Sérgio Filipe Faustino Tavares

Bachelor of Computer Science and Engineering

Contributions to the development of an integrated toolbox of solvers in Derivative-Free Optimization

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser:	Vítor Manuel Alves Duarte, Assistant Professor, NOVA University of Lisbon
Co-advisers:	Ana Luísa da Graça Batista Custódio, Associate Professor, NOVA University of Lisbon
	Pedro Abílio Duarte de Medeiros, Associate Professor, NOVA University of Lisbon

Examination Committee

Chair:	Professor Ludwig Krippahl, NOVA University of Lisbon
Members:	Professor Vasco Pedro, University of Évora
	Professor Vítor Duarte, NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

July, 2020

Contributions to the development of an integrated toolbox of solvers in Derivative-Free Optimization

This document was created using the (pdf)$\LaTeX$ processor, based on the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To life itself, that always has a curious way of teaching us the lessons we need to learn.

Acknowledgements

This master thesis was the longest and most complex project I have done so far. Many were the lessons I learned during its course and I come out of it feeling a lot more confident and competent. I appreciate all the support I received and therefore would like to make several acknowledgments.

I acknowledge the support of Fundação para a Ciência e a Tecnologia, under the project *BoostDFO: Improving the performance and moving to newer directions in Derivative-Free Optimization* (PTDC/MAT-APL/28400/2017).

A big praise for FCT NOVA, for all I learned and the amazing time I had while in this House, along with all the opportunities it provided me. A word of praise also for CMA (Centre of Mathematics and Applications) and NOVA-LINCS, for supporting my work.

To my adviser Vítor Duarte and both co-advisers, Pedro Medeiros and Ana Luísa Custódio, I am grateful to all for this opportunity and all the advice given. You showed me what research is and what it is like to be part of a research team, which I really enjoyed.

Also, a very special thanks to my co-adviser Ana Luísa Custódio, principal investigator of this project. She always has a motivational word at the right times and is always available despite her busy schedule. Thank you for sharing with me your huge passion for Mathematics, and for Optimization in particular. It is passion that makes a real difference and has been inspiring to work with you.

A word of gratitude to Teresa Romão for her help and advice on user testing and usability issues.

A big thanks to all our Anonymous Testers, that endured more than one and a half hour of tasks. And also to my friend Vasco Soares, that was the victim of the pilot user test, to make sure everything was well planned.

To my parents, for all the support and encouragement they always gave to my studies, in particular during my thesis. Thank you!

Last but not least I'd like to show my gratitude for all the amazing people I met along this path, especially those that I am proud to call friends. They have the incredible power of turning good times into amazing ones, and making awful times quite bearable. I give special praise for the best of them: Nuno Pulido, Pedro Correia and André Santos. And also a big praise for André Lopes for being my companion on the battlefield for a few exhausting months.

There was once this final destination, that after all discovered it was the Path.

Unknown Author

Abstract

This dissertation is framed on the ongoing research project *BoostDFO - Improving the performance and moving to newer dimensions in Derivative-Free Optimization*. The final goal of this project is to develop efficient and robust algorithms for Global and/or Multiobjective Derivative-free Optimization. This type of optimization is typically required in complex scientific/industrial applications, where the function evaluation is time-consuming and derivatives are not available for use, neither can be numerically approximated. Often problems present several conflicting objectives or users aspire to obtain global solutions.

Inspired by successful approaches used in single objective local Derivative-free Optimization, we intend to address the inherent problem of the huge execution times by resorting to parallel/cloud computing and carrying a detailed performance analysis. As result, an integrated toolbox for solving single/multi objective, local/global Derivative-free Optimization problems is made available, with recommendations for taking advantage of parallelization and cloud computing, providing easy access to several efficient and robust algorithms and allowing to tackle harder Derivative-free Optimization problems.

Keywords: Derivative-free Optimization, multi/single objective, global/local optimization, numerical algorithms, parallelization, cloud computing, integrated toolbox

Resumo

Esta dissertação insere-se no projecto científico *BoostDFO - Improving the performance and moving to newer dimensions in Derivative-Free Optimization*. O objectivo final desta investigação é desenvolver algoritmos robustos e eficientes para problemas de Optimização Sem Derivadas Globais e/ou Multiobjectivo. Este tipo de optimização é tipicamente requerido em aplicações científicas/industriais complexas, onde a avaliação da função é bastante demorada e as derivadas não se encontram disponíveis, nem podem ser aproximadas numericamente. Os problemas apresentam frequentemente vários objectivos divergentes ou os utilizadores procuram obter soluções globais.

Tendo por base abordagens prévias bem-sucedidas utilizadas em Optimização Sem Derivadas local e uniobjectivo, pretende-se abordar o problema inerente aos grandes tempos de execução, recorrendo ao paralelismo/computação em cloud e efectuando uma detalhada análise de desempenho. Como resultado, é disponibilizada uma ferramenta integrada destinada a problemas de Optimização Sem Derivadas uni/multiobjectivo, com optimização local/global, incluindo recomendações que permitam tirar partido do paralelismo e computação em *cloud*, facilitando o acesso a vários algoritmos robustos e eficientes e permitindo abordar problemas mais difíceis nesta classe.

Palavras-chave: Optimização Sem Derivadas, uni/multiobjectivo, optimização local/global, algoritmos numéricos, paralelismo, computação em cloud, ferramenta integrada

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The present work is framed within the research project *BoostDFO: Improving the performance and moving to newer dimensions in Derivative-Free Optimization*, supported by *FCT – Fundação para a Ciência e a Tecnologia* [PTDC/MAT-APL/28400/2017]. The main goal is to improve the efficiency and the robustness of existent algorithms designed to solve derivative-free optimization problems.

Derivative-free optimization (DFO) is a specific field of nonlinear optimization characterized by the absence of information about derivatives. In this scientific domain, the function to optimize can be nonsmooth, or, even when smooth, derivatives may be unavailable for use or impossible to numerically approximate (given, for instance, the presence of noise). The objective function can even work as a black-box model, only providing the result of point evaluations [15]. As we can deduce, typical methods used in nonlinear optimization (like steepest descent or Newton method) cannot be applied given the absence of derivatives and the impossibility of its efficient approximation. Thus, different algorithmic approaches are required to address this class of problems.

Applications include solving complex technical problems of industrial or scientific nature whose goal is ideally to find a global minimum, fitting one or several objectives. Constraints can also be present [15]. The main challenge with this sort of problems comes from function evaluations, which are usually very expensive – computationally and time-consuming. Each evaluation could take from a few seconds or minutes, to hours or more. For this reason, the state-of-the-art includes problems with only a few hundred variables at best, which is several orders of magnitude below nonlinear optimization with derivatives [15].

Currently, several implementations of DFO solvers are available to the community. In this work we consider SID-PSM [19, 23], DMS/boostDMS [10, 22], GLODS [20] and MultiGLODS [21], all property of the research team, implemented in MATLAB [39].

These are all Directional Direct Search methods, differing on being suited for local or global optimization and on the number of objective functions considered (one or several).

This work and the related research project are intended to increase DFO approaches' competitiveness and corresponding use in practical applications.

1.1 Problem Statement

Directional Direct Search is a DFO class of algorithms. In a simplified way, these algorithms proceed by evaluating points in a discrete neighborhood of the current best point. Coordinate directions are an example of such discrete neighborhoods. If a better point is found in the evaluation procedure, meaning that it decreases the value of the objective function, then the point is accepted and a new discrete neighborhood is defined around it, repeating the evaluation procedure. Otherwise, all the points in the discrete neighborhood have been evaluated (with no decrease found), the current point is maintained and the size of the neighborhood is decreased. The procedure continues until a given threshold is reached, related to the neighborhood size.

As can be inferred, all the (possibly) computationally expensive function evaluations can be processed at the same time in each iteration. More than that, they are independent from each other, which calls for embarrassingly parallel approaches.

Nowadays, the access to cloud computing is getting easier and closer to the general public. The opportunity for on-demand customization based on the user needs and resources configures a promising cost-efficient solution, while eliminating the need to possess and maintain a physical infrastructure [38, 42].

As researchers and other intended users of our optimization solvers don't necessarily have access to powerful machines or could be interested in a low-cost approach for experimentation, designing and testing a cloud solution would be of great interest.

At the present time, all the algorithms have MATLAB code implementations available that allow the customization of several parameters. The available solution requires the user to have at least basic programming skills, MATLAB knowledge and to be familiar with the different algorithmic frameworks. Although these are reasonable expectations for our intended users, it would be beneficial to offer a more interactive and direct way of running the codes while aggregating them in a toolbox, under a Graphical User Interface with a recommendation system.

1.2 Objectives

In order to make the current algorithms more efficient and able to address problems of higher dimensions, which means a higher number of problem variables, parallel versions were considered. As a first step, the developed solutions were tested experimentally on a physical machine (local), and afterwards on the cloud. The goal was to distinguish the conditions for which the parallelism is useful and measure the expected improvements, as well as to recommend an adequate number of processors to tackle a given problem.

Regarding the cloud solution, it would be interesting to consider the number of processors as a function of the cost of each machine. However, this is not possible to address since the number of evaluations/iterations required for solving a problem is not known in advance and cannot be estimated.

The graphical interface is expected to advise the user on the best alternatives for solving the problem: serial or parallel run (when available) and parameters customization. One feature of particular relevance is providing a proper estimate on the number of processors that are required for a good performance, according to the specified data for the problem. The user should provide the function to optimize, as well as any available information about its level of smoothness (smooth, nonsmooth or unknown).

As a result of this work, we expect to be able to *move to newer dimensions* and tackle harder problems.

1.3 Expected Contributions

- Code optimization of the serial version of the SID-PSM algorithm.
- Experimental analysis of several options and parameter choices regarding SID-PSM's performance.
- Parallel versions of SID-PSM, including a detailed analysis of expected gains and cloud testing.
- Implementation of a parallel version with a complete variant, where the discrete neighborhood is always fully explored.
- Design of a code-integrative toolbox featuring a Graphical User Interface with a recommendation system. This part should include all four algorithms (SID-PSM, DMS/BoostDMS, GLODS and MultiGLODS).

1.4 Document Structure

In this first chapter, we provided all context, and clarified challenges and objectives on the problem considered.

Chapter 2 includes all theoretical background and related work necessary for a complete understanding of this work.

Chapter 3 details the analysis of the SID-PSM algorithm, describing the code optimization performed, results from parameter comparison, details on the benchmark process and the identified opportunities for parallelization.

Chapter 4 comprises all the parallelization work accomplished, detailing the algorithm's original structure, the parallel versions implemented, the study on the recommendation system and finally the evaluation performed of the parallel gains.

Chapter 5 is related to the implementation of the code-integrative toolbox. Includes details on the design approach followed, a description of the final product obtained and the results and conclusions derived from the evaluation process.

The last Chapter (Chapter 6) is a summary of all the contributions that derive from this work, including the most important results, concluded by a short section on future work.

Appendixes include all elements required for the user testing performed on our toolbox. Appendix A contains the test script. Appendix B is the questionaire applied afterwards (*System Usability Scale*), taken from the original publication [12].

Annex I includes the complete user guide for the toolbox developed.

CHAPTER 2

Related Work

This chapter provides the theoretical background and references the related work required for a better understanding of the present thesis. In Section 2.1, we introduce several subjects related to Derivative-Free Optimization, such as different problem classes and methods for addressing them. In Section 2.2, basic concepts of parallelism and parallel architectures are revised. A brief overview on cloud computing is included in Section 2.3 and, in Section 2.4, MATLAB's available parallel commands and respective underlying functionality are addressed, as well as other related tools required.

2.1 Derivative-Free Optimization

DFO problems are characterized by two main features: unreliability or inexistence of derivatives, and an expensive cost of function evaluations. Typically, a black-box model is assumed: information about the objective function is only available at the evaluated points.

It is currently an area of great demand for robust and efficient algorithms given the increasing complexity in mathematical modeling, higher sophistication of scientific computing and an abundance of legacy and proprietary codes where information is limited. The diversity of applications includes problems in engineering, mathematics and operations research, physics, chemistry, economics, medicine, cognitive science, computer science and several other fields [8, 11, 27, 29, 34, 57].

Assuming derivative information was available, derivative-based algorithms would clearly be more efficient than DFO methods. However, given that today's practical problems are often complex, nonlinear, not providing an analytical expression for the derivatives of the function to be minimized, neither allowing their reliable numerical estimation (using, for instance, finite-difference methods), there is an ever-growing need for efficient and robust DFO implementations.

2.1.1 Classes of Problems

Problems can be classified based on their level of smoothness, a property related to the corresponding differentiability. A smooth function is defined as being continuously differentiable up to some needed order in its domain (could be of second order or infinity, depending on the problem) [62]. This is an interesting property in the sense that smooth functions, even when derivatives are not available for use, allow the local adjustment of functions based on the available data, which can be easily minimized and constitute the core of trust-region methods [45]. In this way it is possible to exploit the characteristics of the function and accelerate the convergence process of an optimization algorithm. Nonsmooth functions are generally more difficult to optimize. They can even be contaminated with noise, meaning that the previous fitting approach would fail. In this work, both classes of functions were considered for testing.

2.1.2 Derivative-Free Optimization Methods

In order to solve DFO problems, several classes of methods are available, whose effectiveness typically depends on the characteristics of the objective function and constraints. Different approaches include deterministic methods, like Directional Direct Search and Trust-region Methods based on polynomial interpolation or regression. Randomized or stochastic methods, such as Evolutionary Strategies or Particle Swarm Optimization, can also be considered [36].

In the present work, we focus only on those that have a well established convergence analysis, particularly Directional Direct Search and Trust-region Methods based on polynomial interpolation or regression.

2.1.2.1 Directional Direct Search

Direct Search methods only use function values to define the next step of the optimization procedure and don't attempt to approximate derivatives, neither to model the objective function in any explicit or implicit way. In the case of Directional Direct Search, the points to be evaluated correspond to directions [63].

In each iteration, a precomputed set of directions (with good geometrical properties) is explored around the best available point x_k (also known as poll center). A new set of points is evaluated, each of them at distance $\alpha_k\|d\|$ of the poll center, $x_k + \alpha_k\|d\|$, where α_k represents a step size parameter. When and if a better point is found, one whose evaluation represents a decrease (or sufficient decrease, depending on the globalization strategy considered) on the value of the objective function f, the iteration stops, and a new poll center x_{k+1} is thus defined. If no point was found better than the current poll center, the step size is decreased and x_{k+1} is set to x_k. In this case, an opportunistic polling strategy was applied, meaning that the first successful point is accepted and the evaluation procedure is stopped. If all the points considered were evaluated and the best one was chosen, a complete polling strategy would be in place. This process continues through several iterations until a stopping criterion is satisfied. To ensure the quality of the computed point as solution to the problem, this criterion should be related with the step size getting lower than a given small threshold [15, 36].

In Figure 2.1 we present a general framework, where *test_descent* can either encompass a strategy based on simple or sufficient decrease to accept new points. A search step can also be included. This is an optional step, not required for establishing the convergence of the algorithmic class but can be used to increase the numerical efficiency of the algorithms. If the search step succeeds in finding a better point than the current poll center, then the poll step is not performed. The minimization of quadratic polynomial models, like it is the case in Trust-region Methods based in polynomial interpolation or regression, has been used as strategy for its definition [19].

The SID-PSM algorithm, makes use of a structure similar to the one presented in Figure 2.1. Further analysis will be provided in the chapter 3.

```
Set parameters $0 < \gamma_{\text{dec}} < 1 \leq \gamma_{\text{inc}}$
Choose initial point $\boldsymbol{x}_0$ and step size $\alpha_0 > 0$
for $k = 0, 1, 2, \ldots$ do
    Choose and order a finite set $\boldsymbol{Y}_k \subset \mathbb{R}^n$                // (search step)
    $\boldsymbol{x}_k^+ \leftarrow$ test_descent$(f, \boldsymbol{x}_k, \boldsymbol{Y}_k)$
    if $\boldsymbol{x}_k^+ = \boldsymbol{x}_k$ then
        Choose and order poll directions $\boldsymbol{D}_k \subset \mathbb{R}^n$          // (poll step)
        $\boldsymbol{x}_k^+ \leftarrow$ test_descent$(f, \boldsymbol{x}_k, \{\boldsymbol{x}_k + \alpha_k \boldsymbol{d}_i : \boldsymbol{d}_i \in \boldsymbol{D}_k\})$
    if $\boldsymbol{x}_k^+ = \boldsymbol{x}_k$ then
        $\alpha_{k+1} \leftarrow \gamma_{\text{dec}} \alpha_k$
    else
        $\alpha_{k+1} \leftarrow \gamma_{\text{inc}} \alpha_k$
    $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k^+$
```

Figure 2.1: A typical Directional Direct Search framework, taken from [36].

2.1.2.2 Trust-region Methods

Trust-region Methods constitute a particular framework of model-based methods. These methods resort to predictions based on models, which work as a replacement of the objective function [36]. In Trust-region Methods for DFO, models are often built from sampling and some type of interpolation or regression techniques, depending on the number of points available for use (those where the objective function has already been evaluated). Thus, the model configures a local approximation of the function and tries to capture its curvature [17].

Each iteration of a Trust-region Method builds a model around the current iterate, x_k, which is minimized in a trust region. This region is frequently defined as a ball, with center in x_k and a given radius [17].

2.1.3 Existing Solvers

Several DFO solvers are available, corresponding to implementations of different methods. Some well-known solvers related to Directional Direct Search and Trust-region Methods are outlined here, with no intent of being exhaustive: "DFO", NEWUOA/BOBYQA/-COBYLA, HOPSPACK, NOMAD, PSWARM [17, 50]. The last three also include parallel implementations.

"DFO"[14] is a Fortran software for local optimization that builds quadratic models by interpolating selected subsets of points and optimizing the resulting trust region model. NEWUOA [47], BOBYQA [49] and COBYLA [48] are all Fortran implementations of Powell's model-based algorithms, with different characteristics. NEWUOA is particularly interesting for unconstrained optimization since it can solve problems with several hundreds of variables.

HOPSPACK [30] is built upon a C++ software framework. The framework allows parallel operations, using MPI for distributed parallelism or multithreading on multicore machines. Multiple algorithms can run simultaneously and share information for performance improvement. HOPSPACK comes with an asynchronous pattern search solver that handles general optimization problems with linear and nonlinear constraints, and continuous and integer-valued variables. The parallel approach is based on distributing the function evaluations to different workers.

NOMAD [37] is a C++ implementation of the Mesh Adaptive Direct Search algorithm (MADS [6]), that incorporates several strategies and solves nonlinear, nonsmooth and noisy optimization problems. Parallelism resorts to MPI.

PSWARM [60] is a pattern search method implemented in MATLAB and C. The search step corresponds to Particle Swarm, an heuristic algorithm, aiming at global search. The poll step relies on coordinate search. The C implementation includes a parallel version that uses MPI.

2.2 Parallelism

A **parallel computer** can be regarded as a set of processors that are able to work cooperatively to solve a computational problem. This definition can include different kind of multicore system, from portable devices to parallel supercomputers with hundreds or thousands of processors, networks of workstations, multi-core CPUs, and embedded systems [28]. Nowadays, due to the ubiquity of parallel processors and the stagnating single-threaded performance of modern CPUs, parallel programming has become increasingly important and computer scientists and engineers are required to write highly parallelized code in order to fully utilize the computational capabilities of current hardware architectures [54].

Instead of writing programs that execute all instructions sequentially, parallelism allows the execution of multiple instructions at the same time. In this way, we can make our programs run faster while exploiting the available resources of nowadays multicore machines, leading to a better use of all computational power available. Or, given a different point of view, scale up to higher dimensions and increase the difficulty of the problems we can solve. Parallelism can be required to speed up the program runtime, to provide scalability or due to memory restrictions, in case a single worker cannot accommodate in memory all data related to the problem and data partitioning is required. This is why, more than an interesting and exotic subarea of computing, parallelism is becoming a universal need to the programming enterprise [28].

2.2.1 Basic Concepts

2.2.1.1 Speedup

Speedup is a standard metric regarding parallel programming. It measures the time gain obtained by running the code in parallel, by comparison to the initial sequential run. Thus, it can be defined as the quotient of the time using a single processor (T_1) over the time measured for p processors (T_p):

$$S = \frac{T_1}{T_P}.$$

2.2.1.2 Efficiency and Cost

Cost and efficiency are two closely related metrics. Cost expresses the total time spent on computations by all cores, $C = T_p \times p$. Efficiency relates sequential time to cost, using the equation $E = \frac{S}{p} = \frac{T_1}{C}$. An efficiency of 100% means that a *linear speedup* was obtained. This is usually the upper bound we can expect (even though some exceptions can be verified, that are referred as *super-linear speedups*).

2.2.1.3 Scalability

There are two kinds of *scalability analysis* that can be performed, depending on the aim of the parallelization. Both approaches are intended to provide an upper bound for possible speedup and efficiency, by studying the behavior of these variables when varying the number of processors or the input data.

Strong scalability concerns a fixed data input and a varying number of processors. It can be evaluated by Amdahl's Law [3]. On the other hand, **weak scalability** focus on increasing the problem size along with the number of processors. Gustafson's law is the starting point for the analysis, on this case [31].

Regarding this dissertation, both approaches are considered of interest, being weak scalability the most directly applicable, since we aim to reduce execution times. Our approach was to uncover a good number of processors based on a problem's dimension (weak scalability) and test the performance of these values for strong scalability. This is based on the characteristics of this kind of problems (computationally expensive function evaluations).

However, first of all, our aim was to analyze if and under what conditions (related to the computational time of function evaluations) the parallel effort is worthwhile.

2.2.2 Parallel Architectures

2.2.2.1 Distributed Memory Systems

In Distributed Memory Systems all nodes are connected through an interconnection network and remote data accesses need to be explicitly implemented through message passing over this network. Each process can only operate on its own local memory. However, in *collective communication*, like data broadcasting or reduction operations (where the same data is shared between nodes), all nodes can participate. Computer clusters and cloud implementations featuring several machines are good examples of this model. In the case of cloud implementations, latency overheads should also be considered. As main advantage, it can be pointed out the potential of scalability and customization of the hardware solution. However, the ratio between the total time spent on computations and the time spent on message transmission (computation-to-comunication ratio) is usually low, because the message passing system over the network results in additional communication overheads [54].

2.2.2.2 Shared Memory Systems

All CPUs can access a common memory space through a shared bus or crossbar switch. Modern multicore machines configure a well-known example of shared memory systems. All processors feature also a smaller local memory for faster access, known as cache. In this way, expensive accesses to the shared memory can be avoided. [54].

2.2.2.3 Data Partitioning

When designing parallel algorithms, it is also essential to choose the most suited partitioning strategy. **Data parallelism** distributes data across different processors or cores which can then operate on their assigned data. Embarrassingly parallel algorithms configure a particular case where all computations are data-independent and can be naturally partitioned into blocks. It is the simplest parallel scheme and requires very little communication: we can use a master-worker approach where the master delivers the data to the corresponding workers and assembles the results in the end. The only communication present is between the master and the workers, making this scheme very easy to implement on practice [55]. **Task parallelism**, on the other hand, assigns different operations to processors which are then performed on the same data. For example, when wanting to apply several tasks to a dataset, it is possible to split it into blocks, each processor executing all tasks in a given block (data parallelism), or to let each processor execute a different task in the whole dataset (task parallelism). To make the best use of parallel resources, a *Job Scheduler* following a specific *load balancing* strategy is often needed. It is responsible for distributing equal amounts of work to each processor (possibly dynamically) and can adopt mixed partitioning strategies [54].

2.3 Cloud Computing

According to the United States' National Institute of Standards and Technology (NIST) definition of cloud computing, "this is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"[41].

The cloud is therefore not a physical entity, but a global network of remote servers meant to operate as a single, seemingly unlimited environment. To the user it works as an abstraction of the underlying infrastructure, that can be available anywhere, at all times. Its use can be in all similar to regular on-site servers, but everything is accessed remotely with all guarantees of availability and security [38].

All the infrastructure is yield by the provider, allowing customization and easy and secure access. The users have therefore no need to maintain their own infrastructure and all the related costs, such as hardware, maintenance, physical space, energy consumption, *etc*, can be cut off. The users are only responsible for the services they actually use, leading to lower costs and increasing productivity. Given the actual dimension of cloud providers, the convenience and cost-control delivered are very difficult to achieve with a common on-site infrastructure [38]. Google Cloud Platform, Amazon Web Services (AWS) and Microsoft Azure are nowadays among the biggest providers of cloud services worldwide.

There are three possible models associated with cloud services: Software as a Service (**SaaS**), Platform as a Service (**PaaS**) and Infrastructure as a Service (**IaaS**) [41].

IaaS represents the lowest-level construct. The user has full control over the provided computer resources and is able to customize the desired infrastructure at will (the provided abstraction, not the underlying cloud implementation). The required systems and software are built upon this infrastructure.

PaaS model offers a specific cloud infrastructure, including network, servers, operating systems and storage, where the user can control the deployed applications and configuration settings for the application-hosting environment.

The SaaS model provides to the user the capability to run the provider's applications on some cloud infrastructure, granting external access to the application via a web browser or a program interface. The level of control allowed is limited to user-specific application configuration settings.

In this research project, tests were also be performed on the Azure Cloud Platform, using the IaaS resources provided. Several infrastructures were considered, with different numbers of cpus, with the intention of distributing the solution and allowing an easy access with some experimental guarantees. Each one includes a single virtual machine, with Windows or Linux operating system and a MATLAB installation. If some specific challenges on using clusters with the current technology (particularly MATLAB Parallel Server) could be overcome, it would be of interest to test additional cloud infrastructures comprising several machines.

2.4 MATLAB and Related Tools

MATLAB is a natural choice for this thesis, since all codes are implemented in this programming language. The syntax is succinct and straightforward, with support for several functionalities, mostly related to mathematical computations and data visualization and analysis. Simple and intuitive graphical user interfaces can be built and several types of parallelism are supported, so it is well suited for our needs.

MATLAB includes an editor and a profiler, ideal for code optimization. It is possible to detect several code problems, from unused variables to memory allocation problems. As there are no explicit memory references in MATLAB, proper allocation is essential to guarantee the efficiency of programs. Also, a specific tool for the development of applications is included, *Appdesigner*. The interface design is made simply by component drag-and-drop and further integration and development is facilitated.

MATLAB can be extended by several domain specific Toolboxes, each one expanding the available features. In this work we required the Parallel Computing Toolbox, that includes all the necessary mechanisms to add parallelism to programs: high-level constructs, like parallel for-loops, special array types, and the possibility of launching several workers either locally or in a cluster. Nevertheless, for configuring and using a cluster, an extra tool is required, namely the MATLAB Parallel Server, which results in additional costs. For this reason, we focused on studying the improvement over one multicore machine, with plenty of available cores.

As we have seen, the characteristics of SID-PSM algorithm seem to involve mostly the use of parallel for-loops, possibly making use of asynchronous executions, depending on the poll strategy. In order to make use of this kind of parallel executions, the Parallel Computing Toolbox provides three essential keywords: *spmd*, *parfor* and *parfeval* [40].

2.4.1 *spmd*

Spmd stands for *single program multiple data*, meaning that each worker can process different inputs in order to get an improved performance. It is the most common style of parallel programming, allowing seamless interleaving of serial and parallel programming. It is also the paradigm that gives the programmer greater control over the program flow, making it possible to address each worker individually.

Typical applications that can benefit from *spmd* are those that require running simultaneous tasks of a program on multiple data sets, when communication or synchronization is required between the workers [40].

In the present work, we focused on the simpler commands *parfor* and *parfeval*, since they are adequate to attain our goals.

2.4.2 *parfor*

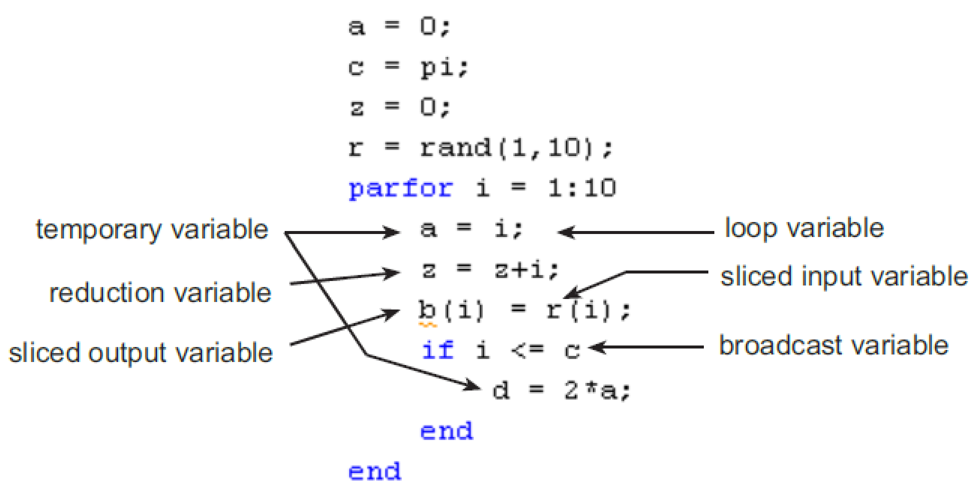
A *parfor-loop* in MATLAB executes a series of statements in the loop body in parallel. A Job Scheduler is responsible for managing the available workers while executing the loop in parallel. Each one of them runs a MATLAB instance in a different core, so the maximum possible number of workers corresponds to the machines' available cores. The number of available workers is defined at the start of the program by calling a static parallel pool, which cannot be modified later. If more workers are needed at a later stage, a new parallel pool should be created [40]. In the case of continuously adjusting the number of processors to best fit our current needs, this wouldn't be an adaptive solution.

Each execution of the body of a *parfor-loop* is an iteration. MATLAB workers evaluate iterations in no particular order and independently of each other. All iterations are always processed, blocking the program until the results are gathered – synchronous approach. *Parfor* command is then more useful on cases of natural parallelism, when a computationally intensive task must be repeated several times, and completed until the end.

To transform a serial program in a parallel one, by means of parallel-loops, all variables inside the loop must be accounted for, in order to prevent concurrency/data-race issues and to minimize communication overheads. The Table 2.1 shows the different variables accepted in the scope of *parfor* loops as well as their description. A code example is provided in Figure 2.2.

Variable Classification	**Description**
Loop	Loop index.
Sliced	Different segments of an array that are accessed in different iterations of the loop.
Broadcast	Variables defined before the loop, whose values are required but never assigned.
Reduction	Variables that accumulate a value across all iterations, *e.g.* counters.
Temporary	Variables created inside the loop and not accessed outside it.

Table 2.1: Variables accepted by *parfor* command, as described in [40].



```
a = 0;
c = pi;
z = 0;
r = rand(1,10);
parfor i = 1:10
    a = i;
    z = z+i;
    b(i) = r(i);
    if i <= c
        d = 2*a;
    end
end
```

Figure 2.2: Example of variables supported by the *parfor* command.

2.4.3 *parfeval*

Parfeval execution model is similar to *parfor*, but with an asynchronous approach based on futures/promises. Each task is sent to an available worker, not blocking the flow of the program, and results can be retrieved later, when the computation is finished, accessing the future objects. Remaining iterations can be canceled at any time.

There are two ways available to approach this scheme, using *parfeval* or *parfevalOnAll*. In both cases all the iterations are divided by the available workers. The difference is related to the way of accessing results. The latter executes all tasks and returns only the complete results. Instead, with *parfeval* we can retrieve the results one by one (after each worker has finished it) which can be convenient to break out of a for loop early. For example, in our optimization procedures, when an opportunistic variant is considered for polling, we can stop the loop early, indicating that a better point was found. Therefore there is no need to wait for all computations to finish, which may be canceled. As the function evaluations are generally computationally expensive, this solution can present a great advantage.

Chapter 3

Analysis of the SID-PSM algorithm

This chapter covers the complete analysis of the SID-PSM algorithm. Sections 3.1 and 3.2 detail the algorithm procedure and the algorithmic options available. Section 3.3 focus on the adopted benchmark strategies. The optimization performed in the code before advancing to its parallelization is described in Section 3.4.

3.1 General Framework

SID-PSM is a directional direct-search method where the poll evaluation follow an ordering induced by simplex gradients [23]. It is suited for solving single objective Derivative-Free Optimization problems, having consistently showed its competitiveness when compared with other DFO solvers [51, 56], even if released only in a serial version. The global convergence analysis guarantees that a subsequence of the sequence of iterates generated by the algorithm will converge to a stationary point of the optimization problem, as proven in [5, 35, 59, 61].

Its current structure can be described in the following steps, that will be further detailed: **Initialization**, **Search Step**, **Poll Step**, **Compute Descent Indicators**, **Order Directions** and **Update the Mesh Size Parameter**. All steps except the Initialization are repeated in each iteration, until a stopping criterion is satisfied. Notice that, for convenience, the order and description of these steps is directly related to the code implementation, being slightly different from the framework proposed in [23]. Only one objective function is accepted but several restrictions can be considered.

3.1.1 Initialization

Choose x_0 and $\alpha_0 > 0$. Choose a set of positive spanning sets D, the set of directions with good geometrical properties. Positive spanning sets for $\mathbb{R}^n$ generate any vector of $\mathbb{R}^n$

through non negative linear combinations of their elements [24]. Select all other necessary constants, including the ones for updating the step size parameter.

3.1.2 Search step

It is based on the minimization of quadratic interpolation or regression models, within a trust region, which size is directly related to the step size parameter. The models are built using sets of previously evaluated points. If a point x, corresponding to the model minimization, satisfies $f(x) < f(x_k)$, then it is accepted as the new iterate, $x_{k+1} = x$, the iteration is declared successful, and the poll step is not performed.

3.1.3 Poll step

This step evaluates the polling set defined as $P_k = \{x_k + \alpha_k d : d \in D_k\}$, following a previously determined order and storing each evaluated point. If a polling point $x_k + \alpha_k d_k$ is found such that $f(x_k + \alpha_k d_k) < f(x_k)$, then the polling procedure is stopped, $x_{k+1} = x_k + \alpha_k d_k$, and the iteration is declared successful. Otherwise the iteration is declared unsuccessful and $x_{k+1} = x_k$.

3.1.4 Compute Descent Indicators

This step is skipped if there are not enough previously evaluated points to allow the computation. Otherwise, compute some form of simplex derivatives [18] in order to obtain a quality descent indicator. This descent indicator is the one used on the ordering step.

3.1.5 Update the Mesh Size Parameter

If the iteration was declared unsuccessful, the step size parameter α_k is decreased (default is to halve it). Otherwise, it is kept constant or increased (default is to keep it constant).

3.1.6 Order Directions

Select the positive spanning set $D_k \in D$ to be used in the next iteration (usually contains $2n + 2$ vectors, where n is the problem dimension). If a descent indicator is available, order the directions in the positive spanning set according to the smallest angle with the descent indicator. These directions will be used to compute the poll points. The established order attempts to evaluate first the most promising points in terms of objective function value, increasing the value of an opportunistic approach.

Finally, return to the search step to begin a new iteration, until a stopping criterion is met.

3.2 Available Strategies

Several strategies are implemented in the SID-PSM algorithm, corresponding to different algorithmic variants that may be of interest, depending on the concrete optimization problem. Having more information about the function to optimize allows a customized approach to the problem. These strategies include the use of a cache, the disable of the search step, additional ordering strategies, the use of an opportunistic or complete approach (regarding the poll step), as well as other specific parameters.

In addition to these strategies, it is possible to consider different stopping criteria (based on the final function value, or the number of evaluations/iterations on the step size parameter) and define the update on the step size.

To choose/confirm the options that were adopted as *default* in the code distribution, an initial comparative study was performed, using the number of function evaluations and the computational times as performance indicators.

The algorithmic strategies tested were as follows:

- Opportunistic vs. complete versions for polling.
- Including or not a search step (based on the minimization of polynomial models).
- Use or not of a cache.
- Two different versions of the routine responsible for the computation of the quadratic models, which will be minimized in the search step (*quad_Frob*)

Initially, eight combinations of algorithmic strategies were tested (without the *quad_Frob* variants). After reducing them to the two best versions, the different *quad_Frob* routines were included, obtaining four algorithmic versions to test in the second phase of this process. The version that provided the best results was chosen as *default* for the distributed version of the code.

The original *default* version is characterized by an opportunistic polling strategy: the first poll point found that guarantees a better solution than the current one is immediately accepted as the poll center for the next iteration, and the remaining poll points are not evaluated. However, one could have opted for a complete strategy (*greedy*): all poll points are tested and the one that corresponds to the greatest decrease of the objective function value (if any) is chosen. Thus, part of the work developed was the implementation of this complete version and the evaluation on its performance. The inclusion of this version also supports the validation process of the parallel version, as their results should match.

Regarding the search step, we wanted to verify its effectiveness and test its combination with the other parameters. As observed in [19], versions including a search step are expected to present superior performance regarding the number of function evaluations, especially for smooth problems.

The effectiveness of the cache was also evaluated through the analysis of hits / misses (*hit ratio*) in addition to the aforementioned methods.

Finally, we tested two versions of the *quad_Frob* routine. According to the profiler, more than 90% of the serial execution time (that ignores function evaluation time) is spent on solving a system of linear equations, whose size can depend quadratically on the number of variables. Thus, this is a concerning part for performance optimization. The first implementation is the original one, which resorts to the computationally expensive method of Singular Value Decomposition [25]. The second one applies the *mldivide* MATLAB function, that automatically chooses a different method (expectedly better) depending on sparsity and other characteristics of the system to be solved. This approach seems to present a tradeoff between computational time (faster) and solution quality (slightly worse). If the quality of the solution wouldn't be significantly affected, it could be interesting to include this more economical alternative.

3.3 Benchmarking

For numerical testing, we gathered 27 academic problems proposed by several sources in the literature [9, 16, 26, 32, 33, 43, 46, 47, 58]. These problems were coded in MATLAB and revised by the research team, including 15 smooth and 12 nonsmooth problems. Each problem was considered with different dimensions: usually 6, 10, 20 and 40, sometimes including 30, 50 and 60, depending on relevance and test duration. We intend to retrieve results based on smoothness and for the general class of all problems. Additionally, we tested a real application problem related to the production of *styrene*, with dimension 8.

Two different systems were used to run the tests. The first one is a machine that belongs to the Centre of Mathematics and Applications of FCT NOVA (*Markov*), with 2 Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz (20 cores, 40 threads total), 200GB of RAM and Linux operating system. *Markov* is shared with other users and can be subject to different loads at any time, incurring in time fluctuations that make results imprecise. In a small test where each condition was tested 5 times, we detected mean fluctuations around 12% in a sequential run, without additional load at the time. For this reason, all tests that needed (precise) computational times were run on Azure Virtual Machines on the *Cloud* (mean of 5% variation, which is acceptable). We used Standard Dv3 Machines with different sizes: 4, 8, 16, 32 cpus. Most tests were run in a Linux operating system, except for the real application that was tested on Windows. The MATLAB version used was 2018*b*.

Besides computational time performance, results regarding the number of function evaluations were reported using data profiles. Data profiles, as described in [44], configure a tool for analyzing the performance of Derivative-Free Optimization solvers when there are limitations on the computational budget, which is often the case. Given a set of problems, data profiles compute and graphically display the percentage of problems that can be solved (within a certain small tolerance), inside a given budget of function evaluations. The cost unit is $n+1$ function evaluations, a representative unit for simplex gradient estimates (that require $n+1$ evaluations) and that easily relates to poll step iterations (in the case of SID-PSM, each iteration may cost up to 2 units as we can evaluate up to $2n+2$ points). Analyzing the obtained graphs, it is possible to detect the solvers that work best given a fixed budget of function evaluations, or the percentage of problems that a solver can eventually solve given unlimited budget. Note that time measures have no impact on data profiles.

3.4 Numerical Experience

3.4.1 Code Optimization

Before advancing to the code parallelization, it is recommended to profile and optimize the serial version of the code, tackling the areas that would most benefit performance [2]. Code optimization performed using MATLAB Profiler, a tool that provides a good outlook on functions' computational times and possible improvements.

For logical operators, short circuiting was applied. In this way, the evaluation of a logical expression is finished as soon as its final result is defined. For example, in the case $A\&\&B$, if A is false there is no need to check the logical value of B and thus we can spare some operations [40].

Regarding arrays (either vectors or matrices), MATLAB allows dynamic array growth, meaning that we can always add or concatenate new elements to an existing array. This seems convenient for the programmer but incurs in unnecessary memory overheads: every time we expand an array, we are allocating memory for a new array of increased size and all the elements are copied for the new memory location. This can be particularly harmful when done inside a loop. The solution is preallocation of memory, which can be made either by explicitly creating an array of the intended size or running a loop backwards, since the first element will automatically cause the allocation of the full array. [1, 13, 40]. This was the single aspect that made the most difference in performance, particularly in the *quad_Frob* routine where the gains were very substantial (as can be seen in Figure 3.1).

At last, memory references are also important to point out, in this case the lack of them. MATLAB does not allow explicit referencing. As a safety mechanism to avoid unintended errors coming from passing references to functions, the only case where a reference is passed is when the corresponding variable is not changed by the body of the function. Thus, all auxiliary variables that were not actually needed were suppressed, in order to avoid unnecessary copies [13, 40].

The plot on Figure 3.1 shows the results obtained by running the complete set of problems in the case of enabling the search step. Considerable gains can be noticed, increasing with problem size. In the case of disabling the search step, no differences were observed.

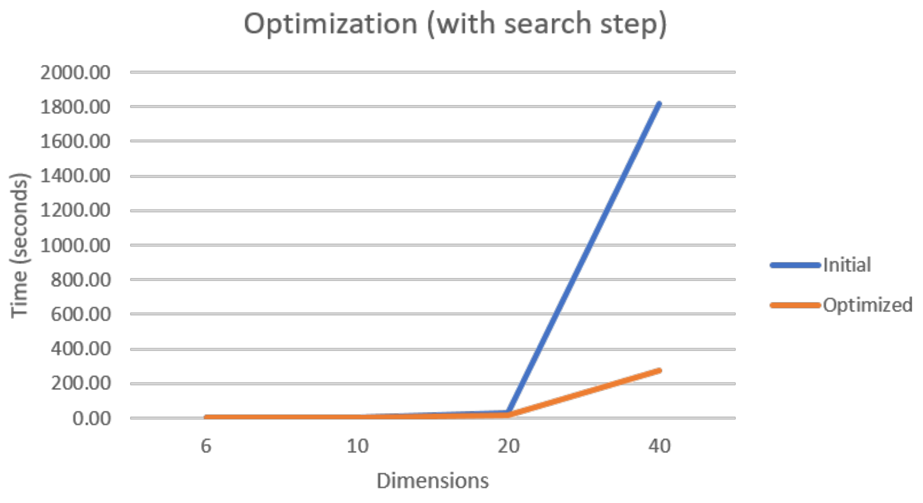


Figure 3.1: Computational time of SID-PSM, when using a search step.

3.4.2 Algorithmic Variants Comparison

Using data profiles [44] and treating every option tested as a different solver, it becomes easy to objectively compare them. To make this approach completely solver independent, it would require knowing in advance the optimal solutions for all problems. Some of them were unavailable in literature. In these cases, we used as solution the best value obtained by any of the solvers, thus maintaining the results comparable. Each data profile was generated in the aforementioned conditions, except now all dimensions were merged, treating each problem with a different size as a different problem (totalizing 108 problems).

As to be expected, complete versions seem to perform badly, while variants considering a search step show a superior performance in both solving problems faster and in the percentage of problems solved. Finally, regarding the use of cache, results seem to be very close. Figure 3.2 depicts the data profiles corresponding to each variant considered.

Additionally, we also tested the percentage of cache hits. In this case, a hit means that one function evaluation was saved by reusing the previous result of an already evaluated point. Our results show hit ratios between 0,69% and 3,14% (see Table 3.1), lowering with the increase on problem dimension. This can be easily explained by the behavior of the algorithm. Each poll step following an unsuccessful iteration will generally have no matches, due to the reduction of the step size. Following a successful poll step, there are only a few points from the previous step that may appear on the new polling set, leading to few matches. Occasionally, points obtained previously by the search step may originate additional matches, with a very low probability. In line with this behavior, the proportion of hits/misses will tend to reduce each time the problem dimensions increase, due to the increase of the number of poll points. There are typically very few hits, but considering the high cost of function evaluations, the use of a cache may still be beneficial.

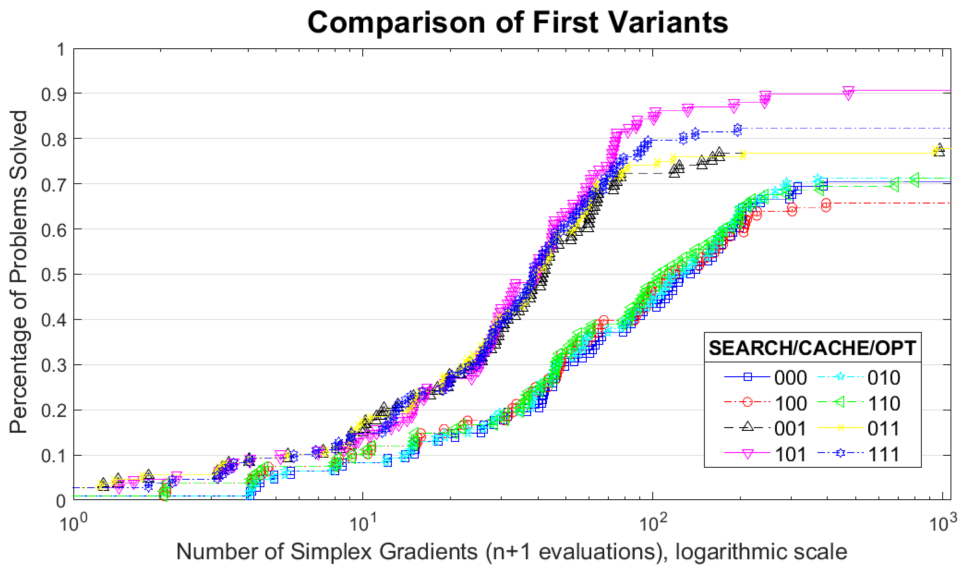


Figure 3.2: Data profile featuring the first eight variants. Legend: SEARCH - 0 absence / 1 presence of the search step; CACHE - 0 absence / 1 presence of a cache; OPT - 0 complete version / 1 opportunistic version.

The behavior of the algorithm was also analyzed independently for smooth and nonsmooth problems. Results for smooth problems are similar to the general case. For nonsmooth functions, deactivating the search step can be a wise choice for small computation budgets (until 80) but afterwards the results are similar.

The two top versions (only difference is cache use) were then chosen for the next phase of the performance assessment of the algorithm and combined with the two variants of the *quad_Frob* routine. The resulting data profile is presented in Figure 3.3.

Focusing on this plot, one can observe that the old *quad_Frob* with cache performs worse than the remaining variants. Notice that this is an important detail as it implies worse final results. The other versions show close curves, meaning either result could be acceptable. Now that this measure is assessed, it is important to understand which option is faster.

In Table 3.1 we present the results regarding each version, averaging all problems on our set by dimension. Except for dimension 6, the new *quad_Frob* versions ($q2$) are significantly faster than the old ones and this difference increases with the increase on the problem dimension.

The last step on this performance assessment was to clarify the role of the cache. We point out that all functions included in this test set have a negligible computational cost (virtually no time).

Recurring to the cache formula:

$$eval\_time \geq cache\_search\_time \times \%hits + (eval\_time + cache\_miss\_time) \times \%misses$$

and isolating the term corresponding to the evaluation time, we get the minimum time of evaluation (in seconds) that justifies the option of a cache (rightmost column on Table 3.1):

$$eval\_time \geq \frac{cache\_miss\_time \times \%misses}{\%hits} + cache\_search\_time$$

Predictably, these rise with dimension and despite the low hit ratios, cache look-ups and misses are almost negligible, leading to very small values of computational time. As we are interested in real application problems that have expensive function evaluations (at least one second), these results indicate that the new *quad_Frob* with an active cache is the wisest choice for the *default* version of SID-PSM algorithm.

After assessing the algorithmic performance of SID-PSM and having optimized the serial version of the code, considering that the dominant cost on the execution of this algorithm is the expensive function evaluations, the poll step becomes the main candidate for parallelization. Its structure is embarrassingly parallel and tackles the bottleneck directly. As MATLAB already applies implicit parallel mechanisms to matrix operations when at least two processors are available, no other worthwhile opportunities were found for this implementation.

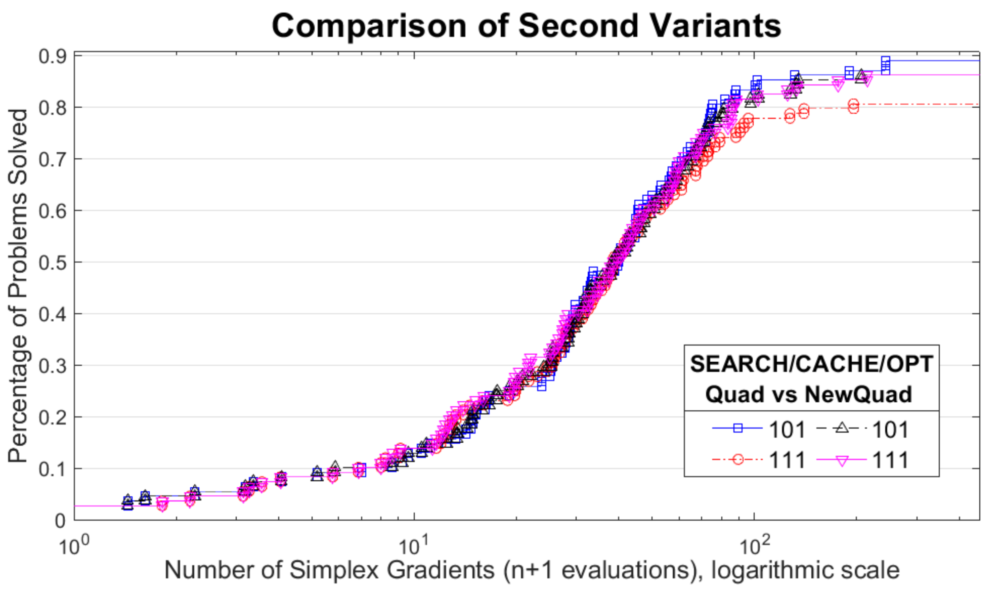


Figure 3.3: Data profile featuring the last four variants. Legend: SEARCH - 0 absence / 1 presence of the search step; CACHE - 0 absence / 1 presence of a cache; OPT - 0 complete version / 1 opportunistic version; the two left curves incorporate the SVD procedure, representing the two right ones the new approach (*mldivide*).

dim	version	time	cache_search	cache_miss	hit_ratio%	min_eval_val
6	101q1	0.66	—	—	—	—
6	**101q2**	**0.18**	—	—	—	—
6	**111q1**	**0.22**	**7.62E-06**	**1.08E-05**	**3.08%**	**0.00035**
6	**111q2**	**0.22**	**7.76E-06**	**1.10E-05**	**3.14%**	**0.00035**
10	101q1	0.97	—	—	—	—
10	**101q2**	**0.53**	—	—	—	—
10	111q1	1.19	7.67E-06	2.18E-05	2.38%	0.00090
10	**111q2**	**0.75**	**8.62E-06**	**2.16E-05**	**1.77%**	**0.00121**
20	101q1	10.80	—	—	—	—
20	**101q2**	**5.42**	—	—	—	—
20	111q1	10.21	1.06E-05	1.11E-04	1.10%	0.00999
20	**111q2**	**4.58**	**1.03E-05**	**1.06E-04**	**1.12%**	**0.00940**
30	101q1	58.66	—	—	—	—
30	**101q2**	**31.91**	—	—	—	—
30	111q1	52.36	1.33E-05	1.02E-04	0.76%	0.01331
30	**111q2**	**27.74**	**1.36E-05**	**9.54E-05**	**0.80%**	**0.01185**
40	101q1	359.45	—	—	—	—
40	**101q2**	**120.29**	—	—	—	—
40	111q1	295.81	1.79E-05	1.17E-04	0.69%	0.01698
40	**111q2**	**132.08**	**1.61E-05**	**1.14E-04**	**0.69%**	**0.01655**

Table 3.1: Computational times and cache hit ratios by dimension. Cache searches and misses represent access times in seconds. Faster versions within each dimension are outlined in **bold**.

Chapter 4

Parallelization of SID-PSM

This chapter covers the process of parallelization of SID-PSM algorithm, featuring the original algorithmic structure (Section 4.1), details on the different schemes considered for parallelization (Section 4.2) and the numerical results obtained (Section 4.3).

4.1 Original Poll Structure

Algorithm 1 corresponds to the original poll step implemented on SID-PSM code. After computing the set of points that should be evaluated (considering the previously established ordering of the poll directions), each point is tested for *feasibility* (*i.e* if it satisfies all the problem constraints, if any), and cache hit. If it is feasible and not in the cache, we proceed to evaluation. The cache is then updated with the point and corresponding objective function value, and success is tested. If an opportunistic strategy is in place, the poll step ends on the first success.

Algorithm 1 Poll Step

```
1:  $success \leftarrow 0$
2:  for $i$ = 1:pointsToEval do
3:      if constrained then
4:          if not feasible then continue
5:      $match \leftarrow search\_cache(x_i)$
6:      if match then continue
7:      $ftemp \leftarrow eval\_point(x_i)$
8:      update_cache($x_i$, $ftemp$)
9:      if $ftemp < f$ then
10:         $success \leftarrow 1$
11:         if opportunistic then break
12: end for
```

This loop is not iteration independent, as updating the cache and evaluating success might raise concurrency issues. Thus, and in order to facilitate the parallel implementation, the serial version was slightly modified. Instead of updating the cache following each evaluation, relevant results (in this case the points, respective objective function values and a logic array to crop irrelevant elements) are stored in memory, allowing the cache to be updated only once in the end of the evaluation process. However, further changes were still required for the parallel implementation (see Section 4.2).

As a final note on the cache use, if it is active, target points are checked before evaluation, otherwise points are always evaluated. Nevertheless, evaluated points are always stored by *default* for computing descent indicators and to be used in the search step, when building the polynomial models.

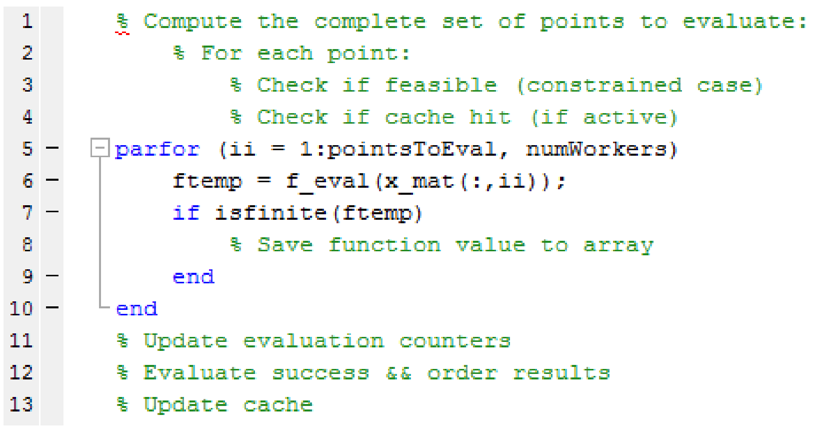
4.2 Parallel Implementation

The parallel implementation is focused on the poll step, which presents an embarrassingly parallel structure. The first step is then to make the cycle iteration independent, while attempting to minimize worker communication. To this end, each worker will receive a point to be evaluated and the objective function. For the complete version of polling, a few global variables and functions (broadcast variables) are also necessary, as well as access to output arrays (sliced output variables). All these variable types are accepted by *parfor*, as seen in Table 2.1. Checking points for feasibility and cache hits one by one may generate unnecessary tasks, in case the point is not evaluated afterwards. Instead, a serial loop will check all points for feasibility and cache hits before proceeding to evaluations, only keeping the ones that will actually be evaluated. Each worker will then evaluate the objective function at the respective point and fill the corresponding parts of the resulting arrays. After the poll step, these arrays will be cropped and reordered for properly storing all results in the cache, at once, exhibiting the same behavior as the serial version. Evaluating poll step success will only be done afterwards, at the end of the current iteration.

Three parallel implementations were created, one relying on a complete polling strategy, other using an opportunistic approach and the last one still considering an opportunistic strategy for polling, but with order guarantees. All three strategies were integrated in the algorithm and can be selected with the respective code parameters.

4.2.1 Complete Polling Version

The complete polling version (see Figure 4.1) is fairly straightforward: *parfor* distributes the points to evaluate to available workers, until all evaluations are performed. As the complete strategy was not chosen as *default* for the serial version of the code, this parallel version was not evaluated. We focused on the other two parallel implementations instead.

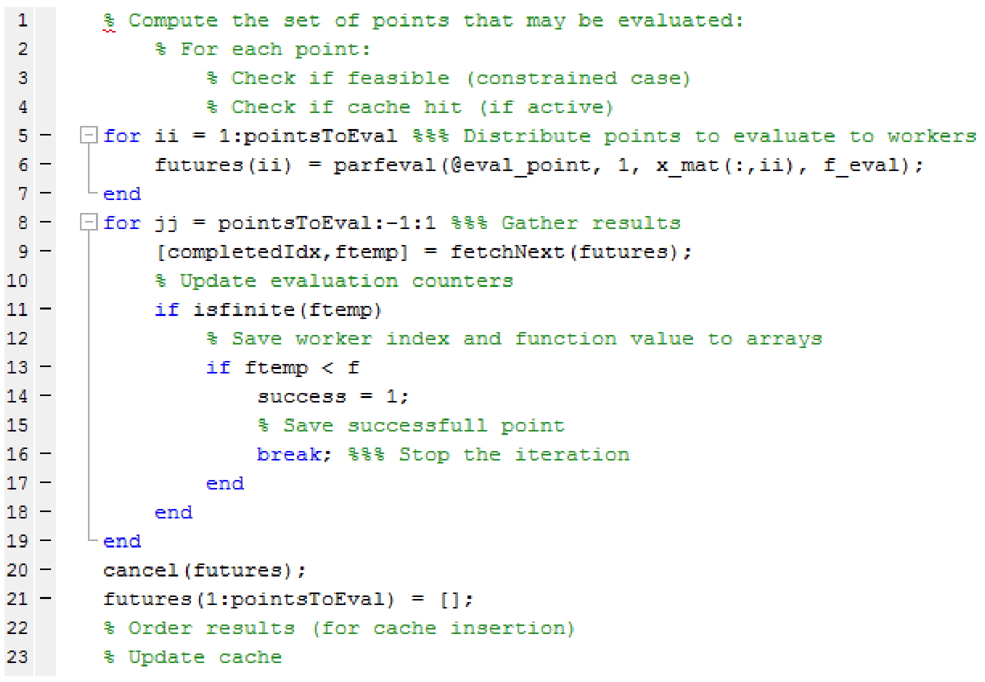


```matlab
% Compute the complete set of points to evaluate:
    % For each point:
        % Check if feasible (constrained case)
        % Check if cache hit (if active)
parfor (ii = 1:pointsToEval, numWorkers)
    ftemp = f_eval(x_mat(:,ii));
    if isfinite(ftemp)
        % Save function value to array
    end
end
% Update evaluation counters
% Evaluate success && order results
% Update cache
```

Figure 4.1: Code snippet with the complete polling version implemented. Comments are also part of the implementation, but the actual code was omitted for shortness.

4.2.2 Opportunistic Polling Version

This version is a bit more complex, due to the inclusion of asynchronous tasks (see Figure 4.2). The first loop (lines 5-7) sends the objective function and each point to the task pool, to be distributed to available workers. Results are fetched one by one, as soon as they are available, justifying the need of a second loop, for retrieving results. *CompletedIdx* represents the index of the task completed, and *ftemp* the objective function value obtained. If the result obtained represents a decrease in the current objective function value, success has been attained and all uncompleted tasks can be discarded. Finally, the loop ends, remaining tasks are canceled and all future objects are cleared.



```matlab
1      % Compute the set of points that may be evaluated:
2          % For each point:
3              % Check if feasible (constrained case)
4              % Check if cache hit (if active)
5  -   for ii = 1:pointsToEval %%% Distribute points to evaluate to workers
6  -       futures(ii) = parfeval(@eval_point, 1, x_mat(:,ii), f_eval);
7  -   end
8  -   for jj = pointsToEval:-1:1 %%% Gather results
9  -       [completedIdx,ftemp] = fetchNext(futures);
10         % Update evaluation counters
11 -       if isfinite(ftemp)
12             % Save worker index and function value to arrays
13 -           if ftemp < f
14 -               success = 1;
15                 % Save successfull point
16 -               break; %%% Stop the iteration
17 -           end
18 -       end
19 -   end
20 -   cancel(futures);
21 -   futures(1:pointsToEval) = [];
22     % Order results (for cache insertion)
23     % Update cache
```

Figure 4.2: Code snippet with the opportunistic polling version implemented. Comments are also part of the implementation, but the actual code was omitted for shortness.

Since the complete polling version is not the code *default*, the opportunistic polling version will be referred further as parallel version.

4.2.3 Opportunistic Polling Version with Order Guarantees

MATLAB documentation refers that tasks generally should be completed by the order in which they were submitted, when using *parfeval* [40]. However, there is no guarantee that this always happens, even with a single worker, as our numerical tests exposed. In rare cases, we observed an order change on the evaluation of the poll points that would conduce to very different final results. This is expected to happen frequently for a higher number of workers, leading to a non-deterministic behavior of the algorithm. Depending on order changes, the algorithm could follow different paths, reaching different final results, both in terms of function value and the total number of function evaluations. Consequently, its behavior is also unpredictable with reference to computational times. Given that our ordering strategy based on descent indicators has proved to be effective on achieving better results [23], we would prefer to keep it in the parallel version. Hence the implementation of this version, which is identical to the opportunistic one, but where results are retrieved by the order they were sent to evaluation, in every cycle. We expect that more evaluations will be completed than in the pure opportunistic version, exposing a tradeoff between speed and quality of the solution for these parallel versions.

The code is very similar to the one of the opportunistic version, except for the ordering process. Every time a result of a function evaluation is received, its order is checked and, if it is out of order, it is placed in a queue in the respective position, waiting for its turn to be processed. When finding a successful point, all results already computed in the queue are discarded alongside the remaining uncompleted function evaluations.

This version will be further referred as parallel_ordered, or simply ordered version.

4.2.4 Debugging

To assess the correctness of the parallel implementations, a simple test was performed: all problems in the test set were solved in parallel, considering only a single worker, and the results respecting to the number of function evaluations, number of iterations and final values obtained were compared to those of the serial version. In the complete polling version, since all points are always evaluated, the order does not matter, making this approach adequate. The opportunistic polling version presented a more challenging case, as results are non-deterministic. The solution found was to retrieve all results at the same time, using *parfevalOnAll*, instead of retrieving one at a time, and process them in the expected order. Regarding the opportunistic polling ordered version, results should match those obtained in the serial run.

4.3 Experimental Results

When a user intends to solve a problem using a parallel implementation, considering the possibility of resorting to cloud services, he needs to decide how many workers to provide to the parallel solver. Thus, it is required to find an "ideal degree of parallelism", limiting the number of workers to a reasonable and expectedly efficient level. A naive approach would consider using a number of workers equal to the number of poll points at each iteration. However, for problems of higher dimensions this might be irrealistic, due to possible limitations in resources, for example in a cloud setting.

In each iteration there are up to $2n+2$ poll points, given a problem of dimension n. If the step size is kept constant for successful iterations and the stopping criterion is based on the step size, there is a fixed number of unsuccessful iterations. For the successful ones, given the good results of the poll ordering strategy, it is possible that most poll steps account only for a few function evaluations (note that an opportunistic strategy is being considered). In this case, it is worth to investigate how many workers will in fact be useful and if there is a predictive rule that we can extract for it, based on the dimension or in the level of smoothness of the problem.

There is also a cost associated to parallelism, as it introduces several overheads. In order to opt for a parallel version, it is necessary that the additional computations have a sufficiently large impact. Thus, it is also important to understand under what conditions parallelism can be valuable. More precisely, for which problem dimension or average function evaluation times do we start benefiting from parallelism?

Answers to these questions correspond to the numerical experiments reported in this section, being a starting point for the GUI development, as the results are included as user recommendations.

4.3.1 Number of Workers

The *default* version selected in Section 3.4.2 was the one used in the numerical tests reported in this subsection. The test battery was run considering dimensions 6 to 50. For each problem, the number of function evaluations per poll step was retrieved. Within each dimension, we computed the average and standard deviation, allowing the adjustment of a linear regression line. Problems were split into smooth, nonsmooth and the complete set (all). Plots were built taking into account two different measures: **Average** (μ) and **Average + Standard Deviation** ($\mu + \sigma$). The latter will be referred to as **upperbound** (see Figure 4.3 and Table 4.1 for the original data). The upperbound version can be particularly interesting as, assuming a Gaussian distribution for the number of evaluated poll points, it ensures that for around 84% of the cases the number of processors provided will be enough to evaluate all poll points at a given iteration (depending on the rounding rule applied).

In both approaches, especially when the standard deviation is added to the average

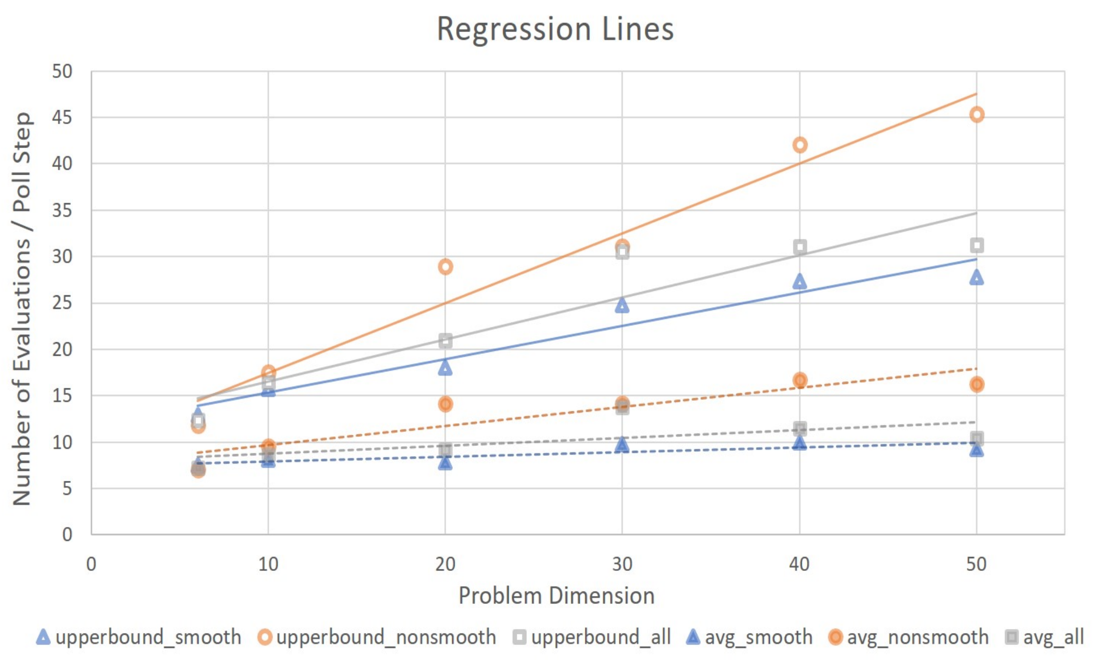


Figure 4.3: Number of function evaluations required by the poll step - average (μ) and upperbound ($\mu + \sigma$), with corresponding regression lines. Dashed lines indicate the average condition.

value, the nonsmooth case requires a considerably higher number of function evaluations per poll step. This might reveal that the descent indicators (computed for exploring first the most promising directions) are more effective on the smooth case (corroborating results described in [18, 23]). As a consequence, a higher number of workers will be required for solving nonsmooth problems, when compared to the smooth case. Table 4.2 reports the results of the linear regression adjustments.

Despite the good adjustment of the regression lines, when considering the approach that adds the standard deviation to the average value (as will be discussed afterwards), the lower dimensions (6 and 10) seem to be somewhat biased in these estimations. For dimension 6, the line indicates 14-15 recommended workers, for a maximum of 14 evaluations in the poll step. For higher dimensions, the ratio between the maximum number of function evaluations at the poll step and the number of required workers increases gradually. For dimension 50, around 35 workers would be required (in the general case), which is about 1/3 of the possible 102 poll evaluations.

The *R-squared* value is a measure between 0 and 1 that estimates how well our regression lines fit the data provided. A value close to one indicates a good linear fit, implying a linear scalability regarding the problem size. The values obtained for the average approach range from 41%-83%, while when considering the upperbound, values are between 88%-96%, depending on the case (see Table 4.2). For the upperbound condition, the obtained R-squared values state the quality of the regression lines as reliable predictors of the number of function evaluations per iteration. However, considering that the number of workers to be recommended needs to be a power of two, we kept both approaches in the analysis, always rounding up the average condition.

Dimension	**6**	**10**	**20**	**30**	**40**	**50**	**Condition**
Smooth	7	8	8	10	10	9	Average
	13	16	18	25	27	28	Upperbound
Nonsmooth	7	10	14	14	17	16	Average
	12	17	29	31	42	45	Upperbound
All	7	9	9	14	11	10	Average
	12	16	21	31	31	31	Upperbound

Table 4.1: Average (μ) and upperbound ($\mu + \sigma$) values considered for the linear regression adjustments.

Average

	Smooth	**Nonsmooth**	**All**
Regression Line	0.05x + 7.42	0.21x + 7.64	0.09x + 7.90
R-Squared (R^2)	0.65	0.83	0.41

Upperbound

	Smooth	**Nonsmooth**	**All**
Regression Line	0.36x + 11.78	0.75x + 9.91	0.45x + 11.97
R-Squared (R^2)	0.94	0.96	0.88

Table 4.2: Regression lines and R^2 obtained.

4.3.2 Parallel vs. Serial Approach

For this evaluation we selected two representative problems of the test set, one smooth and another nonsmooth. Despite different problems showing variations on serial execution times due to their particular structure, the large computational times associated to function evaluations would make the serial part of the program irrelevant, what justifies disabling the search step, as it accounts for most of the serial time. We note that these problems are part of an academic test set, thus the corresponding function evaluation times are insignificant (approximately zero).

The numerical study considered the problem dimensions (ranging from 6 to 60), different numbers of workers (powers of two from 2 to 16) and the two parallel opportunistic versions. Results are shown in Table 4.3 and allow us to compare the time for the three program versions, when the time for objective function evaluations is irrelevant. This puts in evidence the overhead induced by the parallel framework.

Notice that both parallel versions presented identical results and these remain constant, independently of the number of workers, suggesting that the parallel overhead observed does not depend on this number. In addition, the total number of function evaluations performed and the final values obtained were practically the same in all three versions enforcing the coherence of the retrieved execution times.

	Number of Workers				
Dimensions	**2**	**4**	**8**	**16**	**Version**
	0.30	0.30	0.30	0.30	serial
6	2.68	2.32	2.39	2.33	parallel
	2.70	2.38	2.36	2.39	par_ord
	0.07	0.07	0.07	0.07	serial
10	5.63	4.61	4.81	4.41	parallel
	5.33	4.60	4.66	4.64	par_ord
	0.41	0.41	0.41	0.41	serial
20	15.16	14.07	14.10	13.77	parallel
	15.38	14.40	14.13	14.48	par_ord
	2.86	2.86	2.86	2.86	serial
30	44.56	40.67	42.10	63.23	parallel
	45.43	41.81	40.75	41.66	par_ord
	11.57	11.57	11.57	11.57	serial
40	65.40	60.49	64.28	60.96	parallel
	67.93	60.83	63.01	61.67	par_ord
	33.01	33.01	33.01	33.01	serial
50	124.04	111.38	110.96	111.53	parallel
	125.58	111.79	112.19	120.53	par_ord
	59.02	59.02	59.02	59.02	serial
60	176.97	156.88	156.97	155.72	parallel
	175.39	154.55	159.97	168.49	par_ord

Table 4.3: Comparison of serial and parallel computational times (overheads), with deactivated search step.

As stated previously, the goal here was to define the point where we start benefiting from parallelism. The idea was to measure the overheads arising from parallel executions and grasp on a minimum function evaluation time that would make those worthwhile. To this end, the following formula was employed in the computations:

$$time_{par} + \frac{eval\_time \times num\_evals}{num\_workers} \leq eval\_time \times num\_evals + time_{seq}$$

Leading to:

$$eval\_time \geq \frac{(\text{time}_{par} - \text{time}_{seq}) \times num\_workers}{num\_evals \times (num\_workers - 1)},$$

where time_{seq} and time_{par} represent the serial and parallel running times observed, excluding the fraction for objective function evaluations.

In Table 4.4 we display the minimum function evaluation times computed. For convenience, these results concern only the opportunistic ordered version, since they were indistinguishable for both parallel versions.

	Number of Workers			
Dimensions	**2**	**4**	**8**	**16**
6	0.014	0.009	0.007	0.007
10	0.017	0.010	0.009	0.008
20	0.022	0.014	0.011	0.011
30	0.035	0.021	0.018	0.017
40	0.036	0.021	0.019	0.017
50	0.050	0.029	0.025	0.025
60	0.062	0.034	0.031	0.031

Table 4.4: Minimum function evaluation times required for parallel gains, obtained by dimension.

Even for the higher dimensions, the biggest time we uncovered was 0.062. This is a very small time in comparison to those expected on real application problems. In conclusion, results emphasize the importance of employing parallel strategies to improve the performance of the SID-PSM algorithm. The practical effectiveness of these strategies will be covered in the next subsection.

4.3.3 Parallel Gains

After understanding for which cases the parallel approach represents an improvement of the numerical performance of the algorithm, the final step was to measure the effective gains obtained on problems similar to those we intend to solve. Additionally, we aimed to test the value of our recommendations regarding the number of workers to be considered, and to enrich the test on which of the two conditions would provide a most cost-effective solution (average or upperbound).

To this end, two sets of experiments were conducted, comparing serial with parallel (opportunistic) executions. The first one respected to the complete test set. Although these are only academic problems, we aimed to approximate our target class as best as possible. These functions were treated as blackbox and time delays were added to account for expensive function evaluations. Results were split between smooth and nonsmooth cases, possibly with different recommendations. The second test respected to a real application problem from chemical engineering, related to *styrene* production and first mentioned in [4].

4.3.3.1 Test Setup and Performance Metrics

As it would be too costly (money and time-wise) to test all academic problems for the intended dimensions, including the time delays, a different approach was followed. Every problem was run 10 times for each condition (number of workers and dimension), with no time delays. The number of parallel cycles was accounted for (within each iteration) in the following way: 0 or 1 during the search step (most times does one evaluation, which does not allow any parallel gain) and $\frac{numEvals}{numWorkers}$ (rounded up) during the poll step, depending on the number of workers in use.

With this information, it was possible to estimate execution times for every problem, given different delays. Results were averaged over the 10 runs. All results presented in this section feature time delays including 0.1 and the range of 1 to 32 seconds, growing in powers of two. Intermediate delays might be omitted to reduce redundancy.

As mentioned before, the sequential part of the algorithm is itself time-consuming, due to lengthy procedures that could not be parallelized. The execution time of the parallel part is directly related to function evaluation times. This means that increasing the time delays associated to function evaluations allows to decrease the impact of the sequential part on the overall time, achieving higher speedups. We include 95% confidence intervals to account for variability, in cases where it can be of interest. Both parallel versions were tested: parallel and ordered.

The procedure to select the number of recommended workers was applied in the following way. For the upperbound condition, results given by the previously estimated regression lines (see Section 4.3.1) were rounded to the nearest power of two. For the average condition, results from the regression lines were always rounded up, to the nearest power of two, in an attempt to account for variability. The differences between the recommendations obtained with each one of the two conditions grow with the problem dimension, making the comparison more interesting for higher dimensional orders. Table 4.5 shows the problem dimensions and the number of recommended workers obtained, that were considered for our tests.

In the remainder of this section, results are presented for a test set comprising 108 problems, with dimensions between 6 and 40. These 108 problems correspond to the 27 problems in our academic test set, run with the five dimensions considered. Label *rec* respects to results computed with the number of workers recommended based on the regression lines, while for rec_{-1} and rec_{+1} results are computed with the previous and next powers of two, respectively. The goal is to test if the recommendation that the toolbox provides guarantees a good performance without compromising too much on efficiency, for a cost-effective solution. When considering dimension 6, only 14 evaluations are performed in each poll step, therefore using more than 16 workers would be totally unnecessary. For coherence with the strategy adopted to report the results, Figure 4.4 and Table 4.6 both use 8 workers as the recommended value for dimension 6, instead of 16 (hence the asterisk * in Table 4.5 for these recommendations). All other results use 16 as recommended, since only the recommendations (middle values) are considered.

Notice that in Table 4.5, when the number of recommended workers is 32, we would need to run the problem set with 16, 32 and 64 workers. However, economical and technical reasons related to cloud use, prevented us from using 64 workers. Thus, as it wouldn't be possible to obtain the three results, we opted to exclude these problems from the test set used for Figure 4.4. This test set comprises then 81 problems, with dimensions between 6 and 20 (dimension 40 was excluded). Comparing plots between upperbound and average conditions is unnecessary, since they are equivalent. For completeness and because results for higher dimensions are also relevant since the need for parallelization should increase, Table 4.6 reports execution referring times only for the recommended values (middle values) and the test set is complete. All times displayed are expressed in hours.

Speedup charts include our complete sample, since they were computed with the number of recommended workers. Speedup values shown represent the average speedup among all included problems (Figure 4.5) and the same applies to efficiency (Figure 4.6). These results were also analyzed distinguishing the smooth and the nonsmooth cases, comprising the different recommendations (Figure 4.7). An additional table was included (Table 4.7), allowing the comparison between speedups when the number of recommended workers varies (only dimension 40).

When testing the parallel version that does not preserve the poll ordering for average and upperbound strategies, we noticed that the quality of the solution was affected. The previously established poll order changes often, leading to different paths and, consequently, the final result obtained might be different, as a consequence of the stopping criterion or of a different local minimum being found. Sometimes a better solution was reached, others its quality decreased. This collection of problems is not typically used for global optimization, so problems are expected to have few local minima, if any. For real applications, where there are no indications on the total amount of local minima, this can have a greater significance. The same effect was also verified in the *styrene* problem, with considerable (negative) impact (more details in the Subsection 4.3.3.3). This was the motivation to create an ordered version, that executes in parallel and yet respects the ordering strategy defined by the serial implementation of the original algorithm. In this way we ensure that the final result computed is the same amongst different runs and equivalent to the one obtained for the serial version.

To evaluate the quality of our final solution, we adopted a specific metric that allows for comparison between the solutions of parallel and serial executions. The following formula was used:

$$\frac{\text{Res}_{Par} - \text{Res}_{Seq}}{\max\left(1, \left|\text{Res}_{Seq}\right|\right)}, \tag{4.1}$$

where Res_i stands for the final computed value of algorithmic version i. Positive values show a detriment on the quality of the final solution, by comparison with the serial version. The result of this metric was intended to be given in percentage, representing a relative variation by problem, given the variability on the orders of magnitude of the final values that can go from 10^2 to 10^{-20} (substantially lower than the precision of our stopping criterion based on the step size - 10^{-5}). In order to address this problem, we used the function *max(1, •)* in the denominator. In this way, the metric represents an absolute error when Res_{Seq} is smaller than one, corresponding in the remaining cases to a relative variation. The use of the factor *1* is definitely arguable, because it directly relates to the type of error that our metric addresses. If we are not interested in a very high precision, this strategy holds, justifying our choice. Reporting only average values for the indicator, by aggregating the results, always incurs in loss of information. Presenting results for all instances (problem and dimension) would be exhaustive. Even so, we report the best and worst results obtained for this metric, in addition to the average values (see Table 4.8), for each of the 10 runs performed. Gain and loss ratios represent the percentage of problems where the final function value computed was better or worse than the one of the serial run, respectively.

Dimension	Smoothness	Average	Upperbound
	S	8	16
6	**NS**	16	16
	ALL	16 *	16 *
	S	8	16
10	**NS**	16	16
	ALL	16	16
	S	16	16
20	**NS**	16	32
	ALL	16	16
	S	16	32
40	**NS**	16	32
	ALL	16	32

Table 4.5: Recommendation on the number of workers to be used for the problems considered, given average and upperbound strategies.

4.3.3.2 Results on the academic test set

Analyzing the results from Figure 4.4, it is clear the advantage of both parallel versions with the increase of the computational time associated to function evaluations. However, for the parallel version (without ordering guarantees), the marginal gain from rec_{-1} to rec is more meaningful than the one from rec to rec_{+1}. This is the expected behavior: by increasing the number of workers, we will always achieve gains in terms of computational time, but these gains will be smaller each time. Our recommendation on the number of workers to use seems to be an adequate cost-effective compromise. For the ordered version this also verifies, although it is somewhat surprising that rec_{+1} takes more time than rec. This happens because of differences on the final solution computed when running the same problem with a different number of workers. The algorithm is following different paths and a lot more parallel cycles are completed for the rec_{+1} case than for rec, explaining why it takes more time.

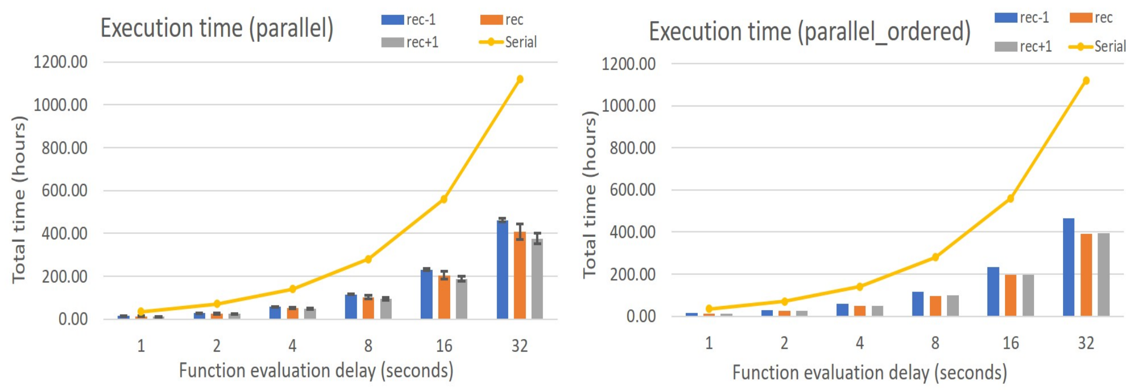


Figure 4.4: Aggregated time results for the academic test set, comparing both parallel versions. The number of workers is abstracted in each bar, according to the recommendations. Black lines represent 95% confidence intervals.

Relevant time gains are noticeable in comparison to the serial version. Intervals for a 95% confidence level show slight time variations among runs for the parallel version. The ordered version reveals constant times (except for small machine fluctuations).

Version	**Condition**	**Time delay**							**Evaluations**
		0.1	**1**	**2**	**4**	**8**	**16**	**32**	
Parallel	average	7.71	38.11	71.89	139.45	274.57	544.80	1085.26	539618
	upperbound	7.30	35.57	66.99	129.83	255.50	506.85	1009.54	524850
Par_ord	average	6.15	33.57	64.03	124.96	246.82	490.53	977.96	480642
	upperbound	6.42	30.78	57.86	112.00	220.29	436.87	870.04	459960
Serial	–	13.74	177.44	232.66	463.11	923.99	1845.77	3689.32	414799

Table 4.6: Aggregated execution times (in hours) and number of function evaluations performed, given both recommendations for defining the number of workers.

Table 4.6 details the values obtained for execution times. Notice that a significant time reduction is achieved even when only considering a 0.1 second delay in function evaluations. We notice that we had previously estimated that parallel gains would be considerable for a function evaluation time above 0.1 (see Subsection 4.3.2). Unexpectedly, the parallel version takes more time than the ordered one, what can be explained by the higher number of evaluations performed, due to different paths taken by the algorithm. Comparing the two recommendation conditions for defining the number of workers to be used, in the majority of cases upperbound outperforms average, as expected, since it uses more workers.

Time delay	**0.1**	**1**	**2**	**4**	**8**	**16**	**32**
average	2.97	5.50	5.83	6.01	6.11	6.16	6.19
upperbound	4.04	7.92	8.44	8.74	8.89	8.97	9.01

Table 4.7: Average speedup for problems of dimension 40.

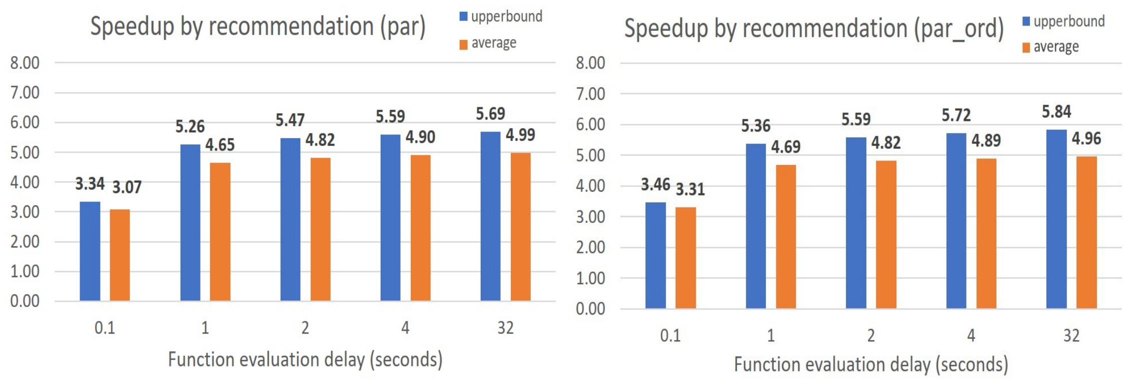


Figure 4.5: Average speedup among all problems, given both recommendations / parallel versions and different time delays for function evaluations.

In agreement with the previous computational time results, speedups also indicate interesting gains (over 3) for a delay of 0.1 seconds and a slightly better overall performance of the ordered version (Figure 4.5). Speedup grows substantially from 0.1 to 1 seconds of time delay, being the subsequent increases less significant. Once more, these results justify the need for a parallel solution. The recommendation on the number of workers to use based on the upperbound condition outperforms the one based on the average, as expected. For problems of dimension 40, where the recommendation is different (Table 4.7), the one based on the upperbound achieves a much higher performance, gaining almost 50% for a 32 second delay on function evaluations. Even at the cost of some efficiency, this represents a significant increase in performance. Overall efficiency is stable across the two recommendations and parallel versions (see Figure 4.6). However, it reveals low values (around 30%). This is due to the opportunistic polling strategy, jointly with the effective ordering. The parallel gains depend on the number of function evaluations performed within each cycle, which will usually be a low value if the ordering strategy works properly and opportunistic polling is considered. Also the execution time between the serial and parallel parts of the algorithm may fluctuate. Speedup and efficiency will then depend on the structure of the problem, constraining our measures to lower limits than those we would usually expect for parallel strategies.

When separating speedup by level of smoothness of the problems, an interesting effect can be observed: speedup is much larger for nonsmooth problems than for smooth ones (see Figure 4.7). Two factors contribute to this effect, being the first related to our test set. A small number of our nonsmooth problems perform only a few iterations, being the majority of these iterations unsuccessful (so the algorithm can finish early). This means that the corresponding polling cycles are complete (maximizing the number of evaluations / cycle). This fact will highly increase the speedup of these problems, resulting in larger speedup values for nonsmooth functions. The second factor is related to the algorithm itself. As observed in Section 4.3.1, the ordering strategy is found to perform better for smooth problems (decreasing the number of evaluations / cycle).

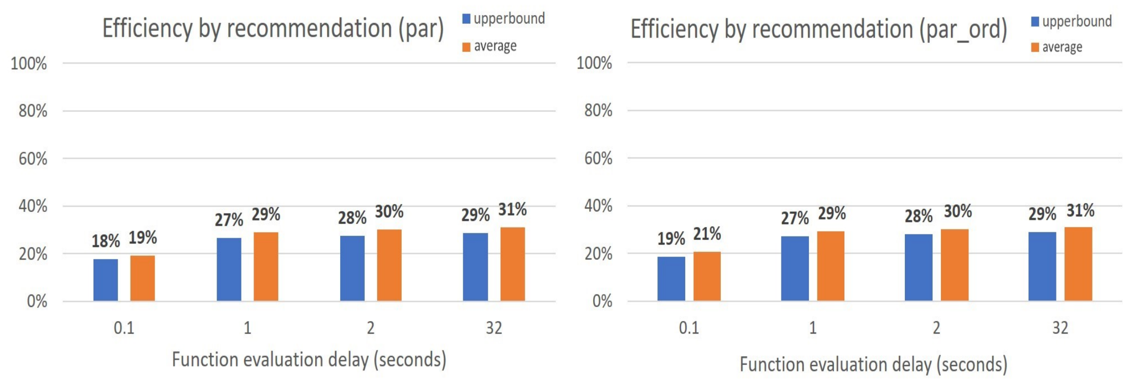


Figure 4.6: Average efficiency among all problems, given both recommendations / parallel versions and different time delays for function evaluations.

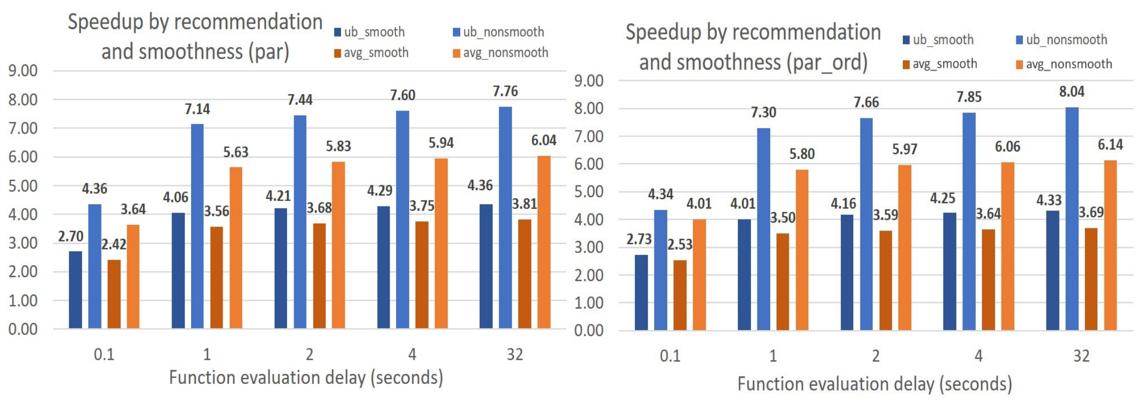


Figure 4.7: Average speedups split by smoothness class.

Additionally, the search step performs very well in smooth problems, which leads to comparatively less poll steps performed than for nonsmooth functions, consequently increasing the sequential part of the algorithm. These results are again consistent across both recommendations and parallel versions.

With all the analysis performed, we are finally able to decide on the strategy regarding the recommendation of the number of workers to use, that will integrate the GUI to be developed. Our goal is to maximize the achieved speedup without losing too much on efficiency, for a cost-effective solution. The upperbound condition seems a good fit since the efficiencies analyzed are very close, with significant speedup gains. Also, the statistical bound provided by this condition is of practical interest, offering a good level of parallelism on each cycle (84% of function evaluations are covered). The expected gains should also be more consistent, since using the average always implies some variability. To conclude, we have chosen the upperbound as our recommendation method for a cost-effective number of workers to be used.

Concerning the quality of the final solution, it was necessary to evaluate the values obtained by the parallel version, since the order changes have impact on the results (see Table 4.8). Gains and losses seem to be well distributed across problems, with around 33%

Run		1	2	3	4	5
Best case	Value	-99.96%	-99.96%	-99.99%	-81.62%	-100.00%
	Problem	*activefaces*	*activefaces*	*activefaces*	*activefaces*	*activefaces*
	Smoothness	ns	ns	ns	ns	ns
	Dimension	20	20	20	20	20
Worst case	Value	36.29%	36.29%	36.29%	86.75%	36.29%
	Problem	*broydn3d*	*broydn3d*	*broydn3d*	*activefaces*	*broydn3d*
	Smoothness	s	s	s	ns	s
	Dimension	20	20	20	10	20
Average		-0.54%	-0.54%	-0.55%	0.20%	-0.48%
Gain ratio		33.33%	33.33%	32.41%	36.11%	28.70%
Loss ratio		32.41%	32.41%	31.48%	29.63%	36.11%

Run		6	7	8	9	10
Best case	Value	-49.61%	-100.00%	-99.68%	-99.99%	-100.00%
	Problem	*activefaces*	*activefaces*	*activefaces*	*activefaces*	*broydn3d*
	Smoothness	ns	ns	ns	ns	s
	Dimension	20	20	20	20	10
Worst case	Value	72.63%	93.86%	16.68%	36.29%	7.83%
	Problem	*activefaces*	*activefaces*	*chaincrescentII*	*broydn3d*	*problem19*
	Smoothness	ns	ns	ns	s	ns
	Dimension	10	10	40	20	10
Average		0.31%	0.44%	-1.03%	-0.83%	-1.22%
Gain ratio		32.41%	30.56%	32.41%	35.19%	38.89%
Loss ratio		33.33%	37.96%	36.11%	33.33%	29.63%

Table 4.8: Results on the quality of the final solution computed for different runs. Each run aggregates all problems and dimensions. The number of workers was selected using the upperbound condition.

on each case (the remainder respects to the percentage of problems where the results are the same). The average value of variation ranges from -1.22% to 0.44% among the several runs. Best and worst cases suggest that some specific problems have the most impact on the average result, and it is essentially the same problem for the best case (*activefaces*, dimension 20). For the worst case, there is more variability between problems, with *broydn3d* appearing the majority of times. No relation could be established between the quality of the final solution and problem dimension. As as example, problem *activefaces* shows a big gain for dimension 20 and occasionally a big loss for dimension 10.

Respecting the choice between parallel versions, results are still inconclusive. Testing a real application problem is necessary and encloses more practical value.

4.3.3.3 Results on a real application problem - *styrene*

As aforementioned, this is a real chemical engineering problem that simulates the production of styrene, described in [4, 7]. The process involves four steps: preparation of reactants, catalytic reactions, a first distillation - where styrene is recovered, and a second distillation - where benzene is recovered. During this second distillation, unreacted ethylbenzene is recycled as an initial reactant on the process. A numerical simulator of this chemical process has been built, one function evaluation corresponding to a complete simulation. Due to the presence of recycling loops like the one of ethylbenzene, the complete process has to be simulated until the final result is provided. The time to get an evaluation often fluctuates, ranging from 10^{-5} to 30 seconds (according to our tests), with an average of 4.5 seconds. The presence of costly and blackbox function evaluations configure this problem as representative of our target class.

The problem has 8 variables (subject to lower and upper bounds), related to the industrial process, and 11 constraints, some process-related (*e.g* environmental norms regarding excesses) and some economical (*e.g* investment value). The main goals were to maximize the net present value (f_1), the purity of produced styrene (f_2) and the overall ethylbenzene conversion into styrene (f_3) [7]. However, since we are working on single objective optimization, we followed the approach used in [4]: to add f_2 and f_3 as constraints imposing minimum lower bounds and consider $-f_1$ as our single objective function (SID-PSM minimizes problems). Further description and a scheme of the production process can be found in [4].

Since serial runs are deterministic (considering the total number of function evaluations and the final solution) and time variations are meaningless in the cloud machines, only one serial run was performed. The problem was run 30 times for each of the two parallel versions evaluated (ordered and parallel) and different number of workers. This larger sample provided us more robustness on the statistical analysis performed.

The presented results show execution times (Figure 4.8), information on worst/best/average runs (Figure 4.9), final function value obtained (Figure 4.12) and total number of function evaluations (Figure 4.10). Concerning the evaluation of parallel gains, speedup and efficiency were also reported (Figure 4.11).

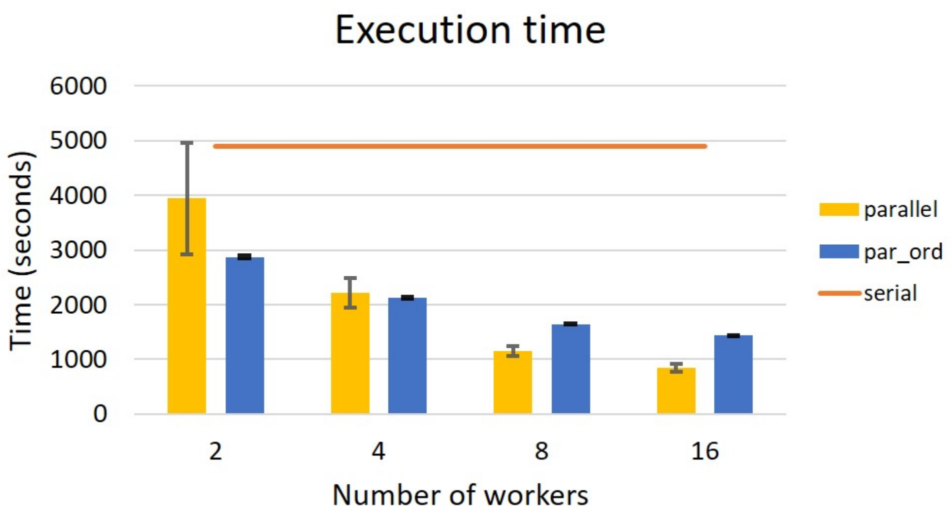


Figure 4.8: Average runtime given different numbers of workers. Black lines represent 95% confidence intervals based on our sample of 30 runs.

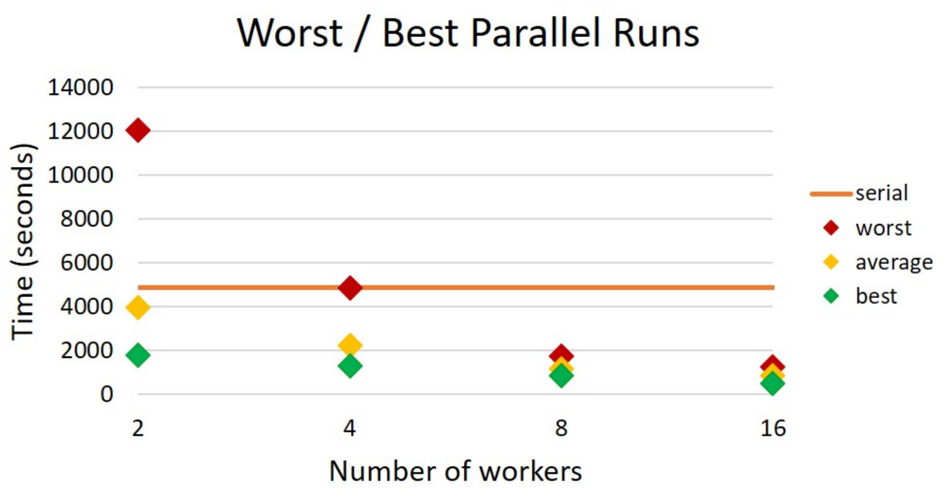


Figure 4.9: Best, worst and average runtimes for the parallel version, considering a sample of 30 runs.

In Figure 4.8 we can observe that the ordered version performs well for a small number of workers (2,4), but the decrease in time gains is progressively smaller. On the other hand, the parallel version is outperformed for low numbers of workers, but achieves much higher gains with their increase (8 and 16). The results regarding 2 and 4 workers are surprising, especially considering that the parallel version performs less evaluations than the ordered one (see Figure 4.10). This can be explained by the variability among function evaluation times, which leads to some outliers in different runs. Higher numbers of workers seem to mitigate this effect, leading to more stable overall execution times, which enforces the idea that a high number of workers is preferable on real application problems. Note that this problem has dimension 8, so each poll step consists on $2n + 2 = 18$ function evaluations. Since the parallel version allows order changes on the evaluations, the algorithm may follow very different paths, possibly performing more or less evaluations and incurring in different execution times. Figure 4.9 illustrates these variations. We can see that there is a great difference between best and worst runs. This difference is mitigated by the increase on the number of workers. However for 8 and 16 workers, the worst run still requires more than twice the time needed for the best run (see Table 4.9).

Number of workers	Best run	Average run	Worst run
2	1779	3946	12050
4	1310	2221	4840
8	849	1150	1745
16	480	844	1273

Table 4.9: Execution times for best, worst and average runs of the parallel version. Time is expressed in seconds. The serial version has an execution time of 4893 seconds.

Concerning function evaluations (Figure 4.10), both versions increase their numbers with the increase of the number of workers. We measure completed evaluations (instead of initiated), so naturally with more workers available we end up with more completions, despite their success. The ordered version starts the same as the serial one (2 workers) and has big increases, especially for 16 workers. This is to be expected, since we wait for evaluations to be completed in the exact order they are assigned. In the meantime, several other function evaluations can be completed. The parallel version shows smaller numbers of function evaluations, possibly due to shorter paths till termination. Note that the final value computed may change with different numbers of workers, also adding some more variability to all measures (see Subsection 4.3.3.4 below).

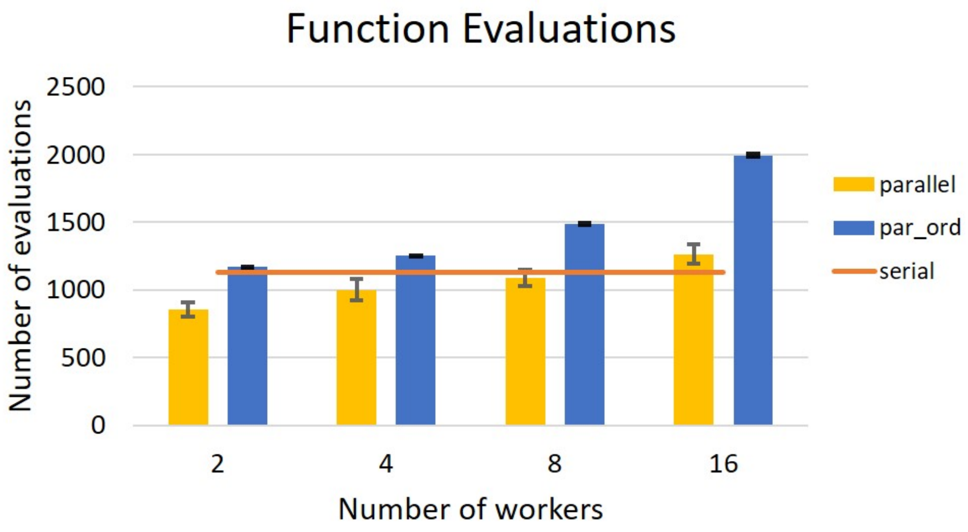


Figure 4.10: Average number of function evaluations performed. Black lines represent 95% confidence intervals based on our sample of 30 runs.

Figure 4.11 displays speedup and efficiency obtained for both versions. These values were computed for each of the 30 runs and then averaged. Although results are equivalent for 2 and 4 workers, for higher numbers of workers the parallel version clearly outperforms the ordered one, both in terms of speedup and efficiency, achieving acceptable levels even for 16 workers. Note that 16 would be our recommended number of workers to address this problem, based on the upperbound regression line. For the parallel ordered version, interesting speedups are achieved for 2 and 4 workers, increasing slowly over the number of workers, along with a great loss in efficiency. Again, this might be explained by two factors: the opportunistic polling strategy and the effectiveness of the poll ordering. As the algorithm always stops right when success is found, we limit the power of parallelism. If a complete strategy was in place, speedup would increase since we would always evaluate all poll points. Given the adoption of these strategies, it wouldn't be possible to aim for a linear speedup, or even estimate a theoretical one.

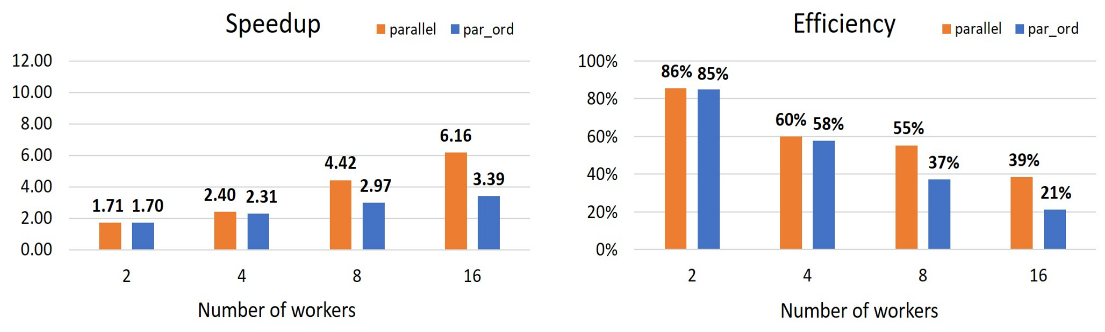


Figure 4.11: Average results on speedup and efficiency obtained for different numbers of workers, considering a total of 30 runs.

A comparison between the final values computed by each version, is shown in Figure 4.12. The ordered version gives us the same result as the serial version, the best result we were able to produce. However, for the parallel version the results obtained were poor (along with some variation). A tradeoff between speed and the quality of the final result seems to occur (in the previous subsection this was not so clear).

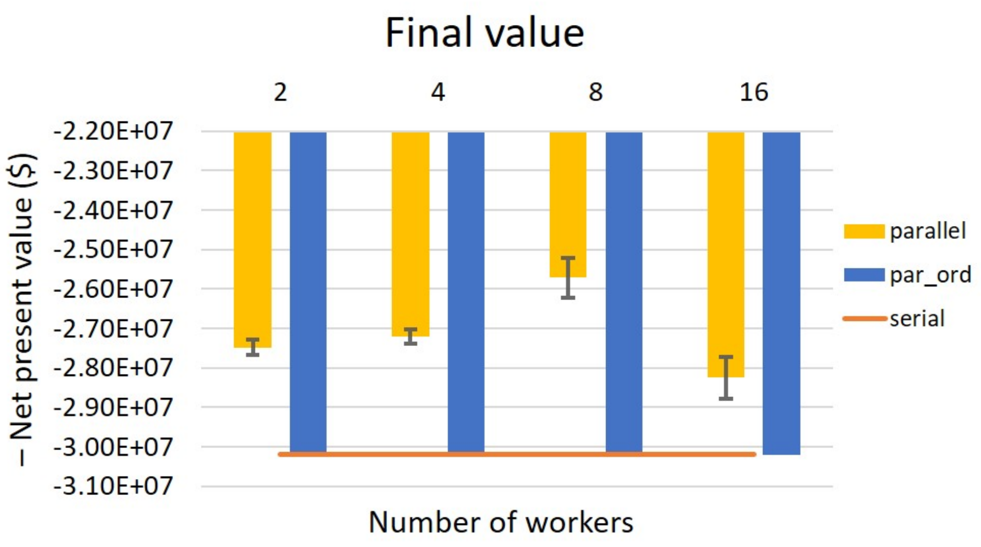


Figure 4.12: Average final value computed (note that the problem is addressed as a minimization one). Black lines represent 95% confidence intervals, based on our sample of 30 runs.

Table 4.10 reports the percentage difference between the final value obtained in the parallel and serial runs, resorting to the metric defined in equation (4.1). Result fluctuations are captured within 95% confidence intervals. Although a similar analysis showed almost no variation for the academic set of problems, a great loss of quality is observed here, when using the parallel version. Depending on the number of workers, our final computed value is between 4.66% to 16.52% worse than the one of the serial version. Additionally, all runs of the problem produced worse solutions.

Number of workers	Final value variation
2	8.31 - 9.65 %
4	9.26 - 10.50 %
8	13.14 - 16.52 %
16	4.66 - 8.22 %

Table 4.10: Final value variation by number of workers, calculated using equation (4.1). A 95% confidence interval addresses fluctuations.

After evaluating the parallel gains obtained on a real application problem, we are finally able to select our *default* version. Even if these tests only comprised a problem, this is more representative of the class of problems that we intend to address. We observed that the parallel version performs much faster, but has the unintended property of compromising the quality of the solution obtained. Our goal would be to improve the performance of the algorithm with a minimum of quality loss. For this reason, the ordered version seemed to be a good compromise between quality and efficiency and was set as *default* in our code distribution. The parallel version is also provided as an option.

4.3.3.4 Final note

While running these tests we noted inconsistencies between serial runs that raised some question about the determinism of the algorithm. This is especially meaningful when working with parallel strategies, since it may have implications on the expected behavior. We already anticipated that the parallel version would be non-deterministic, but both the serial and the ordered version were supposed to be deterministic. Through several specific small tests, the following became clear: different serial behavior should be expected between different MATLAB versions; different machines or operating systems have no impact on the result, given the same MATLAB version. However, results change with the number of workers available, what we did not expect. After a comprehensive search and some additional tests, we could understand that the *quad_frob* routine, and in particular MATLAB *mldivide* routine, both used on the search step, were incurring in small variations due to precision issues. MATLAB programs already take advantage of available workers to speedup programs through implicit parallelism mechanisms. In this case, the result of a search step could be slightly changed, leading to a different path and, consequently, to a different result. This is true for all versions, including the ordered one, and serial executions, when changing the number of workers available. For consistency, we always force serial runs to use a single worker. Considering the ordered version, this effect wasn't noticeable in the *styrene* problem, and had a slight influence on times and evaluations in the test battery, as previously discussed.

Chapter 5

BoostDFO Toolbox

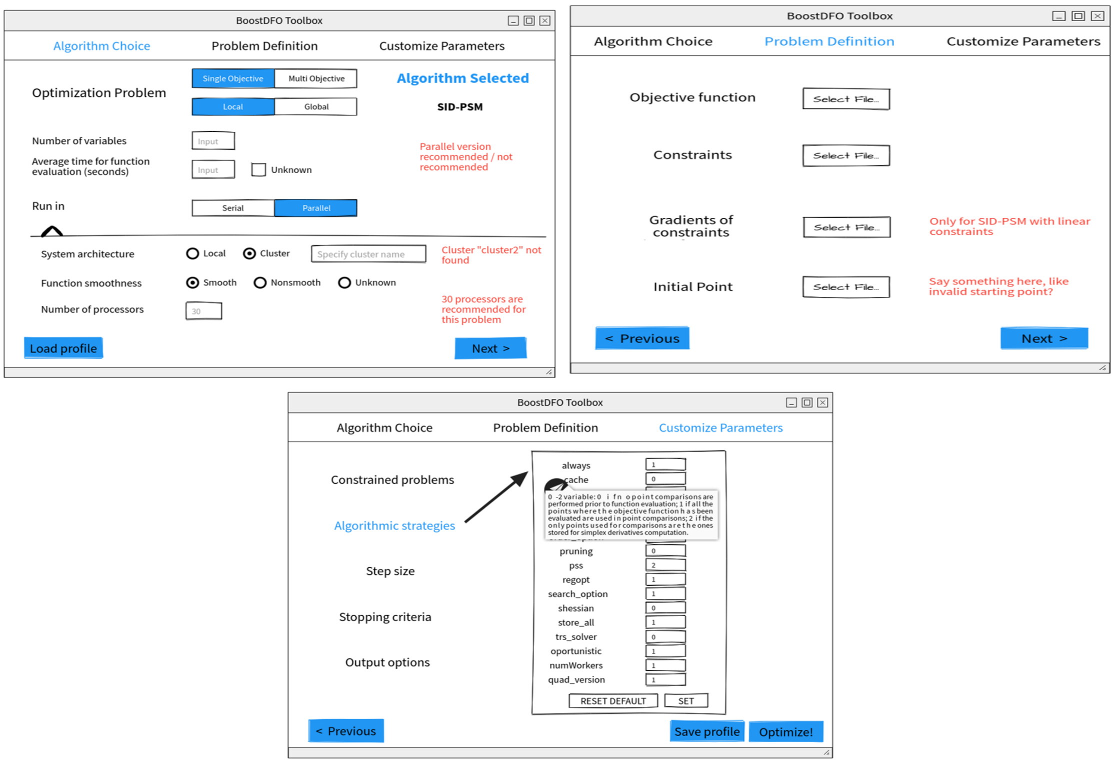
Part of the present work focused on the design of a code-integrative toolbox, featuring a Graphical User Interface. The main goal is to provide to the user an interactive and easy way of running the different Derivative-free Optimization solvers, property of the research team, while aggregating them under an application. The toolbox should provide information related to algorithmic choices, leading to the best use of the codes, and include a recommendation system for parallelism. This recommendation system is based on the comprehensive study performed in Chapter 4. Our target users are mainly optimizers that need to address complex real application problems, but also researchers that require results for complete sets of academic problems. The toolbox is suited for local/global and single/multi objective Derivative-free Optimization, including the codes SID-PSM [19, 23], DMS/BoostDMS [10, 22], GLODS [20] and MultiGLODS [21].

In this chapter, we describe all steps related to the development of the application. We start by analyzing the design process, from beginning to end (Section 5.1). Then, the final version of the toolbox is presented and thoroughly described, along with all the included features (Section 5.2). Finally, the evaluation performed on the application is referred on Section 5.3.

5.1 Design Process

The first step of the design process was to outline the specifications. Aiming towards a user-centered design, we started by defining our target users (as aforementioned) and addressing the information requirements of each of the four algorithms. The user needs to be able to choose between the different algorithms provided, configure parallel options (when available), interact with the different algorithmic choices by setting the corresponding parameters and provide the distinct files that define a problem: objective function(s), constraints, bounds and initial points. Information about all parameters and our recommendation for an appropriate number of workers to use should also be included, in order to best guide the user in the customization process. A set of required features was generated by the research team, as well as a rough sketch design that would contain all necessary information. Following these specifications, mockups were produced, including three different screens: algorithm selection, problem selection and parameter customization. These mockups were designed with *MockFlow*, a tool for sketching application interfaces, and are exhibited on Figure 5.1. After revising the mockups with the research team, the development of the application initiated using *Appdesigner*, a MATLAB tool suited for application development.

The considered approach followed a spiral design model. While maintaining a linear development structure, this model conveys some flexibility due to the intermediate feedback. At every phase of the process, the design gets more complex and is evaluated before proceeding to the next iteration. Ideally, the evaluation should always be done by users. However, as the application layout is not too complex and the research team is very familiar with the user needs and characteristics, the intermediate evaluations were done within the research team. Only the evaluation of the final design was done by users, through user testing (see Subsection 5.3). The development stage included two rounds of feedback with the research team, and some additional feedback rounds with the principal investigator, who also qualifies as a target user. With all suggestions well integrated, resulting in a robust design, the final interface will be presented in the next section.



0 -2 variable: 0 i f n o point comparisons are performed prior to function evaluation; 1 if all the points where the objective function has been evaluated are used in point comparisons; 2 if the only points used for comparisons are the ones stored for simplex derivatives computation.

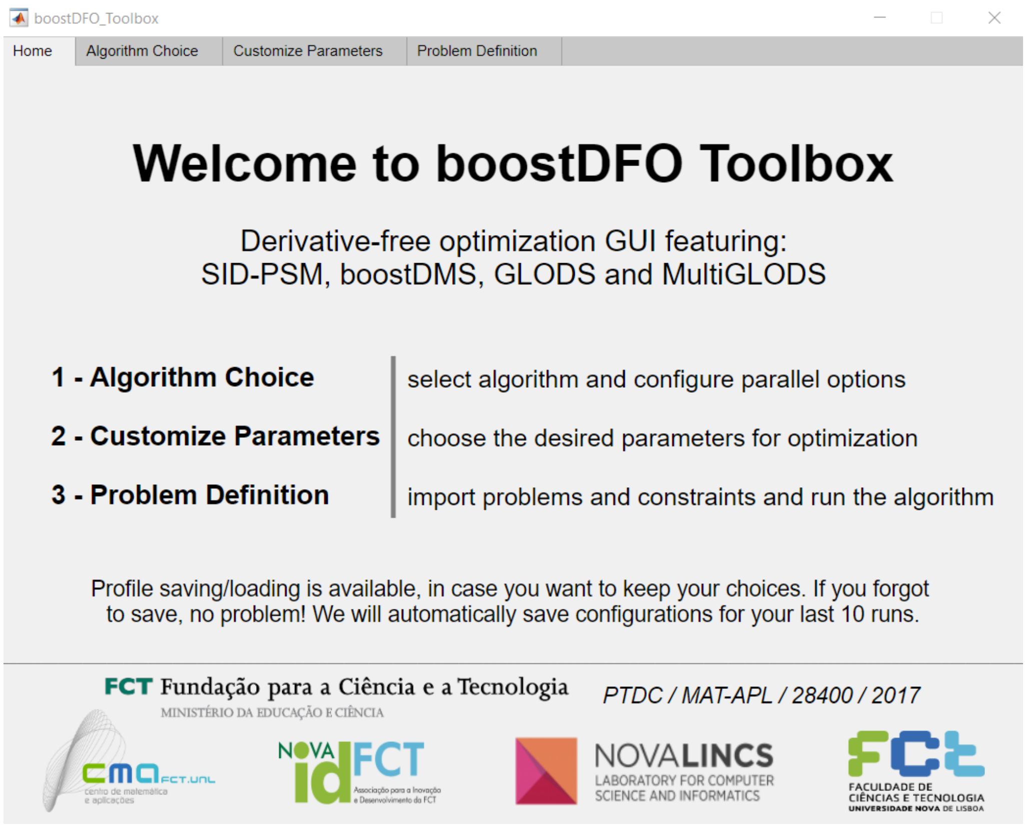
Figure 5.1: Final sketches obtained as a starting point for development (mockups).

5.2 Final Product

The present toolbox was developed using MATLAB version R2019b. Compatibility with previous versions of MATLAB is not guaranteed, due to recent features added by Mathworks. Depending on the solver selected, some additional MATLAB toolboxes might be needed by the user: parallel executions require the Parallel Computing Toolbox, BoostDMS requires the Optimization Toolbox and GLODS and MultiGLODS both require the Statistics and Machine Learning Toolbox. The GUI is tested to work on Windows, Linux and MacOS systems.

5.2.1 Features and Layout

The toolbox features a simple four screen layout: *Home*, *Algorithm Choice*, *Parameter Customization* and *File Selection*.



Welcome to boostDFO Toolbox

Derivative-free optimization GUI featuring:
SID-PSM, boostDMS, GLODS and MultiGLODS

1 - Algorithm Choice	select algorithm and configure parallel options
2 - Customize Parameters	choose the desired parameters for optimization
3 - Problem Definition	import problems and constraints and run the algorithm

Profile saving/loading is available, in case you want to keep your choices. If you forgot to save, no problem! We will automatically save configurations for your last 10 runs.

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA EDUCAÇÃO E CIÊNCIA
PTDC / MAT-APL / 28400 / 2017
cma FCT.UNL centro de matemática e aplicações
NOVA id FCT Associação para a Inovação e Desenvolvimento da FCT
NOVALINCS LABORATORY FOR COMPUTER SCIENCE AND INFORMATICS
FCT FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

Figure 5.2: *Home* screen layout.

The first one (Figure 5.2) is a general home page, featuring a simple user tutorial, information about project funding and research centers involved. The user should be able to quickly understand how to use the toolbox. However, if the user needs additional help or requires a more detailed explanation on the provided features, we also maintain a complete user guide on the main folder (see Annex I).

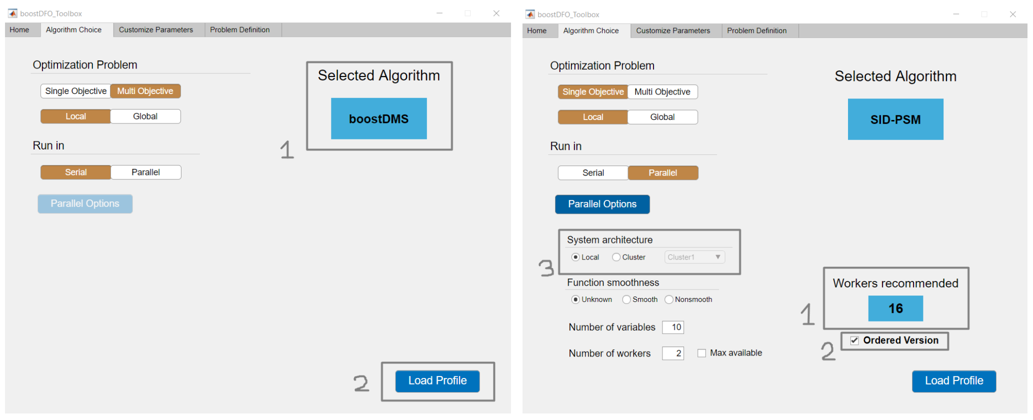


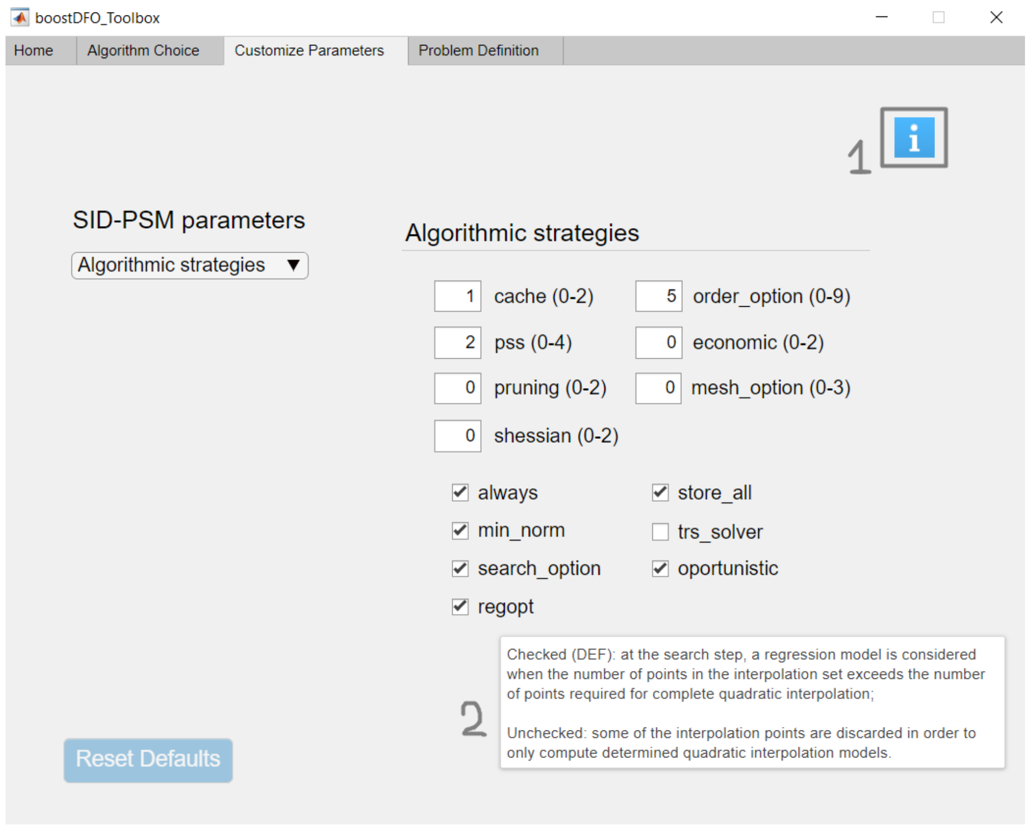
Figure 5.3: *Algorithm Choice* screen layout. Left figure shows the default layout; right figure shows parallel options. **Outlined features:** Left figure - **1.** algorithm selection. **2.** load profile. Right figure - **1.** worker recommendation. **2.** parallel version option. **3.** local machine or cluster selection.

The next screen, *Algorithm Choice* (Figure 5.3), allows the choice of the intended algorithm, depending on the general class of optimization problem to be addressed (local/global, single/multi objective). Parallel options are also available, for now only for SID-PSM algorithm, including the number of processors to use, the recommendation on a cost-effective number of processors, the kind of parallel version to use (keeping or not the poll order) and the system architecture (local or cluster execution). Previously configured clusters (using the MATLAB configuration tool) are automatically detected. In this way it is possible to use online cloud services, or other customized systems.

The recommendation on the number of CPU cores to use depends on the number of variables of the problem and its level of smoothness (smooth, nonsmooth or unknown). According to the data provided, our wizard generates and displays a recommended value (based in the analysis performed in Chapter 4). However, the user can always select a different number of workers, as long as there are enough CPU cores in the machine. For clusters, it is not possible to statically verify the maximum number of cores available, so the toolbox allows any input value and the algorithms will run either with the selected value or the maximum number of cores available, in case there are less workers available.

Parameter Customization (see Figure 5.4) is about parameter display and customization. Depending on the selected algorithm, the corresponding parameters are presented along with informative tooltips on the different options. The information icon on the top right corner (see Figure 5.4) notifies the user on how to find these helping tooltips: hovering the mouse on top of the desired parameter. It is always possible to reset all parameters to the initial default values.

Finally, *File Selection* allows the the user to provide the necessary files (Figure 5.5). Files uploaded can contain the objective function(s), the initial points, bounds and remaining constraints (depending on the algorithm). Also, special parameters that depend on these files can be customized. Information on correct file building is presented in the



1 cache (0-2)
2 pss (0-4)
0 pruning (0-2)
0 shessian (0-2)
5 order_option (0-9)
0 economic (0-2)
0 mesh_option (0-3)

☑ always
☑ min_norm
☑ search_option
☑ regopt
☑ store_all
☐ trs_solver
☑ oportunistic

Checked (DEF): at the search step, a regression model is considered when the number of points in the interpolation set exceeds the number of points required for complete quadratic interpolation;

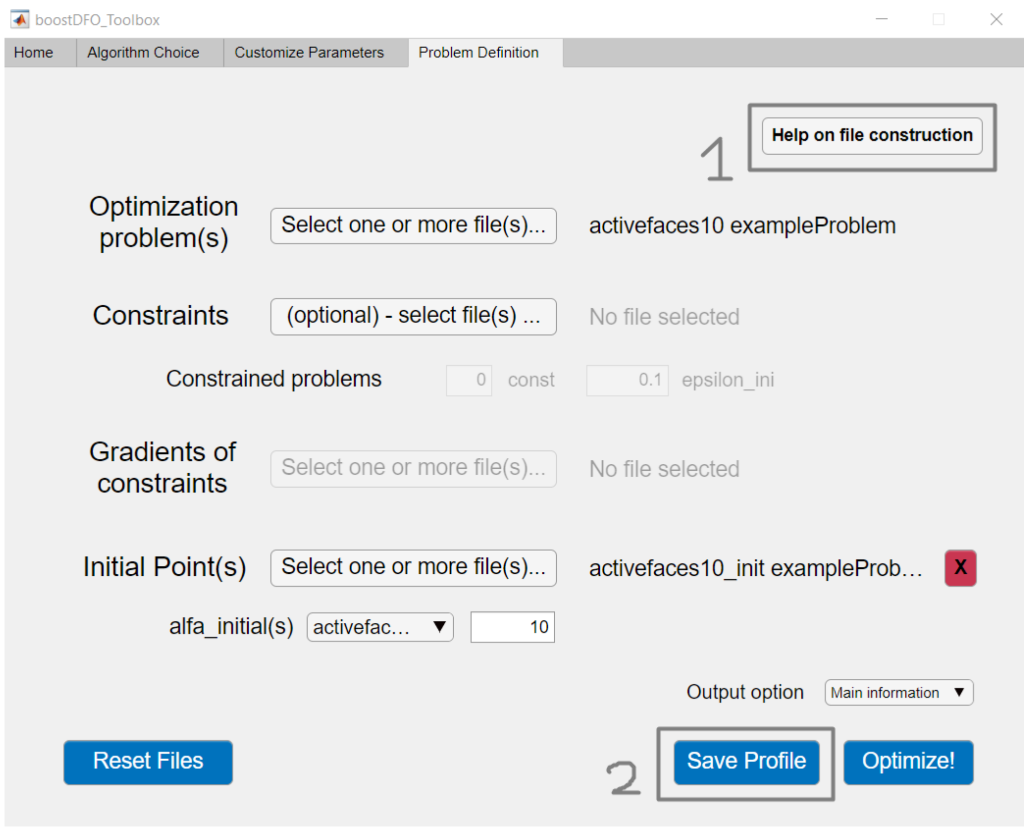
Unchecked: some of the interpolation points are discarded in order to only compute determined quadratic interpolation models.

Figure 5.4: *Parameter Customization* screen layout. **Outlined features: 1.** info button that informs the user about helping tooltips. **2.** a helping tooltip with all the information on the selected parameter.

user guide, that is automatically opened if the user clicks the *"Help on file construction"* button on the top right corner (element 1 of Figure 5.5). One or more files can be selected, allowing the user to solve multiple problems. In this way, we fulfill the needs of both optimizers: the ones that need to address a real application problem, and those that require results from test sets.

When pressing the *"Optimize!"* button, the run begins, automatically saving the profile and creating a new folder within the *reports* folder, duly identified with the name of the selected algorithm and the timestamp. Inside, the user can find a *.csv* file with the final information on all selected problems (one or several), along with the output files generated by the algorithm, identified by problem name. These files might contain the progression of the approximation to the problem solution, plots, error logs and MATLAB variables related to the number of iterations, total function evaluations performed or current variables that allow the problem run to be continued at a later stage. These outputs depend on the algorithm selected and on the success of the run.

In case the user needs to repeat runs, we include a feature of saving/loading profiles. These profiles allow the user to save all the information provided across the different screens: algorithm (including parallel options), parameters and files selected. In this


boostDFO_Toolbox
Home
Algorithm Choice
Customize Parameters
Problem Definition
1
Help on file construction
Select one or more file(s)...
(optional) - select file(s) ...
Select one or more file(s)...
Select one or more file(s)...
X
Reset Files
2
Save Profile
Optimize!

Optimization problem(s) activefaces10 exampleProblem

Constraints No file selected

Constrained problems 0 const 0.1 epsilon_ini

Gradients of constraints No file selected

Initial Point(s) activefaces10_init exampleProb...

alfa_initial(s) activefac... 10

Output option Main information

Figure 5.5: *File Selection* screen layout. **Outlined features: 1.** help button that automatically opens the user guide. **2.** save profile button.

way, the user can easily retrieve all configurations selected in a previous run and use them to repeat an experiment or as a starting point for a new optimization. Additionally, the profiles of the last 10 executions are automatically saved, in case they are eventually needed.

Table 5.1 shows a summary of all the features provided by the toolbox.

Feature List
Simple tutorial and complete user guide
Algorithm selection (class of the optimization problem)
Parallel options, including:
Recommendation on the number of cores
Architecture selection (local/cluster)
Selection of parallel version
Parameter customization
Single/multiple file selection
Information on parameters and file options (helping tooltips)
Save/Load parameter profiles
Profile autosave (last 10 runs)
Execution of the different optimization algorithms

Table 5.1: List of features included in the toolbox.

5.2.2 File System Layout

In order to comply with the complexity of integrating all the algorithms and different features, a specific file system layout is used, upon which the toolbox is built (see Figure 5.6). The main directory includes the *toolbox.mlapp* file that launches the application, along with the user guide and some hidden image files (logos of the supporting institutions). The folder *scripts* includes all files needed on the application startup. The file (*constants.m*) contains all static messages and paths necessary and the four different *.mat* files enclose the default parameter values to initiate each of the algorithms.

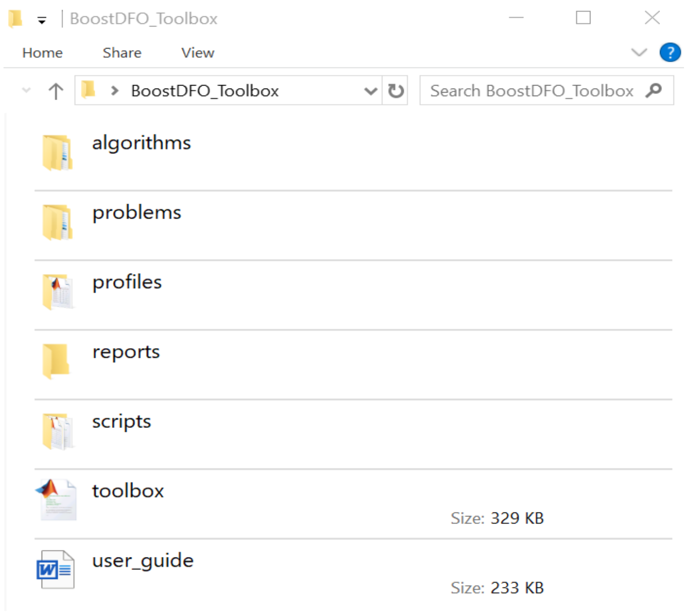


Figure 5.6: File system layout.

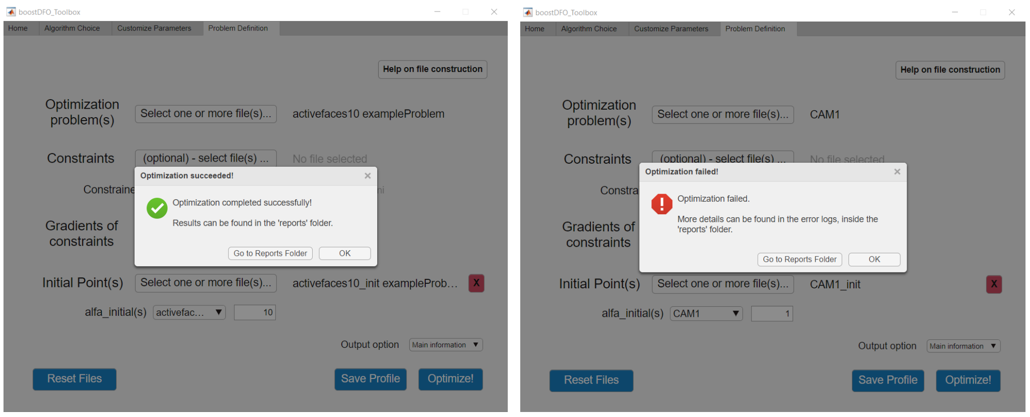
The folder *algorithms* contains codes corresponding to the four algorithms, duly adapted to integrate the interface.

On *problems*, the user can find several example problems already coded, to test the different algorithms. *Profiles* is the folder where all parameter profiles are stored. It already includes the *defaultProfile* that also loads on startup, and an *autosave* folder where the application automatically saves the parameters used in the last ten runs performed. Finally, the *reports* folder is where the user can find the optimization results.

5.2.3 Error Handling and Feedback

The application captures a comprehensive number of error cases that may arise. Informative feedback is always presented to the user, allowing for both understanding its causes and finding out possible solutions. Figure 5.7 shows an example of feedback given in the cases of success and failure of a run.

If an error is detected on startup (*e.g.* MATLAB path is not set to the root of the file system), the application does not open and displays an error message, forwarding the user to possible solutions.



activefaces10 exampleProblem
No file selected
activefaces10_init exampleProb…
alfa_initial(s): activefac… 10
Output option: Main information

Optimization succeeded!
Optimization completed successfully!
Results can be found in the 'reports' folder.

CAM1
No file selected
CAM1_init
alfa_initial(s): CAM1 1
Output option: Main information

Optimization failed!
Optimization failed.
More details can be found in the error logs, inside the 'reports' folder.

Figure 5.7: Feedback received after the execution. Pressing *"Go to Reports Folder"* automatically opens the folder where the results are stored.

While optimization is running, possible errors, either due to incorrect file construction or other unexpected case, are captured and stored in an error log, along with the algorithm outputs. In this way, information on the errors is always given to the user without blocking the application flow, eliminating the need for recovery.

5.3 Evaluation

Evaluation of the toolbox was made through user testing. Three testers were selected that classified as typical users: optimizers that are familiar with Derivative-free Optimization and solvers, with some degree of contact with the algorithms integrated in the toolbox, but with no prior knowledge on the toolbox application. The tests were performed remotely, using *Zoom*, a video conference platform, requiring users to share the screen with the developer. The test script was followed by a quick questionnaire and both items were shared on *Google Forms*. The procedure was thorough, taking between one and a half and two hours to complete.

The test itself was a set of eight different tasks scenarios, organized on five topics (see Appendix A). Each task scenario represents a typical task that users might perform using the application. The information provided was minimal, to assess how easily users could understand all the necessary steps on their own, or resorting to the helping tools that the application provides (helping tooltips and user guide). However, some hints were provided at times by the developer in order to reduce the possible frustration due to the length of the test. For the same reason, some tasks were sometimes skipped, depending on the performance of the user. If the user had already explored the feature to be tested or had already understood perfectly all the steps he would have to take, that task could safely be skipped without compromising the test.

At the end, the *System Usability Scale* (SUS) questionnaire [12] was applied. This is a standard and well validated questionnaire that quantitatively measures usability, in a scale of 0 to 100 (see Appendix B). It is very quick and simple to apply, comprising only ten questions. The version used and how the results are interpreted are part of the original article [12].

By conducting this user test procedure, it was possible to gather both qualitative and quantitative feedback on the toolbox.

5.3.1 Results

The general feedback obtained was that the toolbox was intuitive, easy to use and its features were well integrated. Additionally, based on the user experience and comments, several practical details were improved (*e.g*, level of information provided and layout details). The main suggestions regarding this constructive feedback were essentially three:

1. Users felt that the original toolbox should include a more detailed guide to detail some of the features. At the time of the tests, the user guide only included help for file construction, as this was the most difficult task to perform.

2. The Load/Save Profile was slightly confusing, as users thought they were supposed to use it but didn't know how. None of the users understood this feature on their own. When explained, all of them responded positively, stating that it was a useful and practical feature.

3. When selecting files corresponding to different optimization problems, it was not clear if they could be picked individually and accumulate or if they should all be selected at once. However, the file selection is followed by a field where the name of the selected files appear, so this issue probably would only appear in a first usage of the toolbox.

Building up on this qualitative feedback, we decided to address the questions in the user guide, including more useful information. Additionally, we felt some details were missing in order to present a complete guide. Thus, all questions were better clarified and integrated, along with new information on relevant features, system requirements and further details. We expect that these points are now clearer and further improvements might be added in the future.

Regarding our quantitative results, we obtained an average of 83.3 points in the SUS scale. SUS questionnaire has been available for a long time (more than 30 years) and the meaningfulness of the corresponding results has been extensively studied. Percentile ranks and the Net Promoter Score [52] are some of the metrics considered for it. Percentile ranks represent a normalized comparative measure, considering a broader universe of applications. Converting our score to a percentile, our toolbox would be inside the 90-95 percentile, which is a very good indicator [52]. The Net Promoter Score evaluates the likelihood of a user recommending the application to another person (colleague, peer) and seems to be correlated to SUS scores [53]. Our score suggests that some of our users could be classified as *Promoters*, what would help spreading the toolbox across the community. Since our sample of testers was reduced, further testing and feedback from more users will impact on the result validation.

Chapter 6

Conclusions

The present work resulted in two major outputs: a parallelized version of the algorithm SID-PSM (including two different parallel strategies) and a code-integrative Toolbox that features a GUI for all four Derivative-free Optimization solvers (SID-PSM, DMS/Boost-DMS, GLODS and MultiGLODS).

Furthermore, it provided a pretext to revise SID-PSM's parameters and select the best default options, through an extensive data profile analysis, and to introduce some performance optimizations on the original code. Several algorithmic options were then analyzed and compared. The version that includes active cache and search step, an opportunistic poll strategy and an improved routine to build the quadratic models demonstrated a superior performance and was thus selected as the default version of the code.

Then we described the introduction of parallelism in SID-PSM and analyzed the potential parallel behavior of the algorithm, so that a recommendation can help a user selecting the number of CPUs to use, even without any prior knowledge about the problem. Building up on the number of function evaluations/poll step cycle, two recommendation methods were proposed: one based on its average and other adding a standard deviation to the average (upperbound). The upperbound approach seemed the best one, given the efficiency observed on our problem test set being very close on both cases, with significant speedup gains for this last approach. Also, the statistical bound provided is of practical interest, offering a good level of parallelism on each cycle (at least 84% of function evaluations covered). The expected gains should also be more consistent, by mitigating the variability present in the average approach.

Additionally, two different parallel version were designed and compared, both of them targeting the poll step: a regular parallel version and one that keeps the order of function evaluations within the poll step, taking advantage of the ordering strategy already in place. The motivation to develop both versions came from the detection of a possible tradeoff between performance, expressed in computational time, and the quality of the final result obtained. The proposed parallel versions were evaluated regarding the recommended number of CPUs and applied to both an academic set of problems and a real industrial application (the production of styrene). We verified relevant parallel gains even for functions with an evaluation time of 0.1 seconds, outlining the advantages of the parallel approach for tackling real application problems.

The results obtained with our test set, following the recommendation provided, reveal a general speedup around 5.70, rising almost to 9 for problems of dimension 40. Also, nonsmooth problems seem to benefit even more from parallelism, pratically doubling the speedup obtained for smooth problems. These results allow us to foresee greater speedups for more complex problems.

The relevance of offering the parallel ordered option to our users is illustrated by the results obtained for the styrene production problem when using 16 CPUs (our recommended value). Accounting only the regular parallel version, a speedup of almost 6 was observed, compared to the serial version, but the solution obtained was almost 6% worse. Following the ordering strategy on function evaluations led to an increase of 55% in the execution time, but this is still less than 1/3 of the total time required by the serial version, guaranteeing the quality of the final solution computed while maintaining a good performance. This was the parallel version adopted as default in our code. However, different problems can exhibit different behaviors and we give the user the possibility to choose the parallelization strategy, based on his expertise and preferences.

As for our toolbox, it was possible to build an intuitive and easy-to-use system with all the intended features. User testing was performed with three representative users, applying a set of task scenarios for qualitative feedback and the SUS questionnaire for a quantitative evaluation. The general feedback was that the features were well integrated and the toolbox would be very practical for solving DFO problems. Also, the constructive feedback provided allowed us to improve it further. We received an average score of 83.3 points in the SUS scale (0-100), what places our application inside the 90-95 percentile rank on a normalized scale and suggests that some users are likely to recommend it to others, helping spread the tool across the DFO community.

6.1 Future Work

After the successful parallelization of the SID-PSM algorithm, a natural direction to proceed would be the parallelization of the remaining solvers, and their inclusion on the toolbox. Additionally, further improvements to the toolbox might be made, as a result of feedback from the community of users. Another research path that could be beneficial is related to taking advantage of clusters and cloud services. In this work it has not been possible to explore different cloud architectures as intended, due to the high cost of MATLAB Parallel Server. Instead, we used a single machine for cloud testing, what can be restrictive if the user needs a very high number of CPUs. Accordingly, it may also not seem advantageous for most users of our codes/toolbox to make use of this Parallel Server. To tackle this issue, it would be valuable to explore a system that would interact with the cloud or clusters and manage its instantiation, deployment and control directly and on-the-fly, thus simplifying the process for the common user.

Bibliography

[1] Y. Altman. *Undocumented Matlab*. URL: https://undocumentedmatlab.com/blog/preallocation-performance. (accessed: 10.07.2019).

[2] Y. Altman. *Accelerating MATLAB® Performance : 1001 Tips to Speed up MATLAB Programs*. Boca Raton: CRC Press, 2015. ISBN: 978-1-4822-1130-6.

[3] G. M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities." In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, pp. 483–485.

[4] C. Audet, V. Béchard, and S. Le Digabel. "Nonsmooth optimization through mesh adaptive direct search and variable neighborhood search." In: *Journal of Global Optimization* 41.2 (2008), pp. 299–318.

[5] C. Audet and J. E. Dennis Jr. "Analysis of generalized pattern searches." In: *SIAM Journal on Optimization* 13.3 (2002), pp. 889–903.

[6] C. Audet and J. E. Dennis Jr. "Mesh adaptive direct search algorithms for constrained optimization." In: *SIAM Journal on Optimization* 17.1 (2006), pp. 188–217.

[7] C. Audet, G. Savard, and W. Zghal. "A mesh adaptive direct search algorithm for multiobjective optimization." In: *European Journal of Operational Research* 204.3 (2010), pp. 545–556.

[8] M. Aziz, W. Hare, M. Jaberipour, and Y. Lucet. "Multi-fidelity algorithms for the horizontal alignment problem in road design." In: *Engineering Optimization* (2019), pp. 1–20.

[9] C Bogani, M. Gasparo, and A Papini. "Generalized pattern search methods for a class of nonsmooth optimization problems with structure." In: *Journal of Computational and Applied Mathematics* 229.1 (2009), pp. 283–293.

[10] C. Brás and A. Custódio. *On the use of polynomial models in multiobjective directional direct search*. Tech. rep. Centre of Mathematics and Applications, NOVA University of Lisbon, 2019. URL: http://ferrari.dmat.fct.unl.pt/personal/alcustodio/BoostDMS.pdf.

[11] R. P. Brito, H. Sebastião, and P. Godinho. "Portfolio management with higher moments: the cardinality impact." In: *International Transactions in Operational Research* 26.6 (2019), pp. 2531–2560.

[12] J. Brooke et al. "SUS - A quick and dirty usability scale." In: *Usability Evaluation in Industry* 189.194 (1996), pp. 4–7.

[13] S. Chapman. *MATLAB Programming for Engineers*. 2nd ed. Pacific Grove, CA: Brooks/Cole-Thomson Learning, 2002. ISBN: 0-534-39056-0.

[14] COIN-OR Project. *Derivative Free Optimization*. URL: http://projects.coin-or.org/Dfo.

[15] A. R. Conn, K. Scheinberg, and L. N. Vicente. *Introduction to Derivative-free Optimization*. Vol. 8. SIAM, 2009.

[16] A. Conn, N. Gould, M. Lescrenier, and P. L. Toint. "Performance of a multifrontal scheme for partially separable optimization." In: *Advances in Optimization and Numerical Analysis*. Springer, 1994, pp. 79–96.

[17] A. L. Custódio, K. Scheinberg, and L. N. Vicente. "Methodologies and software for derivative-free optimization." In: *Advances and Trends in Optimization with Engineering Applications*. Ed. by T. Terlaky, M. F. Anjos, and S. Ahmed. MOS-SIAM Book Series on Optimization. Philadelphia: SIAM, 2017.

[18] A. L. Custódio, J. E. Dennis Jr., and L. N. Vicente. "Using simplex gradients of nonsmooth functions in direct search methods." In: *IMA J. Numer. Anal.* 28 (2008), pp. 770–784.

[19] A. L. Custódio, H. Rocha, and L. N. Vicente. "Incorporating minimum Frobenius norm models in direct search." In: *Comput. Optim. Appl.* 46 (2010), pp. 265–278.

[20] A. L. Custódio and J. A. Madeira. "GLODS: global and local optimization using direct search." In: *Journal of Global Optimization* 62.1 (2015), pp. 1–28.

[21] A. L. Custódio and J. A. Madeira. "MultiGLODS: global and local multiobjective optimization using direct search." In: *Journal of Global Optimization* 72.2 (2018), pp. 323–345.

[22] A. L. Custódio, J. A. Madeira, A. I. F. Vaz, and L. N. Vicente. "Direct multisearch for multiobjective optimization." In: *SIAM Journal on Optimization* 21.3 (2011), pp. 1109–1140.

[23] A. L. Custódio and L. N. Vicente. "Using sampling and simplex derivatives in pattern search methods." In: *SIAM Journal on Optimization* 18.2 (2007), pp. 537–555.

[24] C. Davis. "Theory of positive linear dependence." In: *American Journal of Mathematics* 76.4 (1954), pp. 733–746.

[25] J. W. Demmel. *Applied Numerical Linear Algebra*. Vol. 56. SIAM, 1997.

[26] J. E. Dennis Jr and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Vol. 16. SIAM, 1996.

[27] D. Fleischman, K. T. Fountaine, C. R. Bukowsky, G. Tagliabue, L. A. Sweatlock, and H. A. Atwater. "High spectral resolution plasmonic color filters with subwavelength dimensions supplemental information." In: *ACS Photonics* 6 (2019), pp. 332–338.

[28] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[29] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley. "Google Vizier: A Service for Black-Box Optimization." In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: Association for Computing Machinery, 2017, 1487–1495. DOI: 10.1145/3097983.3098043.

[30] G. A. Gray, J. D. Griffin, M. Taddy, M. Martinez-Canales, and T. G. Kolda. *HOPSPACK: Hybrid optimization parallel search package*. Tech. rep. SAND2008-8057. Sandia National Laboratories (SNL-CA), 2008.

[31] J. L. Gustafson. "Reevaluating Amdahl's law." In: *Communications of the ACM* 31.5 (1988), pp. 532–533.

[32] M. Haarala. *Large-scale Nonsmooth Optimization: Variable Metric Bundle Method With Limited Memory*. 40. University of Jyväskylä, 2004.

[33] W. Hare and C. Sagastizábal. "Benchmark of some nonsmooth optimization solvers for computing nonconvex proximal points." In: *Pacific Journal on Optimization* 3 (2006), pp. 545–573.

[34] S. D. Karamintziou, A. L. Custódio, B. Piallat, M. Polosan, S. Chabardès, P. G. Stathis, G. A. Tagaris, D. E. Sakas, G. E. Polychronaki, G. L. Tsirogiannis, O. David, and K. S. Nikita. "Algorithmic design of a noise-resistant and efficient closed-loop deep brain stimulation system: a computational approach." In: *PLoS ONE* 12.e0171458 (2017), pp. 1–26.

[35] T. G. Kolda, R. M. Lewis, and V. Torczon. "Optimization by direct search: New perspectives on some classical and modern methods." In: *SIAM Review* 45.3 (2003), pp. 385–482.

[36] J. Larson, M. Menickelly, and S. M. Wild. "Derivative-free optimization methods." In: *Acta Numerica* 28 (2019), pp. 287–404.

[37] S. Le Digabel. "Algorithm 909: NOMAD: Nonlinear Optimization with the MADS algorithm." In: *ACM Transactions on Mathematical Software* 37.4 (2011), pp. 1–15.

[38] D. C. Marinescu. *Cloud Computing: Theory and Practice*. Morgan Kaufmann, 2017.

[39] MathWorks. *MATLAB*. URL: https://www.mathworks.com.

[40] MathWorks. *MATLAB Documentation*. URL: https://www.mathworks.com/help/.

[41] P. Mell, T. Grance, et al. "The NIST Definition of Cloud Computing." In: (2011).

[42] Microsoft. *Microsoft Azure*. URL: https://azure.microsoft.com/en-us/overview/. (accessed: 27.06.2019).

[43] J. J. Moré, B. S. Garbow, and K. E. Hillstrom. "Testing unconstrained optimization software." In: *ACM Transactions on Mathematical Software* 7.1 (1981), pp. 17–41.

[44] J. J. Moré and S. M. Wild. "Benchmarking derivative-free optimization algorithms." In: *SIAM Journal on Optimization* 20.1 (2009), pp. 172–191.

[45] J. Nocedal and S. Wright. *Numerical Optimization*. Springer Science & Business Media, 2006.

[46] M. J. D. Powell. "Direct search algorithms for optimization calculations." In: *Acta Numerica* 7 (1998), 287–336. DOI: 10.1017/S0962492900002841.

[47] M. J. Powell. "The NEWUOA software for unconstrained optimization without derivatives." In: *Large-scale Nonlinear Optimization*. Springer, 2006, pp. 255–297.

[48] M. J. Powell. "A direct search optimization method that models the objective and constraint functions by linear interpolation." In: *Advances in Optimization and Numerical Analysis*. Springer, 1994, pp. 51–67.

[49] M. J. Powell. "The BOBYQA algorithm for bound constrained optimization without derivatives." In: *Cambridge NA Report NA2009/06, University of Cambridge, Cambridge* (2009), pp. 26–46.

[50] L. M. Rios and N. V. Sahinidis. "Derivative-free optimization: a review of algorithms and comparison of software implementations." In: *Journal of Global Optimization* 56.3 (2013), pp. 1247–1293.

[51] B. Sauk, N. Ploskas, and N. Sahinidis. "GPU parameter tuning for tall and skinny dense linear least squares problems." In: *Optimization Methods and Software* (2018), pp. 1–23.

[52] J. Sauro. *5 Ways to Interpret a SUS Score*. URL: https://measuringu.com/interpret-sus-score/. (accessed: 26.03.2020).

[53] J. Sauro. *Predicting Net Promoter Scores from System Usability Scale Scores*. URL: https://measuringu.com/nps-sus/. (accessed: 26.03.2020).

[54] B. Schmidt. *Parallel Programming : Concepts and Practice*. Cambridge, MA: Morgan Kaufmann Publishers, an imprint of Elsevier, 2018. ISBN: 978-0-12-849890-3.

[55] G. Sharma and J. Martin. "MATLAB®: a language for parallel computing." In: *International Journal of Parallel Programming* 37.1 (2009), pp. 3–36.

[56] Y. Sun, N. V. Sahinidis, A. Sundaram, and M.-S. Cheon. "Derivative-free optimization for chemical product design." In: *Current Opinion in Chemical Engineering* 27 (2020), pp. 98–106.

[57] P. Taesler, J. Jablonowski, Q. Fu, and M. Rose. "Modeling implicit learning in a cross-modal audio-visual serial reaction time task." In: *Cognitive Systems Research* 54 (2019), pp. 154–164.

[58] P. L. Toint. "Some numerical results using a sparse matrix updating formula in unconstrained optimization." In: *Mathematics of Computation* 32.143 (1978), pp. 839–851.

[59] V. Torczon. "On the convergence of pattern search algorithms." In: *SIAM Journal on Optimization* 7.1 (1997), pp. 1–25.

[60] A. I. F. Vaz and L. N. Vicente. "A particle swarm pattern search method for bound constrained global optimization." In: *Journal of Global Optimization* 39.2 (2007), pp. 197–219.

[61] L. N. Vicente and A. Custódio. "Analysis of direct searches for discontinuous functions." In: *Mathematical Programming* 133.1-2 (2012), pp. 299–325.

[62] E. W. Weisstein. *Smooth Function*. URL: `http://mathworld.wolfram.com/SmoothFunction.html`. (accessed: 09.07.2019).

[63] M. Wright. "Direct search methods: Once scorned, now respectable." In: *Numerical analysis: Proceedings of the 1995 Dundee Biennial Conference in Numerical Analysis*. Addison-Wesley. 1996, pp. 191–208.

Appendix A

User Testing Script

User Testing Script

BoostDFO_Toolbox is a graphical user interface (GUI) developed as an aiding tool for running optimization codes. It is suited for solving single/multi objective and local/global Derivative-free Optimization problems, featuring the codes *SID-PSM*, DMS/*boostDMS*, *GLODS* and *MultiGLODS*. Some coded example problems are already provided with the distribution, with the purpose of helping the user to run his/her own problems. In the next screens you will be asked to perform some tasks using this GUI.

1. <u>Solving a specific problem</u>

A - The user wants to run the problem *ackley* with the algorithm *GLODS,* with a maximum of 500 evaluations. At the end, the user checks the results obtained.

B -The user runs the problem *exampleProblem*, including the corresponding constraints. This is a single objective, local optimization problem. At the end, the user checks the results obtained.

2. <u>Solving a set of problems</u>

A - The user runs a set of problems, chosen from the ones already provided. The algorithm to choose is *MultiGLODS*. At the end, the user checks the results obtained.

B - The user wants to perform multi objective, local optimization in a set of problems from those already provided. When selecting the initialization files, he/she should select both the correct and some extra incorrect files. At the end, the user checks the results obtained.

3. Solving a new problem

The user prepares all the necessary files to run his/her own problem. Any kind of optimization may be performed. Then, the user runs the problem and checks the results. If the problem optimization is time-consuming, a small stopping criterion may be used.

3. Solving a new problem (applied only if the user doesn't have his/her own problem)

The user prepares all the necessary files to run the problem which data is provided below. The optimization will be performed with the algorithm *boostDMS*. In the File Selection screen, the *list* option should take the value 2.

Problem data:

Dimension – **30 variables**

Objective function:

```
numVar = length(x);
g      = 1 + 9*sum(x(2:numVar))/(numVar-1);
f(1)   = x(1);
f(2)   = g * ( 1-(x(1)/g)^2);
F      = f';
```

Bounds – ***[0,…,0] e [1,…,1] (problem dimension)***

Initial Point: ***none***

4. Solving a parallel problem – part 1

The user wants to run the problem *brownal30* (dimension 30) using parallel computing. This problem classifies as smooth. The parallel ordered version should be used, and the stopping criteria is a maximum of 800 evaluations. Additionally, the user should change three parameters (at will) within the parameter customization area.

4. Solving a parallel problem – part 2

The user intends to run again the previous problem with the same options but including a different stopping criterion.

5. Continue a previous run

The first task was to run the problem *ackley* with the algorithm *GLODS,* allowing a maximum of 500 evaluations. Now the user wants to perform more 1500 evaluations, restarting from the previous result.

Now that you have completed all tests on the interface, we ask you to fill a quick usability questionnaire. Thank you for your kind collaboration!

Appendix B

System Usability Scale (SUS) questionnaire

System Usability Scale

	Strongly disagree				Strongly agree
1. I think that I would like to use this system frequently	1	2	3	4	5
2. I found the system unnecessarily complex	1	2	3	4	5
3. I thought the system was easy to use	1	2	3	4	5
4. I think that I would need the support of a technical person to be able to use this system	1	2	3	4	5
5. I found the various functions in this system were well integrated	1	2	3	4	5
6. I thought there was too much inconsistency in this system	1	2	3	4	5
7. I would imagine that most people would learn to use this system very quickly	1	2	3	4	5
8. I found the system very cumbersome to use	1	2	3	4	5
9. I felt very confident using the system	1	2	3	4	5
10. I needed to learn a lot of things before I could get going with this system	1	2	3	4	5

Using SUS

The SU scale is generally used after the respondent has had an opportunity to use the system being evaluated, but before any debriefing or discussion takes place. Respondents should be asked to record their immediate response to each item, rather than thinking about items for a long time.

All items should be checked. If a respondent feels that they cannot respond to a particular item, they should mark the centre point of the scale.

Scoring SUS

SUS yields a single number representing a composite measure of the overall usability of the system being studied. Note that scores for individual items are not meaningful on their own.

To calculate the SUS score, first sum the score contributions from each item. Each item's score contribution will range from 0 to 4. For items 1,3,5,7,and 9 the score contribution is the scale position minus 1. For items 2,4,6,8 and 10, the contribution is 5 minus the scale position. Multiply the sum of the scores by 2.5 to obtain the overall value of SU.

SUS scores have a range of 0 to 100.

The following section gives an example of a scored SU scale.

Annex I

Toolbox User Guide

BoostDFO_Toolbox User Guide

BoostDFO_Toolbox is a code-integrative Graphical User Interface (application) suited for local/global and single/multi objective Derivative-free Optimization. Its main goal is to provide interactive and easy access to a suite of solvers that users can apply to their problems. It features four different solvers, property of the research team: **SID-PSM**, **BoostDMS**, **GLODS**, and **MultiGLODS** (more information on the **Included Solvers** section).

Contents

System Requirements

The present toolbox was developed using Appdesigner on MATLAB version R2019b. It was tested to work on Windows, Linux and MacOS. In order to use this GUI, the user should have MATLAB 2019 (or a newer version) installed. Also, for using the different codes, some additional MATLAB toolboxes might be necessary.

Parallel executions require the Parallel Computing Toolbox;

BoostDMS requires the Optimization Toolbox;

GLODS and **MultiGLODS** require the Statistics and Machine Learning Toolbox.

Compatibility with previous versions of MATLAB is not guaranteed, due to the new features added by Mathworks.

Installation

After extracting the contents of the *.zip* folder, everything is set.

Even though *toolbox.mlapp* is the file that launches the toolbox, it relies on the file system included in this folder. **It is not recommended to change/delete any original folders/files**, as this may compromise the app functioning (except for the example problems; those can be safely edited/deleted). New files generated by the toolbox are safe to delete (profiles, **excluding *defaultProfile***, and reports).

Opening the Toolbox

In the main directory (*boostDFO_Toolbox*), open the file *toolbox.mlapp*.

Sometimes MATLAB is not set to automatically open this type of files. If this verifies, the user can choose one of two options:

find MATLAB path folder to open this file with MATLAB;

open the MATLAB command window, type the *appdesigner* command to open the development environment, open the *toolbox* and click *Run*.

After the first use, MATLAB should be automatically configured to open *.mlapp* files.

Solvers Included

This toolbox includes four solvers, suited for different classes of Derivative-free Optimization problems. More information on each one can be found at:

SID-PSM	http://www.mat.uc.pt/sid-psm
BoostDMS	http://www.mat.uc.pt/dms http://ferrari.dmat.fct.unl.pt/personal/alcustodio/BoostDMS.pdf
GLODS	http://ferrari.dmat.fct.unl.pt/personal/alcustodio/GLODS.htm
MultiGLODS	http://ferrari.dmat.fct.unl.pt/personal/alcustodio/multiglods.htm

Parallel Versions

At the moment, **SID-PSM** is the only algorithm that features a parallel version. The other parallel versions will be added in the future, as soon as they are developed.

Features

The toolbox includes four different screens: *Home*, *Algorithm Choice*, *Parameter Customization* and *File Selection*. Those are designed to allow a quick and easy use of the algorithms.

In the first one, *Home*, you can find a quick user tutorial. Below we give a better explanation on the main features.

Solving single/multiple problems

The toolbox can be used to solve either one or more optimization problems. In this way, we can meet the demands of optimizers that need to address a real application problem (in any field), as well as researchers that require results from a complete test set.

On the *File Selection* screen, you can find selection buttons followed by text fields on the right. When a file is selected, the corresponding name appears on the right side. If you then select another file, they will switch. The selection of several files should be done at once, using the keys CTRL + LEFT MOUSE CLICK or mouse dragging. In the right side, the name of the different files selected will now appear.

For information on how to build the files correctly, see section **Instructions on File Building** below.

Helping tooltips

Many options and parameters are available for user customization. You can always use the recommended defaults. However, if you need details on what a specific parameter/option does, just hover the mouse on top of the corresponding text and a helping tooltip will appear.

Parallel Options (only available for SID-PSM)

Cluster selection

When solving problems resorting to parallelism, you may use your own machine (local) or a cluster. The clusters should be previously configured (using the MATLAB configuring tool) and are automatically detected. In this way it is possible to use online cloud services, or other customized systems.

Worker recommendation

This is a recommendation provided on a cost-effective number of processors. In case you are using cloud services or would like to spare some available cpus, this is an indicator for a good performance. If you don't have these restrictions and have more cpus than those recommended, you can use them for increased performance, up to the total number of function evaluations per poll step completed by the algorithm (*2*problemDimension* + 2, in case of SID-PSM). There will be no performance gain after this point.

The number of variables and level of smoothness of the problem(s) directly impact the recommendation. If you are solving multiple problems with parallel options, please make sure that all of them have the same number of variables and level of smoothness.

The recommendation provided is based on numerical results obtained on a test set comprising academic problems, which were analyzed as a whole and split by level of smoothness. This recommendation ensures that the number of workers is enough for completing the poll step in a single pass in 84% of the cases (assuming it follows a Gaussian distribution).

Profiles (save / load / autosave)

In case you have the need to repeat runs, we include a feature of saving/loading Profiles. These profiles save all the information provided across the different screens - algorithm chosen, parameters and files selected.

The last 10 executions are automatically saved (*autosave* folder), in case they are eventually needed.

Instructions on File Building

This section includes all the instructions on building the files necessary to solve Derivative-free Optimization problems. Each subsection features a different algorithm. Examples are provided for all of them, both here and inside the *Problems\Examples* folder. All files are coded in MATLAB.

SID-PSM

The set of files *exampleProblem* was prepared as an example of a constrained problem to be solved with *SID-PSM*.

It relates to the constrained optimization problem with two variables:

$$\begin{array}{ll} \textbf{Min} & (x_2 - x_1^2)^2 \\ \textbf{s.t.} & -2 <= x_1 <= 0 \\ & x_2 <= 1 \end{array}$$

The initial point considered is [-1.2 1]'.

Problem (objective function) – required

This file codes the objective function, defining the problem. It can be located anywhere on the computer, but the other files to load must be on the same directory. One or several problems can be solved at once (all from the same directory).

It receives a point *x* and outputs the result of the evaluation of that point (single value).

e.g.

```
function f = exampleProblem(x)

    f = (x(2)-x(1)^2)^2;
```

Constraints - optional

This file includes all constraints that apply to the problem. It should be named as **problemname*_const*, replacing **problemname** by the name of the corresponding problem (in our example, it would be *exampleProblem_const*). It is possible to solve both constrained and unconstrained problems in the same run, by uploading only some constraints files. *SID-PSM* requires no bounds for initialization, but these can be inserted as constraints.

It receives a point *x* and outputs the constraints' value at the given point (a column vector of *c* values, where *c* is the number of constraints). Constraints are always written in the form $C_i(x) \leq 0$.

e.g.

```
function [c_const] = exampleProblem_const(x)

        c_const = [ ];

        c_const(1) = x(1);

        c_const(2) = -x(1)-2;

        c_const(3) = x(2)-1;

        c_const = c_const';
```

Gradients of Constraints - optional (required in the constrained case)

This file contains the gradients of the problem's constraints. It should be named as **problemname*_const_grad*. This file is required for solving constrained problems, since it is used to build the poll directions.

It receives a point *x* and outputs the gradients of the constraints evaluated at *x* columnwise (a matrix with of *n**c values, where *n* (rows) is the number of variables and *c* (columns) is the number of constraints).

e.g.

```
function [grad_c] = exampleProblem_const_grad(x)

        grad_c = [ ];

        grad_c(:,1) = [1 0];

        grad_c(:,2) = [-1 0];

        grad_c(:,3) = [0 1];
```

Initialization (initial point) - required

This file includes only the initial point, as a single variable named *x_initial*. It should be named **problemname*_init*. An initialization file must be provided to each problem loaded.

The initial point is a column vector with the corresponding coordinates.

e.g.

```
x_initial = [-1.2 1]';
```

BoostDMS

Problem (objective functions) – required

This file codes the different objective functions to optimize, defining the problem. As this algorithm is suited for multiobjective optimization, several objective functions might be included. The file can be located anywhere on the computer, but other files to load must be on the same directory. One or several problems can be solved at once (all from the same directory).

It receives a point *x* and outputs the result of the evaluation of that point, given the different objective functions (column vector of values).

e.g.

```
function [F] = ZDT1_example(x)

        numVar = length(x);

        g        = 1 + 9*sum(x(2:numVar))/(numVar-1);

        f(1)     = x(1);                                %function 1

        f(2)     = g*(1-sqrt(x(1)/g));                  %function 2

        F        = f';
```

Constraints - optional

This file includes all constraints that apply to the problem, except for problem bounds. These are provided in the initialization file. This file should be named **problemname*_const*, replacing **problemname** by the name of the corresponding problem. It is possible to solve both constrained and unconstrained problems in the same run, by uploading only some constraints files.

It receives a point *x* and outputs the constraints' values at the given point (column vector of *c* values, where *c* is the number of constraints). Constraints are always written in the form $C_i(x) \leq 0$. Check ***SID-PSM*** section for an example, since this part works similarly.

Initialization (initial points + bounds) - required

This file includes the initial point, as a single variable named *x_initial*, as well as the problem bounds. It should be named **problemname*_init*. An initialization file must be provided to each problem loaded.

The initial point is a column vector with the corresponding coordinates. The problem bounds (lower and upper) are also column vectors, where the size of the vector corresponds to the number of problem variables. Bounds are always required, being set to *-inf* or *+inf* if they do not apply to a given problem. Variable *x_initial* must be present, but may be empty depending on the *list* option chosen:

> **When *list* == 0** – the user can provide its own list of initial points to start the optimization procedure. Variable *x_initial* is then a matrix where each column corresponds to a different point. If variable *x_initial* is empty, the initial point is the middle point defined by the bounds.
>
> **When *list* == 1/2/3** – variable *x_initial* should be empty (if not, it will be ignored).
>
> **When *list* == 4** – the file to provide is the output of a previous run of the problem *BoostDMS_lastiteration_*problemname*.m* (relocated to the problem directory and renamed as **problemname*_init*). This file allows the user to continue running a problem that stopped for an unknown reason (*e.g.* hidden constraints) or with a more precise stopping criterion, by saving all the relevant variable information. Besides the initial points and bounds, this option features some additional information and is not intended to be user created.

In the case that the ***pareto_front*** option is inactive, only one point will be generated by the algorithm (rather than an approximation to the complete Pareto front). Thus, only the first point on *x_initial* would be considered as initialization.

e.g. ***(for list == 1/2/3)***

```
x_initial = [];

lowerbound = zeros(30,1);

upperbound = ones(30,1);
```

GLODS

Problem (objective function) - required

This file codes the objective function, defining the problem. It can be located anywhere on the computer, but the other files to load must be on the same directory. One or several problems can be solved at once (all from the same directory).

It receives a point *x* and outputs the result of the evaluation of that point (single value).

e.g.

```
function f = becker_lago(x)

f = (abs(x(1))-5)^2+(abs(x(2))-5)^2;
```

Initialization (initial points + bounds) - required

This file includes the initial point, as a single variable named *x_initial*, as well as the problem bounds. It should be named **problemname*_init,* replacing **problemname** by the name of the corresponding problem. An initialization file must be provided to each problem loaded.

The initial point is a column vector with the corresponding coordinates. The problem bounds (lower and upper) are also column vectors, where the size of the vector corresponds to the number of problem variables, and are always required. Variable *x_initial* must be present, but may be empty depending on the *list* option chosen:

> **When *list* == 0** – the user can provide its own list of initial points to start the optimization procedure. Variable *x_initial* is then a matrix where each column corresponds to a different point. If variable *x_initial* is empty, the initial point is the middle point defined by the bounds.
>
> **When *list* == 1/2/3** – variable *x_initial* should be empty (if not, it will be ignored).
>
> **When *list* == 4** – the file to provide is the output of a previous run of the problem *glods_lastiteration_*problemname*.m* (relocated to the problem directory and renamed as **problemname*_init*). This file allows the user to continue running a problem that stopped for an unknown reason (*e.g.* hidden constraints) or with a more precise stopping criterion, by saving all the relevant variable information. Besides the initial points and bounds, this option features some additional information and is not intended to be user created.

e.g. ***(for list == 1/2/3)***

```
x_initial = [];

lowerbound = -10*ones(2,1);

upperbound = 10*ones(2,1);
```

MultiGLODS

Problem (objective functions) - required

This file codes the different objective functions to optimize, defining the problem. As this algorithm is suited for multiobjective optimization, several objective functions might be included. The file can be located anywhere on the computer, but other files to load must be on the same directory. One or several problems can be solved at once (all from the same directory).

It receives a point *x* and outputs the result of the evaluation of that point, given the different objective functions (column vector of values).

e.g.

```
function [F] = ZDT1_example(x)

        numVar = length(x);

        g       = 1 + 9*sum(x(2:numVar))/(numVar-1);

        f(1)    = x(1);                             %function 1

        f(2)    = g*(1-sqrt(x(1)/g));               %function 2

        F       = f';
```

Initialization (initial points + bounds) - required

This file includes the initial point, as a single variable named *x_initial*, as well as the problem bounds. It should be named **problemname*_init,* replacing **problemname** by the name of the corresponding problem. An initialization file must be provided to each problem loaded.

The initial point is a column vector with the corresponding coordinates. The problem bounds (lower and upper) are also column vectors, where the size of the vector corresponds to the number of problem variables, and are always required. Variable *x_initial* must be present, but may be empty depending on the *list* option chosen:

> **When *list* == 0** – the user can provide its own list of initial points to start the optimization procedure. Variable *x_initial* is then a matrix where each column corresponds to a different point. If variable *x_initial* is empty, the initial point is the middle point defined by the bounds.
>
> **When *list* == 1/2/3** – variable *x_initial* should be empty (if not, it will be ignored).
>
> **When *list* == 4** – the file to provide is the output of a previous run of the problem *multiglods_lastiteration_*problemname*.m* (relocated to the problem directory and renamed as **problemname*_init*). This file allows the user to continue running a problem that stopped for an unknown reason (*e.g.* hidden constraints) or with a more precise stopping criterion, by saving all the relevant variable information. Besides the initial points and bounds, this option features some additional information and is not intended to be user created.

e.g. ***(for list == 0)***

```
x_initial(:,1) = 0.2*ones(30,1);

x_initial(:,2) = 0.6*ones(30,1);

lowerbound = zeros(30,1);

upperbound = ones(30,1);
```

Research Team

The research team responsible for the development of *boostDFO_Toolbox* is:

Ana Luísa Custódio (Universidade NOVA de Lisboa, CMA)

Vítor Duarte (Universidade NOVA de Lisboa, NOVA LINCS)

Pedro Medeiros (Universidade NOVA de Lisboa, NOVA LINCS)

Sérgio Tavares (Universidade NOVA de Lisboa)

Acknowledgements

Support for this project was provided by *Fundação para a Ciência e a Tecnologia* (Portuguese Foundation for Science and Technology) under the grant PTDC/MAT-APL/28400/2017.

We also acknowledge all the researchers responsible for the development of the algorithms integrated in the toolbox (information can be found in the *readme* files inside the corresponding algorithm folder).