Multi-Objective Tabu Search 2: First Technical Report

Christos Tsotskas,
Timoleon Kipouros,
Mark Anthony Savill

COA Report No 1
June 2015

Propulsion Engineering Centre
School of Aerospace, Transport and Manufacturing
Cranfield University
Cranfield
Bedford MK43 0AL
England

COA Report No 1
June 2015

Multi-Objective Tabu Search 2: First Technical Report

Christos Tsotskas, Timoleon Kipouros, Anthony Mark Savill
Propulsion Engineering Centre
School of Aerospace, Transport and Manufacturing
Cranfield University
Cranfield
Bedford MK43 0AL
ISBN 978-1-907413-34-6

contact e-mails:
c.tsotskas@cranfield.ac.uk
t.kipouros@cranfield.ac.uk
mark.savill@cranfield.ac.uk

Abstract

The purpose of this document is to describe Multi-Objective Tabu Search 2 (MOTS2), which is a native mutli-objective optimiser. It has been developed to tackle a variety of real-world problems of engineering interest. The design and implementation are presented, followed by verification, validation and user instructions. At a glance, it involves introduction to the algorithm, explains configuration settings and structure, and results interpretation. Then, the optimiser is tested against a series of mathematical test functions in order to verify its functionality. The main goal is to demonstrate and assess the performance and applicability of the optimiser. The next step is to use MOTS2 on a real-world case, where the performance of optimising a 2D airfoil is validated and illustrated.

Contents

List of Figures

List of Tables

1 Introduction

This document serves as an accompanying resource of the new multi-objective optimisation algorithm, called Multi-Objective Tabu Search 2 (MOTS2), which is a multi-objective variant of the original Tabu Search [13]. It can be considered as a combined extension of numerical analysis tools and artificial intelligence optimisation methods and techniques[19]. The main purpose is to introduce and describe MOTS2 without compromising in understanding and technical detail. Throughout this document, the terminology from [4] is used. Furthermore, all the files, directories and options appear in bold, a list of which is presented in the Appendix, below. Unless it is specified explicitly, the order of appearence of components in the files corresponds to the actual order of definition, within the optimiser, both for decision variables and objective function values.

Tabu search methods belong to the class metaheuristics optimisers[19]. They perform stochastic and local search optimisers or, alternatively, intensive local search. Originally, they were designed to manage heuristics of hill climbing, but they were adopted to manage heuristics of neighbourhood exploration. In fact, they can be considered as a strategy that controls a collection of embedded heuristic techniques. The former family of optimisers first introduced the concept of memory in metaheuristics, where the Reactive and Parallel derivatives emerged[3, 2, 12]. This algorithm is based on the single-objective Tabu Search (TS), introduced in [10, 5], and its multi-objective variant, as described in [13], where a detailed explanation can be found in. Moreover, MOTS2 includes the improvements (local search enhancements for Diversification Move) discussed in [15] and, given any parallel framework, it can operate in parallel mode saving elapsed time.

Additionally, integrated memories are manipulated and designed as databases, which makes the implementation very flexible and sustainable. Also, a new method called "kick" was added (see A.2.18 and A.2.19).

The structure of the document follows. The first part documents the development phase of the optimiser, where its structure and important routines are described. The third chapter explains how to perform a quick-run with MOTS2 by using a minimal combination of settings, whereas the following chapter explains in greater detail how to fully exploit all the features provided. The latter chapter describes the structure of configuration settings and input files, where every entry/entity is described along with a simple example. Considering that configuration settings are appropriate, the fifth chapter illustrates how to perform multi-objective optimisation by using MOTS2. The next chapter informs the user how to interpret and understand the generated output of the optimiser. The verification phase of MOTS2 is presented in Section B.1. Two families of test functions for two and three objectives are used, where MOTS2 delivered satisfactory results. Finally, the effectiveness of the optimiser is demonstrated on a real-world case, where MOTS2 is applied on a case of airfoil shape optimisation with two objectives.

2 The implementation of MOTS2

MOTS2 was implemented in two stages; First the memories were created, as the core data structure of the algorithm. Then the algorithmic structure was built around them. Since the concept of memories is so important, it is sensible to employ object oriented techniques. Because of the diversity of cases the optimiser will be called on, the optimiser is implemented in C++ programming language, which was selected for structural, linking and interfacing purposes. The code is portable, cross-platform and can be delpoyed on any computational architecture.

2.1 Describing the different types of memories

This section describes the concept of hierarchical memories. In total, there are 5 types of memories within the optimiser, each with slightly different purpose. These are called the Short Term Memory (STM), Medium Term Memory (MTM), Long Term Memory (LTM), Intensification Memory (IM) and HISTORY containers, or memory banks. All memorie share two common features; duplicate decision variables points are not allowed, but identical objective function values might exist, and their attributes are user-defined variables and do not change as the optimisation progresses.

The first, STM, is the collection of Tabu points. These are black-listed points that the optimiser is not allowed to use as long as they remain inside the memory. The size of STM is selected before the optimisation starts and remains fixed until the end. In principle, it implements a stack data structure and contains information about the decision variables, only. During every iteration, a new point is pushed into to the top of the memory and, if the memory is full, the last point is poped. So, the newly points are located at the top, and the elder is at the bottom.

Three types of memories, MTM, IM and HISTORY, have identical implementation, but they are populated differently. They all implement a dictionary (hash table) data structure, which holds a number of entries. Each entry is defined as an assembly of decision variables and their corresponding objectives. Moreover, all the entries are listed in asending order, based on the decision variables, only. The collection of the optimal non-dominated points, which are usually called Pareto Front (PF) points is stored in MTM. This memory holds the final output of the optimiser, which is a trade-off among the selected objectives and presents their interplay. All the points for the Intensification Move (see below) live in IM, which is used for the local-search features. Finally, HISTORY logs all the evaluated points (both feasible and not). This gives the big picture of the optimisation search, which is frequently combined with data mining techniques so as to extract information that will guide the search further. In addition HISTORY is used for post-processing and other purposes.

The last type of memories, LTM, is used for the global-search features. This holds the visiting frequency of various regions of the design space. Each variable range is equally split into a number of regions, whose frequency is monitored and is used under certain occasions (see Diversification Move below). The number of regions is set beforehand and remains the same until the end. This simply informs the optimiser (and the user) which areas are the most and least frequently visited and forces the optimiser to explore the most unknown regions.

The different types of memories and their relations, with regard to an example 2d objective space, are depicted in Figure1. All these memories are used to assist critical decisions during the optimisation process.

2.2 Algorithmic Structure

The optimiser depends on memories and performs a search by combining a systematic local search along with stochastic elements so as to intelligently search the entire design space. The flow of the aforementioned procedures is shown in Figure 2. The search starts from Base Point (BP) and three memory containers (STM, MTM, HISTORY). All these are used on every iteration, whereas the other parts are called when special conditions are met. It starts with blank memories, and as the optimisation search progress, it gradually builds a knowledge base about the design space and the objective space. Then, the search is guided from the performance of the decision vari-

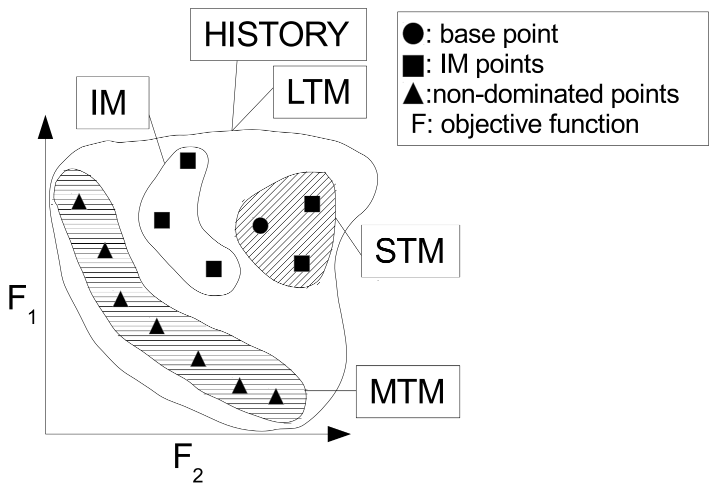


Figure 1: Tabu Search memories

ables point in the objective space so as to discover even better performance. All the stages are guided by the BP. Around the BP, the adjacent candidate design points are investigated and evaluated. Then, the corresponding objective values are sorted according to domination criteria of multi-objective optimisation [9] and the appropriate memories for the current iteration and the next BP is resolved. The previous BP and all the recently generated points are inserted into the appropriate memory containers. Hence, MOTS2 progresses effectively through the objective space while not wasting precious computational time.

In general, the design space is explored in a stochastic way, while recently visited points (called Tabu points) are avoided so as to guarantee more exploitation of the unknown design space. More importantly, this type of optimisers explicitly use memory (and history to some broader extent) throughout the optimisation search[1]. This can be particularly beneficial in terms of computational budget when the number of iterations increases and/or when direct search[2] is the only possible way. In fact, the local search scheme - in

[1]initially this seems complicated and taking advantage of it is not very straight forward

[2]this class of optimisers considers the evaluation of objective functions as black-box procedure

this implementation; Hooke and Jeeves [11], which is particularly efficient for continuous parameters - is combined with stochastic elements and other enhancements.

The optimiser also keeps track of statistics during the process, which direct the search according to the discovered landscape of the design space. At the top level, the optimiser employs a mechanism for local and global search. Thus, it could be considered a hybrid, since it expands the original definition[6]. In practice, the local search part is performed more frequently than its global counterpart. However, the global part can be considered as an enhancement to cover more general cases. The statistics determine the progress of the optimiser by activating certain behaviour when special conditions are met; they are mostly used to detect design points around the current search point, within relatively short distance, whereas the search mechanisms attempt to discover good design points in the entire design space. Consequently, the functionality of MOTS2, as depicted in Figure 2, results in better performance throughout the optimisation process.

Aggregated information will be used in future steps to guide the search, when certain conditions are triggered. This procedure keeps repeating until the stopping criteria are met. Depending on the nature of the application, these are usually the elapsed time, the number of evaluations, the number of consecutive failures to find a better point, number of iterations or a combination of them. During every iteration a fraction of the flow of the algorithm is executed every time, and the rest runs when certain conditions are met. The core of the optimiser is the Update Memories, Hooke and Jeeves-, Intensify- and Reduce-Move, the remaining parts are algorithmic enhancements, which speed-up the search.

In the simplest approach, the optimisation starts off from a datum design point[3], which is a combination of parameters of well-known performance. Following the operation of local-search optimisers, the selection of the datum design is the most crucial part, before actually launching the optimisation process, as this could potentially trap the algorithm at an early stage. Then, a couple of new points are generated and evaluated. One of them will be the BP, throughout the current iteration loop, and the rest are accordingly forwarded to the memories. Thereafter, the optimisation keeps iteratively going on.

[3]this should be a valid point

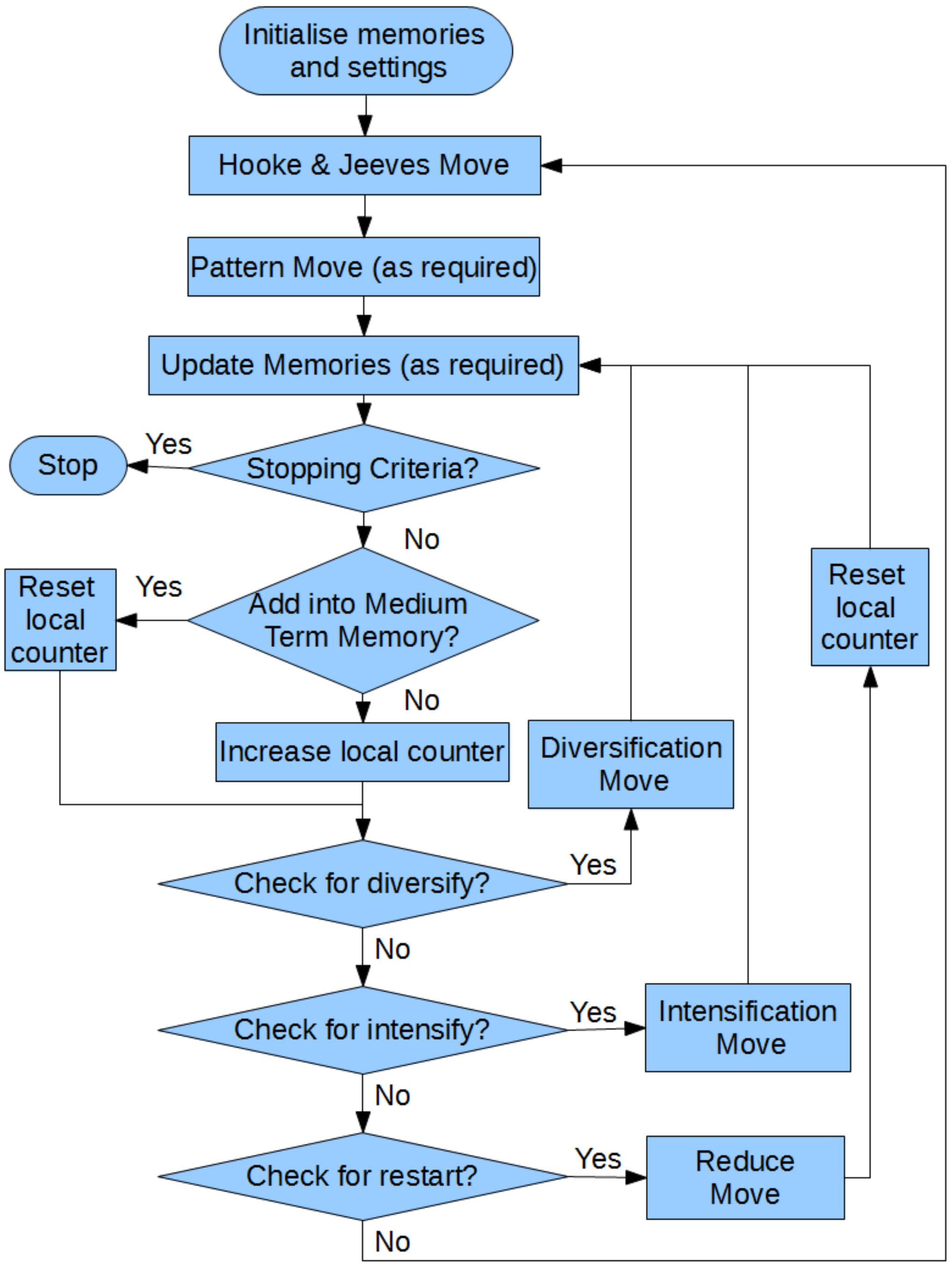


Figure 2: MOTS2 algorithm flow diagram

2.2.1 Frequently Performed Moves

The following parts take place on a regular basis:

Hooke and Jeeves Move is the most important part of the local search scheme as it occurs on every iteration and requires plenty objective function evaluations, which can be expensive. This is the most time-consuming part of the optimisation process. Starting from the BP, a couple of valid and non-tabu points are generated by combining the current BP and the current Search Step (SS, see A.2.4) as illustrated in Figure 3. In theory, up to twice the population size of the design parameters (see **nVar** below) can be generated, but since some of them could either belong to STM or be invalid, they are excluded from the following stages of Hook and Jeeves Move. Then, a few of the recently created points are randomly selected and evaluated in order to save some computational budget (see **n_sample** below) and added into the appropriate memory banks. These points are within the close vicinity of the BP and this is the local search phase of the optimiser. The **n_sample** can be up to 2***nVar**, but since this will use considerable computational budget, a lesser value should be selected, but not too low because it will compromise in local exploration. Among the recently evaluated points, one of them will be randomly selected as the BP and the search continues.

Pattern Move: This is just an enhancement of the Hooke and Jeeves Move where the next BP will be quickly resolved. Whenever Hooke and Jeeves Move takes place for second time, the following BP is generated by combining information from the last two BPs. It takes place every other iteration (once in every two iterations). In this way, the search may be accelerated along known downhill directions. In fact, it calculates the gradient direction between the last and penultimate BP and applies the same change to generate a new point that will be evaluated.

Update Memories: At the end of every iteration the newly resolved BP is inserted into the base memory bank, STM, MTM, LTM, IM and HISTORY (should it fulfil their corresponding conditions). New points are popped and pushed into STM following the principles of data stack. All the points within IM are filtered based on Pareto dominance, whereas

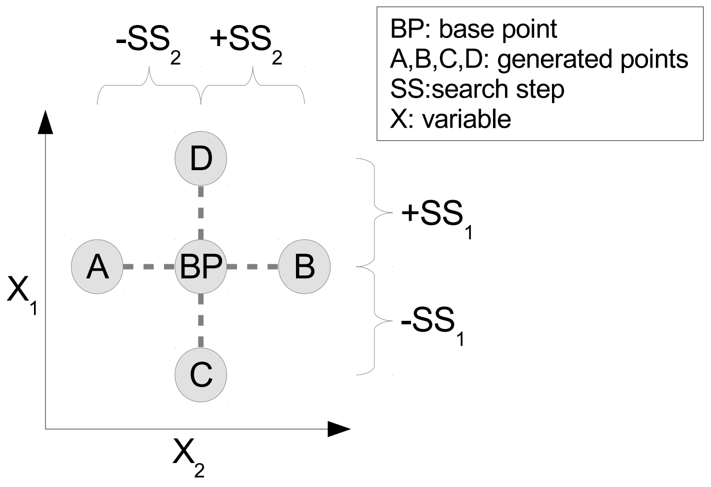


Figure 3: Hooke and Jeeves Move in the design space

HISTORY keeps track of all the records. The dominated points from MTM are also filtered out.

During the execution of the algorithm, the memory containers are enriched with information that will be exploited on later stages. Therefore, a zero-knowledge search starts and as the optimiser runs, it learns about the intrinsic features of the design space from the containers iteration-by-iteration. This results in a knowledge base and according to the principles of artificial intelligence, this is the best method of a heuristic search to be performed.

2.2.2 Conditionally Performed Moves

Besides the Moves mentioned above, a number of complementary Moves is carried out when certain conditions are met. These Moves can escape MOTS2 from local minima and perform local and global search, and step refinement, respectively. Although the frequency is defined by the user, in order to be consistent with the original definition of Tabu Search, their numerical value should increase in the order of appearance.

Intensify Move: By definition, contrary to single-objective optimisation, during multi-objective optimisation several points form the Pareto-

Front. However, during every iteration, only one of them might be the BP. Therefore, the remaining points that dominate the current trade-off, but have not been selected as BPs, are stored into IM. Whenever the search cannot discover any new nor non-tabu point, another point from that back-up container is selected randomly as the following BP. Therefore, the search returns back to the vicinity of the most promising points discovered so far and picks-up the search thereafter. This should be the most frequently performed move.

Diversify Move: Instead of finding a better point, within a short range, a new non-tabu point is randomly generated from least explored region of the design space. Here, the information stored in LTM is utilised in order to generate the new point. The design space is explored as equally as possible. This is the global search phase of the optimiser and its frequency depends on the problem.

Restart Move: Whenever the search fails to discover a new good point with the current SS for a large number of iterations, a new BP is randomly resolved similarly with Diversify Move. Then, the SS is refined accordingly. This should be the rarest performed move because SS is only reduced. By performing this move the optimiser is gradually narrowing down the exploration range and focuses on certain regions, where it is expected to find a better optimum.

2.3 Constraints and Objectives Handling

Throughout this document only minimisation of objectives will be considered, due to the duality between maximisation and minimisation. So, for maximisation problems, one simply needs to reformulate the original problem in a way that the quantities to be maximised should be multiplied by (-1). Hence, minimising the negative of any objective equals to maximising the same objective. In addition, any number of objectives (above 2) can be minimised. It is worthy to mention that multi-objective optimisers can also perform single objective optimisation just by using the same numerical value for all the objectives.

Furthermore, MOT2 can deal with both soft and hard constraints. The former have to be programmed within the source code and the optimiser needs to be compiled again. Although the range of variability also belongs to the soft constraints, the range of each parameter can be set individually without the need for compiling (see A.3). Whenever there is a violation of constraints, the optimisation automatically assigns a very large penalty value (see A.5) to the set of variables that triggered the constraint, without running the respective variables set over the actual evaluation tool/method. The hard constraints are related to the objective function evaluation tool/method and are set independently of the optimiser[4]. Since the optimiser performs minimisation of the objectives, it is highly advisable the evaluation tool to assign a different penalty value at the same order of magnitude as for the soft constraints. This will prevent any confusion to the optimiser. Hence, the optimiser can deal with any type of optimisation problem, both unconstrained and constrained that involves real parameters.

3 Unique features

This version of MOTS2 has a number of features that differentiate it from the original variation, which was presented in [13]. First, they all intend to make the structure of the optimiser more flexible, in order to link with external and internal tools and software. Then, they attempt to

- enhancements
- kick
- data-base representation, will facilitate the implentation of additional modules from the field of data mining.
-

References

[1] A. Auger, J. Bader, D. Brockhoff, and E. Zitzler. Theory of the Hypervolume Indicator: Optimal μ-Distributions and the Choice of the Reference Point. In *Foundations of Genetic Algorithms (FOGA 2009)*, FOGA '09, pages 87–102, New York, NY, USA, 2009. ACM. A.2.13

[4]The penalty values of the invalid designs should be of the same order of magnitude as the soft constraints. However, both should be set to a numerically large value.

[2] R. Battiti and G. Tecchiolli. The reactive tabu search. *ORSA Journal on Computing*, 6(2):126–140, 1994. http://joc.journal.informs.org/content/6/2/126.full.pdf+html. 1

[3] J. Brownlee. *Clever algorithms nature-inspired programming recipes*. Lulu Enterprises, 2011. 1

[4] C. A. Coello Coello, G. B. Lamont, and D. A. Veldhuizen. *Evolutionary algorithms for solving multi-objective problems*. Springer, New York, 2nd ed. edition, 2007. 1

[5] A. M. Connor and D. G. Tilley. A Tabu search method for the optimization of fluid power circuits. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, 212(5):373–381, 1998. 1

[6] K. Deb. *Multiobjective optimization using evolutionary algorithms*. Wiley, New York Chichester, 2001. 2.2

[7] M. Drela. Xfoil subsonic airfoil development system, April 7, 2008. B.2.1

[8] M. Drela. Xfoil - an analysis and design system for low reynolds number airfoils. June 1989. B.2.1

[9] M. Geilen, T. Basten, B. Theelen, and R. Otten. An algebra of pareto points. *Fundamenta Informaticae*, 78(1):35–74, 2007. 2.2

[10] F. Glover and M. Laguna. *Tabu search*. Kluwer Academic Publishers, 3rd printing edition, 1999. 1

[11] R. Hooke and T. A. Jeeves. Direct Search Solution of Numerical and Statistical Problems. *Journal of the ACM (JACM)*, 8(2):212–229, 1961. 2.2

[12] D. M. Jaeggi, C. Asselin-Miller, G. T. Parks, T. Kipouros, T. Bell, and J. Clarkson. *Multi-objective Parallel Tabu Search*, pages 732–741. Search. Springer, 2004. 1

[13] D. M. Jaeggi, G. T. Parks, T. Kipouros, and P. J. Clarkson. The development of a multi-objective tabu search algorithm for continuous optimisation problems. *European Journal of Operational Research*, 185(3):1192–1212, 2008. 1, 3, B.1.1, B.2.1

[14] A. Jameson and J. C. Vassberg. Computational fluid dynamics for aerodynamic design: Its current and future impact. In *39th AIAA Aerospace Sciences Meeting and Exhibit*, 8-11 January 2001. B.2.1

[15] T. Kipouros, D. M. Jaeggi, W. N. Dawes, G. T. Parks, A. M. Savill, and P. J. Clarkson. Biobjective Design Optimization for Axial Compressors Using Tabu Search. *AIAA Journal*, 46(3):701–711, 2008. 1

[16] J. Knowles, L. Thiele, and E. Zitzler. A tutorial on the performance assessment of stochastic multiobjective optimizers. Technical report, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH) Zurich, 2005. B.1.1

[17] M. López-Ibáñez, L. Paquete, and T. Stützle. *Exploratory Analysis of Stochastic Local Search Algorithms in Biobjective Optimization*, pages 209–222. Experimental Methods for the Analysis of Optimization Algorithms. Springer, Berlin, Germany, 2010. B.1.1, B.2.1

[18] O. Mersmann, H. Trautmann, B. Naujoks, and C. Weihs. Benchmarking evolutionary multiobjective optimization algorithms. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8, 2010. B.1

[19] S. Russell. *Artificial Intelligence : A Modern Approach*. Prentice Hall, Upper Saddle River, 3rd edition, 2010. 1

[20] T. W. Sederberg and S. R. Parry. Free-form deformation of solid geometric models. *SIGGRAPH Comput.Graph.*, 20(4):151–160, 1986. B.2.1

[21] P. R. Spalart and D. R. Bogue. The role of cfd in aerodynamics, off-design. *The Aeronautical Journal*, 107(1072):323–329, 2003. O: Anglais. B.2.1

[22] E. Zitzler, K. Deb, and L. Thiele. Comparison of Multiobjective Evolutionary Algorithms: Empirical Results. *Evolutionary Computation*, 8(2):173–195, 2000. B.1.1

A How to use MOTS2

All files are regular plain text files, where all the values are separated by one(or more) blank space or any other non-printable character.

A.1 Preparing MOTS2 before launching

MOTS2 has a number of user-defined and self-adjusted attributes.

A.2 configuration.txt (this file is mandatory)

The most important text file to prepare before running the algorithm is the configuration.txt. Its structure and the values on each line orchestrate the functionality the optimiser and define required files. Obviously, more control is provided if more files are used, as explained below, at the expense of additional user effort.

This is the only file that contains exactly 19 lines of arithmetic and non-arithmetic values. Each value should be in one line ONLY followed by new line (Enter key). This file contains the optimisation settings as follows: (herein the words exist for ease of understanding and should not be present in the actual file)

1	diversify	15
2	intensify	5
3	reduce	50
4	SS	0.1
5	SSRF	0.5
6	save_step	15
7	n_sample	6
8	nVar	10
9	nObj	2
10	n_of_loops	0
11	n_of_evaluations	60000
12	n_of_consecutive_improvements	0
13	assessment	HV
14	nRegions	4
15	STM_size	5
16	LogType	full
17	Starting point	1
18	maximum_improvements	1300
19	maximum_duplicates	1000

(a) explaining configuration.txt file, columns correspond to line number, description and value, respectively

```
15
5
50
0.1
0.5
15
6
10
2
0
60000
0
HV
4
5
full
1
1300
1000
```

(b) example configuration.txt (just the rightmost column)

Figure 4: configuration file

A.2.1 diversify

(positive integer number) defines the number of consecutive objective function evaluations (or simply evaluations) upon which no optimal point was found. When this limit is reached the code quasi-randomly generates a new point and assigns that as the new BP. This is related to the Diversify Move in 2.2.2.

A.2.2 intensify

(positive integer number) defines the number of consecutive evaluations upon which no optimal point was found. Whenever this limit is reached the code picks up randomly a point from IM as the following BP. This is related to the Intensify Move in 2.2.2.

A.2.3 reduce

(positive integer number) defines the number of consecutive evaluations upon which no optimal point was found. Whenever this limit is reached the code quasi-randomly generates a new point and assigns that as the new BP and reduces the step for each variable by SSRF (see below). This is related to the Reduce Move in 2.2.2.

A.2.4 SS

(two options [0 or any real number between 0 and 1.0]) represents the Search Step value for each variable as a percentage of the range between upper and lower bound for the respetive variable. If set to 0, then the user should provide the **start_step.txt** file, where individual step values for each variable should be contained. If the user specifies a fraction, then the same step refinement as a percentage of the range of each variable will be applied on each variable! for instance, if the range for design variable V is between 2 and 4, SS=0.1 means that the SS for the specific variable will be (4-2)*0.1=0.2.

A.2.5 SSRF

(real number between 0 and 1.0) represents the SS Retain Factor (SSRF), which is the percentage change of the initial SS value for each variable. Following the example above, SSRF=0.5 means that the SS will be 0.2*0.5=0.1. If 3 consecutive reductions occur this equals 0.2 * (0.5 * 0.5 * 0.5).

A.2.6 save_step

(positive integer number) defines the frequency the memory files will be updated during the search. Inside the **memories** directory all the files described in Subsection A.9.1 will be created (read for more details). The larger this value is, the quicker the execution. Usually, updating all the files takes a couple of seconds. So, if the turnaround time of the objective function evaluation is significantly larger, this value will not make any difference in terms of turn around time. All the files are stored under the **memories** directory. The values of **nVar** and **nObj** (see below) directly affect the functionality of the optimiser to store information about its progress. The larger they are set, the more time is required to save all the memories. This can delay the termination of the optimisation in cases of low complexity as the objective functions' evaluation time is shorter than the saving time (e.g. when the objective function evaluation takes a couple of milliseconds, such as benchmark functions). Hence, setting **save_step** to a large value is advisable when the evaluation time is quick.

A.2.7 n_sample

(positive integer number) During Hook and Jeeves Move (see 2.2.1), up to 2***nVar** (see below) new points are generated. In order to save computational time, this population is split in smaller sets and they are evaluated (in batches) one by one until certain criteria are met. So, this value represents in how many design points will be contained in each batch. If the division is not complete, the last batch will contain the modulo part of the 2 * **nVar** / **n_sample**. Obviously, **n_sample** should be larger than 1 and less than 2***nVar** (which means that every member of the population will be sampled from the first round) In practice, **n_sample**=2 * **nVar** should be used only in very complicated cases.

A.2.8 nVar

(positive integer number) number of elements that define the design vector. Currently, it is assumed that all the variables are real numbers. Properly setting the **SS** and **SSRF** values (see above) could turn the problem into mixed integer form, but this is not advisable.

A.2.9 nObj

(positive integer number) number of elements that define the objective vector

A.2.10 n_of_loops

(positive integer number) the search stops after the respective number of iterations. If set 0, the limit is considered infinite.

A.2.11 n_of_evaluations

(positive integer number) the search stops after the respective number of objective function evaluations. If set 0, the limit is considered infinite.

A.2.12 n_of_consecutive_improvements

(positive integer number) the search stops after the respective consecutive number of failed improvements of the PF. If set 0, the limit is considered infinite. Obviously, if all values (n_of_loops, n_of_evaluations, n_of_consecutive_improvements) are set 0, the search performs an endless loop and turns out to be a heuristic procedure.

A.2.13 assessment

(only HV) The quality of the discovered trade-off is assessed with a method that is based on HyperVolume indicator (HV)[1]. Currently there are no other methods implemented. So, leave this value unaltered.

A.2.14 nRegions

(positive integer number) represents the number of clusters the search domain will be divided into for EACH variable. The larger the value the more coarse the new point generation will be. For instance, if value V1 ranges between 2 and 3, nRegion=4 will produce clusters [2.0, 2.25] [2.25, 2.5] [2.5, 2.75] [2.75, 3]. Currently, the same number of nRegions is applied on each variable range.

A.2.15 STM_size

(positive integer number) defines how many points will be considered Tabu for every point generation. The larger the value the more aggressive the search will be.

A.2.16 LogType

(only full) it exports a number of files in **monitor_data** directory related to the search progress. Currently, there are no other options implemented.

A.2.17 starting point

(two options - binary number [0,1]) if starting point is set to 0 the optimiser will automatically generate a random starting point (within the permitted bounds for each variable separately). If the value is set to 1, then the optimiser will read from a datum starting point from the file **datum_design_vector.txt** which should reside in the root directory.

A.2.18 maximum_improvements

(positive integer number) matches with the option below

A.2.19 maximum_duplicates

(positive integer number) The two values above are particularly useful for cases which are dominated by multi-modality. **Maximum_improvements** indicate the amount of consecutive non-improvements of the discovered PF. Once this value is met, the optimiser refines **SS**. **Maximum_duplicates** represents the amount of permitted duplicates per objective value and is used similarly. These two features are used together in order to avoid situations when the search is trapped. The step refinement occurs either when the combination of { **maximum_duplicates** and 10% of **maximum_improvements** } OR { **maximum_improvements** } conditional statements are met, whichever comes first. Setting these two values very high will virtually "deactivate" their effect.

A.3 design_vector_ranges.txt (this file is mandatory)

specifies the minimum and the maximum allowed values for each variable, which is part of the design vector. The permitted range for each decision variable can vary. It should contain exactly 2***nVar** values in a fashion that the fist couple values correspond to the lower and upper bound of the first variable. For instance, the 3rd and 4th values define the range of the 2nd variable and so on.

A.4 reference_point.txt (this file is mandatory)

This file is related to the quality assessment of the optimisation process, which is a user-defined assessment method/quality indicator. In contrast to the other files, this one contains only objective function values of some arbitrary reference. It is combined with the PF points in order to consider improvements during the search. It should contain exactly **nObj** values. The first value corresponds to the first objective and so on.

A.5 failed_objective_vector.txt (this file is mandatory)

This file contains the penalty values which will be assigned to the invalid decision variables point by the optimiser when the designs are either invalid or violate problem's constraints. Typically, for a minimisation problem, the values are set to a relatively large number for each component. It is crucial to mention that the models (which are actually the objective function evaluation) should also provide a setting for penalty values (of the same order of magnitude) when the execution of the model is not successful. These two penalty settings, frequently called soft and hard penalty) should be different for ease of interpreting the results. It should contain exactly **nObj** values.

A.6 start_step.txt (whenever SS=0)

This file contains exactly **nVar** values. As expected, the i-th value corresponds to the i-th variable, respectively. In addition, this file is related with the value of the **SS** setting in the **configuration.txt**. Here, the user specifies explicitly the **SS** for each variable separately as a percentage of the range of each variable. It is worthy to note that the user is responsible for the consistency of the input values. It is good practice to perform sensitivity analysis before setting individual weights.

A.7 datum_design_vector.txt (whenever starting point=1)

This file is the most important for the optimisation process because of the inherent operation of local search optimisers. Whenever the starting point value of the configuration file equals 1, then the optimiser will read the **datum_design_vector.txt** file and will assign the initial point for the search. It should contain exactly **nVar** values.

A.8 Running MOTS2

Here, the difference between a framework and stand-alone application is arbitrary and there are many grey zones. In the simplest approach the framework defines the concept, whereas the application defines the fundamental functionality that end-users are concerned about. Contrary, an application performs specific, well-defined instructions. Since the framework optimises a case, in order to avoid confusion, MOTS2 will be considered as a framework throughout this document.

Although, by implementation, MOTS2 was developed as an optimiser (a single application), it's structure resembles a framework. However, it can be seen as a framework by itself, or it can be linked with other frameworks, too. Arguably, it started as an application but after a number of enhancements, where each one could be an application by itself, it seems to be more as a framework. In fact, since MOTS2 can serve as a skeleton for more complicated problems, the term framework is more appropriate.

For simplicity, the user can edit all the configuration files and just execute MOTS2. This is done by running **mots2.exe** from the console. All the internal applications/modules are activated/deactivated according to the aforementioned files. Immediately after the launch, all the configuration and input settings are displayed (mainly for revision/cross-check purposes) and the optimisation steps are performed. The end-user can interrupt the whole process at anytime just by terminating **mots2.exe** either from the terminal or the task manager. Following the **save_step** user's setting (see A.2.6), which can affect the turnaround time, the output data can be collected anytime. It is a good practice to save all the configuration, input and output files, before attempting to run another case.

MOTS2 can run in two modes:

Regular On the command line just typing **mots2.exe** will perform optimisation from scratch.

Resume On the command line, typing **mots2.exe -r** will resume optimisation from a certain stage, under the condition that the folder memories and the respective files exist.

Regarding restart, it is important to mention that the case setup, for instance number of objectives and variables, STM size,... etc should not be altered. In order to restart the optimisation the **memories** folder should be present within the current directory, the files **STM.txt**, **MTM.txt**, **HISTORY.txt**, **LTM.txt** and **IM.txt** should be within **memories** folder and the file **checkpoint.out** should exist within the **monitor_data** folder (see A.9.2). The first time MOTS2 runs, the **restart** file is created automatically under the **monitor_data** folder. Then, the optimiser will read through all the files, load them into the respective memories and will randomly choose one point from **MTM.txt** as the initial point.

The final trade-off file is **TS.txt** and will be generated/updated automatically throughout the process and will remain in the root folder when the optimisation finishes.

A.9 Interpreting the output

After running the optimisation, a number of files will be created under the **memories** and **monitor_data** directory. All data from STM, MTM, LTM, IM, HISTORY and some complementary data are stored within **memories**, whereas **monitor_data** are used to monitor the progress of the optimisation and keep track of statistics. Automatically generated files, ending in *.plt extension, for plotting purposes reside in each directory. All these files can be used both for live monitoring of the optimisation search and for post-processing the results.

A.9.1 Memories directory

All the files are related to the optimisation process. Each document is a regular ASCII file and it should be read row-by-row. There are two families of files; the regular and the numbered ones with the prefix “snap”. The former correspond to the current (most recently updated) versions of the **memories**. The latter are instances of an earlier stage of the optimisation progress and the number corresponds to the respective state[5], when the snapshot was taken. In other words, its a snapshot of #evaluations (e.g. **HISTORY_snap20041.txt**). Under this directory, the following 7 types of files will be created:

BASE.txt contains the BP for every iteration. The entries are listed in a reversed order, so that the most recent BP is at the top of the document. This document is comprised of the following columns :

- 1 integer for the iteration counter
- **nVar** elements for the variables
- **nObj** elements for the corresponding objectives
- 1 string that indicates in which region the BP belongs to (currently deactivated)
- 1 string that describes how the BP was assigned

STM.txt contains all the designs that are considered tabu (already visited and they cannot be selected as a BP for the current iteration)

- **nVar** elements for the variables

IM.txt contains all the designs that reside into the Intensification Memory

- **nVar** elements for the variables
- **nObj** elements for the corresponding objectives

MTM.txt contains all the pairs of decision variables and objective function values that belong to the PF discovered so far

- **nVar** elements for the variables

[5]The total number of objective function evaluations performed so far. Without loss of generality, it represents the computational budget invested and is an indication of the overall progress of the optimisation search.

- **nObj** elements for the corresponding objectives

MTMfrequencies.txt contains the multiplicity of the objectives of the PF. This file should be advised for cases with multi-modal objectives (there are many combinations of the design vectors that deliver exactly the same fitness and might trap the optimiser - such as ZDT4). In order to face multi-modality, properly setting the **maximum_improvements** and **maximum_duplicates** (see A.2.18 and A.2.19) will mitigate this problem.

- **nObj** elements for the corresponding objectives
- corresponding frequency

LTM.txt contains all the decision variables that reside into the Long Term Memory

- **nVar** elements for the variables

HISTORY.txt contains all the pairs of decision variables and objective function values so far (both feasible and infeasible) and their corresponding objectives.

- **nVar** elements for the variables
- **nObj** elements for the corresponding objectives

A.9.2 Monitor_data directory

All the files are related to the status of the optimiser, are created automatically and are regular ASCII files. These are mainly statistics which explain how the optimisation progresses. It is good practice to advise (and plot) these files for understanding the meaning of the configuration parameters and how these can affect the success of the optimisation process.

evals.out evaluations file

- iteration counter
- total number of designs evaluated (the infeasible designs are not considered)

- number of design violations (whenever it appears)

hypervolume.out (currently deactivated)

i_local.out local counter file

- iteration counter
- local counter value

im_size.out contains the total number of points into the Intensification Memory

- iteration counter
- IM size

intensify.out counts how many times the Intensification Move was used

- iteration counter
- number of the performed Intensification Moves

memory_status.txt checks the state of memories (this is the only file that is not read line-by-line) and it is used for debugging MOTS2

reduce.out counts how many times the Reduction Move was used

- iteration counter
- number of the performed Reduction Moves

step_size.out logs the current SS

- iteration counter
- **nVar** elements, each element corresponds to the current SS for the specific variable

update_memories.out (currently deactivated)

basePoint.out similar to BASE.txt but in proper order with fewer information(used for debugging)

- iteration counter
- **nVar** elements for the variables

checkpoint.out this file is used when MOTS2 is resumed and contains sequentially the following items:

- iteration counter
- **nVar** elements for the decision variables
- diversification counter
- intensification counter
- reduction counter
- i_local counter
- **nVar** elements for the SS for each variable

diversify.out counts how many times the Diversification Move was used

- iteration counter
- number of the performed Diversification Moves

quick.out contains an aggregated view of all the aforementioned information. This is particularly useful in Unix environment to monitor the progress live in a single terminal. Columns correspond to:

- iteration counter
- number of evaluations
- number of violations
- evaluations to iterations ratio
- MTM size
- i_local counter

- IM size
- diversification counter
- intensification counter
- reduction counter

B Verification and Validation of MOTS2

B.1 Verifying MOTS2

By implementation, MOTS2 belongs to the category of stochastic local search algorithms. Hence, it is not possible to predict the search path, since this is expected to be random and different every time one performs the optimisation. The optimiser makes decisions based on the information contained into the memories and from statistics it gathers. Therefore, the performance of the optimiser will be assessed by carrying out statistical analysis[18]. This is a standard and consistent way that can compare any optimiser under a common basis. The turnaround times are not considered as performance metric.

B.1.1 Benchmarking the family of ZDT functions for 2-objectives optimisation

More specifically, the methodology described in [16] was followed. Also, as suggested in [22], each optimisation scenario with test functions was executed 50 times. The combinations of configuration settings used for the verification phase are the same used in [13]. Although these are neither unique, nor optimal, they can satisfactorily achieve the target goals of delivering the target trade-off. It is common practice to test the family of ZDT[22] benchmark functions. Among them, ZDT4 test function is the most difficult to resolve for extremely multi-dimensional landscapes, as stated in [22]. First the collection of PFs for each case was used to produce the Empirical Attainment Function (EAF), applied on the data. The attainment surfaces are depicted in Figures 5 to 9, followed by the HV boxplots. The respective figures were generated by using the external plotting tool described in [17].

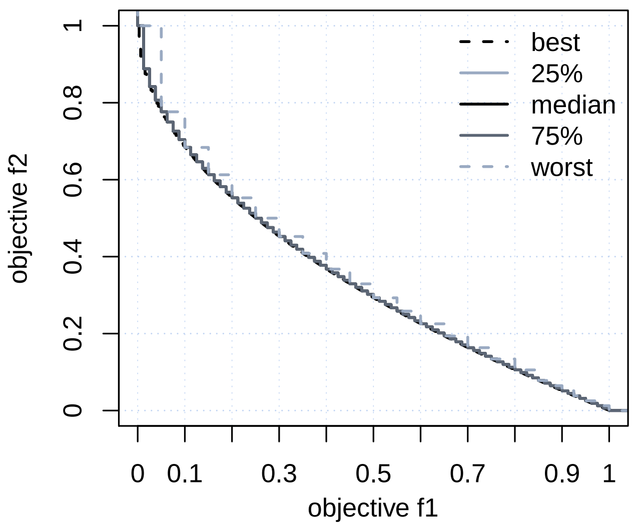


Figure 5: ZDT1 EAF

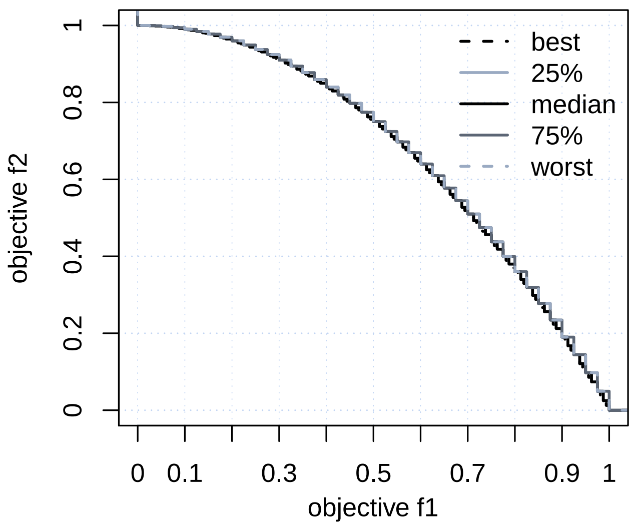


Figure 6: ZDT2 EAF

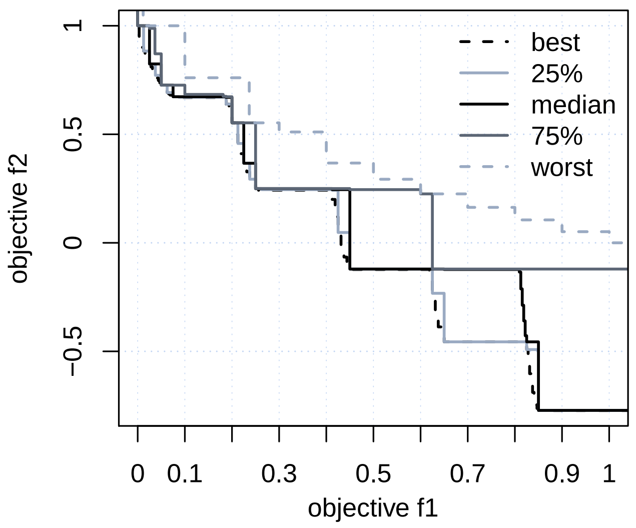


Figure 7: ZDT3 EAF

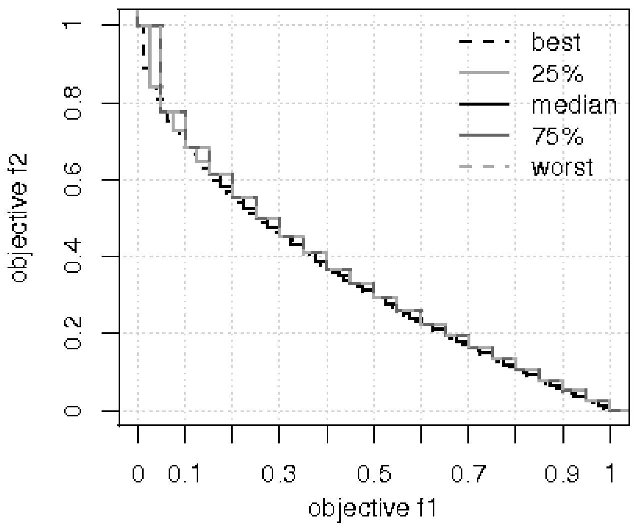


Figure 8: ZDT4 EAF

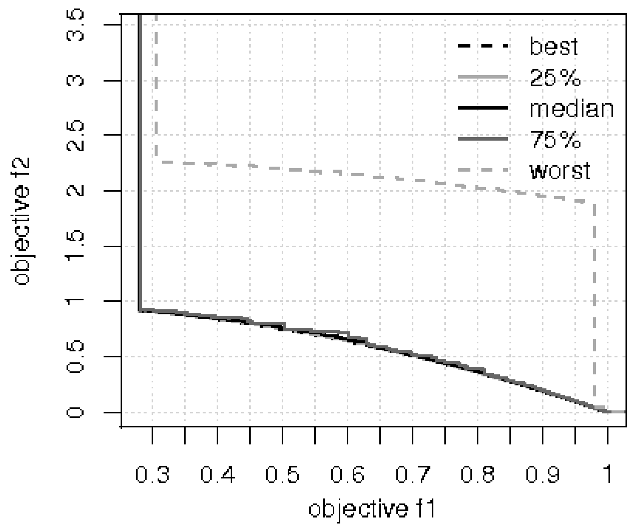


Figure 9: ZDT6 EAF

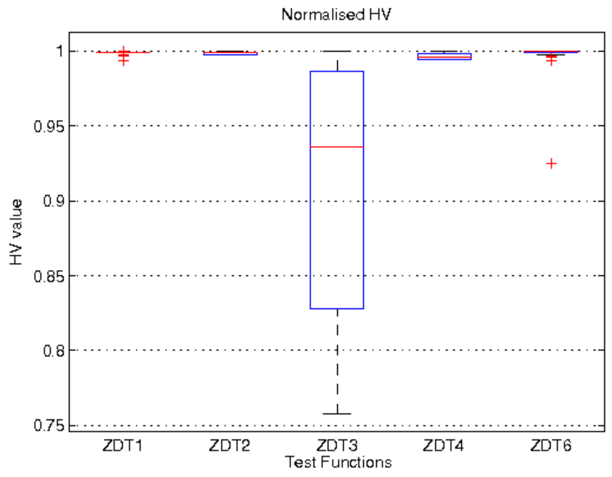


Figure 10: Normalised HV boxplots for ZDT Test Functions

B.2 Validating MOTS2

The test functions used in the previous chapter can be used for verifying the functionality of the optimiser but they do not reflect the needs of real-world applications. The former are simple and computationally cheap to evaluate, whereas the latter can be of arbitrary complexity and extremely costly to use. In addition, this high abstraction level can lead to severe loss of information about the nature of the original problem. So, it seems interesting to use MOTS2 on a real case and observe its behaviour and results.

This chapter presents the effectiveness of the optimiser when used on real-world applications. Currently, only one validation case is presented. As more cases emerge, these will be inserted in this Section.

B.2.1 Airfoil Shape Optimisation

A relatively simple scenario that proves the effectives of the optimiser in real-world application is the shape/profile optimisation of an airfoil. This is an aerodynamic applications that measures the performance of an airfoil by changing its shape (a combination of upper and lower surface). More specifically, the focus is on the maximisation of lift and minimisation of drag of a 2D airfoil, which are conflicting in nature. In open literature there are several techniques which can derive these two quantities, but the choice of the best/most appropriate one, in terms of accuracy, is out of scope. Herein, the goal is to validate MOTS2 and demonstrate its linking with external software.

A brief description of the tools used follows. The two objectives are derived by using a software called XFOIL [7, 8]. It belongs to the category of panel methods for low-speed inviscid flow, where the body is discretised in terms of singularity distribution on the surface. Compared to conventional Computational Fluid Dynamics (CFD) methods, where the flow is calculated around the test object over a large surrounding field, panel computations around the surface are only required. This method has certain advantages for specific cases. The only prerequisite is to discretise the surface, while the computational cost is relatively low. The downside is that only subsonic flows of low Mach number can be studied. Consequently, inaccurate skin friction estimations, wave drag predictions, lift coefficient estimation will eventually lead to poor results in the optimisation process.

The physical representation of the airfoil surfaces is a very challenging task. The points should be distributed mostly in leading and trailing edge because the velocity changes rapidly in these regions. Imposing additional constraints could help, without mitigating completely the generation of irregular shapes. For instance, cusp-like configurations that will alleviate drag due to Mach waves formation cannot be proposed by the optimisation process. By using exact coordinates of the upper and lower airfoil surface velocity distribution is calculated. However, this will be investigated in future. In order to minimise the complexity, the Free Form Deformation (FFD)[20] geometrical representation technique is used to parameterise the different geometrical airfoil profiles. Nevertheless, this application can deliver good results for low velocity cases, and demonstrate the applicability of the optimiser in real-world applications.

A brief description of the assembly of aforementioned software follows. The shape of the airfoil is obtained by using the FFD method. A new geometrical arrangement is generated by adjusting the control points that manipulate the whole surface. Then, a series of exact 2D coordinates are generated for the upper and lower surface of the airfoil. Then, these are used as input on XFOIL, where the given airfoil is combined with multiple angles of attack, and the average lift and drag coefficients are obtained. Regarding the optimisation search, this is considered as a single objective functions' evaluation of a design point and, thus, will be repeated many times. This is the most computationally intensive part of the whole optimisation search and can be considered as a modular black box.

As usually in the optimisation search, the optimiser will strategically propose various combinations of inputs, FFD control points here, which will be mapped internally with the respective objectives by using the assembly described above. Inside the optimiser's body the fitness, airfoil performance metrics here, received are normalised over the metrics of the airfoil profile of NACA 0012, which is the datum design point. Hence, the results of the optimiser are expressed in terms of relative improvement. This will be repeated until the end of the computational budget, which is a fixed number of objective function evaluations. For minimisation purposes, the lower the objective value below 1.0, the better the design is from the initial performance

A brief description of the assembled framework will be described. The shape of the airfoil is obtained by using the FFD parameterisation method. A new geometrical profile is generated by adjusting 8 control points that manipulate the whole surface. Then, a series of coordinates are generated, which will be used to calculate the pressure distribution of the pressure over the airfoil surfaces and will deliver the respective lift and drag for the target configuration. Simply, the optimiser will propose various combinations of control points and will request the respective fitness, as described above. This will continue until stopping criteria are met. Using less accurate but faster methods (such as XFOIL) in the early stages of the design process is preferred. Thereafter, more accurate simulation models should be considered and, ultimately, the computationally expensive CFD tools could be employed. Clearly, optimising the behaviour of aerodynamic problems via a high fidelity CFD tool is of paramount importance in the whole engineering design process, as described in[14, 21].

In summary, 8 decision variables uniquely characterise the profile of a 2D airfoil, which represents an individual geometrical shape of an airfoil. Essentially, the geometrical arrangement is represented via FFD, which is a technique that manipulates any shape in a free-form manner [20], and the variables are the control points. The objectives are to maximise lift coefficient (cL) and minimise drag coefficient (cD) coefficient. In the Figures below the cL maximisation is equal to minimisation of the negative cL. The search starts from a datum point (a well known geometry) of the airfoil NACA 0012 and the optimisation is expressed in terms of relative improvement; any newly generated geometry is compared against the datum design and the ratio of new coefficients over the datum ones is considered as objective.

Regarding the constraints, only hard constraints and range of variability were applied here. More specifically, two thickness limits of the airfoil are specified at 25% and 50% of its length. These are hard-integrated into the objective function evaluation model (XFOIL). Besides that, the range of each parameter is between -0.4 and 0.3.

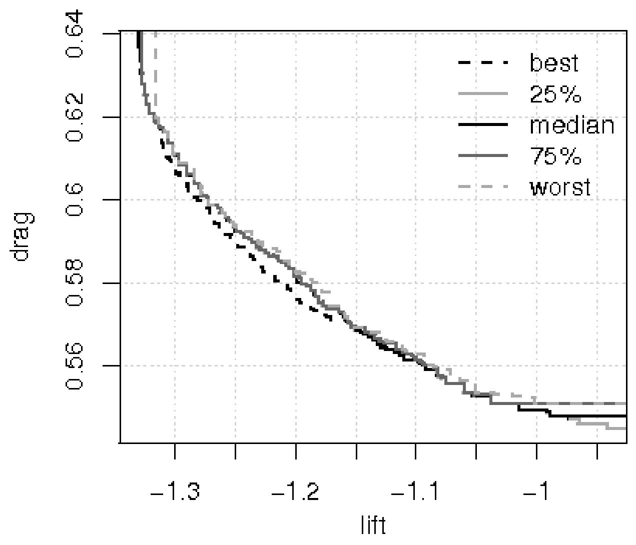
The configuration parameters of the optimiser are listed in Table 1 and follow the same logic as described in [13]. The first three values are related to the search strategy. The lesser the number, the more frequently the respective move will be performed. The case ran for about 3000 objective function evaluations and the obtained PFs are depicted below. The initial step was resolved by previous studies, which carried out sensitivity analysis on XFOIL. The retained factor and random sampling were set arbitrarily. According to developer's experience, halving the SS and acquiring about as many samples as the number of variables turns out to be a decent choice. In this specific case, the turnaround time of one objective function evaluation can take up to 1 minute. This number is not very long, but in more complex problems it could be of the order of magnitude of hours or days. So, a relatively low upper limit of evaluations was used. The ability of the optimiser to reveal a PF as close to the (unknown) true PF is the most important aspect for efficient and reliable multi-objective optimisation. Whenever the true PF is not known, delivering an approximation set as close to the corner of interest (bottom

Table 1: Configuration settings for airfoil optimisation

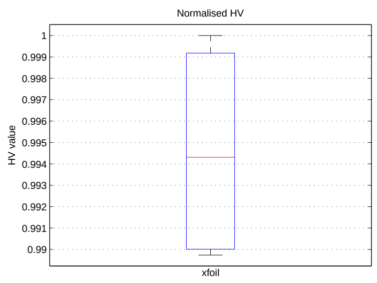
description	MOTS2 setting
performing diversification move after # iterations	25
performing intensification move after # iterations	15
performing reduction move after # iterations	45
initial search step	0.07
search step retain factor	0.5
# of random samples	6
# of variables	8
# of objectives	2
max objective function evaluations	3000
# of regions in Long Term Memory	4
Short Term Memory size	15

left in the trade-off Figures) is satisfactory. Again, the number of individual regions and and STM size were chosen based on previous experience and following previous studies. Herein, the LTM regions are split based on the range of the first variable. Although this strategy seems to be good here, this is not always the case. In fact, deeper sensitivity analysis should be performed in order to resolve the most important variables or combination of variables and divide the search space according to that.

The same statistical tesgin was applied again. The EAF plots were produced by using the tool described in [17] and along with the HV boxplots are depicted in Fig. 1. From the shape of the PF, MOTS2 presents a robust behaviour and seems to be appropriate for this type of engineering problems.



(a) EAF



(b) HV boxplot

Figure 11: XFOIL performance assessment