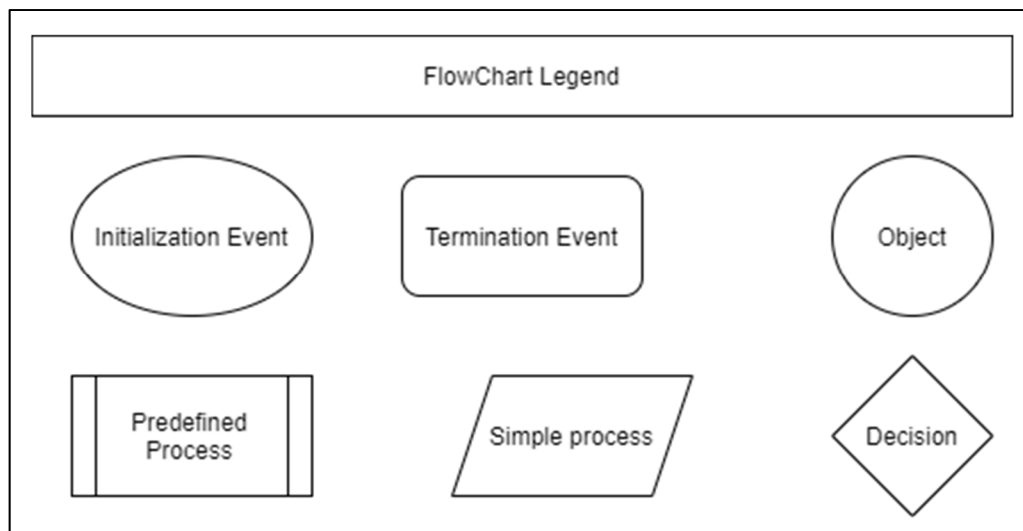


Figure 3.2 - Legend of the Components in the Flowchart

This diagram shows the convention used in the design of the various flowcharts. This convention was used instead of the formal UML shapes for reasons of keeping the toolchain used for the design process as streamlined as possible. In fact, the use of the formal UML shapes would have required the introduction of a new software as well as the modification of the development workflow. The solution that was finally chosen focuses on implementing the UML format despite the use of a custom convention for the elements.

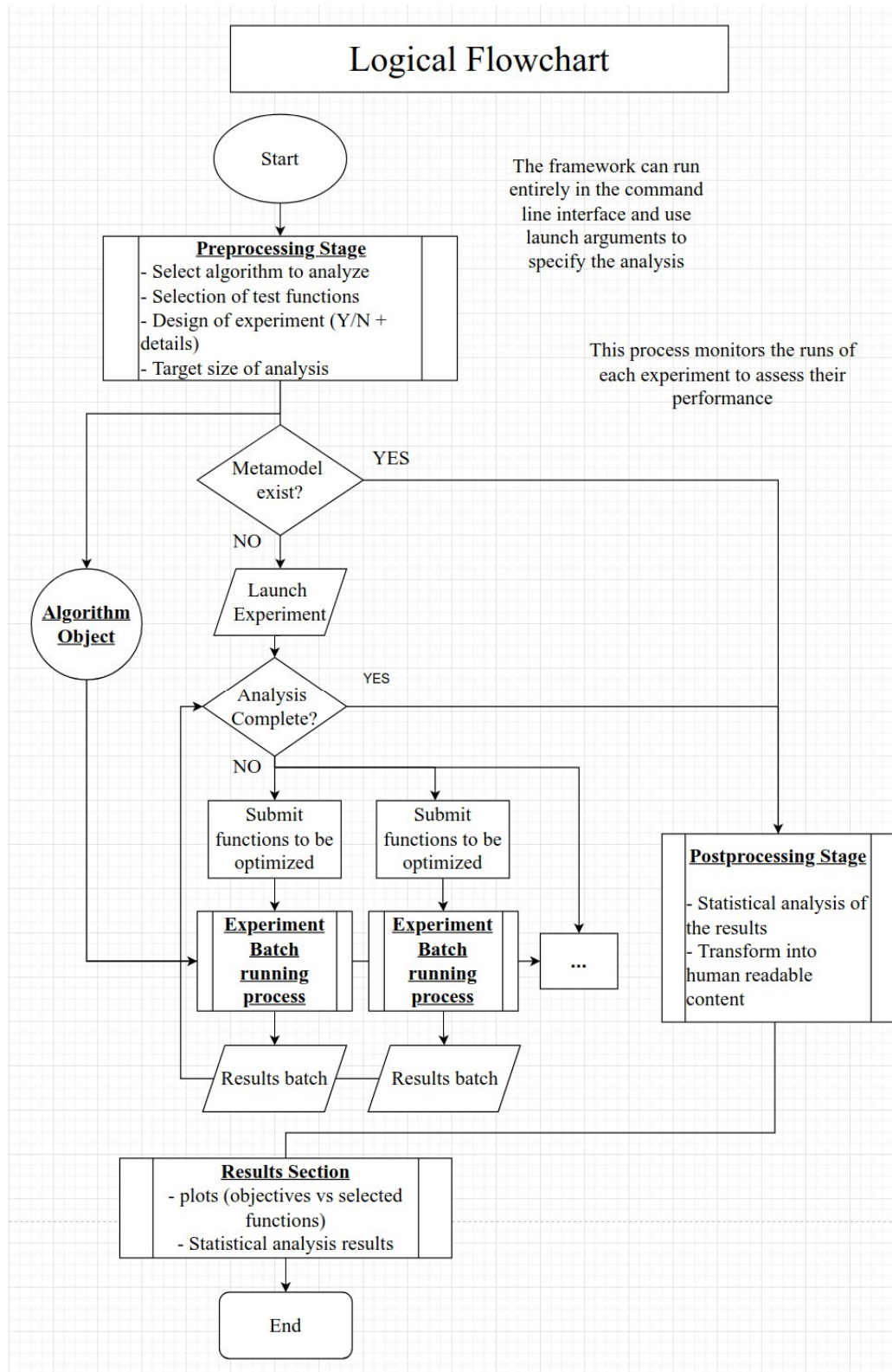


Figure 3.3 - Main Logical View of the Framework

The flowchart above displays the main logical view of the framework to specify its requirements. The most important aspects to notice are the need for an algorithm module and the preprocessing module. The importance of these two elements was realised while producing the diagram which reinforces the importance of producing a good architecture. These two modules are specified in more details in the following figures. Another point to note is the fact that the actual runs of the proposed algorithm on the various test problems will be executed simultaneously in batches using parallel computing. The framework is therefore using a hybrid between the object oriented and flow-based programming paradigms. Example objects are the algorithm module as well as the test problems. They indicate the object-oriented nature of the framework. The flow-oriented nature of the program can be observed by looking closely at the flow by which the experiments are run. All the parallel processes can be modelled as sets of operations that perform calculations on data structures containing the attributes of each experiment. These attributes are:

1. The metaheuristic object being tested;
2. The tests to apply for the specific run;
3. The results obtained after the run;
4. The performance data collected about the specific run.

To accomplish this, multiple processes and threads will be accessing the experiment data structure quasi simultaneously. Semaphores (which can essentially be understood as system level variables) can be used to synchronize these activities.

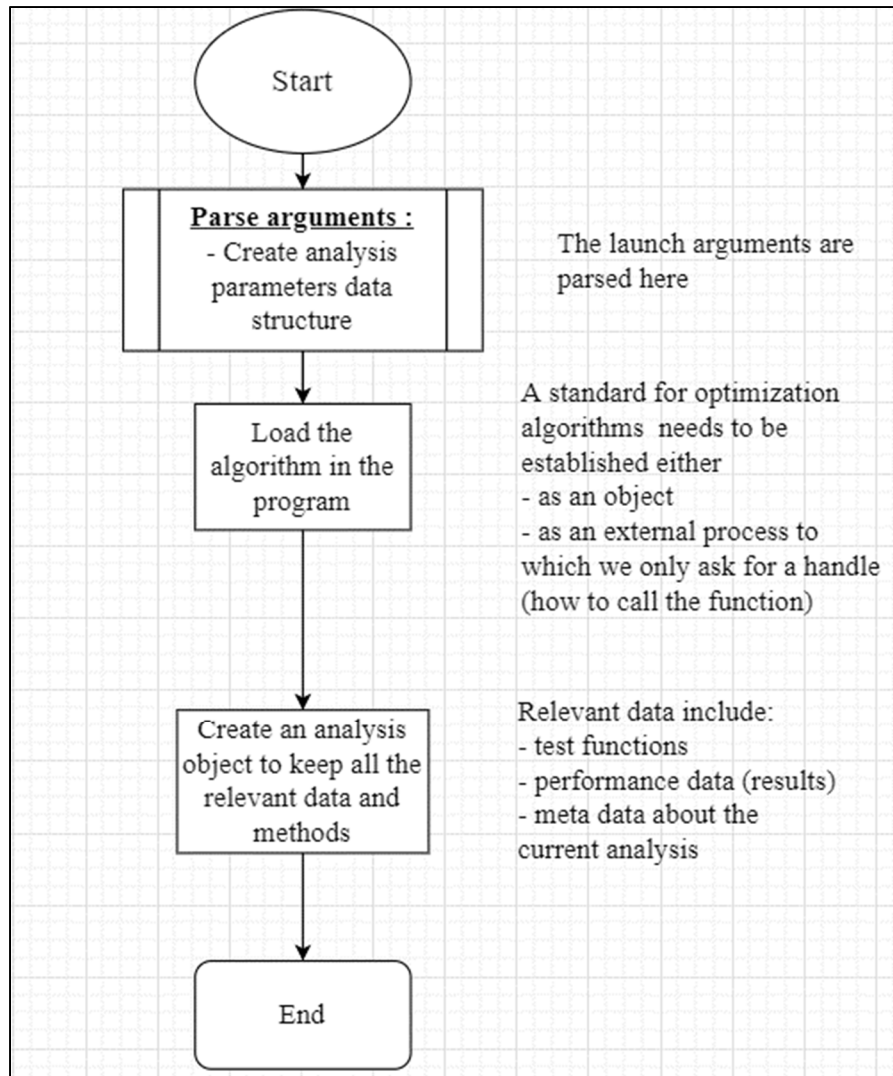


Figure 3.4 - The Preprocessing Module

The preprocessing module plays an important role in the framework as it sets up the environment required for the experiments to be run correctly and parses the inputs given by the user. For this reason, its development involves the user more than the other modules and the design decisions taken at this level will determine the design of the rest of the framework.

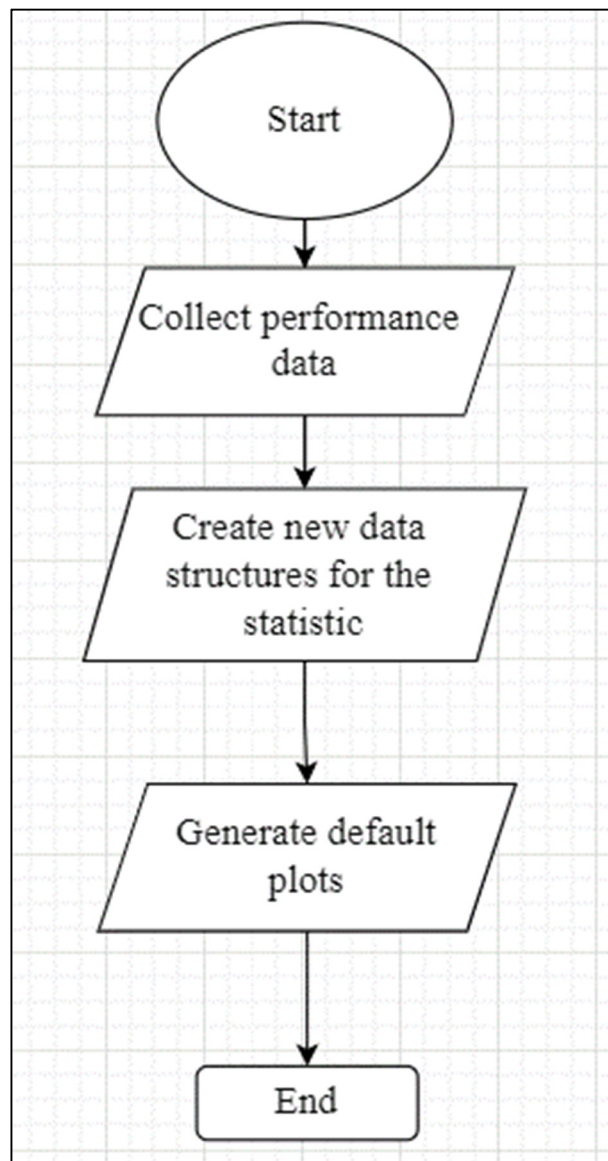


Figure 3.5 - The Postprocessing Module

The post processing module, like the preprocessing module is important because it communicates directly with the user.

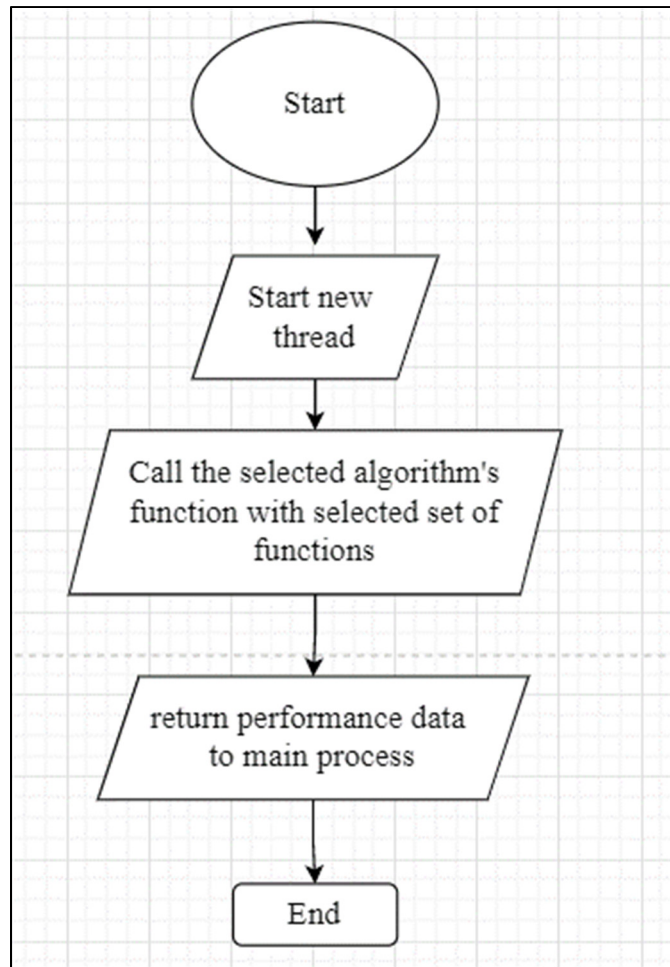


Figure 3.6 - The Experiment Running Process

The diagram above showcases the experiment batch running process for one experiment. As displayed in the main flowchart, all the runs will be performed in different threads to leverage the power of parallel computing and reduce the execution time of the whole process. The various threads will be synchronized, and the data will be stored in custom data structures containing all the required information as specified above.

The Algorithm module below (Figure 3.7) specifies the contents and format of the algorithm object that the user has to provide as argument to the framework when calling it.

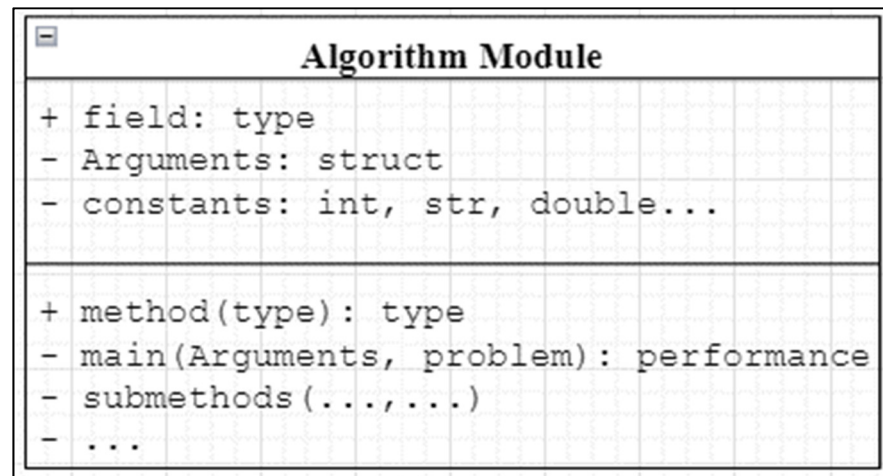


Figure 3.7 - The Algorithm Module

This architecture is a hybrid between the flow oriented and object-oriented paradigms. This is to leverage the advantages of both paradigms as outlined in section 1.1.

Beyond the architecture of the program itself, the lifecycle used by the development team is also an important consideration. In the case of this research, an agile lifecycle was preferred since the requirements were not all completely defined from the start and the ability to refine the architecture based on the results of the various experiments in the testing phase is important.

Therefore, the life cycle chosen was Kanban because it fulfills the agile requirement and provides the required flexibility for a research project. It also makes it possible to reassess the priorities of the various tasks and functionalities being treated to make sure that the stages of the project are fulfilled correctly.

The importance of a well-designed and documented architecture is emphasized in this project as it makes the software more accessible to future contributions and facilitates collaboration. It is also associated with reduced development cost in industry as well as improved quality and reliability. In fact, it is much easier to avoid breaking a program when its architecture is known compared to trying to modify an obscure software. an added benefit of well-designed and documented architectures is therefore the reduction of the

maintenance cost associated with it. For example, less time is invested fixing unattended consequences of code changes. The following paragraph expands on this by discussing the notion of technical debt.

In their paper entitled, “Technical Debt: from Metaphor to Theory and Practice”, Philippe et al. explain the impact that skipping the good practices of software development can have on the product (2012). The following figure outlines the technical debt landscape as defined in the paper and organized by how “visible” they are. The visibility characteristics relates to how readily detectable they are by usual identification tools like static code analysers.

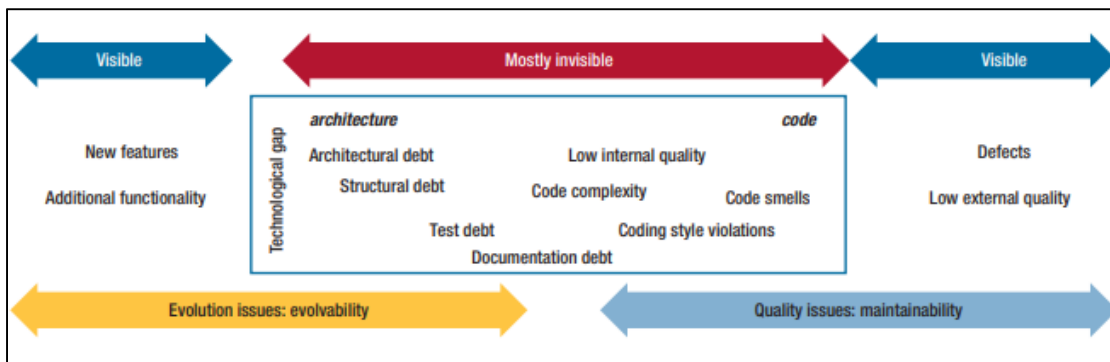


Figure 3.8 - The Technical Debt Landscape as proposed by Philippe (2012)

To address this issue of technical debt, the practice of refactoring is recommended. The article also recommends paying attention to the design phase of the development process and to use iterative design lifecycles as they provide the opportunity to fix non ideal processes and weakly implemented standards and protocols. The article also brings to light the fact that the use of iterative lifecycles does not automatically improve technical debt and that there is a need to specifically dedicate some activities for refactoring.

3.1. Requirements tracking

For tracking the requirements of the framework, multiple tools have been explored to select the right solution that would conform to the lifecycle of the project as well as its level of complexity. Following is a list of notable tools that have been considered:

1. Jira with Confluence. This tool enables the integration of the entire development environment from the version control resources like GitHub to the task management utilities and specification and scheduling documents. A communication functionality is also included. (Atlassian, n.d.)
2. Forecast PM as in project management is a management solution that integrates project management tools and integrates an artificial intelligence for process automation purposes. It is important to note that this is a commercial software(Forecast, n.d.).
3. Visure is a very comprehensive solution that offers many features for project management as well as requirements tracking (Hewitt, 2014). It uses a process driven approach by pushing the users to define the processes by which the requirements are supposed to be managed and enforces them in the workflow presented to access the requirements. It also comes with a library of standards that can be applied to the requirements management process. This has the benefit of simplifying the process control procedures of the users.

From the tools presented above, Jira is most interesting and accessible as it is particularly well suited for the lifecycle used in this research namely Kanban. It will therefore be used to integrate the tools that are already in use like Slack, Draw.io, and GitLab.

3.2. Conclusion

The 4+1 views model of Kruchten was used to model the system being proposed in this research. It was established that a hybrid between flow based, and object-oriented programming would be ideal for the proposed framework. The notion of technical debt was discussed to identify the potential risks this might pose to the project. Finally, options for requirements tracking were identified and the Kanban lifecycle was chosen for the development of the proposed framework.

CHAPTER 4

EXPLORATORY LANDSCAPE ANALYSIS

As this research aims to streamline the benchmarking process of metaheuristics, Alternative strategies of classifying the test functions are explored. A numerical method is selected instead of the qualitative approaches described in the literature review above because numerical methods lend themselves to better statistical analysis and are more repeatable. Exploratory landscape analysis (ELA) enables the classification of test functions using numerical methods and automation. This method has been developed to determine parameters which can reliably predict the performance of metaheuristic algorithms on other problems with similar parameters (Kerschke & Trautmann, 2016). In addition, the high level features discussed in the literature review such as the basin sizes and multimodality are debated and can be evaluated differently by different experts(Kerschke & Trautmann, 2016). Therefore, this method (ELA) will be used as the standard test function classification method in this research.

Other benefits of using numeric landscape analysis features are the possibility of combining the features with data from benchmarking the metaheuristics on test functions to build predictive and selection models for matching future optimization problems with the appropriate metaheuristics (Kerschke & Trautmann, 2016).

To automate the ELA process in this research, a software library named FLACCO (Feature-Based Landscape Analysis of Continuous and Constrained Optimization) will be integrated into the code of the framework. FLACCO is written in the R language while the Framework being developed in this research is written in Python. To bridge this compatibility gap, the Rpy2 framework is imported into the program to handle the collaboration between the two languages. FLACCO is capable of calculating at least 16 feature sets which are basically

vectors containing similar features. Example of feature sets which can be computed with FLACCO are `ela_meta`, `ela_distr`, `nbc`, `gcm`, `cm_angle` etc.... The FLACCO package also comes with ready-made visualization functions to better understand the characterization of the test functions. Example visualizations can be seen below.

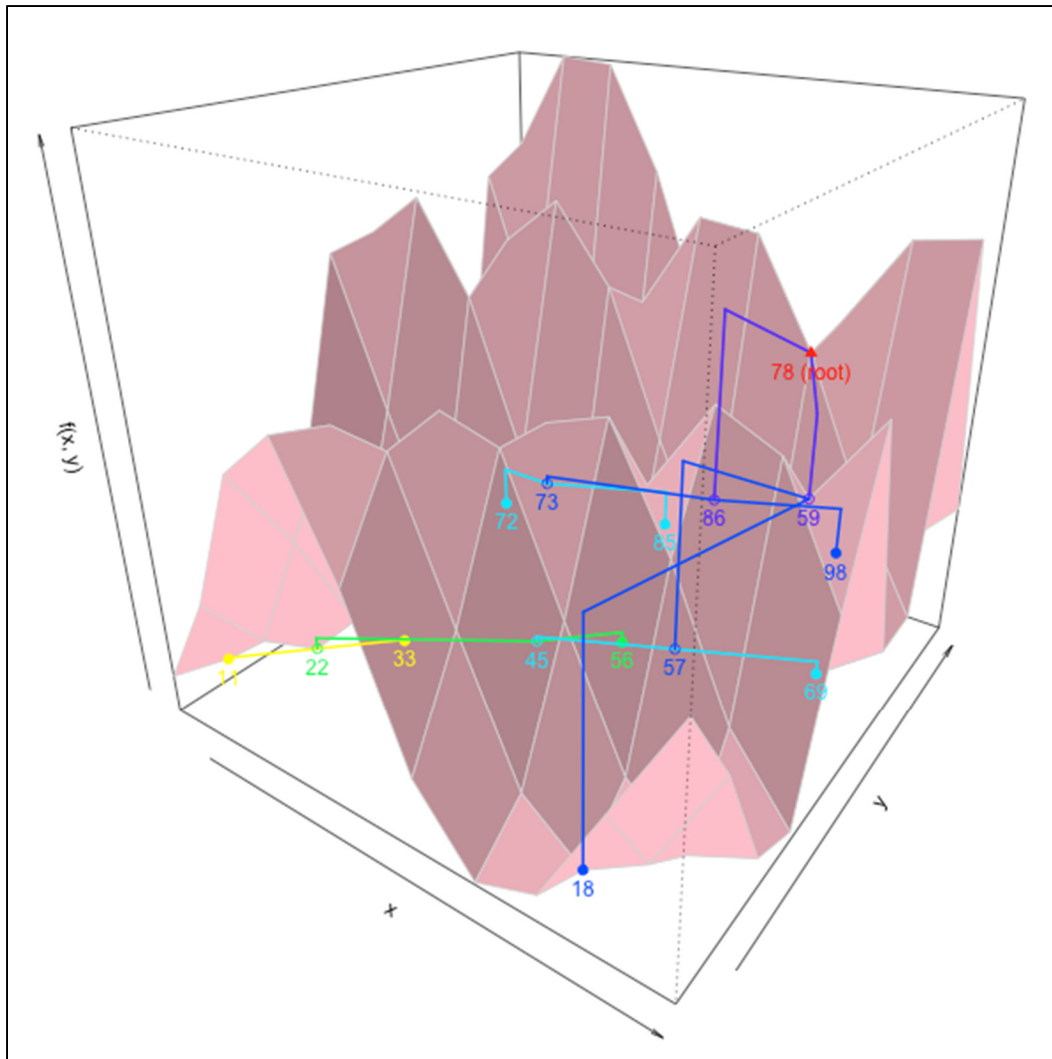


Figure 4.1 - 3D Barrier Tree Visualization by FLACCO (kerschke, 2015/2021)

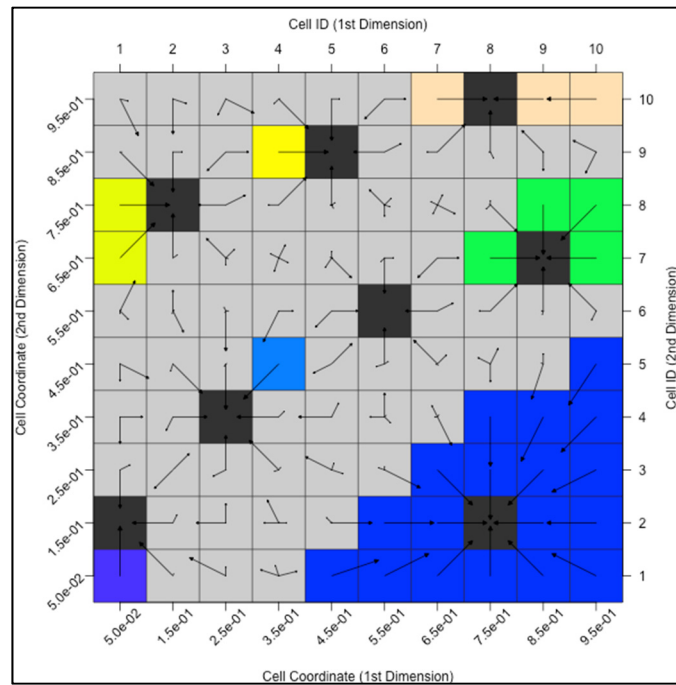


Figure 4.2 - Cell Mapping Visualization by FLACCO (kerschke, 2015/2021)

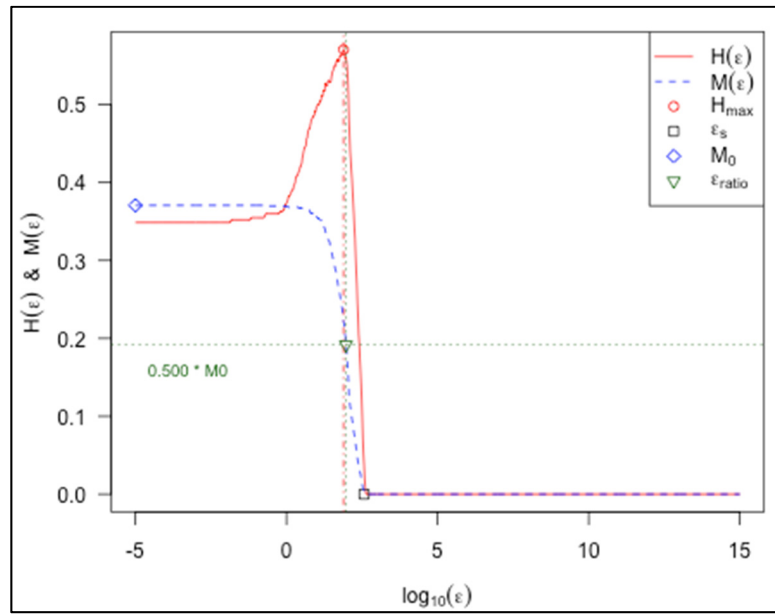


Figure 4.3 - Information Content Plot by FLACCO (kerschke, 2015/2021)

FLACCO also has the flexibility of configuring control parameters for each of its available feature sets (Kerschke & Trautmann, 2016). This makes it possible to adapt features to the specific landscape of the problems at hand.

4.1. Conclusion

The Integration of FLACCO into MDAF is discussed. FLACCO is a great addition to the proposed framework as it enables a quantitative assessments of problem instances and makes the application of NFLT a more objective process.

CHAPTER 5

IMPLEMENTATION OF THE METAHEURISTICS DESIGN AND ANALYSIS FRAMEWORK

This research developed a software Framework for the analysis of metaheuristics. To fulfill this objective, it includes features to automatically calculate performance metrics of optimization algorithms and visualization functions which enable the operator to get a deeper understanding of the results and the test function characteristics.

The program implements recommendations outlined in the first chapter of this paper like the use of valid statistical methodology recommended by (Brownlee, 2007). For example, the performance calculations are repeated 30 times by default with different initial conditions for each test function and the average as well as standard error of the performance metric are kept as the real performance characteristics. This ensures that the stochastic nature of some algorithms is captured. The performance metrics calculated by the framework are:

- The CPU time of the calculations: this determines exactly how much time the CPU of the computer spent specifically on tasks related to the algorithm being tested excluding the time spent on any other applications running on the computer. This is to ensure the repeatability of the performance results obtained independently from the load the computer is operating under.
- Number of calls to the objective function: this metric is calculated since some algorithms have more complex logic but require few real calculations of the objective function (e.g., surrogate assisted algorithms) while other algorithms have simpler logic but require frequent calls to the objective function. This metric therefore captures the impact of how difficult the objective function is to evaluate.

- The Quality of achieved results: this metric measures how good the obtained results are as some algorithms can find a solution quickly but would oscillate around the optimum while other algorithms take more steps to find a solution of greater quality.
- The convergence rate of analyses: This metric is used because some algorithms diverge and are not able to find the solution depending on multiple factors like the step size or the initial conditions. Therefore, the percentage of trials which found a solution is included as a metric.

The framework also comes with functions capable of automatically calculating a numerical representation of all the test functions. This is accomplished by integrating the FLACCO framework to MDAF. As explained in the previous chapter, FLACCO is written in the R programming language and the integration with MDAF (written in Python) is accomplished with RPy2 which is a compatibility framework between the two languages. Using FLACCO, the ELA feature sets presented in Figure 5.1 can be calculated. A subset can be chosen among them depending on the analysis to be conducted like in the following case study.

Feature class	Name	Num. features
ela_distr [25]	<i>y</i> -distribution	5
ela_level [25]	levelset	20
ela_meta [25]	meta-model	11
nbc [17]	nearest better clustering (NBC)	7
disp [17]	dispersion	18
ic [33]	information content	7
basic [20]	basic	15
limo [20]	linear model	14
pca [20]	principal component analysis	10
cm_angle [16]	cell mapping angle	10
cm_conv [16]	cell mapping convexity	6
cm_grad [16]	cell mapping gradient homog.	6
gcm [16]	generalized cell mapping	75
bt [20]	barrier tree	90

Figure 5.1 FLACCO Feature Sets Provided By FLACCO (Tanabe, 2021)

Plotting functions are also included in the framework which enable the users to visualize the test functions as shown in Figure 5.2 below.

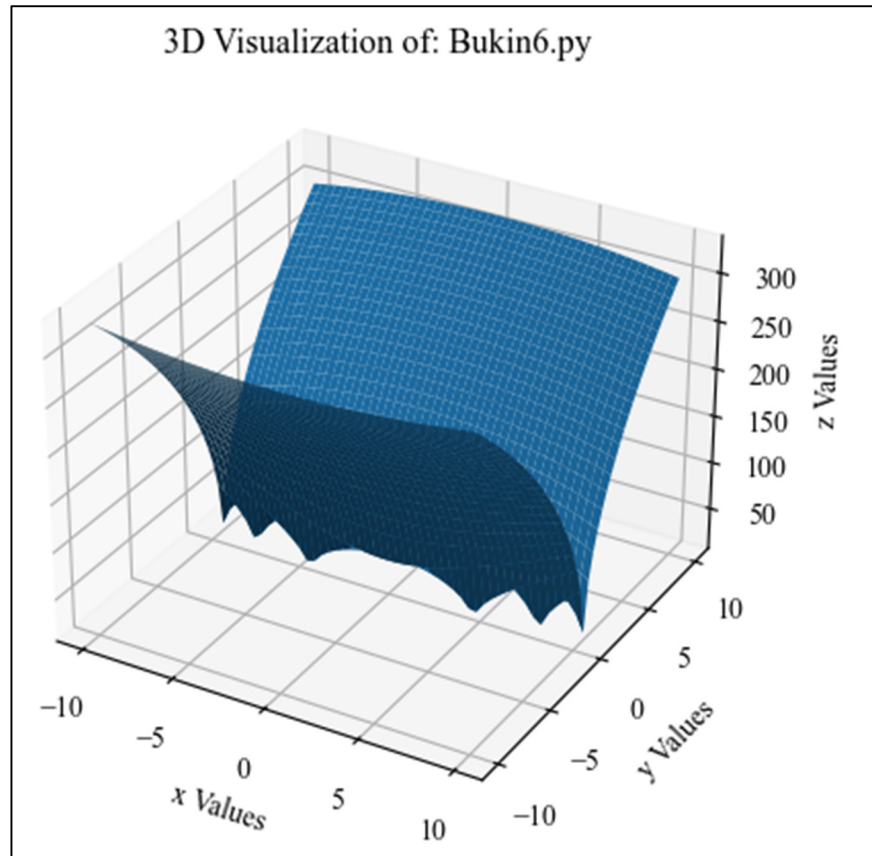


Figure 5.2 - Test Function Visualization produced by MDAF

Radar plots are also available to visualize the ELA representation of various test functions and compare them to other functions. The radar plots represent each feature set which are vectors with the elements plotted at the respective angles. Multiple functions can be automatically plotted together. This feature can also be used to verify that a benchmarking analysis contains a good representation or sample of all possible feature set values. Figure 5.3 shows an example of ELA representation radar plots.

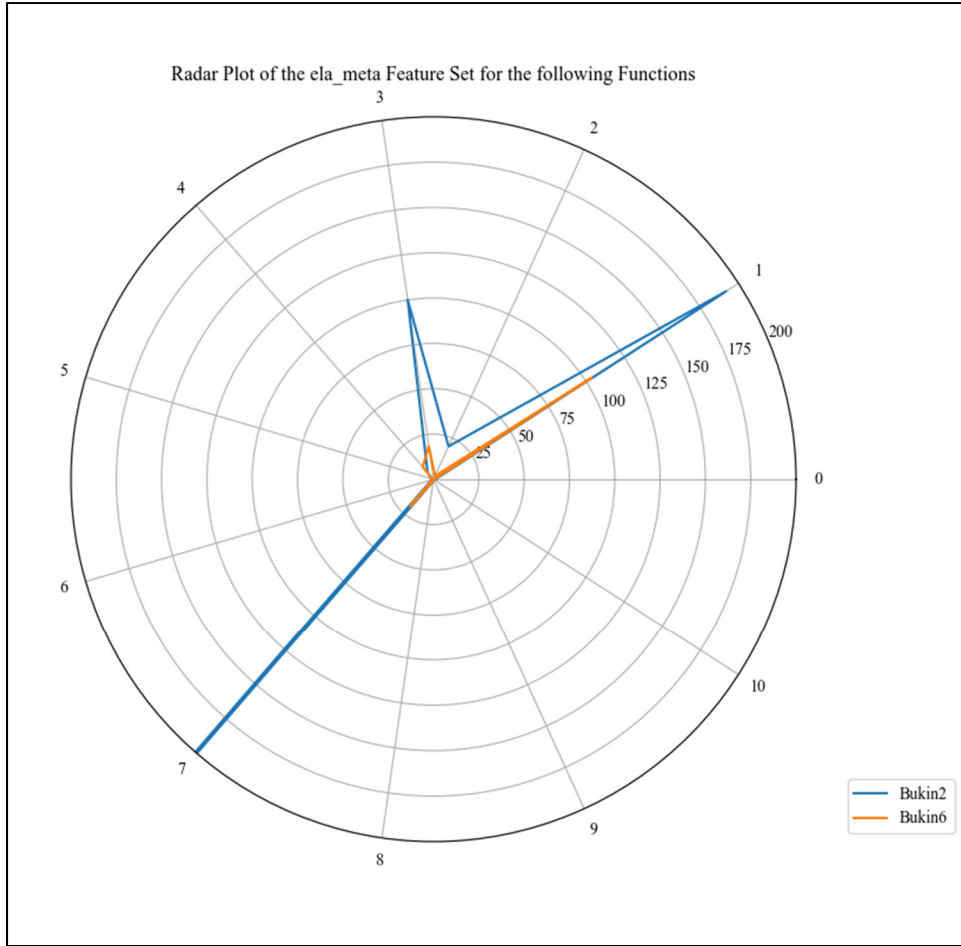


Figure 5.3 - Radar Plot of Bukin2 and Bukin6 test functions' ela_meta feature set by MDAF

5.1. Default Test Functions

MDAF comes with 27 built-in test functions which can be used to benchmark new metaheuristics. When launching an analysis, these preprogrammed test functions can be referenced using the '@' symbol in front of the function name. Preprogrammed test functions include Bukin6, and Brown as visualized in the following figures. The functions were taken from (Hussain et al., 2017) and (Jamil & Yang, 2013).

The user also has the flexibility of adding external test functions to its analyses. To this end, the new test functions need to be organised following APPENDIX IV. It must accept a list as

input with each element of the list representing a factor of the optimization problem. For the function to be compatible with the plotting routines inside MDAF, the function must be 3 dimensional. External libraries such as matplotlib and SciPy can be used for higher dimensional data.

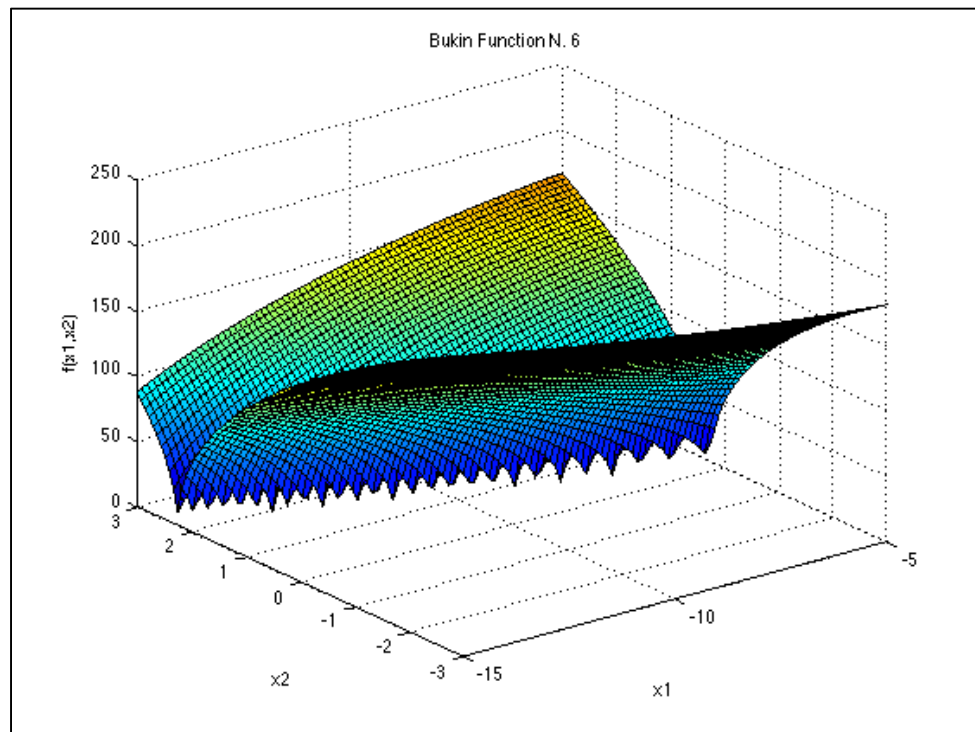


Figure 5.4 - Bukin6 Visualization(*Bukin Function N. 6*, n.d., p. 6)

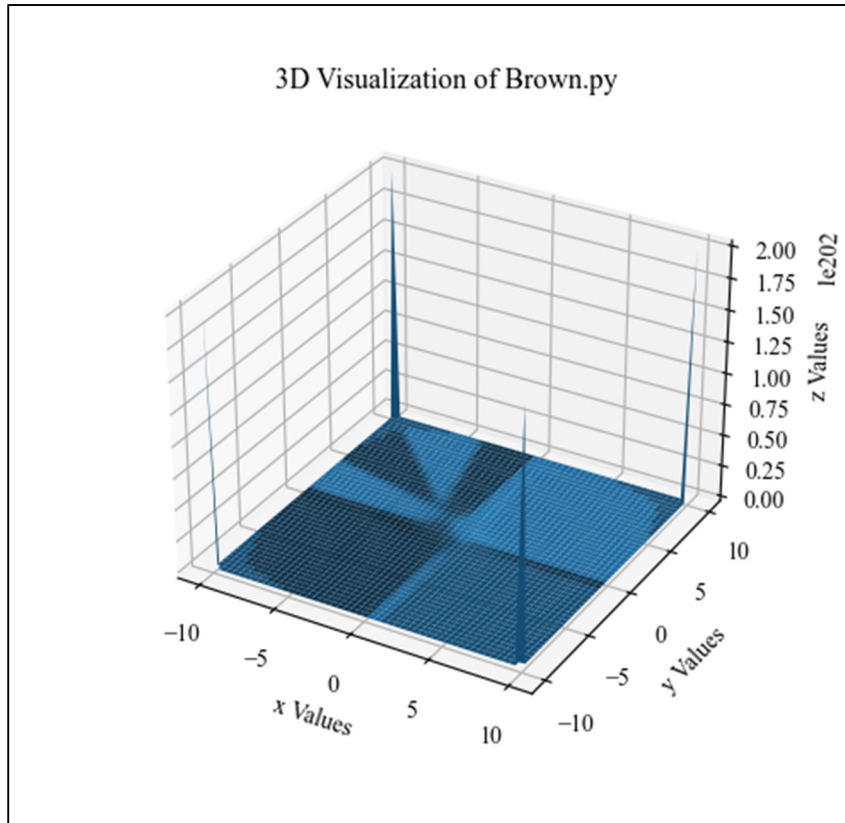


Figure 5.5 - Brown Function Visualization

5.2. PyPI Packaging

The code of the framework is packaged for the PyPI repository. It can be downloaded from Git and installed using the python preferred installer program (pip). This makes it possible to the operator to use the framework in the python interactive shell. The package metadata as well as Python init files can be found in APPENDIX II and APPENDIX X.

5.3. Experimental Planning

The framework has the ability to run experiment plans (EP) by calling the `exp` function. The code of this function is available in APPENDIX III of this report. A list of the test functions on which the experiments need to be run is provided by the operator as an argument. Preprogrammed functions need to be specified as explained in the previous sections. The operator is expected to use the plotting methods provided by the framework in the analysis planning stage. The `exp` method runs the optimization algorithm 30 times by default or a user-

defined number of times on each test function provided. It then calculates the average and standard deviation for all the benchmarking metrics. This makes it possible to capture the stochastic nature of certain algorithms. In fact, as stochastic algorithms can have varying performance on the same problem due to the impact of the random parameters like the temperature in simulated annealing, the repeated benchmarking runs makes this phenomenon detectable for analysis.

5.4. Automated Testing

Automated tests were designed and implemented to guide the development process of the program. They are included in this report as they can be used to quickly validate changes to the code. This simplifies the continuous improvement of the framework by automating repetitive tests that need to be performed to validate the changes. APPENDIX V presents the automated testing algorithms used for the design of MDAF. The *unittest* library was used for this feature as well as *doctests*. The *unittest* library was applied to testing the framework's methods while *doctests* were used for the integrated test functions. Figure 5.6 below shows a view of the testing environment. On the left sidebar, the list of automated tests is displayed with their status. In this particular example, all the tests were successful.

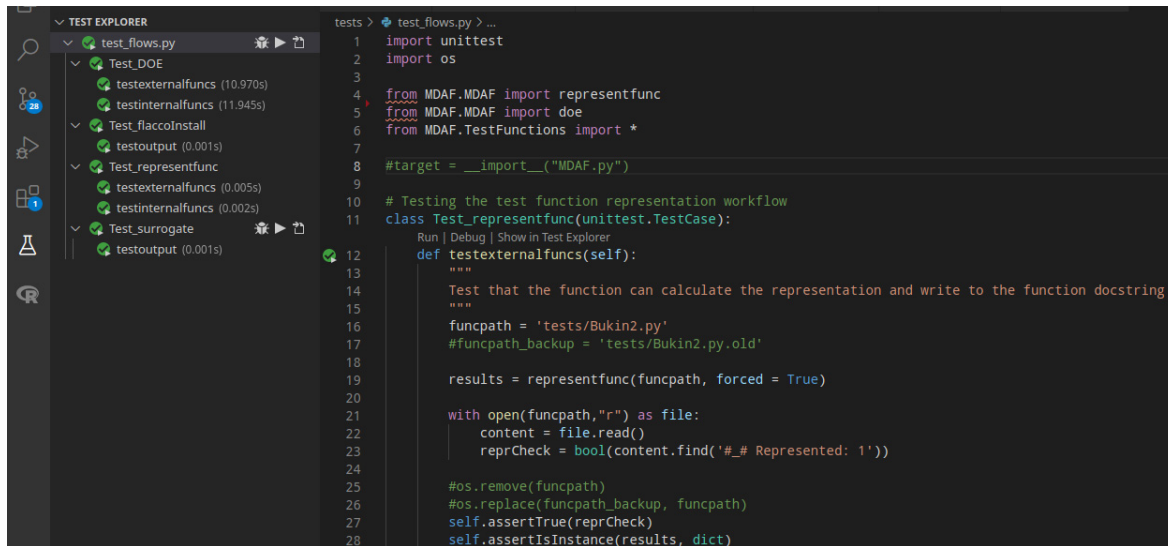


Figure 5.6 - Snapshot of the Automated Tests Results in VS Code

Doctests were also used in the implementation of the preprogrammed test functions. The *Doctests* were executed using the following command. The results of automated testing can be observed in the figure below.

Algorithm 5.1. PyTest Command

```
python -m pytest --doctest-modules --doctest-continue-on-failure
```

The options of the command have the following meaning:

- “doctest-modules”: automatically runs the doctests in all python modules in the current directory
- “doctest-continue-on-failure”: prevents the process from stopping when any one test fails.

In Figure 5.7 below, the results of running the doctests for the integrated test functions of MDAF is displayed. We can see at the bottom of the figure that all 28 tests passed successfully. This can also be observed by the small green dot beside each test function name listed in the figure. The percentages listed at the right of the figure show the progress of the overall testing process. Therefore, we know that the test has completed when the last line with the *Zirilli*

test function is displayed. The *100%* value on the right shows that this was the last function being tested.

```
[remi@Centurion TestFunctions]$ pytest --doctest-modules --doctest-continue-on-failure
===== test session starts =====
platform linux -- Python 3.9.6, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /home/remi/Documents/MDAF-GitLAB
plugins: anyio-3.3.0
collected 28 items

Ackley2.py . [ 3%]
Alpine.py . [ 7%]
Brown.py . [ 10%]
Bukin2.py . [ 14%]
Bukin4.py . [ 17%]
Bukin6.py . [ 21%]
Easom.py . [ 25%]
Keane.py . [ 28%]
Leon.py . [ 32%]
Matyas.py . [ 35%]
McCormick.py . [ 39%]
Miele_Cantrell.py . [ 42%]
Periodic.py . [ 46%]
PowellSingular2.py . [ 50%]
Price1.py . [ 53%]
Price2.py . [ 57%]
Quartic.py . [ 60%]
Rastriring.py . [ 64%]
Scahffer.py . [ 67%]
Schwefel.py . [ 71%]
Sphere.py . [ 75%]
Step.py . [ 78%]
Step2.py . [ 82%]
Styblinski-Tang.py . [ 85%]
SumSquare.py . [ 89%]
Wayburn.py . [ 92%]
Zettle.py . [ 96%]
Zirilli.py . [ 100%]

===== warnings summary =====
../../../../usr/lib/python3.9/site-packages/packaging/version.py:127: 2578 warnings
  /usr/lib/python3.9/site-packages/packaging/version.py:127: DeprecationWarning: Creating a LegacyVersion has been
  deprecated and will be removed in the next major release
    warnings.warn(

-- Docs: https://docs.pytest.org/en/stable/warnings.html
===== 28 passed, 2578 warnings in 1.42s =====
[remi@Centurion TestFunctions]$
```

Figure 5.7 - Automated Python *Doctest* Results

5.5. Efficient computation

As the computations involved in benchmarking an optimization algorithm are heavy and can be time consuming, parallel computing has been extensively employed in the implementation of the framework to leverage the power of modern CPUs more effectively. The framework will automatically determine the number of cores available on the machine and evenly spread the load and maximize throughput. This is achieved using the python multiprocessing library. For this goal, the relevant functions of the framework are designed to be picklable. Meaning that they can be serialized by the python pickle module(*Pickle — Python Object Serialization —*

Python 3.10.1 Documentation, n.d.). Figure 5.8 showcases the task manager of the computer running MDAF while benchmarking a PSO algorithm. Numbered 1 to 16 are the CPU cores all showing close to 100% utilisation (annotated 1). This attribute of MDAF is dynamic and automatically adapts to the machine on which it is running by detecting the number of cores available and adjusting its multithreading and multiprocessing logic accordingly. Annotated 2 are the processes that have been spawned by MDAF. They all run independently from each other and the results are collected by one of the processes playing a managerial role.

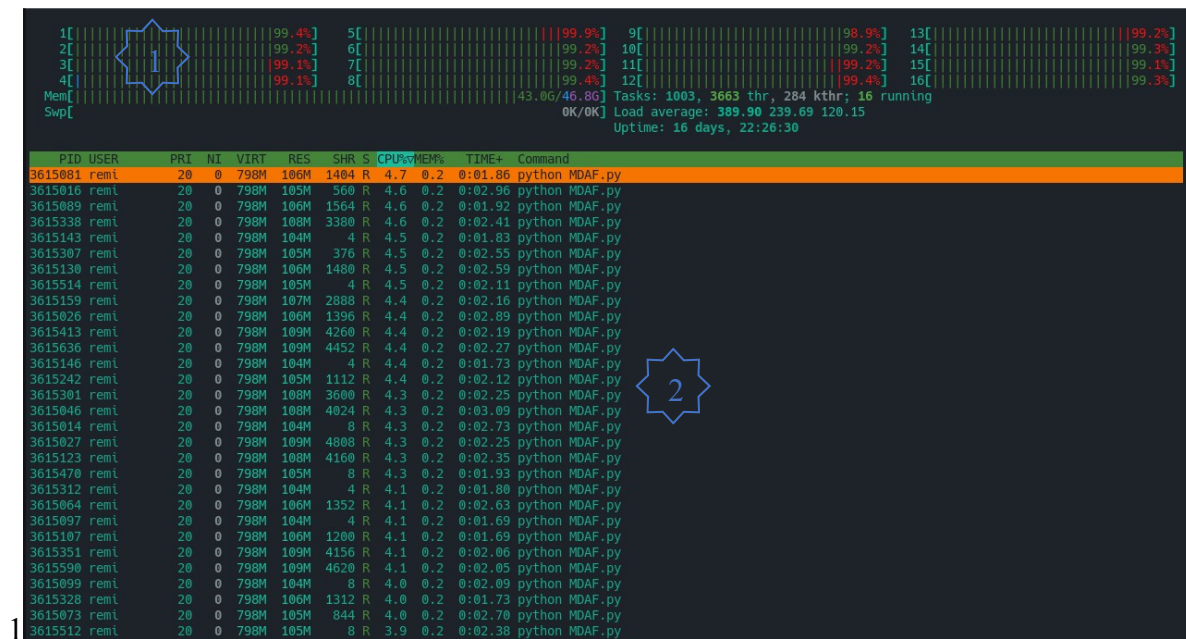


Figure 5.8 - Optimally Loaded Central Processing Unit (CPU)

5.6. Conclusion

The implementation of the proposed framework is discussed as well as its features and usage recommendations. The framework can benchmark algorithms with a variety of metrics such as the number of objective function evaluations, the CPU time of computation, the quality of achieved results, and the convergence rate of analyses. A default set of test functions are provided with the proposed framework containing 27 functions such as The Bukin and Ackley test functions. Features such as the multiprocessing abilities of the framework are also discussed.

CHAPTER 6

CASE STUDY: BENCHMARKING THE PARTICLE SWARM ALGORITHM WITH DIFFERENT PARAMETERS TO DETERMINE THE EFFICIENT VALUES

The specific algorithm being investigated in this case study can be found in APPENDIX IX. The algorithm will be benchmarked using the MDAF framework which implements the recommendations outlined in the literature review. For example, special attention was used in selecting the measures of performance as recommended by (Brownlee, 2007). This measure will be the number of calls to the test function as this measure is independent of the machine on which the benchmark is being computed. A maximum number of iteration parameter is set to 10'000 because the analysis is looking for effective parameters. MDAF allows the user to set this value to any number. Parameters that require the algorithm to perform more than 10'000 iterations before achieving a suitable solution are considered ineffective. This has the benefit of keeping the runtime of the analysis manageable as a study can run for days if no limits are applied.

6.1. Methodology

The following steps can be implemented to reproduce this study.

1. Select test functions from the preprogrammed library or generate python files for new test functions that are to be included in the Analysis;
2. The “visualize2D” function in MDAF can be used at this stage to quickly visualize the shape of a test function;
3. Create a list object in python containing the paths to all the selected test functions; Preprogrammed library test functions should be specified with the symbol “@” before the function name (e.g., @bukin.py);
4. Make sure to have called the install FLACCO function at least once since installing MDAF;

5. Calculate the features of all the selected test functions by calling the “representfunc” function of MDAF;
6. The plot function method of MDAF, can be used to generate a radar plot of the selected benchmarking test functions’ FLACCO feature sets;
7. The steps 1 to 5 can be repeated until a good set of test functions is identified. This list will be referred to as the experiment list;
8. Call the *exp* function from MDAF with the experiment list as argument as well as any arguments which are supposed to be passed to the metaheuristic algorithm;
9. MDAF will then proceed to calculate the performance of the algorithm on the test functions and return a dictionary containing all the performance data;
10. This case study utilises MDAF for parameter tuning and the additional script for this is outlined in APPENDIX VIII.

6.2. Results

The below figures display the output of running the Visualize2D method of MDAF. The method is used to investigate test functions while putting together the experiment list. It can also be useful to assess newly proposed test functions.

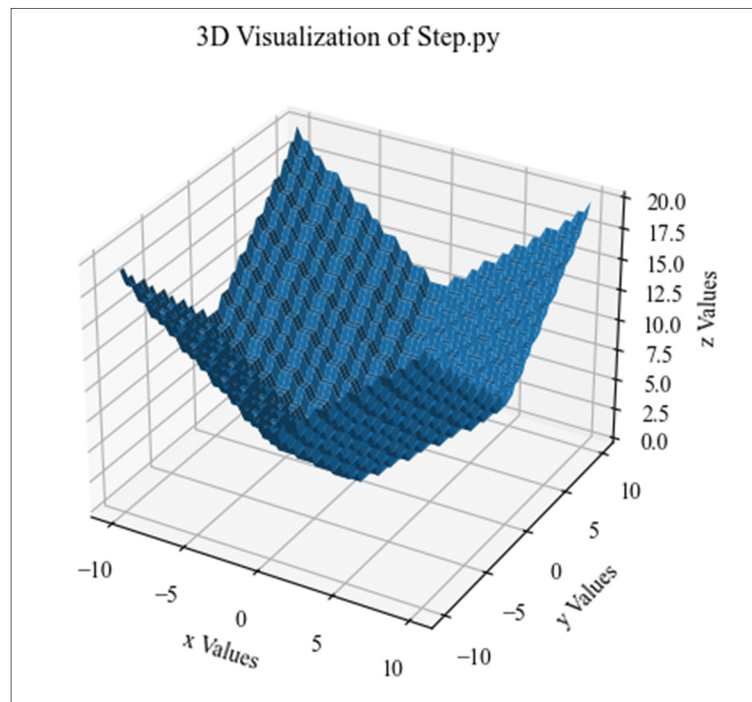


Figure 6.1 - Result of the Visualize2D function for the Step Test Function

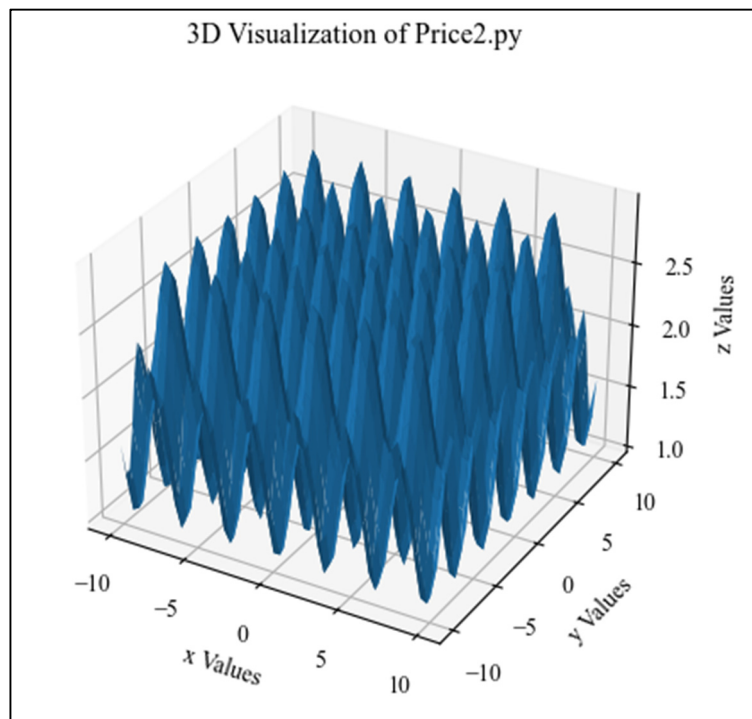


Figure 6.2 - Result of the Visualize2D function for the Price2 Test Function

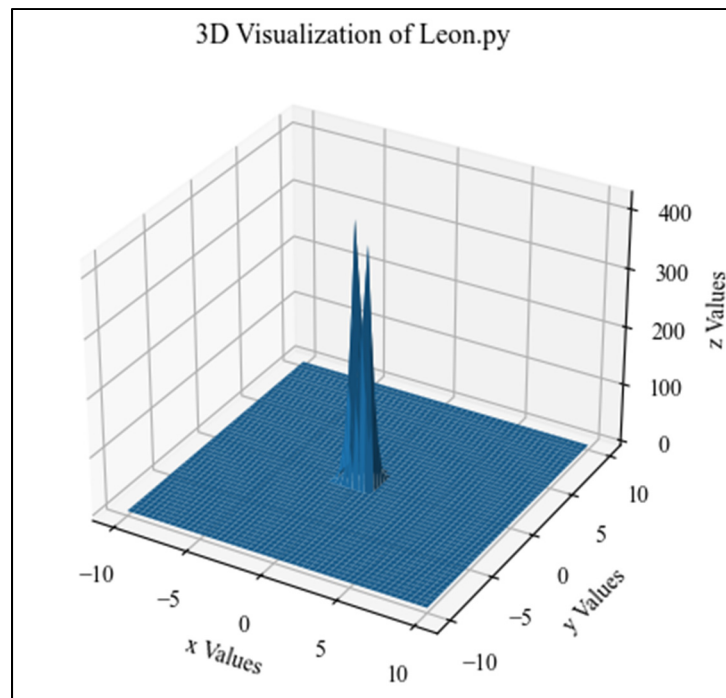


Figure 6.3 - Result of the Visualize2D function for the Leon Test Function

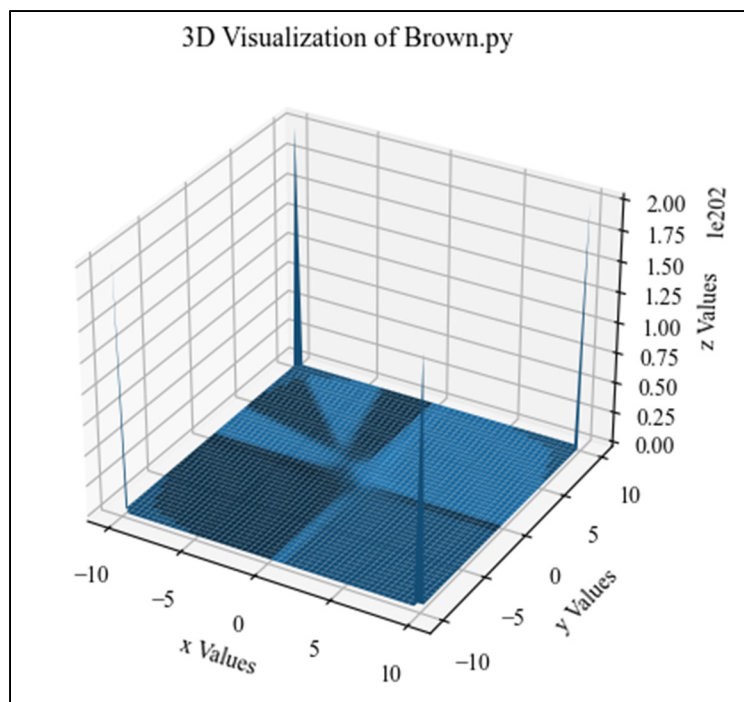


Figure 6.4 - Result of the Visualize2D function for the Brown Test Function

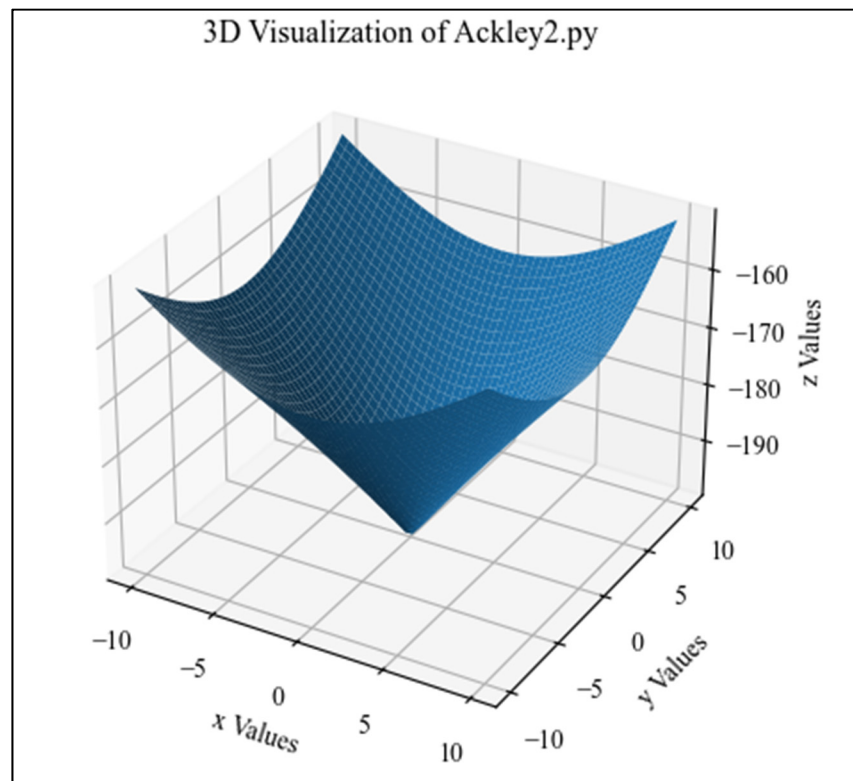


Figure 6.5 - Result of the Visualize2D function for the Ackley2 Test Function

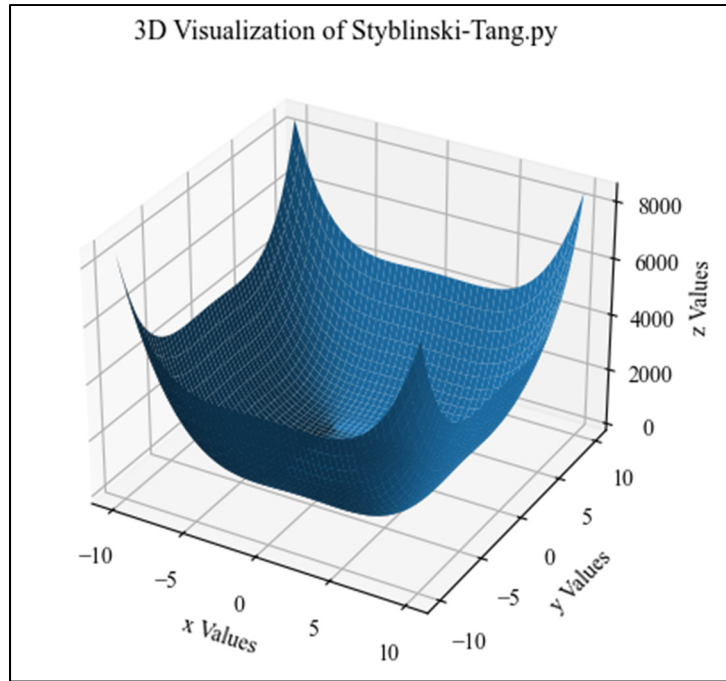


Figure 6.6 - Result of the Visualize2D function for the Styblinski-Tang Test Function

The following figure displays the outputs of the *plotfuncs* method of the framework to investigate the FLACCO feature set values of the test functions. This plot represents the gcm feature set being considered in this analysis. Please note that the algorithm of the “plotfuncs” method is available in the appendix. Each line on the plots represents a test function and for each test function, the feature values are marked on their respective axes.

These plots can also be used to assess the distribution of the test functions across the feature space. The actual values for all the features for all the functions will be presented in table format in APPENDIX VI.

As explained by Kerschke and Trautmann, it is necessary to combine the feature sets to make sure that the test functions are fully characterized as each feature set only capture a small subset of relevant characteristics for metaheuristic algorithms(Kerschke & Trautmann, 2016).

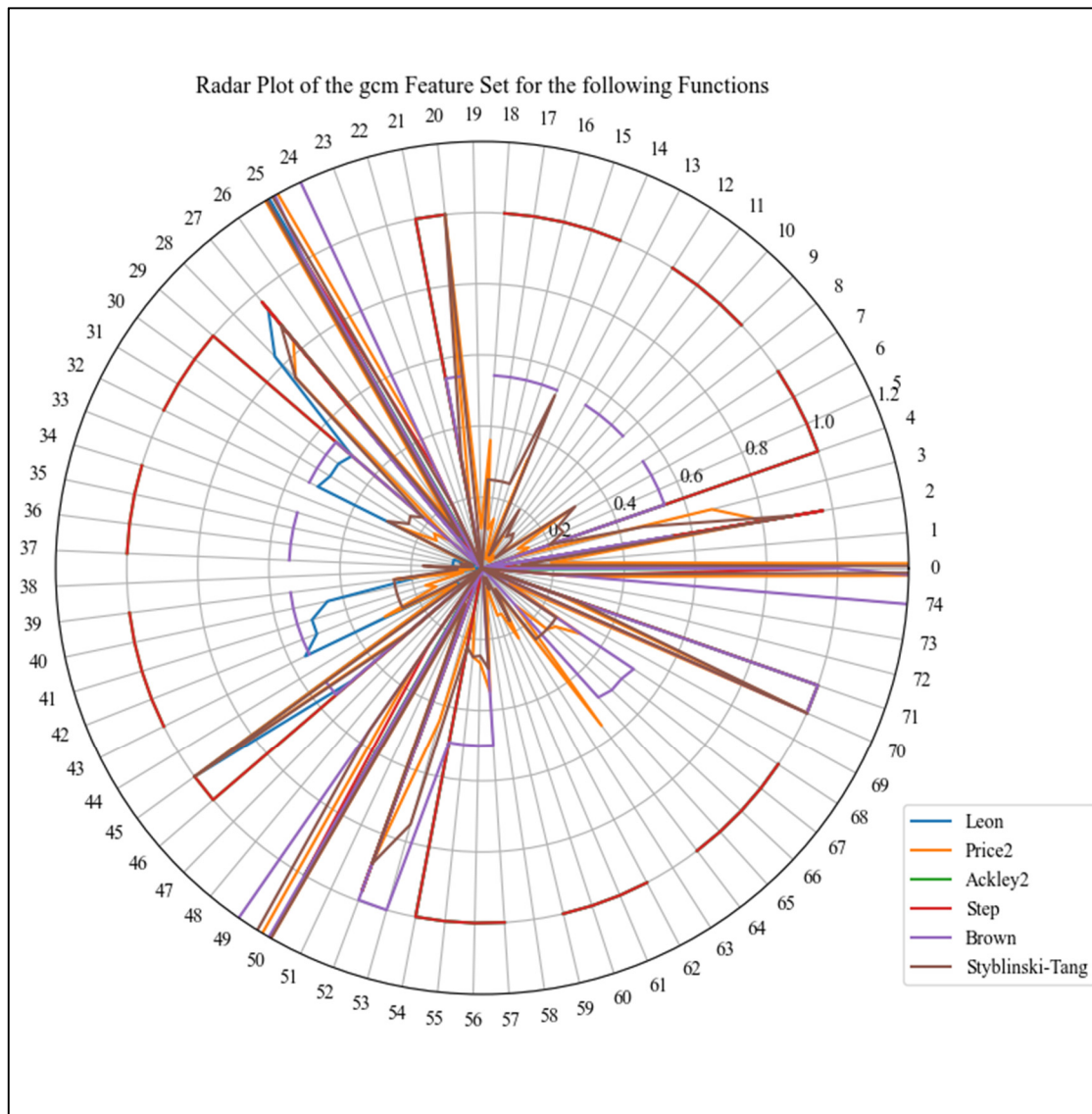


Figure 6.7 - Result of the plotfuncs Function for gcm features

From the plot above, it is apparent that the selected set of test functions fills up the space of possible values for the gcm feature set. The gcm feature set is quite extensive and could be used on its own for a quantitative model of metaheuristic performance as it contains a large number of features which are well distributed. The scale of some of its features (24 and 25), however, need to be normalized as they might overly influence the model. It is also important to note that some features are calculated by FLACCO to a value of “NaN” meaning not a number. This special value is used to represent the result of undefined operations. All features

with such values need to be addressed before building a quantitative model. This can be achieved by dropping all such features which is a standard technique. Other methods of dealing with *NaN* values exist which are not addressed in this paper as they are not part of the study being conducted.

The following data represents the results of benchmarking the PSO Algorithm using MDAF for the following test functions: Leon, Price2, Brown, Ackley2, Styblinski-Tang, and Step. The impact of the *c1* and *c2* parameters of the PSO algorithm on its effectiveness at optimizing test functions is studied. The optimization effectiveness is measured by the number of calls from the algorithms to the test function. This data is plotted on heatmaps with the *c1* values on the x-axis and *c2* on the y-axis with the colors from violet to red showing the number of calls to the test function.

Each test function studied in this case is evaluated 16 times for each point on the map and the average statistic is used for mapping. The number of trials was set to 16 for practical purposes since increasing this value resulted in exponentially longer running times. The standard error of the average statistics is also calculated for each point and displayed on another heatmap. The standard error statistic is calculated using the following formula:

$$Standard\ error_{each\ point} = \frac{standard\ deviation_{each\ point}}{\sqrt{sample\ size}} \quad (3)$$

With *sample size* = 16, and standard deviation calculated automatically using the python statistics library. Therefore, each test function produces two heatmaps and the same *c1* and *c2* values are used across all test functions. The specific *c1* and *c2* values used in the experiment as well as all the performance data used for the following plots can be consulted in APPENDIX VII.

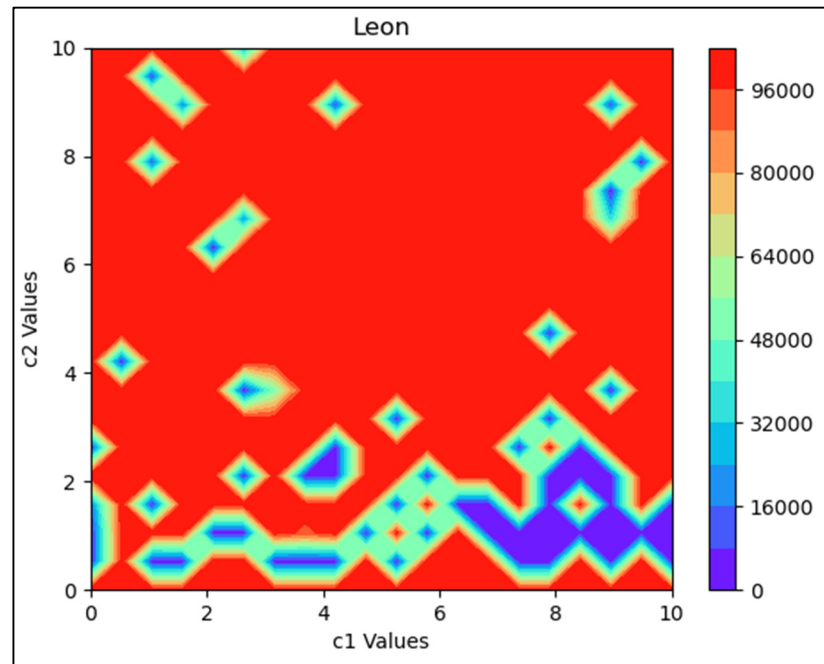


Figure 6.8 - Average Number of Function Calls for Solving the Leon Test Function vs c_1 and c_2 Parameter Values

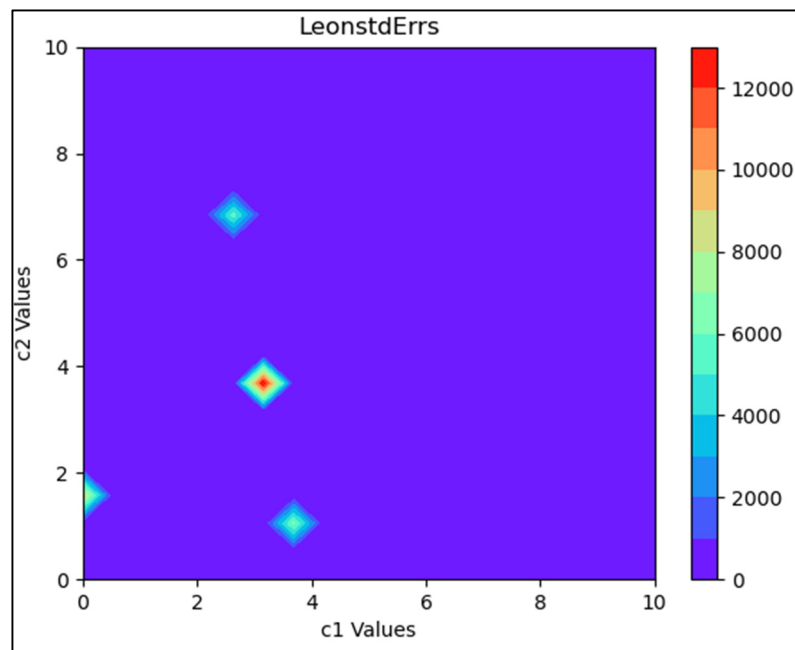


Figure 6.9 - Standard Errors for the Number of Function Calls for Solving the Leon Test Function vs c_1 and c_2 Parameter Values

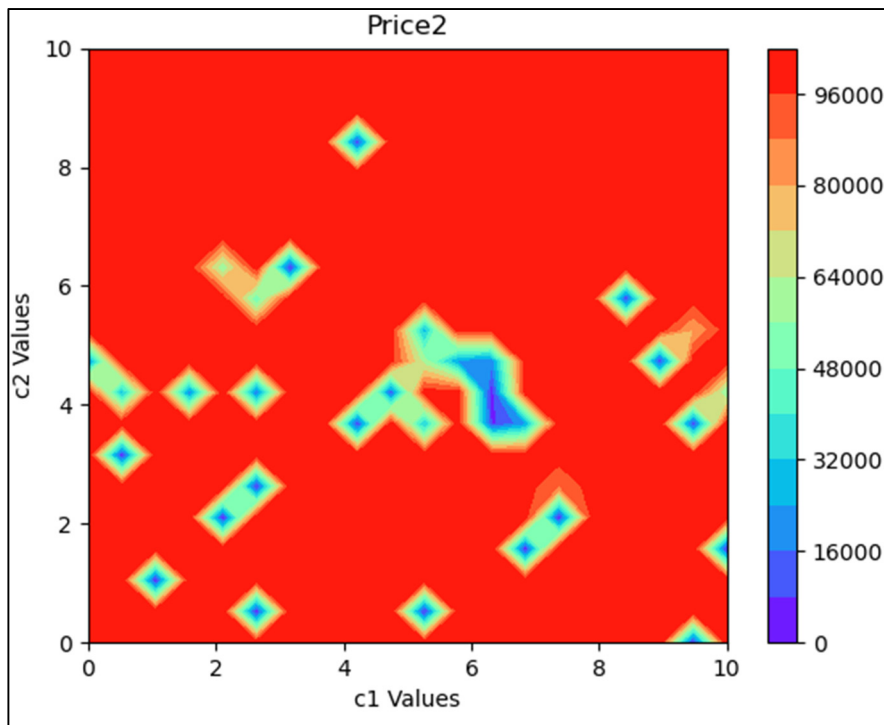


Figure 6.10 - Average Number of Function Calls for Solving the Price2 Test Function vs c1 and c2 Parameter Values

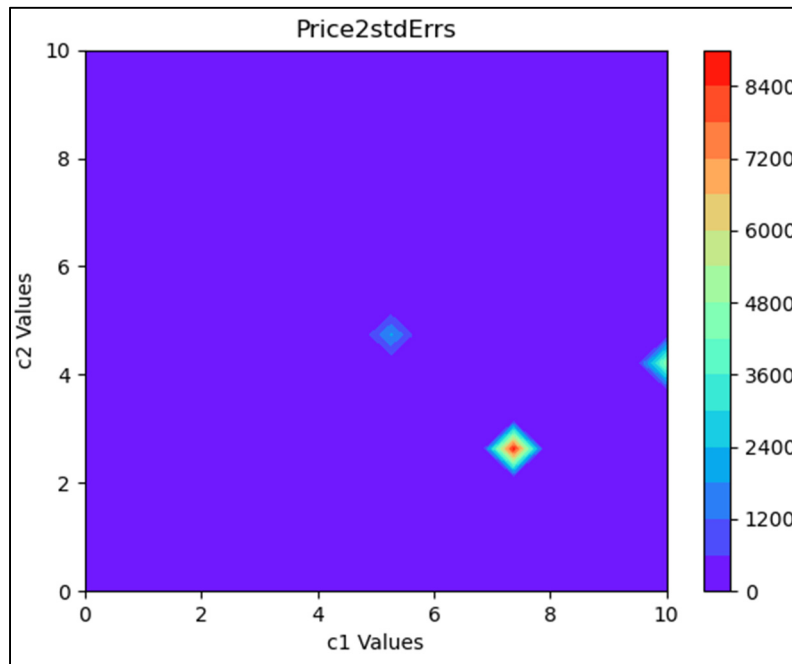


Figure 6.11 - Standard Errors for the Number of Function Calls for Solving the Price2 Test Function vs c1 and c2 Parameter Values

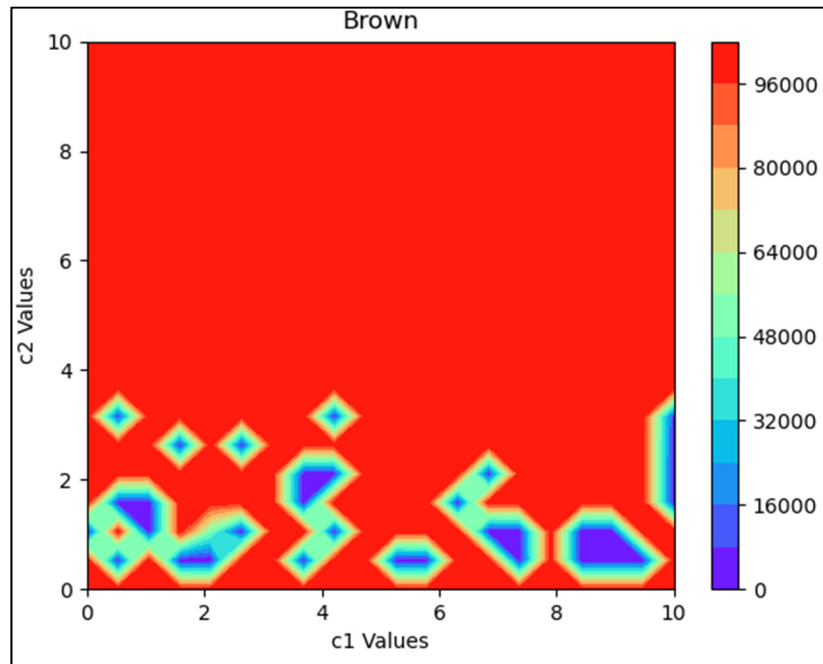


Figure 6.12 - Average Number of Function Calls for Solving the Brown Test Function vs c1 and c2 Parameter Values

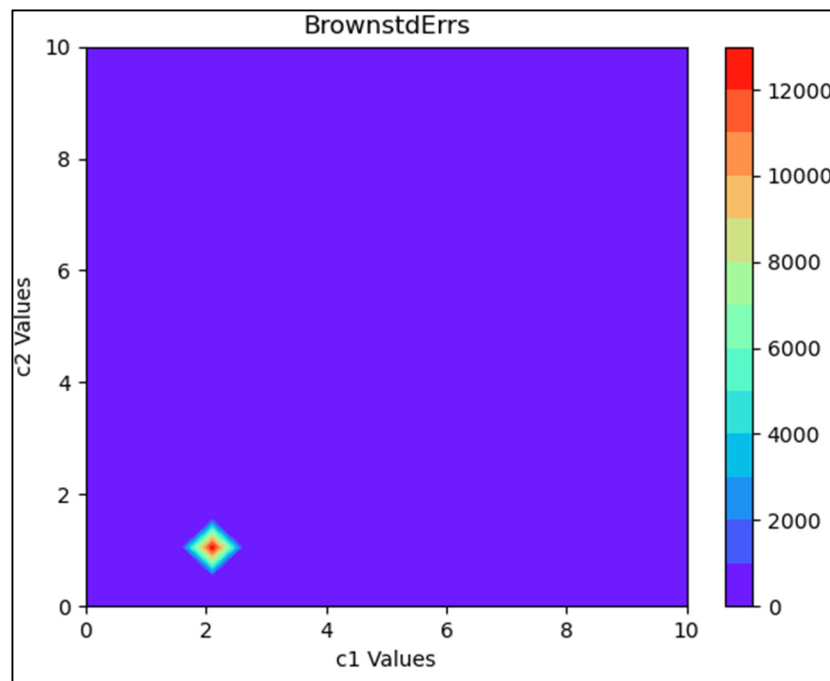


Figure 6.13 - Standard Errors for the Number of Function Calls for Solving the Brown Test Function vs c1 and c2 Parameter Values

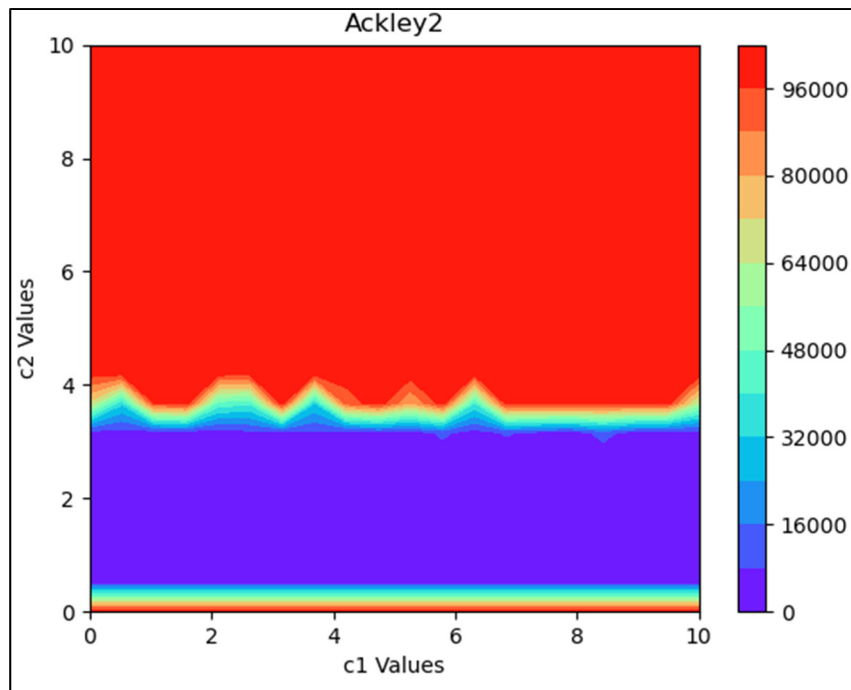


Figure 6.14 - Average Number of Function Calls for Solving the Ackley2 Test Function vs c1 and c2 Parameter Values

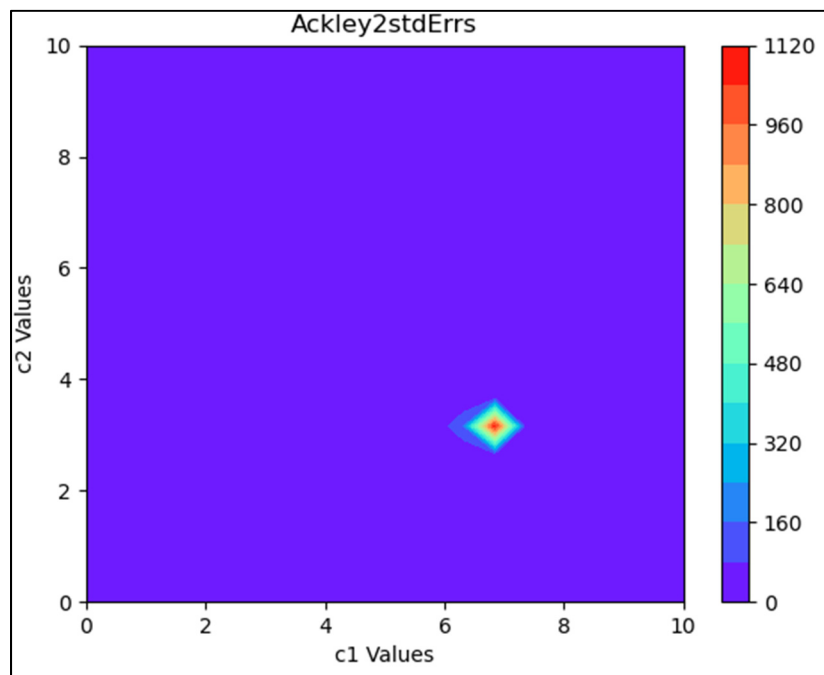


Figure 6.15 - Standard Errors for the Number of Function Calls for Solving the Ackley2 Test Function vs c1 and c2 Parameter Values

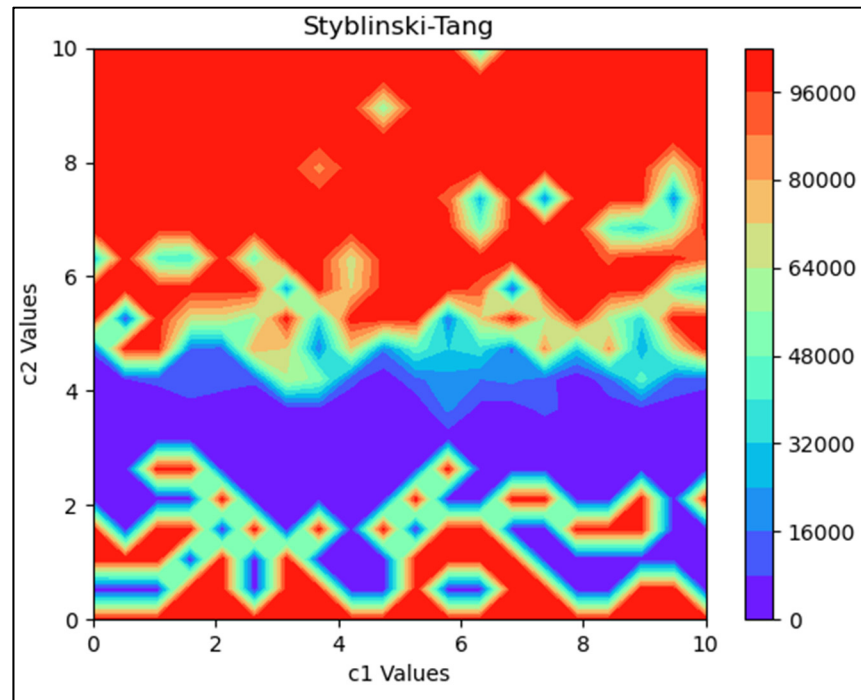


Figure 6.16 - Average Number of Function Calls for Solving the Styblinski-Tang Test Function vs c_1 and c_2 Parameter Values

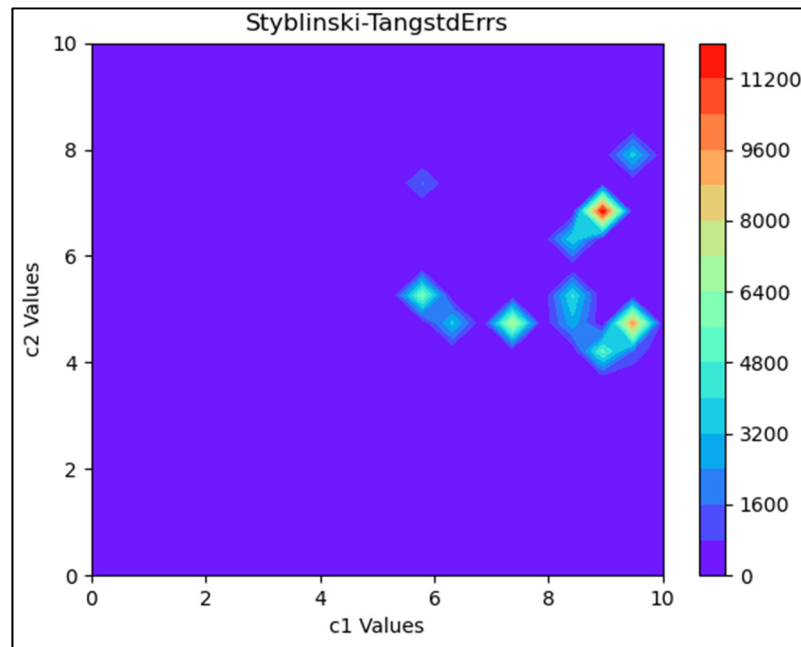


Figure 6.17 - Standard Errors for the Number of Function Calls for Solving the Styblinski-Tang Test Function vs c_1 and c_2 Parameter Values

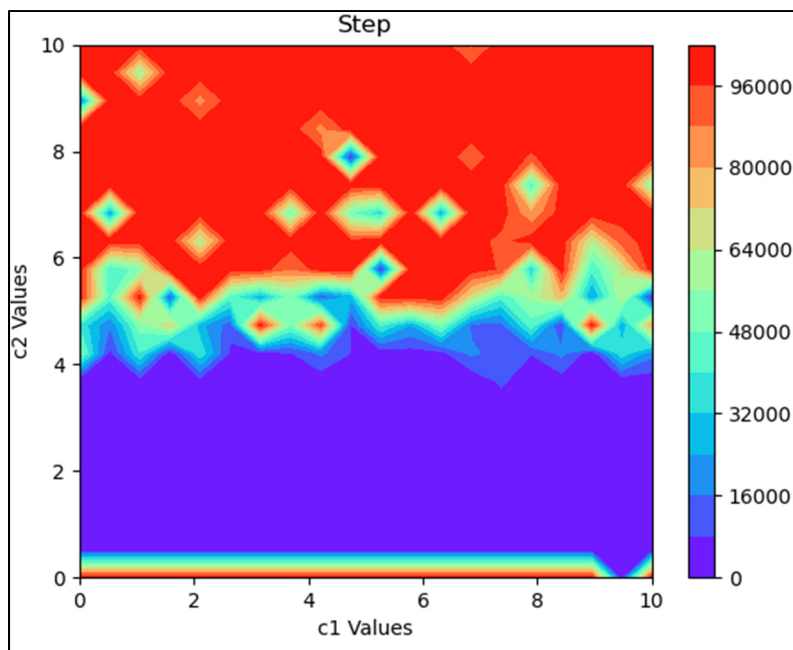


Figure 6.18 - Average Number of Function Calls for Solving the Step Test Function vs c1 and c2 Parameter Values

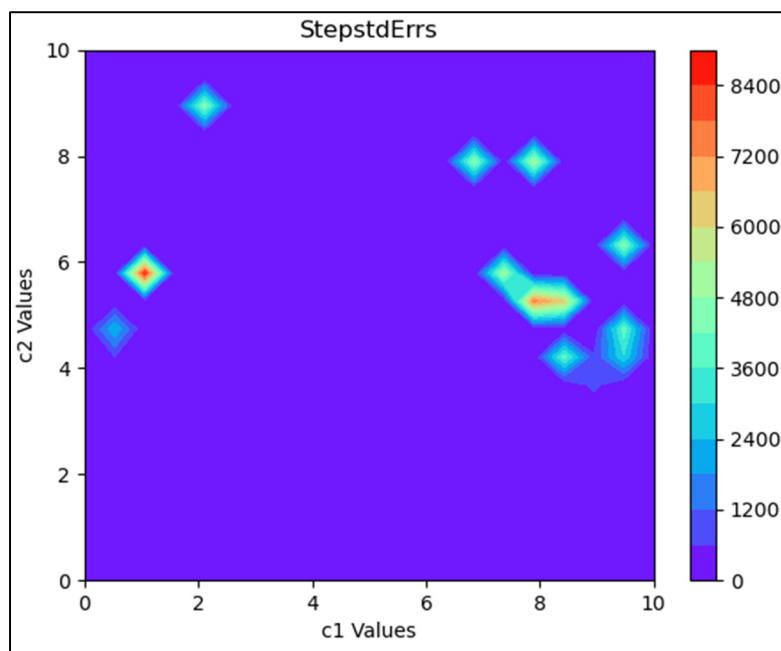


Figure 6.19 - Standard Errors for the Number of Function Calls for Solving the Step Test Function vs c1 and c2 Parameter Values

6.3. Discussion

The heatmaps displayed in the results section show the performance of the PSO algorithm at various $c1$ and $c2$ parameter values. These parameters are used to calculate the “velocity” of each point at each iteration and $c1$ and $c2$ are respectively used as weights to determine the importance of each point’s previous best and the historical best point of the algorithm. Please see APPENDIX IX for the full PSO algorithm.

As explained above, the performance of the PSO algorithm is measured by the number of times it needed to call the function being optimized for each run. This measure is taken 16 times for each point (unique combination of $c1$ and $c2$ values) and the average is used for analysis. This is a good way to detect the stochastic nature of some algorithms. As PSO does not highly rely on randomness to achieve global optimization but rather utilises a population of points, the relatively small number of deviations observed for most functions analysed makes sense. For example, Step function’s results show that only a small subset of the possible combinations between $c1$ and $c2$ generate high levels of deviation in the performance of the algorithm between runs with the same parameters.

From the Leon test function results, we observe that most combinations of $c1$ and $c2$ yield a poor performance of PSO. The few effective combinations are almost randomly distributed across the parameter space. This could be explained by the shape of the test function. As it can be observed in Figure 6.3, the Leon test function’s shape is flat everywhere except very close to the optimal. This makes it very difficult for PSO to optimize it and the performance of the algorithm is therefore expected to be poor (high number of calls to the test function). This is consistent with the observed results from MDAF, and the few parametric regions of good performance can be explained by random chance as the initial positions of all points of the algorithm is random.

The Price2 test function results also show a poor performance of PSO. This is also to be expected as this function is very chaotic, and algorithm can very easy remain trapped in a local optimal. In fact, the heatmap for this test function shows that the huge majority of combinations

for c_1 and c_2 yield poor performance by PSO with small random regions of effective values mostly concentrated towards the lower values of c_2 . This means that this test function is easier to optimize when each particle follows its path to the best solution instead of all converging towards a historical general best point. This is explained by the chaotic nature of the test function since PSO being less social (higher c_1 values) makes it explore a wider area of possibilities compared to more social combinations of c_1 and c_2 (higher values of c_2).

The Brown test function is showing more significant results as a concentration of effective parameters can be observed for lower values of the c_2 parameter. This is similar to the case of Price2. In fact, the shape of Brown (Figure 6.4), despite being very different from Price2, is also challenging for PSO to optimize as it does not provide a good slope to follow toward the optimal. The improvement introduced for smaller values of the c_2 parameter can be explained by the fact that it pushes the algorithm to be more exploratory as each point follows a more independent route from the others.

Ackley2's results are showing a meaningful trend as the combinations with c_2 values below three always produce effective performances of PSO. This indicates that the c_1 parameter's value does not matter for this test function for values ranging from zero to ten. This can be explained by the shape of the Ackley2 test function. As is shown in Figure 6.5, Ackley2 does not have local minima. Therefore, low values for the c_2 parameter ensure that the algorithm follows a gradient descent towards the optimal solution as quickly as possible as the points in the population fall down the funnel all together instead of being "distracted" by the general historical best point. In fact, high values of the c_2 parameter can cause the particles to overshoot the optimal. To further understand what happens in the "ideal" region for the parameters c_1 and c_2 of PSO for this function, a specific benchmarking study was performed on Ackley2 for c_1 values ranging between 0 and 3 and c_2 values going from 0 to 10. The following heatmap was produced showing a complex region with values ranging from 400 calls to 6000. All these points could be considered acceptable parameters for the algorithm depending on other constraints on the design. In fact, for such small variations in the performance, other factors might have a bigger impact on the performance of PSO on similar algorithms. MDAF can also

be used in this scenario to evaluate these other factors like the values of the weight parameter, and the number of particles used by the algorithm.

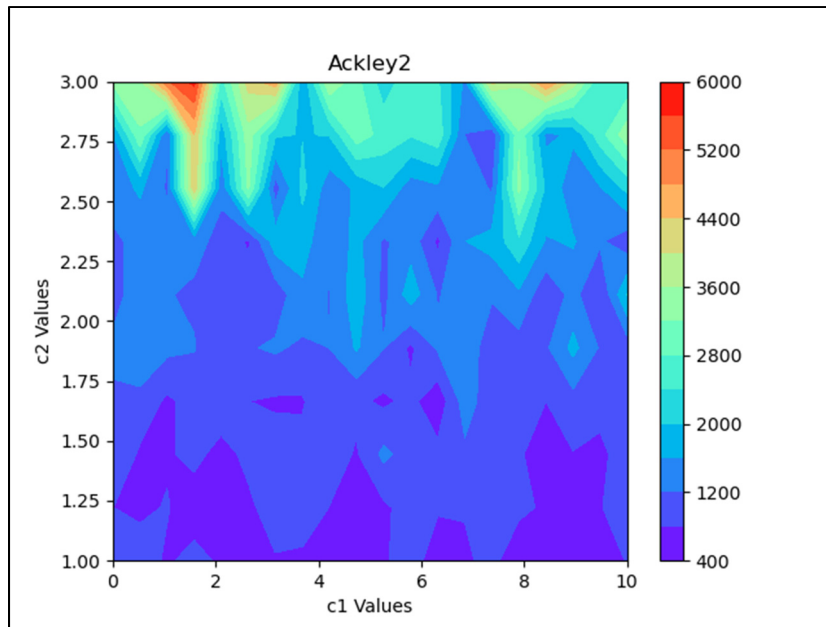


Figure 6.20 - PSO Average Number of Function Calls Heatmap for Sub Region of Interest for the Ackley2 Test Function

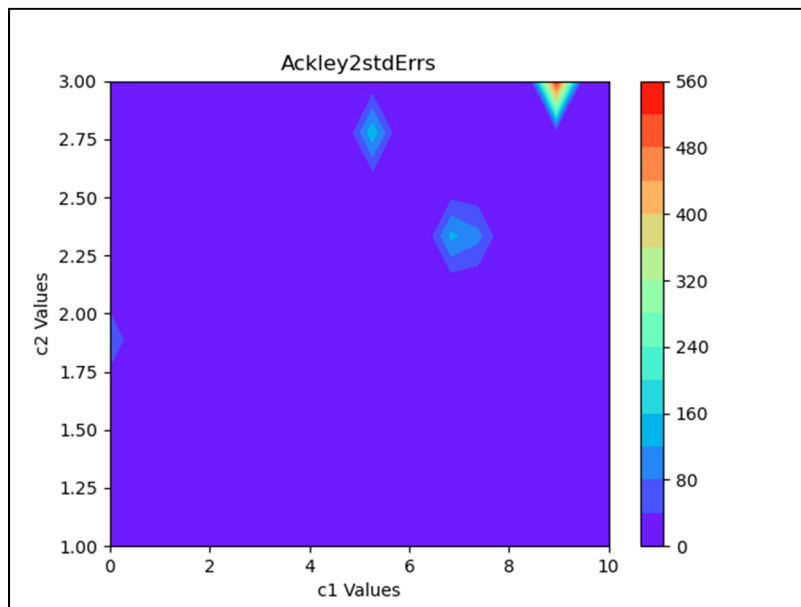


Figure 6.21 - Standard Error Heatmap for the Above Average Values of the Performance of PSO on Ackley2

The Styblinski-tang test function also shows a similar behavior of being mostly influenced by the value of the $c2$ parameter. In fact, it has a shape that is loosely similar to Ackley2 since there is no local minima in this function. The effective values of $c2$ for this test function are between two and four.

The step function being very similar to Ackley2, and Styblinski-Tang also outputted similar performance data. The effective values of $c2$ for this function are between 0.5 and 4.

As discussed above, the $c2$ parameter had the most importance on the performance of PSO on the studied functions. It was also observed that similar test functions generated similar performance maps for the PSO algorithm. This agrees with the prediction outlined by the NFLT. As the trends outlined are good for building rules of thumbs, it is also relevant to note that the heatmaps showcased specific cases of $c1$ and $c2$ parameter combinations like with the Step function. In fact, despite the fact that most optimal combinations happen with $c2$ below 4, some combinations like $c1$ at 5.2 and $c2$ at 7.9 yields effective performance. This can be useful in exceptional cases.

Observing the gcm feature set computed with the integrated FLACCO module of MDAF, it is apparent that the Step and Ackley2 test functions are very similar. In fact, the traces of both these functions on Figure 6.7 are almost identical. It is therefore expected that PSO will have similar performances on both functions. This is because ELA features capture the important characteristics for predicting the performance of optimization algorithms on the function in question. As can be observed in the results section (Figure 6.18, and Figure 6.14), the performance of PSO is very similar for both functions. The figure below compares the feature sets for the Step and Ackley2 test functions as well as the performance heatmaps of PSO to solve them at various values of $c1$ and $c2$.

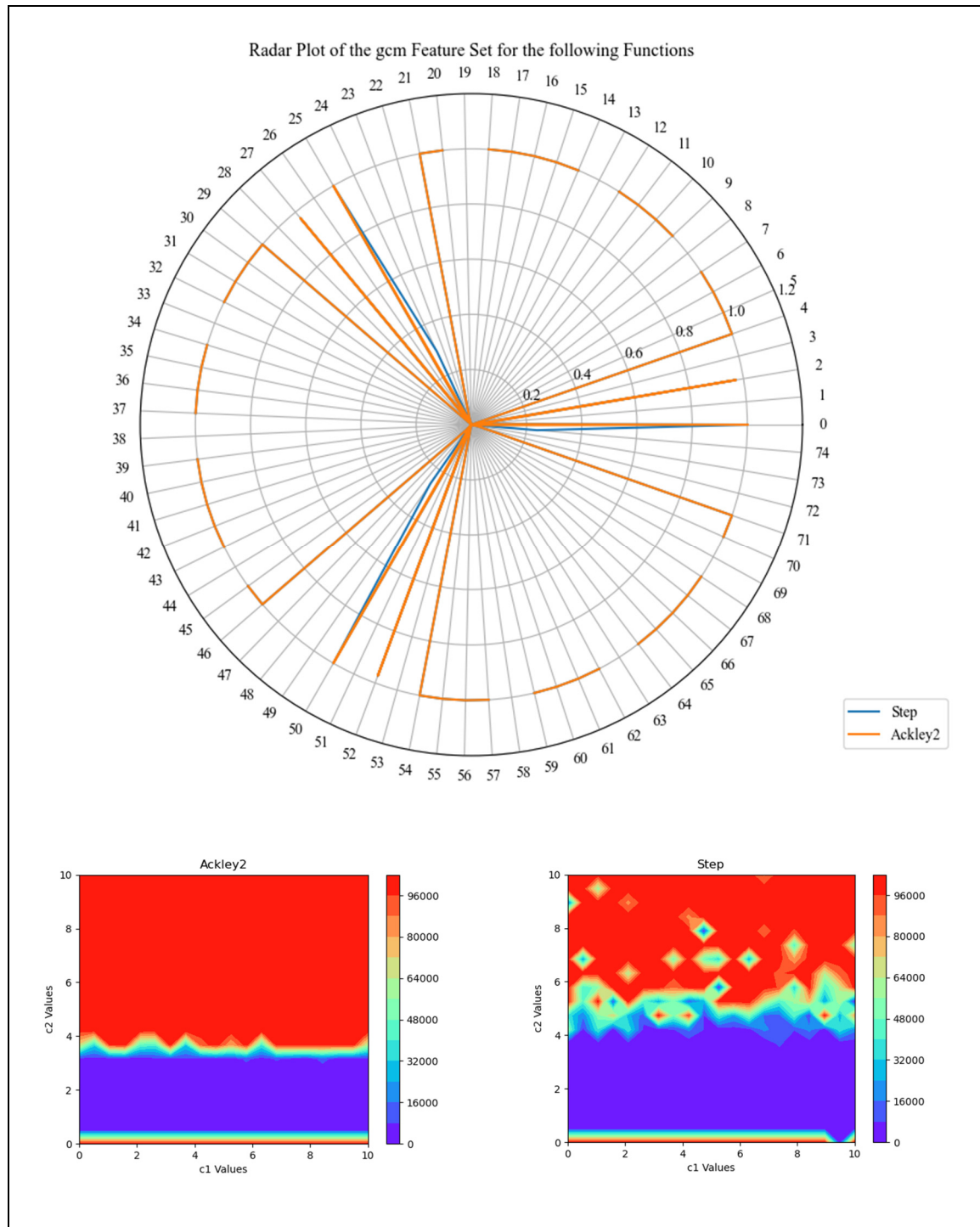


Figure 6.22 - Comparison of the Representation and Performance of the Ackley2 and Step Test Functions

The gcm feature set radar plots overlaps for both functions and the performance characteristics of PSO for both show a similar pattern. Indeed, insights obtained from the study of the Ackley2 test function can be effectively applied to the Step function. This is in line with the NFLT which implies that optimization algorithms have similar performances on problems of the same category. It also shows that ELA feature sets can be an effective method of classifying functions to be optimized. It is relevant to note the variations between both cases as the Step function is more complex than Ackley2 and therefore has a more intricate performance heat map than Ackley2. However, these slight differences are to be expected as these are two different functions and a careful selection of optimal parameters from the simpler function (Ackley2) applies very well to the more complex one (Step).

6.4. Test Case Conclusion

it appears that the most effective parameters to be used with PSO vary depending on the type of function to be optimised despite the existence of some popular combinations of $c1$ and $c2$ which work well on a wide array of function types. This highlights the usefulness of MDAF as it makes it possible to readily assess the performance of a proposed algorithm on various problem types. The addition of FLACCO to the framework improves its usefulness further as it is also possible to generate a quantitative representation of an arbitrary problem to be optimized using exploratory landscape analysis and comparing this representation to available test functions. A suitably similar test function can then be used to perform a meta optimization (the optimization of the parameters of an optimization algorithm) either manually or automatically using MDAF. This would be very useful for costly functions like multi-domain simulations which often need to be optimized in the design process of various products. For example, computational fluid dynamics simulations are often conducted to optimize the shape of formula1 airfoils. The same type of studies is also conducted in aerospace for optimizing the shape of turbo compressor blades and turbine vanes. As these simulations can be quite costly to compute, it is important to make sure that the algorithm being used for optimization is tuned to achieve the ideal solution with as few objective function evaluations as possible.

CONCLUSION AND PERSPECTIVES

This research explored algorithm benchmarking using quantitative analysis in an attempt to simplify and streamline this process. A Framework (MDAF) was developed using the Python programming language to automate the computation of the benchmarks of given algorithms. This framework contains various features like visualization helper functions, and test function characterization methods. For example, the framework has default 3D and radar plotting routines that can generate standardized visualizations of any mathematical numerical function.

MDAF facilitates the implementation of the best practice guidelines outlined in the first chapter of this paper by automating the benchmarking process and implementing reliable benchmarking metrics like the number of objective function calls required by an algorithm to achieve a solution. Other Benchmarking metrics are also implemented into MDAF like the CPU time of analysis as well as the convergence rates. The CPU time is considered a reliable metric because it measures specifically the time that was spent by the CPU on the analysis processes only and excludes other operating system runtimes. Therefore, this metric is repeatable regardless of other processes on the analysis computer unlike the runtime metric which is often used. Finally, the convergence rate indicates what proportion of test runs achieved the desired results against the number of runs which diverged.

Following the development of the Framework, a case study was performed using MDAF to benchmark a PSO algorithm. This case study showed that the framework was capable of benchmarking the performance of optimization algorithms for various parameter values. Therefore, the framework can be utilised in many types of analyses for metaheuristics like parameter optimization.

This work can be extended and improved in many ways to achieve a better understanding of any metaheuristic. For example, regression models of the performance of metaheuristics can be implemented using an engine like TensorFlow. The inputs of such a model would be the ELA features of a test function and the model could output a predicted performance or a suitability score. Autoencoders can be used to eliminate non important ELA features in this use case. Another use case would be to utilize MDAF in the design of an algorithm selection model which would automatically determine the best algorithm to be used based on the ELA feature representation of a test function. Such a model could find applications in any industry which performs optimization analyses. In fact, such a model would take the guessing work out of choosing which solvers to use for given optimization problems.

APPENDIX I. SIMULATED ANNEALING PSEUDO CODE

Algorithm-A I-1 Simulated Annealing Pseudo Code

```
/*  
Borrowed Code: Simulated Annealing pseudo code  
The following lines have been borrowed from (Luke, 2013)  
*/  
1.  $t \leftarrow \text{temperature, initially a high number}$   
2.  $S \leftarrow \text{some initial candidate solution}$   
3.  $Best \leftarrow S$   
4. repeat  
    1.  $R \leftarrow \text{Tweak}(\text{Copy}(S))$   
    2. if  $\text{Quality}(R) >$   
         $\text{Quality}(S)$  or if a random number chosen from 0 to 1  $<$   
         $e^{\frac{\text{Quality}(R) - \text{Quality}(S)}{t}}$  then  
        3.  $S \leftarrow R$   
        4. Decrease  $t$   
    5. if  $\text{Quality}(S) > \text{Quality}(Best)$  then  
        1.  $Best \leftarrow S$   
5. until  $Best$  is the ideal solution, we have run out of time, or  $t \leq 0$   
6. return  $Best$   
/* End of the Borrowed Code */
```

APPENDIX II.

PACKAGE META DATA

Algorithm-A II-1 Package Meta Data for MDAF

Package meta data:

```
1. [metadata]
2. name = MDAF
3. version = 0.1
4. description =A Framework for the Analysis and Benchmarking of
  Metaheuristics
5. url = https://git.rehounou.ca/remi/MDAF
6. author = Remi Ehounou
7. author_email = remi.ehounou@outlook.com
8. license = MIT
9. long_description = file: README.md
10. long_description_content_type = text/markdown
11. classifiers =
12.     Programming Language :: Python :: 3
13.     License :: OSI Approved :: MIT License
14.     Operating System :: OS Independent
15.
16.
17. [options]
18. package_dir =
19.     = .
20. include_package_data = True
21. packages = find:
22. python_requires = >=3.6
23. install_requires =
24.     numpy
25.     sklearn
26.     matplotlib
27.     rpy2 == 3.4.4
```

APPENDIX III.

MDAF CODE

Algorithm-A III-1 MDAF Algorithm Implementation

MDAF main Code

```

1.
2.     from os import path
3.     from os import sys
4.     import importlib.util
5.     import multiprocessing
6.     import time
7.     import re
8.     from numpy import random as rand
9.     from numpy import array, isnan, NaN, asarray, linspace,
        append, meshgrid, ndarray
10.    import statistics
11.    from functools import partial
12.    import shutil
13.
14.    # Surrogate modelling and plotting
15.    import matplotlib.pyplot as plt
16.    from sklearn.neural_network import MLPRegressor
17.    from sklearn.model_selection import train_test_split
18.
19.    # Test function representation
20.    from rpy2 import robjects as robjs
21.    from rpy2.robjects.packages import importr
22.    from rpy2 import rinterface
23.
24.    # Test function characteristics
25.    import statistics as st
26.
27.
28.    def installFlacco(mirror =
        'https://utstat.toronto.edu/cran/'):

```

```

29.         utils = importtr('utils')
30.         utils.install_packages('flacco', repos=mirror)
31.         utils.install_packages('list', repos=mirror)
32.         utils.install_packages('lhs', repos=mirror)
33.         utils.install_packages('plyr', repos=mirror)
34.         utils.install_packages('RANN', repos=mirror)
35.         utils.install_packages('numDeriv', repos=mirror)
36.         utils.install_packages('e1071', repos=mirror)
37.
38.     class counter:
39.         #wraps a function, to keep a running count of how many
40.         #times it's been called
41.         def __init__(self, func):
42.             self.func = func
43.             self.count = 0
44.
45.         def __call__(self, *args, **kwargs):
46.             self.count += 1
47.             return self.func(*args, **kwargs)
48.
49.     def simulate(algName, algPath, funcname, funcpath, args,
50.                 initpoint):
51.         # loading the heuristic object into the namespace and
52.         # memory
53.         spec = importlib.util.spec_from_file_location(algName,
54.                                                         algPath)
55.         heuristic = importlib.util.module_from_spec(spec)
56.         spec.loader.exec_module(heuristic)
57.
58.         # loading the test function object into the namespace and
59.         # memory
60.         testspec =
61.             importlib.util.spec_from_file_location(funcname, funcpath)
62.         func = importlib.util.module_from_spec(testspec)
63.         testspec.loader.exec_module(func)

```

```

60.         # defining a countable test function
61.         @counter
62.         def testfunc(args):
63.             return func.main(args)
64.
65.         # using a try statement to handle potential exceptions
        raised by child processes like the algorithm or test functions or
        the pooling algorithm
66.         try:
67.             #This timer calculates directly the CPU time of the
            process (Nanoseconds)
68.             tic = time.process_time_ns()
69.             # running the test by calling the heuristic script
            with the test function as argument
70.             quality = heuristic.main(testfunc, initpoint, args)
71.             toc = time.process_time_ns()
72.             # ^^ The timer ends right above this; the CPU time is
            then calculated below by simple difference ^^
73.
74.             # CPU time in seconds
75.             cpuTime = (toc - tic)*(10**-9)
76.             numCalls = testfunc.count
77.             converged = 1
78.         except:
79.             quality = NaN
80.             cpuTime = NaN
81.             numCalls = testfunc.count
82.             converged = 0
83.         return cpuTime, quality, numCalls, converged
84.
85.     def measure(heuristicpath, funcpath, args, connection,
        sampleSize = 30):
86.         '''
87.         This function runs a set of optimization flows for each
            test function. it returns the mean and standard deviation of the
            performance results
88.         '''

```



```

89.
90.         #defining the heuristic's name
91.         heuristic_name =
           path.splitext(path.basename(heuristicpath))[0]
92.
93.         #defining the test function's name
94.         funcname = path.splitext(path.basename(funcpath))[0]
95.
96.         # Seeding the random module for generating the initial
           point of the optimizer: Utilising random starting point for
           experimental validity
97.         rand.seed(int(time.time()))
98.
99.         # guetting the representation of the function
100.        funcChars = representfunc(funcpath)
101.
102.        n = funcChars['dimmmensions']
103.        upper = funcChars['upper']
104.        lower = funcChars['lower']
105.
106.        if not isinstance(upper, list): upper = [upper for i in
           range(n)]
107.        if not isinstance(lower, list): lower = [lower for i in
           range(n)]
108.
109.        scale = list()
110.        for i in range(n):
111.            scale.append(upper[i] - lower[i])
112.
113.
114.        # Defining random initial points to start testing the
           algorithms
115.        initpoints = [[rand.random() * scale[i] + lower[i] for i
           in range(n)] for run in range(sampleSize)] #update the inner as
           [rand.random() * scale for i in range(testfuncDimmmensions)]
116.        # building the iterable arguments

```

```

117.         partfunc = partial(simulate, heuristic_name,
118.             heuristicpath, funcname, funcpath, args)
119.
120.         n_proc = multiprocessing.cpu_count() # Guetting the
121.             number of cpus
122.         with multiprocessing.Pool(processes = n_proc) as pool:
123.             # running the simulations
124.             newRun = pool.map(partfunc, initpoints)
125.
126.             cpuTime = array([resl[0] for resl in newRun])
127.             quality = array([resl[1] for resl in newRun])
128.             numCalls = array([resl[2] for resl in newRun])
129.             converged = array([resl[3] for resl in newRun])
130.
131.             cpuTime = cpuTime[~(isnan(cpuTime))]
132.             quality = quality[~(isnan(quality))]
133.             numCalls = numCalls[~(isnan(numCalls))]
134.             converged = converged[~(isnan(converged))]
135.
136.             results = dict()
137.             results['cpuTime'] = array([statistics.fmean(cpuTime),
138.                 statistics.stdev(cpuTime)]) if cpuTime.size > 0 else array([])
139.             results['quality'] = array([statistics.fmean(quality),
140.                 statistics.stdev(quality)]) if quality.size > 0 else array([])
141.             results['numCalls'] = array([statistics.fmean(numCalls),
142.                 statistics.stdev(numCalls)]) if numCalls.size > 0 else array([])
143.             results['convRate'] = array([statistics.fmean(converged),
144.                 statistics.stdev(converged)]) if converged.size > 0 else array([])
145.
146.             connection.send((results, newRun, funcChars))
147.
148.         def writerepresentation(funcpath, charas):
149.             # Save a backup copy of the function file
150.             shutil.copyfile(funcpath, funcpath + '.old')
151.
152.             # create a string format of the representation variables

```

```

148.         representation = ''
149.         for line in list(charas):
150.             representation += '\n\t#_# ' + line + ': ' +
                repr(charas[line]).replace('\n', '')
151.             representation += '\n\n\t#_# Represented: 1\n\n'
152.
153.         # Creating the new docstring to be inserted into the file
154.         with open(funcpath, "r") as file:
155.             content = file.read()
156.             docstrs = re.findall(r"def
main\(..*?\):.*?'''(.*)'''.*?return\s+.*?", content, re.DOTALL)[0]
157.             docstrs += representation
158.             repl = "\\1"+docstrs+"\t\\2"
159.
160.         # Create the new content of the file to replace the
            old. Replacing the whole thing
161.         patrn = re.compile(r"(def
main\(..*?\):.*?'''.*?return\s+.*?\n|$)", flags=re.DOTALL)
162.         newContent = patrn.sub(repl, content, count=1)
163.         # Overwrite the test function file
164.         with open(funcpath, "w") as file:
165.             file.write(newContent)
166.
167.     def representfunc(funcpath, forced = False):
168.         if (funcpath.find('@') == 0): funcpath =
            path.dirname(__file__) + '/TestFunctions/' + funcpath[1:]
169.
170.         #defining the function name
171.         funcname = path.splitext(path.basename(funcpath))[0]
172.         # loading the function to be represented
173.         spec = importlib.util.spec_from_file_location(funcname,
            funcpath)
174.         funcmodule = importlib.util.module_from_spec(spec)
175.         spec.loader.exec_module(funcmodule)
176.

```

```

177.         # Finding the function characteristics inside the
           docstring
178.         if funcmodule.main.__doc__:
179.             regex = re.compile(r"#_#\s?(\w+):(.+)?\n") # this
           regular expression matches the characteristics already specified in
           the docstring section of the function -- old exp:
           "#_#\s?(\w+):\s?([-+]?(d+(\.\d*)?|\.\d+)([eE] [-+]?d+)?) "
180.             chars = re.findall(regex, funcmodule.main.__doc__)
181.             results = {}
182.             for charac in chars:
183.                 results[charac[0]] =
           eval(charac[1].replace('nan', 'NaN'))
184.
185.             # Automatically generate the representation if the
           docstrings did not return anything
186.             if not ('Represented' in results):
187.                 print("Warning, the Representation of the Test
           Function has not been specified\n==\n*****Calculating the
           Characteristics*****")
188.                 n = int(results['dimensions'])
189.                 blocks = int(1+10/n)
190.                 if blocks < 3: blocks=3
191.
192.                 # Importing FLACCO using rpy2
193.                 flacco = importr('flacco')
194.
195.                 # creating the r functions
196.                 rlist = robjs.r['list']
197.                 rapply = robjs.r['apply']
198.                 rvector = robjs.r['c']
199.                 r_unlist = robjs.r['unlist']
200.                 rtestfunc =
           rinteface.rternalize(funcmodule.main)
201.
202.             # Verify if a list of limits has been specified
           for all dimensions or if all dimensions will use the same
           boundaries

```

```

203.             if (type(results['lower']) is list):
204.                 lowerval =
205.                     r_unlist(rvector(results['lower']))
206.                 uperval =
207.                     r_unlist(rvector(results['upper']))
208.             else:
209.                 lowerval = results['lower']
210.                 uperval = results['upper']
211.
212.             X = flacco.createInitialSample(n_obs = 500, dim =
213.                 n, control = rlist(**{'init_sample.type' : 'lhs',
214.                 'init_sample.lower' : lowerval, 'init_sample.upper' : uperval}))
215.             y = rapply(X, 1, rtestfunc)
216.             testfuncobj = flacco.createFeatureObject(**{'X':
217.                 X, 'y': y, 'fun': rtestfunc, 'lower': lowerval, 'upper': uperval,
218.                 'blocks': blocks, 'force': forced})
219.
220.             # these are the retained features. Note that some
221.             # features are being excluded for being problematic and to avoid
222.             # overcomplicating the neural network.... the feature sets are
223.             # redundant and the most relevant ones have been retained
224.             # the excluded feature sets are: 'bt',
225.             'ela_level'
226.
227.             # feature sets that require special attention:
228.             'cm_angle', 'cm_grad', 'limo', 'gcm' (large set with some nans),
229.             featureset =
230.             ['cm_angle', 'cm_conv', 'cm_grad', 'ela_conv', 'ela_curv', 'ela_distr', '
231.             ela_local', 'ela_meta', 'basic', 'disp', 'limo', 'nbc', 'pca', 'gcm', 'ic']
232.             pyfeats = dict()
233.             for feature in featureset:
234.                 rawfeats =
235.                 flacco.calculateFeatureSet(testfuncobj, set=feature)
236.                 pyfeats[feature] = asarray(rawfeats)
237.
238.             writerepresentation(funcpath, pyfeats)
239.
240.

```

```

225.         for feat in results.keys():
226.             if isinstance(results[feat], ndarray):
227.                 results[feat] =
228.                     results[feat].reshape(results[feat].shape[:-1])
229.         return results
230.
231.
232.
233.     def exp(heuristicpath, testfunctionpaths, args,
234.             measurementSampleSize = 30):
235.         for i, funpath in enumerate(testfunctionpaths):
236.             if funpath.find('@') == 0:
237.                 testfunctionpaths[i] = path.dirname(__file__) +
238.                     '/TestFunctions/' + funpath[1:]
239.
240.             if (heuristicpath.find('@') == 0): heuristicpath =
241.                 path.dirname(__file__) + '/SampleAlgorithms/' + heuristicpath[1:]
242.
243.             #defining the function's name
244.             funcnames = [path.splitext(path.basename(funcpath))[0]
245.                           for funcpath in testfunctionpaths]
246.
247.             #defining the heuristic's name
248.             #heuristic_name =
249.                 path.splitext(path.basename(heuristicpath))[0]
250.
251.             # logic variables to deal with the processes
252.             proc = []
253.             connections = {}
254.
255.             # loading the test functions into the namespace and
256.             memory
257.             for idx, funcpath in enumerate(testfunctionpaths):
258.                 funcname = funcnames[idx]
259.                 # Creating the connection objects for communication
260.                 between the heuristic and this module

```

```

254.         connections[funcname] =
multiprocessing.Pipe(duplex=False)
255.         proc.append(multiprocessing.Process(target=measure,
name=funcname, args=(heuristicpath, funcpath, args,
connections[funcname][1], measurementSampleSize)))
256.
257.         # defining the response variables
258.         responses = {}
259.         failedfunctions = {}
260.
261.         # Starting the subprocesses for each testfunction
262.         for idx,process in enumerate(proc):
263.             process.start()
264.
265.         # Waiting for all the runs to be done
266.         for process in proc: process.join()
267.
268.         for process in proc:
269.             run = process.name
270.             if process.exitcode == 0: responses[run] =
connections[run][0].recv()
271.             else:
272.                 responses[run] = "this run was not successful"
273.                 failedfunctions[run] = process.exitcode
274.                 connections[run][0].close()
275.                 connections[run][1].close()
276.
277.
278.         # display output
279.         print("\n\n|||| Responses: [mean,stdDev] ||||")
280.         for process in proc: print(process.name + "____\n" +
str(responses[process.name][0]) + "\n_____")
281.
282.         #return the performance values
283.         return responses
284.

```

```

285.     def plotfuncs(funcpaths, feature, low_limit = 0, high_limit =
286.         200):
287.         pi = 3.141592653589793
288.         for i, funpath in enumerate(funcpaths):
289.             if funpath.find('@') == 0:
290.                 funcpaths[i] = path.dirname(__file__) +
291.                     '/TestFunctions/' + funpath[1:]
292.                 funcnames = [path.splitext(path.basename(funcpath))[0]
293.                     for funpath in funcpaths]
294.                 representations = {}
295.                 for idx, funpath in enumerate(funcpaths):
296.                     representations[funcnames[idx]] =
297.                         representfunc(funpath)[feature]
298.                 # generate a list of the categories of the plot
299.                 elements = list(representations.values())
300.                 categories = [str(i) for i in
301.                     list(range(len(elements[0])))]
302.                 # creating the plot figure
303.                 fig = plt.figure(figsize = (12,8))
304.                 ax = plt.subplot(polar = "True")
305.                 for idx, func in enumerate(representations):
306.                     vals = representations[func]
307.                     vals = [float(v) for v in vals]
308.                     # get the number of dims of the plot
309.                     N = len(vals)
310.                     # repeat the first value to close the circle
311.                     vals += vals[:1]
312.                     #calculate the angles for each category
313.                     angles = [n/float(N)*2*pi for n in range(N)]
314.                     angles += angles[:1]
315.                     #creating the polar plot

```



```

317.         ax.plot(angles,vals)
318.
319.         # X ticks
320.         plt.xticks(angles[:-1], categories)
321.
322.         #ax.set_rlabel_position(0)
323.
324.         # y ticks
325.         # set dynamic scaling for each dimension
326.         plt.ylim(low_limit,high_limit)
327.
328.         plt.title("Radar Plot of the "+feature+ " feature for the
           following Functions")
329.         plt.legend(funcnames)
330.         plt.show(block=True)
331.         return representations
332.
333.     def visualize2D(funcpath, min = -10, max=10):
334.         if funcpath.find('@') == 0:
335.             funcpath = path.dirname(__file__) + '/TestFunctions/'
           + funcpath[1:]
336.
337.         # loading the test function object into the namespace and
           memory
338.         testspec =
           importlib.util.spec_from_file_location(path.splitext(path.basename(
           funcpath))[0], funcpath)
339.         func = importlib.util.module_from_spec(testspec)
340.         testspec.loader.exec_module(func)
341.
342.         # create the 2D mx
343.         x = linspace(min,max)
344.         y = linspace(min,max)
345.         X, Y = meshgrid(x, y)
346.         vals = array([[X[i][j],Y[i][j]] for i in
           range(X.shape[0]) for j in range(X.shape[1])])

```

```

347.         z = array([func.main(arg) for arg in vals])
348.         Z = z.reshape([50,50])
349.         fig = plt.figure()
350.         ax = plt.axes(projection='3d')
351.         ax.plot_surface(X,Y,Z)
352.         plt.show()
353.
354.
355.
356.     def model(features, doe_data):
357.
358.         X_train, X_test, y_train, y_test =
            train_test_split(features, doe_data, random_state=1)
359.
360.         regr = MLPRegressor(random_state=1,
            max_iter=500).fit(X_train, y_train)
361.
362.         score = regr.score(X_test, y_test)
363.         return (score, regr)
364.
365.
366.     if __name__ == "__main__":
367.         #plotfuncs(['@Bukin2.py', '@Bukin6.py'], 'ela_meta')
368.         testfuns =
            ['@Bukin2.py', '@Bukin6.py', '@Leon.py', '@Miele_Cantrell.py', '@Brown.
            py', '@Keane.py', '@McCormick.py']
369.         #perf = exp('@SimulatedAnnealing.py', testfuns, {"t":
            1000, "p": 0.95, "objs": 0}, measurementSampleSize=30)
370.         visualize2D(testfuns[1])
371.         #feats = array([representfunc(testfun) ['ela_meta'] for
            testfun in testfuns])
372.
373.         perfs = array([[perf[func][0] ['cpuTime'] [0],
            perf[func][0] ['numCalls'] [0], perf[func][0] ['quality'] [0],
            perf[func][0] ['convRate'] [0]] for func in perf.keys()])
374.         features = array(feats)
375.

```

```
376.         model(features, perfs)
377.     # %%
```

APPENDIX IV.

SAMPLE STRUCTURE OF PREPROGRAMMED TEST FUNCTIONS

Algorithm-A IV-1 The Bukin2 Test Function Implementation

Bukin2 function:

```
1. def main(args):  
2.     '''  
3.  
4.     #_# dimmensions: 2  
5.     #_# upper: [-5, 3]  
6.     #_# lower: [-15, -3]  
7.     #_# minimum: [-10,0]  
8.     #_# Represented: 0  
9.  
10.    '''  
11.    return 100*(args[1]-0.01*args[0]**2+1)+0.01*(args[0]+10)**2
```

APPENDIX V.

AUTOMATED TESTS ALGORITHMS

Algorithm-A V-1 The Unittest Automated Testing Script

Automated Testing Script

```
1.      import unittest
2.      import os
3.
4.      from MDAF.MDAF import representfunc
5.      from MDAF.MDAF import exp
6.
7.      #target = __import__("MDAF.py")
8.
9.      # Testing the test function representation workflow
10.     class Test_representfunc(unittest.TestCase):
11.         def testexternalfuncs(self):
12.             """
13.             Test that the function can calculate the
14.             representation and write to the function docstring
15.             """
16.             funcpath = 'tests/Bukin2.py'
17.             #funcpath_backup = 'tests/Bukin2.py.old'
18.
19.             results = representfunc(funcpath, forced = True)
20.
21.             with open(funcpath,"r") as file:
22.                 content = file.read()
23.                 reprCheck = bool(content.find('#_# Represented:
24.                 1'))
25.
26.             #os.remove(funcpath)
27.             #os.replace(funcpath_backup, funcpath)
28.             self.assertTrue(reprCheck)
29.             self.assertIsInstance(results, dict)
```

```

29.         def testinternalfuncs(self):
30.             """
31.             Test that the function can calculate the
               representation and write to the function docstring
32.             """
33.             funcpath = '@Bukin2.py'
34.             funcverify = 'MDAF/TestFunctions/Bukin2.py'
35.             #funcpath_backup = 'tests/Bukin2.py.old'
36.
37.             results = representfunc(funcpath, forced = True)
38.
39.             with open(funcverify,"r") as file:
40.                 content = file.read()
41.                 reprCheck = bool(content.find('#_# Represented:
1')')
42.
43.                 #os.remove(funcpath)
44.                 #os.replace(funcpath_backup, funcpath)
45.                 self.assertTrue(reprCheck)
46.                 self.assertIsInstance(results, dict)
47.
48.
49.
50.         # Testing the flacco installation workflow
51.         class Test_flaccoInstall(unittest.TestCase):
52.             def testoutput(self):
53.                 """
54.                 Test that the flacco packages are able to install
               automatically
55.                 """
56.                 #installFalcoo()
57.
58.
59.         # Testing the DOE execution workflow
60.         class Test_DOE(unittest.TestCase):
61.             def testexternalfuncs(self):
62.                 """

```

```

63.         Test that it can execute a DOE and output the
           dictionary of the results
64.         """
65.         testfunctionpaths = ["tests/Bukin2.py"]
66.         heuristicpath = "tests/SimulatedAnnealing.py"
67.         args = {"t": 1000, "p": 0.95, "objs": 0}
68.         data = exp (heuristicpath, testfunctionpaths, args)
69.         self.assertIsInstance(data, dict)
70.
71.     def testinternalfuncs(self):
72.         """
73.         Test that it can execute a DOE and output the
           dictionary of the results
74.         """
75.         testfunctionpaths = ["@Bukin2.py"]
76.         heuristicpath = "@SimulatedAnnealing.py"
77.         args = {"t": 1000, "p": 0.95, "objs": 0}
78.         data = exp (heuristicpath, testfunctionpaths, args)
79.         print(data)
80.         self.assertIsInstance(data, dict)

```

APPENDIX VI.

FLACCO Features Data

Table-A VI-1 The gcm Feature Set Values

	Leon	Price2	Ackley2	Step	Brown	Styblinski-Tang
1	1	7	1	1	1	4
2	0.02777778	0.19444444	0.02777778	0.02777778	0.001371742	0.11111111
3	0.97222222	0.80555556	0.97222222	0.97222222	0.541838134	0.88888889
4	0	0.66666667	0	0	0	0.47222222
5	1	0.09454594	1	1	0.543209877	0.20065046
6	1	0.14285714	1	1	0.543209877	0.25
7	1	0.11484541	1	1	0.543209877	0.24278043
8	1	0.29185606	1	1	0.543209877	0.31378867
9	nan	0.07241316	nan	nan	nan	0.04700162
10	1	0.02777778	1	1	0.543209877	0.11111111
11	1	0.04761905	1	1	0.543209877	0.13194444
12	1	0.02777778	1	1	0.543209877	0.11111111
13	1	0.11111111	1	1	0.543209877	0.19444444
14	nan	0.03090826	nan	nan	nan	0.04166667
15	1	0.33333333	1	1	0.543209877	0.52777778
16	1	0.02777778	1	1	0.543209877	0.25

	Leon	Price2	Ackley2	Step	Brown	Styblinski-Tang
17	1	0.14285714	1	1	0.543209877	0.25
18	1	0.11111111	1	1	0.543209877	0.25
19	1	0.36111111	1	1	0.543209877	0.25
20	nan	0.11275243	nan	nan	nan	0
21	1	1	1	1	0.543209877	1
22	1	0.18140016	1	1	0.543209877	0.31378867
23	0.02777778	0.02777778	0.02777778	0.02777778	0.001371742	0.02777778
24	0	0	0	0	0	0
25	0.02	0.396	0.021	0.293	7.939	0.19
26	2	6	1	1	1	4
27	0.05555556	0.16666667	0.02777778	0.02777778	0.001371742	0.11111111
28	0.94444444	0.83333333	0.97222222	0.97222222	0.541838134	0.88888889
29	0.83333333	0.75	0	0	0	0.75
30	0.48352763	0.11878172	1	1	0.543209877	0.21908552
31	0.5	0.16666667	1	1	0.543209877	0.25
32	0.5	0.15022098	1	1	0.543209877	0.24131732
33	0.51647237	0.26560387	1	1	0.543209877	0.29827985
34	0.02329546	0.05358846	nan	nan	nan	0.03423605
35	0.08333333	0.02777778	1	1	0.543209877	0.02777778

	Leon	Price2	Ackley2	Step	Brown	Styblinski-Tang
36	0.08333333	0.04166667	1	1	0.543209877	0.0625
37	0.08333333	0.02777778	1	1	0.543209877	0.02777778
38	0.08333333	0.08333333	1	1	0.543209877	0.16666667
39	0.0	0.02324056	nan	nan	nan	0.06944444
40	0.16666667	0.25	1	1	0.543209877	0.25
41	0.44444444	0.05555556	1	1	0.543209877	0.25
42	0.5	0.16666667	1	1	0.543209877	0.25
43	0.5	0.13888889	1	1	0.543209877	0.25
44	0.55555556	0.30555556	1	1	0.543209877	0.25
45	0.07856742	0.0860663	nan	nan	nan	0
46	1	1	1	1	0.543209877	1
47	0.48352763	0.26560387	1	1	0.543209877	0.29827985
48	0.02777778	0.02777778	0.02777778	0.02777778	0.001371742	0.02777778
49	0	0	0	0	0	0
50	0.02	0.335	0.02	0.261	7.537	0.557
51	1	4	1	1	2	4
52	0.02777778	0.11111111	0.02777778	0.02777778	0.002743484	0.11111111
53	0.97222222	0.88888889	0.97222222	0.97222222	0.997256516	0.88888889
54	0	0.44444444	0	0	1	0.75
55	1	0.12066382	1	1	0.5	0.22466354

	Leon	Price2	Ackley2	Step	Brown	Styblinski-Tang
56	1	0.25	1	1	0.5	0.25
57	1	0.2662074	1	1	0.5	0.24621096
58	1	0.34692138	1	1	0.5	0.28291453
59	nan	0.09797158	nan	nan	0.0	0.02445002
60	1	0.05555556	1	1	0	0.02777778
61	1	0.13888889	1	1	0	0.0625
62	1	0.13888889	1	1	0	0.02777778
63	1	0.22222222	1	1	0	0.16666667
64	nan	0.06804138	nan	nan	0.0	0.06944444
65	1	0.55555556	1	1	0	0.25
66	1	0.13888889	1	1	0.488340192	0.25
67	1	0.25	1	1	0.5	0.25
68	1	0.26388889	1	1	0.5	0.25
69	1	0.33333333	1	1	0.511659808	0.25
70	nan	0.08784105	nan	nan	0.0164894585	0
71	1	1	1	1	1	1
72	1	0.23343294	1	1	1	0.28291453
73	0.02777778	0.02777778	0.02777778	0.02777778	0.002743484	0.02777778
74	0	0	0	0	0	0

	Leon	Price2	Ackley2	Step	Brown	Styblinski-Tang
75	0.022	0.279	0.024	0.239	12.115	0.204

APPENDIX VII.

PSO PERFORMANCE DATA

Table-A VII-1 c1 Parameter Values Used for the Case Study

[illegible]

104

[illegible]

Table-A VII-3 Average Number of calls Required to Optimize the Leon Test Function using PSO for each c1 and c2 Parameter variations

[illegible]

Table-A VII-5 Average Number of calls Required to Optimize the Price2 Test Function using PSO for each c1 and c2 Parameter variations

Table-A VII-7 Average Number of calls Required to Optimize the Brown Test Function using PSO for each c1 and c2 Parameter variations

[illegible]

Table-A VII-8 Standard Errors for the Average Numbers of calls Statistics Required to Optimize the Brown Test Function using PSO for each c1 and c2 Parameter variations

[illegible]

Table-A VII-9 Average Number of calls Required to Optimize the Ackley2 Test Function using PSO for each c1 and c2 Parameter variations

Table-A VII-10 Standard Errors for the Average Numbers of calls Statistics Required to Optimize the Ackley2 Test Function using PSO for each c1 and c2 Parameter variations

[illegible]

Table-A VII-13 Average Number of calls Required to Optimize the Step Test Function using PSO for each c1 and c2 Parameter variations

OVFL	251	301	271	631	771	1291	1231	52761	37781	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL	11561	OVFL	OVFL
OVFL	251	381	471	201	357	791	4004	5095	16427	43151	40441	OVFL	20261	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL
OVFL	291	321	331	791	591	961	2772	39361	51141	OVFL	51601	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL	57081	OVFL
OVFL	291	471	301	221	241	791	5851	2391	70401	8841	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL
OVFL	281	371	371	561	1721	2006	4291	37261	26601	OVFL	OVFL	60191	OVFL	OVFL	OVFL	OVFL	OVFL	84780	OVFL
OVFL	281	191	381	511	781	1001	7621	5901	11381	42551	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL
OVFL	221	411	401	531	461	1181	2021	4011	OVFL	24641	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL
OVFL	291	341	461	291	1071	2371	1181	6991	48791	50001	87721	OVFL	48091	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL
OVFL	311	291	441	1271	361	961	2031	17641	92971	9791	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL	84611	OVFL	OVFL
OVFL	261	181	581	1081	251	1081	4931	7861	6361	24911	OVFL	97411	47711	OVFL	11	OVFL	OVFL	OVFL	OVFL
OVFL	351	271	521	741	1461	511	1491	2911	40651	OVFL	631	OVFL	33381	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL
OVFL	291	571	361	601	851	1311	4131	4351	23331	OVFL	98081	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL
OVFL	191	351	301	621	351	2471	281	5131	50761	OVFL	OVFL	OVFL	21601	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL
OVFL	151	261	371	641	471	2131	1261	17661	14111	66821	OVFL	OVFL	OVFL	OVFL	OVFL	91567	OVFL	OVFL	85231
OVFL	211	271	371	271	641	1681	9641	12261	12971	41311	95005	93351	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL
OVFL	311	381	411	331	481	661	654	8331	43891	57912	30591	OVFL	80241	41321	94990	OVFL	OVFL	98285	OVFL
OVFL	288	391	481	301	401	604	3801	17969	5221	75284	OVFL	OVFL	OVFL	OVFL	OVFL	99986	OVFL	OVFL	OVFL
OVFL	161	331	291	321	681	2171	7226	2348	OVFL	20871	41041	62591	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL
91	315	351	331	1041	1771	1451	1851	36542	22626	62961	64291	91678	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL	OVFL
OVFL	251	171	331	511	441	1981	2921	18641	79371	921	OVFL	OVFL	OVFL	52271	OVFL	OVFL	OVFL	OVFL	OVFL

Table-A VII-14 Standard Errors for the Average Numbers of calls Statistics Required to Optimize the Step Test Function using PSO for each c1 and c2 Parameter variations

0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	19.21	0.00	322.78	567.38	2296.25	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	586.42	0.00	0.00	0.00	8425.36	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	151.05	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	4456.14	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00</																

APPENDIX VIII.

CASE STUDY CODE

Algorithm-A VIII-1 Case Study Python Script used with MDAF to Generate the Results

```
testfuns = ['@Ackley2.py', '@Alpine.py', '@Brown.py', '@Bukin2.py', \
 '@Bukin4.py', '@Bukin6.py', '@Keane.py', '@Leon.py', '@Matyas.py', \
 '@McCormick.py', '@Miele_Cantrell.py', '@Periodic.py', \
 '@PowellSingular2.py', '@Price1.py', '@Price2.py', '@Quartic.py', \
 '@Rastriring.py', '@Scahffer.py', '@Schwefel.py', '@Sphere.py', \
 '@Step.py', '@Step2.py', '@Styblinski-Tang.py', '@SumSquare.py', \
 '@Wayburn.py', '@Zettle.py', '@Zirilli.py']

selections = testfuns['@Leon.py', '@Price2.py', '@Ackley2.py',
 '@Step.py', '@Brown.py', '@Styblinski-Tang.py']

# Measurement sample size
print("Input the measurement sample size Default(16): ")
s = int(input() or 16)
print(s)
#PSO population size
print("\n\nInput the PSO population size Default(10): ")
pop = int(input() or 10)
print(pop)

#visualize2D('@Ackley2.py', -10,10)
#feats = array([representfunc(testfun, True)['ela_meta'] for testfun
in testfuns])
#plotfuncs(['@Bukin2.py', '@Bukin6.py'], 'ela_meta')

# Initialize output data model
# Value iterations
print("\n\nInput the max min and number of elements for the first
parameter iterations. Default(0,10,20): ")
vmax = int(input() or 0)
vmin = int(input() or 10)
vnum = int(input() or 20)
print(vmax, vmin, vnum)
###
print("\n\nInput the max min and number of elements for the second
parameter iterations. Default(0,10,20): ")
vmax2 = int(input() or 0)
vmin2 = int(input() or 10)
vnum2 = int(input() or 20)
print(vmax2, vmin2, vnum2)
### First parameter
iterations = linspace(vmax, vmin, vnum)
### Second parameter for mapping
iterations2 = linspace(vmax2, vmin2, vnum2)
x,y = meshgrid(iterations, iterations2, indexing='ij')

avgs = {}
```

```

    devs = {}
    for i in selections:
        avgs[path.splitext(path.basename(i))[0].replace('@', '')] = []
        devs[path.splitext(path.basename(i))[0].replace('@', '')] = []

    # Loop over range of meta parameters
    for i in range(len(iterations)):
        vectavgs = {}
        vecdevs = {}
        for c in selections:

            vectavgs[path.splitext(path.basename(c))[0].replace('@', '')] = []

            vecdevs[path.splitext(path.basename(c))[0].replace('@', '')] = []
            for j in range(len(iterations2)):
                #initialize the particles
                args = {'high':100, 'low':-100, 't':0.0001, 'p':0,
                'iter_max':10000, 'pop_size':pop, 'dimensions':6, 'c1':x[i,j],
                'c2':y[i,j], 'neededQuality':100, 'sigma':0.001, 'wmax':0.9,
                'wmin':0.4, 'w':0.75}
                # Run the simulation
                perf = exp('@PSO-Annealing.py', selections, args, \
                measurementSampleSize=s)
                perfs = {}
                deviations = {}
                for func in perf.keys():
                    perfs[func] = [perf[func][0]['cpuTime'][0], \
                perf[func][0]['numCalls'][0], perf[func][0]['quality'][0], \
                perf[func][0]['convRate'][0]]

                    deviations[func] = [perf[func][0]['cpuTime'][1], \
                perf[func][0]['numCalls'][1], perf[func][0]['quality'][1], \
                perf[func][0]['convRate'][1]]

                    vectavgs[func].append(perfs[func][1])
                    vecdevs[func].append(deviations[func][1]/(s**0.5))

            for func in perf.keys():
                avgs[func].append(vectavgs[func])
                devs[func].append(vecdevs[func])

    # Generate 2D heatmaps
    for idx, name in enumerate(list(avgs.keys())):
        heatmap(x,y,array(avgs[name]),name)
        heatmap(x,y,array(devs[name]),name+'stdErrs')

    # Save the csv files
    with open("Data/"+ "Xs.csv", "w") as f:
        writer = csv.writer(f)
        writer.writerows(x)
        f.close()
    with open("Data/"+ "Ys.csv", "w") as f:
        writer = csv.writer(f)
        writer.writerows(y)

```

```
f.close()
for idx, name in enumerate(list(avgs.keys())):
    with open("Data/"+name+"Avgs.csv", "w") as f:
        writer = csv.writer(f)
        writer.writerow(avgs[name])
    f.close()
    with open("Data/"+name+"Devs.csv", "w") as f:
        writer = csv.writer(f)
        writer.writerow(devs[name])
    f.close()
```

APPENDIX IX.

PSO ALGORITHM

Algorithm-A IX-1 The PSO Algorithm

```
import math as m
import numpy as np
from numpy import random as r
import time
import copy as cp

r.seed(int(time.time()))
route = list()

#Ackley2 for rapid testing
def func(args):
    return -200 * m.e**(-0.02 * m.sqrt(args[0]**2 + args[1]**2))

def Quality(func, Sc,objective):
    func_output = func(Sc)
    if hasattr(func_output, '__iter__'):
        if (objective is not list):
            objective = [objective for x in range(len(func_output))]
            error = max([func_output[i]-objective[i] for i in range(len(func_output))])
        else:
            error = func_output - objective
    return 1/(abs(error)+0.00000001)

def move(p,args,best):
```

```

v = []
for d in range(args['dimensions']):
    v.append(args['w']*p['v'][d] + args['c1'] * r.random() * (p['best'][d] - p['params'][d]) \
        + args['c2'] * r.random() * (best['params'][d] - p['params'][d]))

return v

def main(func, S, args):

    high = args["upper"] if hasattr(args["upper"], '__iter__') else [args["upper"] for i in
range(args['dimensions'])]
    low = args["lower"] if hasattr(args["lower"], '__iter__') else [args["lower"] for i in
range(args['dimensions'])]
    vmax = [val/3 for val in high]
    vmin = [val/3 for val in low]
    t = args["t"]
    twprob = args["p"]

    route = list()
    #initialize the particles
    particles = []
    for i in range(args['pop_size']):
        p = {}
        p['params'] = np.array([r.uniform(low[i],high[i]) for i in range(args['dimensions'])])
        p['fitness'] = 0.0
        p['v'] = [0.1 for i in range(args['dimensions'])]
        particles.append(p)

    Best = cp.deepcopy(particles[0])
    Q = 0

```



```

route.append(cp.deepcopy(Best['params'][:]))
for iter in range(args['iter_max']):
    # Dynamic linear update of the inertia Weight factor
    #args['w'] = (args['iter_max'] - iter)*(args['wmax']-args['wmin'])/args['iter_max'] +
args['wmin']

    for p in particles:
        Q = Quality(func, p['params'],args['objs'])

        # Particle best
        if (Q > p['fitness']) or (i==0):
            p['fitness'] = Q
            p['best'] = p['params']

        # general best
        if Q > Best['fitness']:
            Best = cp.deepcopy(p)
            route.append(Best['params'][:])

        # move the particles to their next location
        v = move(p,args,Best)
        for x in range(args['dimensions']):
            if p['v'][x] > vmax[x]: p['v'][x] = vmax[x]
            if p['v'][x] < vmin[x]: p['v'][x] = vmin[x]
            p['params'][x]+=v[x]
            if p['params'][x] > high[x]: p['params'][x] = high[x]
            if p['params'][x] < low[x]: p['params'][x] = low[x]
        p['v'] = v

    if p['fitness'] > args['neededQuality']:
        break

```

```
return Quality(func, Best['params'],args['objs']), route
```

```
if __name__ == "__main__":
    # import plot stuff
    import matplotlib.pyplot as plt
    from mpl_toolkits import mplot3d
    import plotly
    import plotly.graph_objs as go

    obje = func
    args = {'upper':100, 'lower':-100, 't':100, 'p':0.2, 'objs':-200, 'neededQuality':100,
'iter_max':10000, 'pop_size':10, 'dimensions':2, 'c1':2, 'c2':2, 'sigma':0.001, 'wmax':0.9,
'wmin':0.4, 'w':0.75}
    a, path = main(obje, 100, args)
    print(a)
    # Plot 2D
    h1 = [i[0] for i in path]
    h2 = [i[1] for i in path]
    plt.plot(h1,h2)
    plt.show()
    # Plot 3D
    #h3 = [func(s) for s in path]
    #trace = go.Scatter3d( x=h1,y=h2,z=h3,mode='markers',marker={'size': 5,'opacity': 0.8,})
    #layout = go.Layout(margin={'l': 0, 'r': 0, 'b': 0, 't': 0})
    #data = [trace]
    #plot_figure = go.Figure(data=data, layout=layout)
    # Render the plot.
```

```
#plotly.offline.iplot(plot_figure)

#fig = plt.figure()
#ax = plt.axes(projection='3d')
#ax.scatter3D(h1, h2, h3, c=h3, cmap='Greens');
#plt.show()
```

APPENDIX X.

PACKAGE INIT FILE

Algorithm-A X-1 MDAF Package Init File

```
from MDAF.MDAF import representfunc
from MDAF.MDAF import exp
from MDAF.MDAF import plotfuncs
from MDAF.MDAF import model
from MDAF.MDAF import visualize2D
from MDAF.MDAF import installFlacco
```

LIST OF REFERENCES

- A-R. Hedar (n.d.). Trid Function. Retrieved August 7, 2020, from http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar_files/TestGO_files/Page2904.htm
- A-R. Hedar (n.d.). GLOBAL OPTIMIZATION TEST PROBLEMS. Retrieved July 31, 2020, from http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar_files/TestGO.htm
- Andrei, N. (2008). An unconstrained optimization test functions collection. *Advanced Modeling and Optimization*, issn: 1841-4311,10(1):2008, pp:147–161.
- Atlassian. (n.d.). A brief overview of Confluence. Atlassian. Retrieved November 29, 2020, from <https://www.atlassian.com/software/confluence/guides/get-started/confluence-overview>
- Auger, A., Hansen, N. (2005). *Performance evaluation of an advanced local search evolutionary algorithm*. IEEE Congress on Evolutionary Computation, September 2-5, Edinburgh, UK, pp:1777–1784. <https://doi.org/10.1109/CEC.2005.1554903>
- Bartz-Belestrin, T. (2003). Experimental Analysis of Evolution Strategies – Overview and Comprehensive Introduction, University of Dortmund, Germany, 36p.
- Basili, V. R., Selby, R. W., & Hutchens, D. H. (1986). Experimentation in software engineering. *IEEE Transactions on Software Engineering*, SE-12(7), 733–743. <https://doi.org/10.1109/TSE.1986.6312975>
- Bertrand Meyer. (1997). *Object-Oriented Software Construction, Second Edition*, Prentice Hall, 1296 p., <https://archive.eiffel.com/doc/oosc/>
- Bianchi, L., Dorigo, M., Gambardella, L. M., & Gutjahr, W. J. (2009). A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing*, 8(2), 239–287. <https://doi.org/10.1007/s11047-008-9098-4>
- Birattari, M., Stützle, T. et al. (2002). A racing algorithm for configuring metaheuristics. *Proceedings of the 4th annual conference on Genetic and Evolutionary Computation*, New-York, USA, pp:11–18.
- Borges, R. M., & Mota, A. C. (2007). Integrating UML and Formal Methods. *Electronic Notes in Theoretical Computer Science*, 184, 97–112. <https://doi.org/10.1016/j.entcs.2007.03.017>

- Bourque, P., & Côté, V. (1991). An experiment in software sizing with structured analysis metrics. *Journal of Systems and Software*, 15(2), 159–172.
[https://doi.org/10.1016/0164-1212\(91\)90053-9](https://doi.org/10.1016/0164-1212(91)90053-9)
- Brownlee, J. (2007). A Note on Research Methodology and Benchmarking Optimization Algorithms, Complex Intelligent Systems Laboratory, Faculty of Information and Communication Technology, Swinburne University.
- Bukin Function N. 6*. (n.d.), Surjanovic, S. and Bingham D., Simon Fraser University, Retrieved September 7, 2021, from <https://www.sfu.ca/~ssurjano/bukin6.html>
- Citation Report*. (2020). Web of Science.
https://apps.webofknowledge.com/CitationReport.do?action=home&product=WOS&search_mode=CitationReport&cr_pqid=5&qid=5&isCRHidden=&SID=7EOfvJq9WTtEVdtZTx4
- Coy, S. P., Golden, B. L., Runger, G. C., & Wasil, E. A. (2001). Using Experimental Design to Find Effective Parameter Settings for Heuristics. *Journal of Heuristics*, 7(1), 77–97. <https://doi.org/10.1023/A:1026569813391>
- Di Gaspero, L., and Schaerf, A. (2002). Writing Local Search Algorithms Using Easylocal++. In S. Voß & D. L. Woodruff (Eds.), *Optimization Software Class Libraries* (pp. 155–175). Springer US. https://doi.org/10.1007/0-306-48126-X_5
- Diego Andrés Alvarez Marín. (2010). *Rastrigin Function*, Simon Fraser University, <https://www.sfu.ca/~ssurjano/rastr.html>
- Dominique Orban. (2002). *CUTEr Website*. <http://www.cuter.rl.ac.uk/problems.html>
- Durkin, T. (1997). What the Media Couldn't Tell You About Mars Pathfinder. 3.
- Eiben, A. E., & Jelasity, M. (2002). A critical note on experimental research methodology in EC. *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, 1, 582–587. <https://doi.org/10.1109/CEC.2002.1006991>
- Forecast. (n.d.). *Help Center | The Basics*. Retrieved November 29, 2020, from <https://help.forecast.it/the-basics>
- Francois, O., & Lavergne, C. (2001). Design of evolutionary algorithms-A statistical perspective. *IEEE Transactions on Evolutionary Computation*, 5(2), 129–148.
<https://doi.org/10.1109/4235.918434>
- Gagnon, I., April, A., & Abran, A. (2020). A critical analysis of the bat algorithm. *Engineering Reports*, n/a(n/a), e12212. <https://doi.org/10.1002/eng2.12212>
- Gagnon Iannick. (2020). A Proposed Framework for the Design and Analysis of Metaheuristics (Unpublished doctoral dissertation). École de technologie supérieure.

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. 431.
- Gandomi, A. H., & Yang, X.-S. (2011). Benchmark Problems in Structural Optimization. In S. Koziel & X.-S. Yang (Eds.), *Computational Optimization, Methods and Algorithms* (pp. 259–281). Springer. https://doi.org/10.1007/978-3-642-20859-1_12
- GEATbx—Genetic and Evolutionary Algorithms Toolbox in Matlab—Main Page. (n.d.). Retrieved July 31, 2020, from <http://www.geatbx.com/>
- GLOBAL World—GLOBALLib. (n.d.). Retrieved July 31, 2020, from <http://www.gamsworld.org/global/globallib.htm>
- Gould, N. I. M., Orban, D., & Toint, P. L. (2015). CUTEst: A Constrained and Unconstrained Testing Environment with safe threads for mathematical optimization. *Computational Optimization and Applications*, 60(3), 545–557. <https://doi.org/10.1007/s10589-014-9687-3>
- Grady Booch, Robert Maksimchuk, Michael Engle, Jim Conallen, Kelli Houston, & Bobbi Young. (2007). *Object-Oriented Analysis and Design with Applications*. <https://www.pearson.com/us/higher-education/product/Booch-Object-Oriented-Analysis-and-Design-with-Applications-3rd-Edition/9780132797443.html>
- Hewitt, P. (2014). BUSINESS BENEFITS OF EFFECTIVE REQUIREMENTS MANAGEMENT. 11.
- Hooker, J. (1995). Testing Heuristics: We Have It All Wrong. *Journal of Heuristics* 1(1): 33–42. *Journal of Heuristics*, 1, 33–42. <https://doi.org/10.1007/BF02430364>
- Hussain, K., Salleh, M. N. M., Cheng, S., & Naseem, R. (2017). Common Benchmark Functions for Metaheuristic Evaluation: A Review. *JOIV : International Journal on Informatics Visualization*, 1(4–2), 218–223. <https://doi.org/10.30630/joiv.1.4-2.65>
- Jamil, M., & Yang, X.-S. (2013). A Literature Survey of Benchmark Functions For Global Optimization Problems. *International Journal of Mathematical Modelling and Numerical Optimisation*, 4(2), 150. <https://doi.org/10.1504/IJMMNO.2013.055204>
- Kaj Madsen—Head of Department•DTU Informatics. (n.d.). Retrieved July 31, 2020, from http://www2.imm.dtu.dk/~kajm/Test_ex_forms/test_ex.html
- kerschke. (2021). flacco: Feature-Based Landscape Analysis of Continuous and Constrained Optimization Problems [R]. <https://github.com/kerschke/flacco> (Original work published 2015)

- Kerschke, P., & Trautmann, H. (2016). The R-Package FLACCO for exploratory landscape analysis with applications to multi-objective optimization problems. *2016 IEEE Congress on Evolutionary Computation (CEC)*, 5262–5269. <https://doi.org/10.1109/CEC.2016.7748359>
- Koppen, M., Wolpert, D. H., & Macready, W. G. (2001). Remarks on a recent paper on the “no free lunch” theorems. *IEEE Transactions on Evolutionary Computation*, 5(3), 295–296. <https://doi.org/10.1109/4235.930318>
- Kruchten, P. (1995). The 4+1 View Model of Architecture. *IEEE Software*, 12, 45–50. <https://doi.org/10.1109/52.469759>
- Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical Debt: From Metaphor to Theory and Practice. *IEEE Software*, 29(6), 18–21. <https://doi.org/10.1109/MS.2012.167>
- Li, X., Engelbrecht, A., & Epitropakis, M. G. (2013). Benchmark Functions for CEC’2013 Special Session and Competition on Niching Methods for Multimodal Function Optimization. 10.
- Lim, W. J., Jambek, A. B., & Neoh, S. C. (2015). Kursawe and ZDT functions optimization using hybrid micro genetic algorithm (HMGA). *Soft Computing*, 19(12), 3571–3580. <https://doi.org/10.1007/s00500-015-1767-5>
- Lu, W., Fu, D., Kong, X., Huang, Z., Hwang, M., Zhu, Y., Chen, L., Jiang, K., Li, X., Wu, Y., Li, J., Yuan, Y., & Ding, K. (2020). FOLFOX treatment response prediction in metastatic or recurrent colorectal cancer patients via machine learning algorithms. *Cancer Medicine*, 9(4), 1419–1429. <https://doi.org/10.1002/cam4.2786>
- Lukasiewicz, M., Glaß, M., Reimann, F., & Teich, J. (2011). Opt4J: A modular framework for meta-heuristic optimization. *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation - GECCO '11*, 1723. <https://doi.org/10.1145/2001576.2001808>
- Luke, S. (2013). Essentials of metaheuristics: A set of undergraduate lecture notes (Second edition, online version 2.0). lulu.com.
- Mendes, S. P., Molina, G., Vega-Rodriguez, M. A., Gomez-Pulido, J. A., Saez, Y., Miranda, G., Segura, C., Alba, E., Isasi, P., Leon, C., & Sanchez-Perez, J. M. (2009). Benchmarking a Wide Spectrum of Metaheuristic Techniques for the Radio Network Design Problem. *IEEE Transactions on Evolutionary Computation*, 13(5), 1133–1150. <https://doi.org/10.1109/TEVC.2009.2023448>
- Meneghini, I. R., Alves, M. A., Gaspar-Cunha, A., & Guimarães, F. G. (2020). Scalable and Customizable Benchmark Problems for Many-Objective Optimization. *Applied Soft Computing*, 90, 106139. <https://doi.org/10.1016/j.asoc.2020.106139>

- Mishra, S. K. (2006). Performance of the Barter, the Differential Evolution and the Simulated Annealing Methods of Global Optimization on Some New and Some Old Test Functions (SSRN Scholarly Paper ID 941630). Social Science Research Network. <https://doi.org/10.2139/ssrn.941630>
- MOEA Framework, a Java library for multiobjective evolutionary algorithms. (n.d.). Retrieved December 30, 2021, from <http://moeaframework.org/>
- Moon, S. J., Hwang, J., Kana, R., Torous, J., & Kim, J. W. (2019). Accuracy of Machine Learning Algorithms for the Diagnosis of Autism Spectrum Disorder: Systematic Review and Meta-Analysis of Brain Magnetic Resonance Imaging Studies. *Jmir Mental Health*, 6(12), e14108. <https://doi.org/10.2196/14108>
- Nannen, V. (2006). A Method for Parameter Calibration and Relevance Estimation in Evolutionary Algorithms. *Genetic and Evolutionary Computation Conference, GECCO 2006, Proceedings*, 8–12.
- Opt4J. (2020). Opt4J - A Modular Framework for Meta-Heuristic Optimization. <http://opt4j.sourceforge.net/>
- Peer, E. S., Engelbrecht, A. P., & van den Bergh, F. (2003). CIRG@UP OptiBench: A statistically sound framework for benchmarking optimisation algorithms. *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, 4, 2386-2392 Vol.4. <https://doi.org/10.1109/CEC.2003.1299386>
- Pickle—Python object serialization—Python 3.10.1 documentation. (n.d.). Retrieved December 27, 2021, from <https://docs.python.org/3/library/pickle.html>
- Russell Eberhart, Yuhui Shi, & James Kennedy. (2001). *Swarm Intelligence—1st Edition*. <https://www.elsevier.com/books/swarm-intelligence/eberhart/978-1-55860-595-4>
- Sala, R., & Müller, R. (2020). Benchmarking for Metaheuristic Black-Box Optimization: Perspectives and Open Challenges. *ArXiv:2007.00541 [Cs, Math]*. <http://arxiv.org/abs/2007.00541>
- Scott, E. O., & Luke, S. (2019). ECJ at 20: Toward a general metaheuristics toolkit. *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 1391–1398. <https://doi.org/10.1145/3319619.3326865>
- Sean Luke, Eric O. Scott, Liviu Panait, Gabriel Balan, & Sean Paus. (2020). *ECJ*. <https://cs.gmu.edu/~eclab/projects/ecj/>
- Sörensen, K., & Glover, F. (2013). *A History of Metaheuristics* (pp. 960–970). https://doi.org/10.1007/978-1-4419-1153-7_1167

- Tanabe, R. (2021). Towards Exploratory Landscape Analysis for Large-scale Optimization: A Dimensionality Reduction Framework. *Proceedings of the Genetic and Evolutionary Computation Conference*, 546–555.
<https://doi.org/10.1145/3449639.3459300>
- The COCONUT Benchmark*. (n.d.). Retrieved July 31, 2020, from
<https://www.mat.univie.ac.at/~neum/glopt/coconut/benchmark.html>
- Tsang, K. K. T. (2018). Basin of Attraction as a measure of robustness of an optimization algorithm. *2018 14th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, 133–137.
<https://doi.org/10.1109/FSKD.2018.8686850>
- Web of Science Core Collections*. (2020). Web of Science.
https://wcs.webofknowledge.com/RA/analyze.do?product=WOS&SID=7EOfvJq9WTtEVdtZTx4&field=TASCA_JCRCategories_JCRCategories_en&yearSort=false
- Zdravko Botev, Dirk Kroese, Thomas Taimre, Jenny Liu, & Sho Nariyai. (2004). *Main*. The Cross-Entropy Toolbox. <https://www.maths.uq.edu.au/CEToolBox/>
- Zhang, P., Wu, H.-N., Chen, R.-P., & Chan, T. H. T. (2020). Hybrid meta-heuristic and machine learning algorithms for tunneling-induced settlement prediction: A comparative study. *Tunnelling and Underground Space Technology*, 99, 103383.
<https://doi.org/10.1016/j.tust.2020.103383>