Universidade de Aveiro Departamento de Engenharia Mecânica
2018

Miguel Jorge Guimarães de Oliveira

Análise de Métodos de Otimização Avançados em Projeto Mecânico

On the Use of Advanced Optimization Methods in Mechanical Design

Universidade de Aveiro Departamento de Engenharia Mecânica
2018

Miguel Jorge Guimarães de Oliveira

Análise de Métodos de Otimização Avançados em Projeto Mecânico

On the Use of Advanced Optimization Methods in Mechanical Design

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Mecânica, realizada sob orientação científica do Prof. Doutor António Gil D'Orey de Andrade Campos, Professor Auxiliar do Departamento de Engenharia Mecânica da Universidade de Aveiro, e do Prof. Doutor João Alexandre Dias de Oliveira, Professor Auxiliar do Departamento de Engenharia Mecânica da Universidade de Aveiro.

Apoio financeiro dos projetos UID/EMS/00481/2013-FCT e CENTRO-01-0145-FEDER-022083

o júri

presidente

Prof. Doutor Joaquim Alexandre Mendes de Pinho da Cruz
Professor Auxiliar da Universidade de Aveiro

Doutor João Filipe Moreira Caseiro
Investigador do CENTIMFE - Centro Tecnológico da Indústria de Moldes, Ferramentas Especiais e Plásticos

Prof. Doutor António Gil D'Orey de Andrade Campos
Professor Auxiliar da Universidade de Aveiro

agradecimentos

Ao Professor Doutor António Gil D'Orey de Andrade Campos, pela orientação, incentivo e disponibilidade sempre demonstrada ao longo deste trabalho. Pelo desafio proposto, que muito me enriqueceu e despertou um gosto especial por algoritmia e programação.

Ao Professor Doutor João Alexandre Dias de Oliveira, pela constante disponibilidade e incentivo demonstrado ao longo deste trabalho. Pela sua visão que muito contribuiu para o enriquecimento do trabalho.

Ao GRIDS, pelos meios disponibilizados e por ser um grupo de investigação dinâmico e em contacto com os alunos.

À Associação BEST Aveiro, na qual vivi alguns dos melhores momentos nesta universidade e que me possibilitou um enorme crescimento e desenvolvimento pessoal. A todas as pessoas que nela tive o privilégio de conviver e que constituem aquilo que a associação simboliza.

A todos os meus amigos, que me acompanharam e proporcionaram momentos inesquecíveis durante estes anos. O vosso apoio foi fundamental.

À minha família, que constitui a minha fundação. Em especial aos meus pais, que sempre me apoiaram incondicionalmente e sem os quais não conseguiria concretizar os meus objetivos.

A todos aqueles que de alguma forma influenciaram e contribuíram para a concretização deste objetivo. Obrigado.

palavras-chave métodos avançados de otimização, projeto mecânico, linguagens de programação, processamento paralelo

resumo

Métodos avançados de otimização têm sido amplamente aplicados ao projeto mecânico, principalmente pela sua capacidade de resolver problemas complexos que técnicas tradicionais de otimização como os métodos baseados em gradiente não apresentam. Devido à sua crescente popularidade, o número de algoritmos encontrados na literatura é vasto. Neste trabalho são implementados três algoritmos distintos, Otimização por Bando de Partículas (PSO), Evolução Diferencial (DE) e Otimização Baseada no Ensino-Aprendizagem (TLBO). Inicialmente, a aplicação destes algoritmos é analisada numa função composta e em três problemas de minimização de projeto mecânico (o peso de um redutor de velocidade, o volume de uma estrutura de três barras e a área de uma placa quadrada com um furo circular).

Além disso, com o aumento do número de algoritmos existentes, a escolha de ferramentas de programação para implementá-los também é vasta e geralmente feita considerando critérios subjetivos ou dificuldades no uso de estratégias de melhoria como processamento paralelo. Deste modo, no presente trabalho é realizada uma análise de ferramentas de programação aplicadas a algoritmos metaheurísticos, utilizando linguagens de programação com distintas características: Python, MATLAB, Java e C++. Os algoritmos e problemas selecionados são programados em cada linguagem de programação, e inicialmente comparados numa implementação de processamento sequencial. Além disso, de forma a analisar possíveis ganhos de desempenho, são implementados procedimentos de processamento paralelo utilizando recursos de cada linguagem de programação.

A aplicação dos algoritmos aos problemas de projeto mecânico demonstra bons resultados nas soluções obtidas. Os resultados, em termos de tempo computacional, de processamento sequencial e paralelo, apresentam diferenças consideráveis entre as linguagens de programação. A implementação de procedimentos de processamento paralelo demonstra benefícios significativos em problemas complexos.

keywords advanced optimization methods, mechanical design, programming languages, parallel processing

abstract

Advanced optimization methods are widely applied to mechanical design, mainly for its abilities to solve complex problems that traditional optimization techniques such as gradient-based methods do not present. With its increasing popularity, the number of algorithms found in the literature is vast. In this work three algorithms are implemented, namely Particle Swarm Optimization (PSO), Differential Evolution (DE) and Teaching-Learning-Based Optimization (TLBO). Firstly, the application of these algorithms is analyzed for a composition function benchmark and three mechanical design minimization problems (the weight of a speed reducer, the volume of a three-bar truss and the area of a square plate with a cut-out hole).

Furthermore, as the scope of available algorithms increases, the choice of programming tools to implement them is also vast, and generally made considering subjective criteria or difficulties in using enhancing strategies such as parallel processing. Thereby an analysis of programming tools applied to metaheuristic algorithms is carried out using four programming languages with distinct characteristics: Python, MATLAB, Java and C++. The selected algorithms and problems are coded using each programming language, which are initially compared in a sequential processing implementation. Additionally, in order to analyze potential gains in performance, parallel processing procedures are implemented using features of each programming language.

The application of the algorithms to the mechanical design problems demonstrates good results in the achieved solutions. In what concerns to the computational time, sequential and processing results present considerable differences between programming languages while the implementation of parallel processing procedures demonstrates significant benefits for complex problems.

"Life begins at the end of your comfort zone."
Neale Donald Walsch

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The framework of the work is presented and the general objectives defined. The structure of the document is described.

1.1 Framework

Optimization stands as a key role in solving engineering problems. From the design of aircraft and aerospace structures, water resources systems to electrical networks, different methods are being applied to obtain the best viable solutions [Rao 2009]. Nowadays, the use of optimization techniques can be transversal to almost every scientific area, from simple personal goals to industrial applications.

In the optimization of mechanical design, depending on the requirements, several objectives can be considered (e.g. weight, strength). Furthermore, if the mechanical system is too complex it can lead to a complicated objective, involving several parameters to optimize and requirements to satisfy. In order to simplify the optimization process, the system is usually split into several subsystems, which in turn are easier to optimize. For example, in a power transmission system, the optimization of the gearbox is computationally and mathematically simpler than the optimization of the whole system [Rao and Savsani 2012]. Different optimization methods or algorithms have been applied to solve mechanical design problems, which can be classified into two different types: traditional optimization methods and advanced optimization methods.

The use of traditional optimization methods, called deterministic algorithms, can be dated to the days of Newton, Lagrange and Cauchy [Rao 2009]. Some algorithms can be classified as gradient-based, as the Newton-Raphson method, which uses the information of the derivatives and values of the function. Gradient-free or direct search algorithms are an alternative, as they do not use any derivative, but only function values. One example of the latter is the Nelder-Mead simplex method [Nelder and Mead 1965]. Overall, traditional optimization techniques have been used with some success in mechanical design problems [Rao and Savsani 2012]. Some advantages have been appointed to traditional methods, as they present good convergence rates and are efficient when searching for local optima [Yang *et al.* 2016]. However, they also present some drawbacks, as getting trapped in local optima, not providing guarantees of global optimality [Yang *et al.* 2018]. Moreover,

as the function gradients might be difficult to estimate when applied to complex problems with a large number of variables and constraints, they are very demanding to solve.

The advanced optimization methods combine a heuristic component with probabilistic transition rules or randomization. These methods have been gaining popularity due to properties traditional methods do not present, allowing quality solutions to be found, but not guaranteeing optimal solutions [Yang *et al.* 2018]. The randomization component allows the algorithms to search globally, avoiding the algorithm to be trapped in local optima. These algorithms can be evolutionary or even nature-inspired, depending on the source of inspiration and on mathematical equations. Examples of advanced optimization methods are the Genetic Algorithms (GA), Particle Swarm Optimization (PSO), Simulated Annealing (SA), Differential Evolution (DE), among others. Analogous to the traditional optimization methods, the use of advanced methods present advantages and disadvantages. From a general perspective, these algorithms are more flexible, simpler and have the capability to deal with more complex problems [Yang *et al.* 2016]. Additionally, the potential of global exploration enhances the chances to reach an optimum solution, as well as the aptitude to deal with a variety and magnitude of problems. Nevertheless, the computational cost tends to be higher than traditional techniques and exact solutions are not repeatable [Yang *et al.* 2016].

Application of the advanced optimization techniques in mechanical design problems has been widely researched, as several authors have studied how to improve the solution of mechanical design optimization problems by employing these techniques. Rao and Savsani [Rao and Savsani 2012] compiled and studied a large number of problems found in the literature using metaheuristic algorithms, such as a gear train, radial ball bearing, Belleville's spring, multi-plate disc clutch brake, robot gripper, pressure vessel, hydrostatic bearing, four stage gear train, among others. Other authors [He *et al.* 2004, Yang and Deb 2010, Baykasoglu 2012, Guedria 2015, Alcántar *et al.* 2017, Zhang *et al.* 2018] have also employed different algorithms and improved variations to a broad range of problems.

When the problem to optimize is relatively simple, its solution might be easily found using analytical methods or even graphic representations. However, if the problem is complex, analytical methods prove to be non-efficient. In the latter scenario, implementation of the optimization methods can be achieved using programming tools that are easily available and automate the process. Nowadays, to solve this kind of problems, general mathematical/technical computing software or programming languages are usually used. Furthermore, it is well known that some programming languages are better suited for a specific application than others, but selecting the optimal programming language involves the consideration of several factors – targeted platform, time of development, readability, writability, among others [Reghunadh and Jain 2011, Sebesta 2012]. Additionally, the majority of programming languages are prepared for multi-paradigm approaches such as procedural or object-oriented programming which might be better suited for some applications than others. However, the choice is generally taken considering subjective criteria or difficulties in using enhancing strategies such as parallel programming tools.

1.2 Objectives

The main goal of this work is to analyze the use of advanced optimization methods in mechanical design problems in order to better understand their characteristics and poten-

tial. This analysis is carried out using three distinct algorithms selected for its simplicity and vast application to mechanical design problems: Particle Swarm Optimization (PSO), Differential Evolution (DE) and Teaching-Learning-Based Optimization (TLBO). Furthermore, it is intended to gain knowledge on available programming tools for the implementation of the algorithms either in terms of development strategies and computational performance. Thereby each algorithm is implemented using different programming languages: Python, MATLAB, Java and C++. Additionally, it is intended to analyze strategies to boost the computational performance, using parallel processing procedures available in each programming language.

1.3 Reading Guidelines

The present document is divided into six chapters, organized as here described.

Chapter 1 - An introduction to the work is presented and its framework described. The general objectives are briefly presented.

Chapter 2 - General concepts used in this work are presented, such as the mathematical formulation of an optimization problem and a method to deal with constrained optimization problems. Moreover, general concepts related to parallel processing are described, as well as parameters used in the evaluation of computational performance.

Chapter 3 - It is presented the mathematical formulations of the implemented advanced optimization methods. Additionally, variants and different strategies of these algorithms are described and literature applications in mechanical design problems are briefly presented.

Chapter 4 - The implementation procedures are described, in particular, the application of parallel processing in the advanced optimization methods. A description of the implemented programming languages and its features is also presented, as well as the mathematical formulation and computational implementation of different applications. Finally, a succinct description of the work's implementation flow is presented.

Chapter 5 - The results and its analysis are presented. Firstly, the advanced optimization methods are compared, based on its efficiency and ability in the resolution of implemented applications. Secondly, the analysis is focused on the computational performance of the algorithms and programming languages, both in sequential and parallel processing implementations.

Chapter 6 - General conclusions and perspectives of future work are presented.

Chapter 2

Fundamentals and General Concepts

A mathematical formulation of an optimization problem and a method to deal with constrained optimization problems are presented. General concepts and approaches in parallel processing are presented, as well as parameters used in the evaluation of performance.

2.1 Fundamentals of Optimization

2.1.1 Mathematical Formulation of an Optimization Problem

Before solving any optimization problem, it is necessary to correctly formulate it and translate it into mathematical language. Mathematically, the properties of a system to be optimized are defined through a function, named objective function. The goal of an optimization problem is to minimize or maximize the objective function. This function relates the properties of the system through D parameters, named design variables or decision variables.

An optimization or mathematical programming problem can be stated as

$$\text{Find } \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_D \end{bmatrix} \text{ which minimizes } f(\mathbf{x}) \,, \tag{2.1}$$

subject to the constraints

$$g_j(\mathbf{x}) \leq 0 \,, \qquad j = 1, 2, ..., m \,, \tag{2.2}$$

$$h_k(\mathbf{x}) = 0 \,, \qquad k = 1, 2, ..., l \,, \tag{2.3}$$

$$x_i^{\min} \leq x_i \leq x_i^{\max} \,, \qquad i = 1, 2, ..., D \,, \tag{2.4}$$

where $\mathbf{x}$ is a D-dimensional vector of the design variables, $f(\mathbf{x})$ is termed objective function, and $g(\mathbf{x})$ and $h(\mathbf{x})$ are known as inequality and equality constraints, respectively. The last

constraints are also referred to as simple constraints or side constraints. In a compact and generic way, an optimization problem can be defined as

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{x}) \,, && && (2.5)\\ \text{subject to} \quad & g_j(\mathbf{x}) \leq 0 \,, && j = 1, 2, ..., m \,, && (2.6)\\ & h_k(\mathbf{x}) = 0 \,, && k = 1, 2, ..., l \,, && (2.7)\\ & x_i^{\min} \leq x_i \leq x_i^{\max} \,, && i = 1, 2, ..., D \,. && (2.8) \end{aligned}$$

An optimization problem that is either subject to side, inequality or equality constraints is said to be a constrained optimization problem. However, some optimization problems are not subject to inequality and equality constraints, thereby known as unconstrained optimization problems. In problems subject to inequality and equality constraints, solutions can be either classified as feasible or unfeasible. If the solutions are feasible, they satisfy the imposed constraints and are thereby a possible valid solution to the problem. However, if the solutions are infeasible, they do not satisfy the imposed constraints and should be rejected. To deal with constrained optimization problems, one of the methods is to rely on an exterior penalty function [Coello 2002].

2.1.2 Penalty Function for Constrained Optimization Problems

The method of the exterior penalty function is a simple and common approach to handle constraints. The idea behind this method is to transform a constrained optimization problem into an unconstrained problem, by adding (or subtracting) a penalty function P to the objective function. Solutions that violate the constraints are penalized and the problem is defined as

$$\begin{aligned} \text{minimize} \quad & F(\mathbf{x}, r_{\mathrm{h}}, r_{\mathrm{g}}) = f(\mathbf{x}) + P(\mathbf{x}, r_{\mathrm{h}}, r_{\mathrm{g}}) \,, && (2.9)\\ \text{subject to} \quad & x_i^{\min} \leq x_i \leq x_i^{\max} \,, \quad i = 1, 2, ..., D \,, && (2.10) \end{aligned}$$

where r_{h} and r_{g} are penalty factors and F designates the augmented objective function. A general formulation of the exterior penalty function is defined as

$$P(\mathbf{x}, r_{\mathrm{h}}, r_{\mathrm{g}}) = r_{\mathrm{h}} \left[\sum_{k=1}^{l} [h_j(\mathbf{x})]^{\gamma} \right] + r_{\mathrm{g}} \left[\sum_{j=1}^{m} \left[\max\{0, g_j(\mathbf{x})\} \right]^{\beta} \right] . \tag{2.11}$$

where, γ and β are positive penalty constants. The penalty function P is non-existent when the constraints' functions h and g are not active.

Furthermore, an exterior penalty function can be classified as dynamic. When the current iteration number is associated with the corresponding penalty factors, normally defined in such a way that the value of the penalty function increases over time. Joines and Houck [Joines and Houck 1994] proposed a dynamic penalty method in which individuals are evaluated at iteration i as

$$F(\mathbf{x}) = f(\mathbf{x}) + (C\, i)^{\alpha} \mathrm{SVC}(\mathbf{x}, \beta) \,, \tag{2.12}$$

where C, α and β are pre-defined constants and $\mathrm{SVC}(\mathbf{x}, \beta)$ is defined as

$$\text{SVC}(\mathbf{x}, \beta) = \sum_{k=1}^{l} H_k(\mathbf{x}) + \sum_{j=1}^{m} G_j^{\beta}(\mathbf{x}) \ , \tag{2.13}$$

and functions H_k and G_j are defined as

$$H_k(\mathbf{x}) = \begin{cases} 0 & \text{if } -\epsilon \leq h_k(\mathbf{x}) \leq \epsilon \\ |h_k(\mathbf{x})| & \text{otherwise} \end{cases} \qquad \text{and} \tag{2.14}$$

$$G_j(\mathbf{x}) = \begin{cases} 0 & \text{if } g_j(\mathbf{x}) \leq 0 \\ |g_j(\mathbf{x})| & \text{otherwise} \end{cases} . \tag{2.15}$$

In this approach, equality constraints are transformed into inequality constraints, where ϵ is the allowed tolerance (normally a very small value). These methods are simple and easy to implement in the advanced optimization methods. However, a disadvantage is related to the necessity of tuning the parameters depending on the optimization problem.

2.2 General Concepts of Parallel Processing

The use of parallel processing implementations is concerned with the need to increase the performance in the computation of applications requiring great processing capacity. A variety of engineering applications, especially of numerical simulation, require lots of resources and the time it takes to solve them may not be affordable.

A simple definition of parallel processing is when two or more activities happen at the same time. When relating parallel processing to computer systems, it means a single system performing multiple independent activities in parallel, rather than sequentially or one after the other (sequential processing), with the goal of decreasing the computational time of a specific task [Williams 2012]. Historically, computers have had one processor, with a single processing unit or core, not allowing to truly run multiple applications simultaneously. However, with the development of technology, computers with multiple core processors on a single chip have become common, allowing to genuinely run more than one task simultaneously.

The task of implementing parallel programs can become more complex when compared to sequential programs, and it does not guarantee an increase in the performance. To ensure that a better performance is achieved with the implementation of a parallel system, the nature of the application and the approach need to be considered.

2.2.1 Approaches to Parallel Processing

A simplified approach to parallel processing is to divide its implementation into two paradigms: multiple threads and multiple processes [Pacheco 2011, Williams 2012]. The first approach uses multiple threads on a single processor, which are commonly designated as lightweight processes. Threads run independently from each other and may run different sequences of instructions. In opposition to multiple processes, threads share the same memory address space and a specific thread's data can be accessed from all threads. An advantage in using multiple threads is that they are lighter and can be created, destroyed and switched faster than processes.

Alternatively, the second approach consists of using multiple processes, where an application is divided into multiple, independent, single-threaded processes that are run at the same time. These processes are then able to communicate, passing messages to each other through interprocess communication channels. A disadvantage on the use of multiple processes is that communication between processes is generally complicated or slow. Furthermore, the initialization of processes might take time and its maintenance requires that the operating system allocates resources to them. However, operating systems typically provide protection between processes in order to avoid that a process unintentionally modifies data associated with another process, as each process has a dedicated memory address space. In Figure 2.1 an illustrative representation of the communication in both approaches is presented.

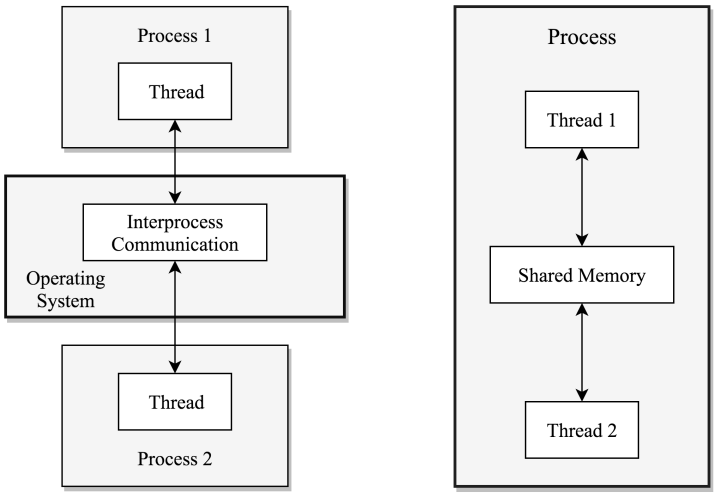


Figure 2.1: Illustrative representation of the communication between (a) processes running in parallel and (b) threads running in parallel in a single process.

Additionally, two types of parallelism can be implemented in both of the described paradigms: task parallelism and data parallelism [Cung *et al.* 2002, Pacheco 2011]. In the case of task parallelism, a program is divided into several instructions that run in parallel, thus reducing the total computational time. However, to implement this strategy it is necessary to guarantee those parallel instructions are not dependent. Data parallelism concerns with the division of data into small chunks to be processed in a set of instructions.

The advanced optimization methods implemented in this work and later described are more suitable for a parallel processing implementation using a data parallelism paradigm. These algorithms are structured in a way that several instructions are applied to a chunk of data while some instructions are order dependent. Thereby, using a data parallelism paradigm it is possible to evaluate the data independently and simultaneously. Furthermore, the implementation of parallel processing techniques follows a shared memory *Multiple Instruction stream/ Multiple Data stream* (MIMD) computer architecture [Duncan 1990]. In a MIMD architecture, computers with multiple processors run, simultaneously and independently, different streams of instructions over several streams of data. The

shared memory architecture allows for interprocess communication by providing a shared memory that each processor can address. Moreover, each processor in a shared memory architecture also has a local memory used as a cache[1].

2.2.2 Evaluation of Performance

As the goal in parallel processing implementations is, usually, to increase the overall performance, objective parameters are used to evaluate the achieved performance. Generally, the evaluation of performance consists of the comparison of the time of execution between sequential and parallel programs.

In terms of evaluating the global performance of a program in a computer with p processors, the main used parameters are the speedup and efficiency. The speedup S_p demonstrates the relationship between the time of execution in sequential processing and the time of execution in parallel processing, showing how many times the parallel program is faster than the sequential program. Speedup is defined as

$$S_\mathrm{p} = \frac{t_\mathrm{s}}{t_\mathrm{p}} \,, \tag{2.16}$$

where t_s is the time of execution of the sequential program and t_p the time of execution of the parallel program. In an ideal parallel program, it is expected to achieve a linear speedup of $S_\mathrm{p} = p$, corresponding to an efficiency e_p of 1, given by

$$e_\mathrm{p} = \frac{S_\mathrm{p}}{p} \,. \tag{2.17}$$

The efficiency is then a parameter that demonstrates the degree of utilization of the available computational resources, representing the fraction of the time spent by the processors in the execution of the parallel program. However, the time consumed by processors is not uniquely dedicated to the execution of the parallel program. Parts of the time of execution is dedicated to other tasks, such as communication between processors or threads, and executing sequential tasks of the program. For these reasons, real programs typically present values of speedup lower than p and efficiency lower than 1.

[1] Cache is a hardware or software component that is used to temporarily store data

Chapter 3

Advanced Optimization Methods

The advanced optimization methods, including their mathematical formulations are described. Variants and different strategies of the algorithms are described and applications of the algorithms in mechanical design problems presented.

3.1 Particle Swarm Optimization

First introduced by Kennedy and Eberhart in 1995 [Kennedy and Eberhart 1995], the Particle Swarm Optimization (PSO) method is based on the communication and social behavior of insects, birds or fish. Each particle or individual behaves both using its own intelligence and the collective intelligence of the swarm. The group of all particles constitute the swarm or population and move through the search range until it finds the best possible solution. If one particle discovers good solutions, it is able to share that information with other particles, which will also be able to follow that particle, even if they are far away in the design space.

The swarm is assumed to have a fixed size of particles, with each particle initially located at a random position in the multidimensional design space. Each particle is represented by two attributes, a *position* and a *velocity*. Particles move around the design space and remember their individual best-discovered position (in terms of objective function value). These individual best positions are shared between particles which then adjust their own positions and velocities based on the best position of the swarm (deterministic) and a randomly chosen acceleration factor (stochastic).

3.1.1 Algorithm Formulation

Kennedy and Eberhart [Kennedy and Eberhart 1995] proposed a *standard* PSO where each particle n is represented by its position vector $\mathbf{x}_n$ and a velocity vector $\mathbf{v}_n$, with equal dimension D representative of the number of design variables. Initially, each particle is randomly generated inside the design space limited by the upper and lower bounds of each design variable. As each particle is defined for its position in the design space, let $\mathbf{x}_{\max}$ be the vector containing the upper bounds and $\mathbf{x}_{\min}$ the vector containing the lower bounds, for each design variable. The initial position of each particle $\mathbf{x}_n^0$ is then defined as

$$\mathbf{x}_n^0 = \mathbf{x}_{\min} + \mathbf{r}\left(\mathbf{x}_{\max} - \mathbf{x}_{\min}\right), \tag{3.1}$$

where $\mathbf{r}$ is a vector of random numbers uniformly distributed in the range [0, 1]. In the implementation presented in this work the initial velocity of each particle $\mathbf{v}_n^0$ is defined similarly to $\mathbf{x}_n^0$ as

$$\mathbf{v}_n^0 = \mathbf{v}_{\min} + \mathbf{r}\left(\mathbf{v}_{\max} - \mathbf{v}_{\min}\right), \tag{3.2}$$

where $\mathbf{v}_{\max} = (\mathbf{x}_{\max} - \mathbf{x}_{\min})/2$ is the vector containing the upper bounds and $\mathbf{v}_{\min} = -\mathbf{v}_{\max}$ the vector containing the lower bounds for each design variable velocity.

At each iteration i, the velocity $\mathbf{v}_n^{i+1}$ of each particle is updated based on the current velocity $\mathbf{v}_n^i$ and position $\mathbf{x}_n^i$, as according to

$$\mathbf{v}_n^{i+1} = \mathbf{v}_n^i + c_1\mathbf{r}_1\left(\mathbf{p}_{\text{best}_n} - \mathbf{x}_n^i\right) + c_2\mathbf{r}_2\left(\mathbf{g}_{\text{best}} - \mathbf{x}_n^i\right), \tag{3.3}$$

where c_1 and c_2 are parameters representing, respectively, the influence of individual experience (cognitive parameter) and the influence of global experience (social parameter). $\mathbf{r}_1$ and $\mathbf{r}_2$ are vectors of random numbers uniformly distributed in the range [0, 1]. $\mathbf{p}_{\text{best}_n}$ represents the best position of particle n obtained during previous iterations, while $\mathbf{g}_{\text{best}}$ represents the best position of all particles in the population. The position $\mathbf{x}_n^{i+1}$ is afterwards updated based on the current position $\mathbf{x}_n^i$ and the updated velocity $\mathbf{v}_n^{i+1}$,

$$\mathbf{x}_n^{i+1} = \mathbf{x}_n^i + \mathbf{v}_n^{i+1}. \tag{3.4}$$

At every iteration, the position of each particle is then evaluated, repeating the cycle until a stopping criterion is satisfied. The PSO implementation is described in the pseudo-code presented in Figure 3.1.

3.1.2 Operational Parameters

3.1.2.1 Population

The number of particles in the population is a parameter initially chosen. A general rule to determine the best number of particles to select does not exist, but experience and the nature of the problem should help to determine the best choice. A good number to select is one that allows the population to cover the design space.

Different studies on the influence of the size of the population in PSO have been made in the past decades, although a consensus was never reached. Trelea [Trelea 2003] stated that in the majority of the problems the success rate (number of times it reaches global optimum) of the algorithm improves significantly with a bigger number of particles, but it also increases the computational time. Chen *et al.* [Chen *et al.* 2015] showed that for problems of low dimension ($D \leq 50$), it is better to use a number of particles bigger than the dimension of the problem, as opposed to problems with high dimension where a smaller number of particles is preferred.

3.1.2.2 Acceleration Parameters

The acceleration parameters c_1 and c_2 control the step each particle can take at every iteration. In the originally proposed PSO, Kennedy and Eberhart [Kennedy and Eberhart

```
Generate initial population with $n$ particles
Initialize each particle with random position $\mathbf{x}_n^0$ and random velocity $\mathbf{v}_n^0$
Set acceleration parameters $c_1$ and $c_2$
Set iteration counter $i = 0$
while stopping criterion do
    for loop over n particles do
        Evaluate the objective function $f(\mathbf{x}_n^i)$
        if $f\left(\mathbf{x}_n^i\right) < f\left(\mathbf{p}_{\text{best}_n}\right)$ or $i == 0$ then
            $\mathbf{p}_{\text{best}_n} = \mathbf{x}_n^i$
        end
    end
    Find $\mathbf{g}_{\text{best}}$ from $n$ particles
    for loop over n particles do
        // Update Velocity
        $\mathbf{r}_1 = \text{rand}(0, 1)$
        $\mathbf{r}_2 = \text{rand}(0, 1)$
        $\mathbf{v}_n^{i+1} = \mathbf{v}_n^i + c_1\mathbf{r}_1\left(\mathbf{p}_{\text{best}_n} - \mathbf{x}_n^i\right) + c_2\mathbf{r}_2\left(\mathbf{g}_{\text{best}} - \mathbf{x}_n^i\right)$
        Check lower and upper bounds of $\mathbf{v}_n^{i+1}$
        // Update Position
        $\mathbf{x}_n^{i+1} = \mathbf{x}_n^i + \mathbf{v}_n^{i+1}$
        Check lower and upper bounds of $\mathbf{x}_n^{i+1}$
    end
    Update iteration counter $i = i + 1$
end
Post-process and output final results
```

Figure 3.1: Pseudo-code for the implementation of Particle Swarm Optimization.

1995] defined these two parameters as being positive constants equal to 2, so that the cognitive and social terms equally influence the new velocity of the particles. Further research was developed to evaluate if these two parameters could be linearly modified or if other values would present better results. Suganthan [Suganthan 1999] studied the effect of linearly decreasing both parameters but concluded that using constant values would present better results, although not necessarily equal to 2.

3.1.2.3 Inertia Weight

The inertia weight w was first introduced by Shi and Eberhart in 1998 [Shi and Eberhart 1998a] to control the balance between local search and global search ability in the velocity of each particle. This new parameter influences the updated velocity $\mathbf{v}_n^{i+1}$, now defined as

$$\mathbf{v}_n^{i+1} = w\mathbf{v}_n^i + c_1\mathbf{r}_1\left(\mathbf{p}_{\text{best}_n} - \mathbf{x}_n^i\right) + c_2\mathbf{r}_2\left(\mathbf{g}_{\text{best}} - \mathbf{x}_n^i\right). \tag{3.5}$$

In the first study developed by Shi and Eberhart, a fixed inertia weight was tested in the range [0, 1.4], demonstrating that, on average, a value in the range [0.9, 1.2] will have a bigger chance to find the global optimum with a reasonable convergence rate. A time decreasing inertia weight in the range of [0, 1.4] also proved to have significant improvements [Shi and Eberhart 1998a]. A further study by the same authors [Shi and Eberhart 1998b] considered a linear decreasing inertia weight in the range of [0.4, 0.9] demonstrating even better results than previous studies. In this formulation a bigger inertia weight at the beginning of the optimization process promotes global exploration, avoiding the algorithm to be trapped in local optimum and a smaller inertia weight at the end enables the algorithm to refine the solution, promoting local exploration.

3.1.3 Variants of the Standard Particle Swarm Optimization

After the introduction of the standard algorithm, several authors proposed modifications or additions to improve the algorithm robustness and its convergence rate. Some authors focused on the addition of parameters to the algorithm, as the case of the inertia weight [Shi and Eberhart 1998a, Trelea 2003], while others proposed quasi-new algorithms.

Veeramachaneni *et al.* [Veeramachaneni *et al.* 2003] proposed a significant modification to the standard algorithm to prevent a premature convergence to local optima. In the newly proposed variant named Fitness-Distance-Ratio-based Particle Swarm Optimization (FDR-PSO), particles move towards nearby particles with a more successful search history, instead of just the best position discovered so far. Experiments proved that this modification improved the algorithm performance compared to the standard formulation.

Liu and Abraham [Liu and Abraham 2005] introduced *turbulence* to PSO to overcome the premature convergence problem, naming the new variant Turbulent Particle Swarm Optimization (TPSO). The idea behind this modification is to control the velocity of the particles implementing a minimum velocity threshold that is adaptively controlled. This way if a particle is trapped in local optima it is able to continue exploring the design space until the algorithm converges.

Knowledge-based Cooperative Particle Swarm Optimization (KCPSO) was introduced by Jie *et al.* [Jie *et al.* 2008] to tackle the premature convergence in complex problems. This variant of PSO simulates the self-cognitive and self-learning process of evolutionary species in a special environment. Particles are divided into sub-swarms to maintain diversity and

carry out local explorations, while information is shared between sub-swarms to maintain global exploration.

To improve the computational cost of PSO in demanding optimization problems, some authors [Schutte *et al.* 2004, Zhou and Tan 2009, Qu *et al.* 2017] proposed a parallel implementation of the standard algorithm.

3.2 Differential Evolution

The Differential Evolution (DE) algorithm was originally introduced by Storn and Price in their nominal papers [Storn 1996, Storn and Price 1997]. DE is a simple evolutionary algorithm of easy understanding and implementation, similar to pattern search and genetic algorithms due to its use of mutation, crossover and selection. As a population-based-stochastic algorithm, DE creates new solutions based on the combination of the progenitor with different members of the population.

3.2.1 Algorithm Formulation

A population of n vectors of D dimension is initially randomly generated and should try to cover the design space as much as possible. At each generation i, a vector $\mathbf{x}_n^i$ represents a position in the design space and is limited by its upper and lower bounds, respectively, $\mathbf{x}_{\max}$ and $\mathbf{x}_{\min}$. The generation of initial vectors is given by

$$\mathbf{x}_n^0 = \mathbf{x}_{\min} + \mathbf{r}\left(\mathbf{x}_{\max} - \mathbf{x}_{\min}\right), \tag{3.6}$$

where $\mathbf{r}$ is a vector of random numbers uniformly distributed in the range [0, 1]. The optimization process is then initialized with DE being divided into three main steps: mutation, crossover and selection.

The mutation scheme allows for a more diversified and robust search in the design space. In this step, for every $\mathbf{x}_n^i$ vector, designated target vector, a mutant vector $\mathbf{v}_n^{i+1}$ is generated by randomly choosing three mutually different vectors $\mathbf{x}_{r_1}^i$, $\mathbf{x}_{r_2}^i$ and $\mathbf{x}_{r_3}^i$, where r_1, r_2 and r_3 are integer values from a sample in the range of [1,2, ..., n]. The mutant vector $\mathbf{v}_n^{i+1}$ is then given by

$$\mathbf{v}_n^{i+1} = \mathbf{x}_{r_1}^i + F\left(\mathbf{x}_{r_2}^i - \mathbf{x}_{r_3}^i\right), \tag{3.7}$$

where F is a positive integer that controls the ratio in which population evolves. This parameter is often referred to as scale factor. The vectors $\mathbf{x}_{r_2}^i$ and $\mathbf{x}_{r_3}^i$ should also be different from the current $\mathbf{x}_n^i$ vector. Depending on the strategy used, $\mathbf{x}_{r_1}^i$ can be chosen randomly from the population or even be the best vector from the previous generation.

To complement the mutation strategy, DE introduces crossover, controlled by a probability parameter C_r that defines the rate or probability for crossover. In this step, trial vector $\mathbf{u}_n^{i+1}$ is generated from two different vectors, the mutant vector $\mathbf{v}_n^{i+1}$ and the target vector $\mathbf{x}_n^i$. The type of crossover can either be binomial or exponential. In the binomial crossover, the trial vector $\mathbf{u}_n^{i+1}$ is generated as

$$u_{n,j}^{i+1} = \begin{cases} v_{n,j}^{i+1} & \text{if } r_j \leq C_\mathrm{r} \\ x_{n,j}^i & \text{otherwise} \end{cases}, \tag{3.8}$$

where r_j is a random number uniformly distributed in the range [0, 1]. Each parameter $j = 1, 2, ..., D$ of the trial vector $\mathbf{u}_n^{i+1}$ is determined independently from each other and to determine which vector contributes a given parameter, C_r is compared to r_j. If r_j is less than or equal to C_r , the trial vector parameter $u_{n,j}^{i+1}$ is inherited from the mutant vector $\mathbf{v}_n^{i+1}$, otherwise the parameter is copied from the vector $\mathbf{x}_n^i$. In the exponential crossover a random position $j = 1, 2, ..., D$ is selected and starting from that position the trial vector $\mathbf{u}_n^{i+1}$ receives a parameter from the mutant vector $\mathbf{v}_n^{i+1}$ until C_r is less than r_j, a random number uniformly distributed in the range [0,1]. From that point forward $\mathbf{u}_n^{i+1}$ copies all the remaining parameters from $\mathbf{v}_n^{i+1}$.

Finally, the selection step decides if the trial vector $\mathbf{u}_n^{i+1}$ replaces the target vector $\mathbf{x}_n^i$ in the population. The objective function is evaluated at $\mathbf{u}_n^{i+1}$ and if its value is less than or equal to the value at $\mathbf{x}_n^i$, $\mathbf{u}_n^{i+1}$ replaces $\mathbf{x}_n^i$ in the population as

$$\mathbf{x}_n^{i+1} = \begin{cases} \mathbf{u}_n^{i+1} & \text{if } f\left(\mathbf{u}_n^{i+1}\right) \leq \mathbf{x}_n^i \\ \mathbf{x}_n^i & \text{otherwise} \end{cases}. \tag{3.9}$$

The procedure repeats up until a termination criterion is satisfied as described in the pseudo-code of Figure 3.2.

3.2.2 Operational Parameters

3.2.2.1 Population

Similar to the PSO algorithm, the size of the population n in DE is an important parameter to achieve satisfying results. According to Storn and Price [Storn 1996] a reasonable size for the population is between $5 \times D$ and $10 \times D$, but the algorithm requires only a minimum of 4 to ensure that there are enough different vectors to work with.

3.2.2.2 Scale Factor

Storn [Storn 1996] initially defined F as a real and constant factor $\in [0, 2]$, but to avoid a premature convergence it is important that F has sufficient magnitude, usually in the range of [0, 1]. Although $F > 1$ is a possible choice, solutions tend to be inferior and computationally less efficient compared to values of $F < 1$. When $F = 1$, the combination of vectors become indistinguishable, reducing the number of mutant vectors by half [Price *et al.* 2005].

3.2.2.3 Crossover Probability

The crossover parameter C_r defines the probability of a parameter of the trial vector being inherited from the mutant vector. Storn [Storn 1996] defined C_r to be a value in the range of [0,1], where a low C_r corresponds to a low crossover rate and a high C_r to a high crossover rate. Storn and Price [Storn and Price 1997] found that using $C_\mathrm{r} = 0.1$ would be a good first choice, but higher values of $C_\mathrm{r} \in [0.9, 1.0]$ increased the speed of convergence and would also make a good first choice to see if a quick solution was found.

```
1  Generate initial population with n vectors of random position x_n^0
2  Evaluate the objective function f(x_n^0)
3  Set scale factor F and crossover probability C_r
4  Set iteration counter i = 0
5  while stopping criterion do
6      for loop over n vectors do
           // Mutation
7          Randomly choose three vectors x_{r1}^i, x_{r2}^i and x_{r3}^i all ≠ x_n^i
8          v_n^{i+1} = x_{r1}^i + F(x_{r2}^i − x_{r3}^i)
9          Check lower and upper bounds of v_n^{i+1}
           // Crossover
10         for j = 1, 2, ..., D do
11             r_j = rand(0, 1)
12             if r_j ≤ C_r then
13                 u_{n,j}^{i+1} = v_{n,j}^{i+1}
14             else
15                 u_{n,j}^{i+1} = x_{n,j}^{i+1}
16             end
17         end
           // Selection
18         Evaluate the objective function f(u_n^{i+1})
19         if f(u_n^{i+1}) < f(x_n^i) then
20             x_n^{i+1} = u_n^{i+1}
21         end
22     end
23     Update iteration counter i = i + 1
24 end
25 Post-process and output final results
```

Figure 3.2: Pseudo-code for the implementation of Differential Evolution.

3.2.3 Strategies and Variants of Differential Evolution

The presented strategy of DE is not the only one existent, as Storn and Price [Storn 1996, Storn and Price 1997] proposed several other strategies that can be adopted depending on the type of problem. In order to classify different strategies of DE the notation $DE/x/y/z$ is introduced, where x specifies the vector to be mutated which can be *rand* (a randomly chosen population vector), *best* (the vector of lowest fitness from the current population) or *rand-to-best* (combination of *rand* with *best*), y is the number of difference vectors used and z denotes the crossover scheme, which can be *bin* (binomial) or *exp* (exponential). Using this notation, the basic DE strategy described in the previous section can be written as DE/rand/1/bin, as others strategies were later proposed by Storn and Price, but described in the work of Babu and Jehan [Babu and Jehan 2003]. These strategies and corresponding changes to the mutation scheme are described in Table 3.1.

Table 3.1: Strategies of the standard Differential Evolution.

Strategy	Mutation Scheme
DE/best/1/exp	$\mathbf{v}_n^{i+1} = \mathbf{x}_{\text{best}}^i + F\left(\mathbf{x}_{r_2}^i - \mathbf{x}_{r_3}^i\right)$
DE/rand/1/exp	$\mathbf{v}_n^{i+1} = \mathbf{x}_{r_1}^i + F\left(\mathbf{x}_{r_2}^i - \mathbf{x}_{r_3}^i\right)$
DE/rand-to-best/1/exp	$\mathbf{v}_n^{i+1} = \mathbf{x}_n^i + \lambda\left(\mathbf{x}_{\text{best}}^i - \mathbf{x}_{r_1}^i\right) + F\left(\mathbf{x}_{r_1}^i - \mathbf{x}_{r_2}^i\right)$
DE/best/2/exp	$\mathbf{v}_n^{i+1} = \mathbf{x}_{\text{best}}^i + F\left(\mathbf{x}_{r_1}^i - \mathbf{x}_{r_2}^i + \mathbf{x}_{r_3}^i - \mathbf{x}_{r_4}^i\right)$
DE/rand/2/exp	$\mathbf{v}_n^{i+1} = \mathbf{x}_{r_1}^i + F\left(\mathbf{x}_{r_2}^i - \mathbf{x}_{r_3}^i + \mathbf{x}_{r_4}^i - \mathbf{x}_{r_5}^i\right)$
DE/best/1/bin	$\mathbf{v}_n^{i+1} = \mathbf{x}_{\text{best}}^i + F\left(\mathbf{x}_{r_2}^i - \mathbf{x}_{r_3}^i\right)$
DE/rand/1/bin	$\mathbf{v}_n^{i+1} = \mathbf{x}_{r_1}^i + F\left(\mathbf{x}_{r_2}^i - \mathbf{x}_{r_3}^i\right)$
DE/rand-to-best/1/bin	$\mathbf{v}_n^{i+1} = \mathbf{x}_n^i + \lambda\left(\mathbf{x}_{\text{best}}^i - \mathbf{x}_{r_1}^i\right) + F\left(\mathbf{x}_{r_1}^i - \mathbf{x}_{r_2}^i\right)$
DE/best/2/bin	$\mathbf{v}_n^{i+1} = \mathbf{x}_{\text{best}}^i + F\left(\mathbf{x}_{r_1}^i - \mathbf{x}_{r_2}^i + \mathbf{x}_{r_3}^i - \mathbf{x}_{r_4}^i\right)$
DE/rand/2/bin	$\mathbf{v}_n^{i+1} = \mathbf{x}_{r_1}^i + F\left(\mathbf{x}_{r_2}^i - \mathbf{x}_{r_3}^i + \mathbf{x}_{r_4}^i - \mathbf{x}_{r_5}^i\right)$

Other than these different strategies of DE, authors have proposed variants of DE to improve the design space exploration ability, convergence rate, among others.

Zhenyu *et al.* [Zhenyu *et al.* 2006] proposed a variant of DE named Self-Adaptive Chaos Differential Evolution (SACDE). This variant adopts a chaos mutation factor, a dynamically changing weighting factor, and introduces an evolution speed factor and an aggregation degree factor of the population to control the parameters F and C_{r}. The chaos mutation factor prevents the algorithm from falling into the local optima as experiments show that the convergence speed of SACDE is significantly superior to DE, while convergence accuracy is also increased.

Epitropakis *et al.* [Epitropakis *et al.* 2011] proposed a proximity induced mutation scheme for DE, named Proximity-Based Differential Evolution (ProDE). In ProDE, neighbors of a parent vector, rather than the randomly chosen vectors, are used to generate the trial vector. To avoid sacrificing exploration capability, a probabilistic approach was

suggested and proved to improve mutation schemes such as DE/rand/1, but fails to show significant improvement when implemented over highly multi-modal or hybrid functions.

Gong and Cai [Gong and Cai 2013] suggested that the mutation scheme can be more useful if two of the vectors are selected based on their fitness, while the third is selected randomly. Their proposed variant, called Ranking-Based Differential Evolution (rank-DE) proposes that, instead of randomized or proximity based approaches, the probability for a vector to be selected in the mutation is calculated from their objective function value rank in the population. The proposed strategy proved to be thus faster and less computationally expensive.

Yang *et al.* [Yang *et al.* 2015] designed an automatic population enhancement scheme that checks each dimension to identify a convergence and diversifies that dimension to a satisfactory level. The proposed mechanism named Auto-Enhanced Population Diversity (AEPD) aids DE to escape from local optima and stagnation. To quantify population diversity, the mean and the standard deviation of individuals' positions are calculated. If the standard deviation is found to be below a threshold, the dimension is called converged.

The mentioned studies and variants of DE are introduced to benefit the algorithm, increasing its robustness and convergence rate. Many more modifications and variants of DE are found in the literature, as it is one of the most famous and more used algorithms.

3.3 Teaching-Learning-Based Optimization

Rao *et al.* [Rao *et al.* 2011, Rao *et al.* 2012] proposed a new optimization algorithm named Teaching-Learning-Based Optimization (TLBO). The proposed algorithm presents the characteristic of being parameter-less (does not require parameters), in opposition to the two previously described algorithms, PSO and DE. According to the authors, TLBO is proposed to obtain global solutions for continuous non-linear functions with less computational effort and high consistency. The algorithm is based on the philosophy of learning and teaching, resembling the influence of a teacher on the output of learners in a class. A teacher is often considered as a highly learned person who shares their knowledge with the learners, whereas the results of the learners are affected by the quality of the teacher. To complement the knowledge provided by the teacher, learners also interact between themselves improving their results.

Similar to other nature-inspired algorithms, TLBO is a population-based algorithm, where a group of learners is considered. Design variables are analogous to the different subjects of the learners and the teacher is considered to be the best solution found so far.

3.3.1 Algorithm Formulation

Initially a population of n learners is randomly generated inside the design space, with each learner having dimension D equal to the number of design variables. At any iteration i, a learner $\mathbf{x}_n^i$ represents the position in the design space and is limited by its upper and lower bounds, respectively, $\mathbf{x}_{\max}$ and $\mathbf{x}_{\min}$. The generation of initial learners is then given by

$$\mathbf{x}_n^0 = \mathbf{x}_{\min} + \mathbf{r}\left(\mathbf{x}_{\max} - \mathbf{x}_{\min}\right), \tag{3.10}$$

where $\mathbf{r}$ is a vector of random numbers uniformly distributed in the range [0, 1]. The

optimization process is then initialized with TLBO being divided into two main phases: teacher and learner phase.

During the first phase, the goal of the teacher is to increase the mean result $\mathbf{m}^i$ of the population in each design variable (subject), where the teacher $\mathbf{t}^i$ is considered the best learner at a given moment. New solutions $\mathbf{u}_n^{i+1}$ are generated based on the difference between the existing mean result $\mathbf{m}^i$ and the current teacher $\mathbf{t}^i$. This difference is represented as $\mathbf{d}_n^i$ and given by

$$\mathbf{d}_n^i = \mathbf{r}\left(\mathbf{t}^i - T_\mathrm{F}\mathbf{m}^i\right), \tag{3.11}$$

where $\mathbf{r}$ is a vector of random numbers uniformly distributed in the range [0, 1]. T_F is a teaching factor that can either be 1 or 2 and is decided randomly with equal probability. The new solution $\mathbf{u}_n^{i+1}$ is then generated according to

$$\mathbf{u}_n^i = \mathbf{x}_n^i - \mathbf{d}_n^i. \tag{3.12}$$

Afterwards, $\mathbf{u}_n^i$ is evaluated and its objective function value compared to the current learner $\mathbf{x}_n^i$ value. If its value is better, it replaces the respective learner in the population, otherwise, it is discarded.

In the learner phase, learners increase their knowledge by interacting with each other. For every learner in the population, a learner $\mathbf{x}_p^i$ is randomly selected among the population, conditioned to be different from the current learner $\mathbf{x}_n^i$. A new solution $\mathbf{v}_n^i$ is then generated from the shared knowledge between the two learners, according to

$$\mathbf{v}_n^i = \begin{cases} \mathbf{x}_n^i + \mathbf{r}\left(\mathbf{x}_n^i - \mathbf{x}_p^i\right) & \text{if } f\left(\mathbf{x}_n^i\right) < f\left(\mathbf{x}_p^i\right) \\ \mathbf{x}_n^i + \mathbf{r}\left(\mathbf{x}_p^i - \mathbf{x}_n^i\right) & \text{if } f\left(\mathbf{x}_p^i\right) < f\left(\mathbf{x}_n^i\right) \end{cases}, \tag{3.13}$$

where $\mathbf{r}$ is a vector of random numbers uniformly distributed in the range [0, 1]. If the new solution $\mathbf{v}_n^i$ is better than the current learner, the last is replaced by the new solution, otherwise, the new solution is discarded. After the learner phase, the updated learners are maintained and become the input of the teacher phase in the next iteration. The steps of TLBO are described in the pseudo-code presented in Figure 3.3.

3.4 Applications in Mechanical Design

Perez and Behdinan [Perez and Behdinan 2007] applied PSO to structural design optimization, using three benchmark problems of 10, 25 and 72 bar truss. The results suggested PSO faired significantly well against different optimization approaches including gradient-based algorithms, convex programming and genetic algorithms. Degertekin and Hayalioglu [Degertekin and Hayalioglu 2013] and Ho-Huu *et al.* [Huu *et al.* 2016] considered the application of TLBO and DE, respectively, to structural design optimization, including the benchmark 10, 25 and 72 bar truss design problems.

Zhou *et al.* [Zhou *et al.* 2006] studied the application of PSO to machining tolerances of a cylinder-piston assembly. PSO demonstrated high efficiency and effectiveness in the studied example.

Rao *et al.* [Rao *et al.* 2011] tested the implementation of TLBO to several design problems, including a four step-cone pulley with four design variables to minimize its weight and a rolling element bearing to maximize the dynamic load carrying capacity. Their study

Generate initial population with n vectors of random position $\mathbf{x}_n^0$
Evaluate the objective function $f\left(\mathbf{x}_n^0\right)$
Find the teacher $\mathbf{t}^0$ for n particles
Set iteration counter $i = 0$
while *stopping criterion* **do**
    `// Teacher Phase`
    $\mathbf{m}^i = \left(\sum_{k=1}^{n} \mathbf{x}_k^i\right)/n$
    for *loop over n vectors* **do**
        $T_{\mathrm{F}} = \mathrm{round}\left[1 + \mathrm{rand}(0,1)(2-1)\right]$
        $\mathbf{r} = \mathrm{rand}(0,1)$
        $\mathbf{d}_n^i = \mathbf{r}\left(\mathbf{t}^i - T_{\mathrm{F}}\mathbf{m}^i\right)$
        $\mathbf{u}_n^{i+1} = \mathbf{x}_n^i - \mathbf{d}_n^i$
        Check lower and upper bounds of $\mathbf{u}_n^i$
        Evaluate the objective function $f\left(\mathbf{u}_n^i\right)$
        if $f\left(\mathbf{u}_n^i\right) < f\left(\mathbf{x}_n^i\right)$ **then**
            $\mathbf{x}_n^i = \mathbf{u}_n^i$
        end
    end
    `// Learner Phase`
    for *loop over n vectors* **do**
        Randomly select $\mathbf{x}_p^i \neq \mathbf{x}_n^i$
        $\mathbf{r} = \mathrm{rand}(0,1)$
        if $f\left(\mathbf{x}_n^i\right) < f\left(\mathbf{x}_p^i\right)$ **then**
            $\mathbf{v}_n^i = \mathbf{x}_n^i + \mathbf{r}\left(\mathbf{x}_n^i - \mathbf{x}_p^i\right)$
        else if $f\left(\mathbf{x}_p^i\right) < f\left(\mathbf{x}_n^i\right)$ **then**
            $\mathbf{v}_n^i = \mathbf{x}_n^i + \mathbf{r}\left(\mathbf{x}_p^i - \mathbf{x}_n^i\right)$
        end
        Check lower and upper bounds of $\mathbf{v}_n^i$
        Evaluate the objective function $f\left(\mathbf{v}_n^i\right)$
        if $f\left(\mathbf{v}_n^i\right) < f\left(\mathbf{x}_n^i\right)$ **then**
            $\mathbf{x}_n^{i+1} = \mathbf{v}_n^i$
        end
    end
    Update iteration counter $i = i + 1$
end
Post-process and output final results

Figure 3.3: Pseudo-code for the implementation of Teaching-Learning-Based Optimization.

demonstrated that TLBO is more effective and efficient than other optimization methods in the mechanical design problems considered.

Rao and Savsani [Rao and Savsani 2012] studied several mechanical design problems using different optimization algorithms, including PSO, DE and TLBO. The problems studied include a gear train design problem, a multi-objective optimization of a radial ball bearing, design optimization of a Belleville's spring to minimize its weight, a multiple disc clutch brake, optimization of a robot gripper geometric dimensions to minimize the difference between maximum and minimum force applied by the gripper for the range of gripper end displacements and a hydrodynamic thrust bearing, among many others.

Saruhan [Saruhan 2014] employed DE to minimize a ball bearing pivot link system weight. Their study concluded that DE proved to be robust and demonstrated the capability to produce an efficient solution to the problem.

Guedria [Guedria 2015] and Kiran [Kiran 2017] applied variants of PSO to a tension/compressing spring design problem, a cylindrical pressure vessel capped at both ends by hemispherical heads to minimize the total manufacturing cost, a welded beam and a speed reducer design problem to minimize its weight.

Chapter 4

Implementation and Applications

The implementation of parallel processing techniques in advanced optimization methods is described. Programming languages and its features are presented. A description of the implemented applications is presented, including the mathematical formulation and computational implementation.

4.1 Configuration of Advanced Optimization Methods

In Chapter 3, a general description and variants of the implemented advanced optimization methods are described. However, the used configuration and parameters are yet to be defined.

As for the PSO (cf. Section 3.1), a standard configuration of the algorithm is implemented with acceleration parameters c_1 and c_2 equal to 2. Additionally, the algorithm is implemented using a linear decreasing inertia weight in the range of [0.4, 0.9]. DE (cf. Section 3.2) is implemented using a standard configuration, with the mutation strategy of DE/rand/1/bin. Operational parameters F and crossover probability C_r are, respectively, equal to 0.5 and 0.7. Finally, TLBO (cf. Section 3.3) is implemented similarly to what is described.

The three algorithms are implemented using a stopping criterion based on the number of function evaluations, rather than the number of iterations/generations. The advantage of the selected stopping criteria is that it enables a fair comparison of the algorithms' efficiency, as this way each one evaluates the objective function exactly the same number of times [Črepinšek *et al.* 2012]. Using the number of function evaluations as a stopping criterion is specifically relevant because, for an equal number of iterations, TLBO evaluates the objective twice than PSO or DE. The selected number of function evaluations was empirically selected and differs from application to application, as they are defined ahead.

4.2 Parallel Processing in Advanced Optimization Methods

When considering a parallel implementation in advanced optimization methods, some issues need to be considered as the goal is to improve the computational performance without

compromising the algorithm's result. The algorithm should operate in such a way that it can be easily decomposed for a parallel implementation. Additionally, it is highly recommended that it presents scalable characteristics [Schutte *et al.* 2004]. Scalability implies that the nature of the algorithm does not place a limit on the number of computational processes to be used, allowing for a full use of the computational resources available. The advanced optimization methods here considered (cf. Chapter 3) are population-based algorithms, thereby well suited for parallel computing as the size of the population can be increased or decreased to maximize the resources available.

A parallel implementation of the algorithm should not have an effect on its operations. The flow of operations should be similar to the sequential implementation, in order to lead to similar results. Furthermore, the selection of the type of parallelism is influenced by the algorithm operations. If the operations of the algorithm are sequence dependent, that is, if the order they appear is mandatory, a parallel processing implementation using task parallelism is not recommended. This characteristic is observed in the implemented algorithms, as in the implementation of PSO, the particle's position can only be updated after the velocity's updating and in DE the crossover operation can only be computed after mutation. Moreover, in TLBO, the learner's phase can only take place after the teacher phase. For these reasons and to guarantee the coherence of the algorithm, data parallelism stands as a more straightforward implementation.

In the case of data parallelism, as the algorithms are population-based (present different solutions at the same time) and different operations are performed over the entire population, a parallel implementation is possible either at the evaluation of the objective function or at specific operations of the algorithm. In PSO, this approach is easily implemented as all the solutions are independent and can be evaluated sequentially. However, in the implementations of DE and TLBO, solutions interact with each other. This dependency observed in DE and TLBO does not allow for a straightforward implementation of parallel processing without compromising the algorithms' order of operations, as each evaluation is either preceded or succeeded by a series of operations that affect individual solutions. As a consequence of this characteristic in the implementations of DE and TLBO, PSO is the only algorithm in which a parallel processing approach is implemented in this work.

PSO operations can be divided into three main steps: the evaluation of the objective function, the update of velocity and the update of position. For the majority of optimization problems, it is expected that the bulk of the computational effort is at the evaluation of the objective function, rather than at the update of velocity or position. For this reason, in the parallel processing implementation of PSO, the algorithm performs the evaluation of the objective function in parallel and the other operations sequentially, similar to the sequential processing implementation. The differences in both PSO implementations, sequential and parallel, are described in Figure 4.1, where dotted lines indicate the parts where they differ.

4.3 Programming Languages and Paradigm

The algorithms are implemented using different programming languages. The programming languages are selected due to its popularity, potential and use in academic environments, as well as using languages with different specifications. Taking these criteria into consideration, four programming languages are considered: Python, MATLAB, Java and

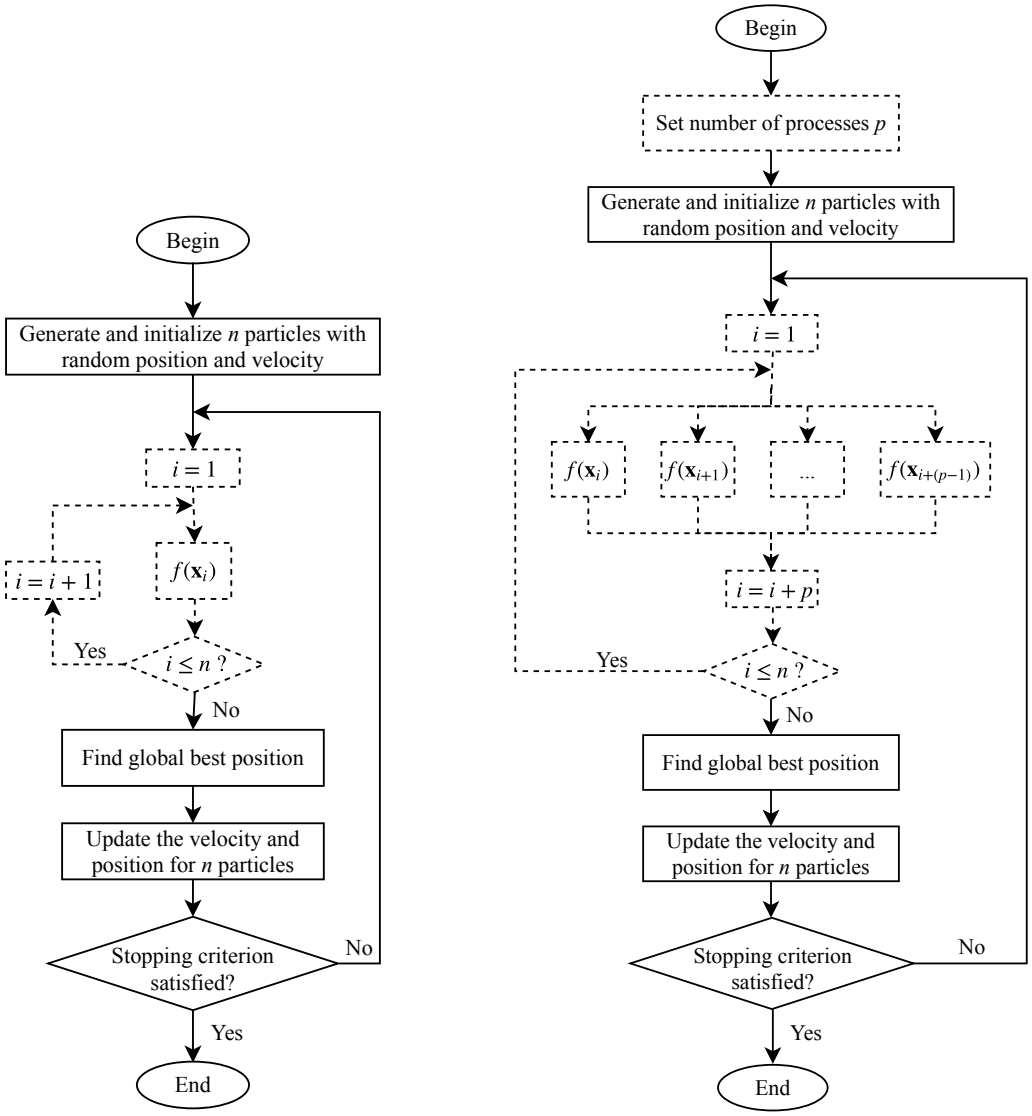


Figure 4.1: Schematic representation of the implementations of Particle Swarm Optimization (a) sequential and (b) parallel algorithms.

C++.

Python is a multi-paradigm, object-oriented, open-source, designed to be an easy-to-read and easy-to-use programming language. Python comes with several features, such as an elegant syntax, making the programs easier to read and write, a large standard library, a variety of basic data types, supports object-oriented programming with classes and multiple inheritance, variables' type are dynamically typed[1] and is an interpreted programming language, among several other features. Python provides built-in modules to implement parallel programming, such as threading module and multiprocessing module. Both modules provide tools to implement parallel programs. However, the multiprocessing module is of more interest, as it implements parallelism based on processes, rather than threads. The multiprocessing offers a vast number of methods that are easily implemented and suitable for different applications.

MATLAB is a multi-paradigm numerical computing environment and a proprietary programming language developed by MathWorks®. It was designed for engineers and scientists, compiling a vast number of toolboxes for different applications. As a programming language, it is enhanced for matrix-based algorithms but supports other features, such as object-oriented programming. Similarly to Python, MATLAB is interpreted and statically typed. Although MATLAB is not free, it is vastly used for educational purposes, either for its capabilities or the user-friendly environment. The implementation of parallel programs is achieved using the Parallel Computing Toolbox™, available in MATLAB, which enables the use of multi-core processors. The toolbox offers parallel for-loops, special array types and parallelized numerical algorithms that are easily implemented in the sequential implementation.

Java is a general-purpose, object-oriented programming language, and is one of the most used and popular programming languages in the world [TIOBE 2018]. It is recognized as being a fast, reliable and secure programming language, as well as having a large standard library available for any programmer and application. Java, in opposition to Python and MATLAB, is compiled and statically typed[2]. Java offers features to easily implement parallel programs using threads. However, the implementation using processes is not straightforward. For this reason, it is used a multiprocessing library available in a GitHub[3] repository [Csomor 2017] for the implementations of parallel processing in Java.

C++ is an enhanced/extended version of the C programming language and one of the most popular nowadays [TIOBE 2018]. It is characterized for being compiled, statically-typed, multi-paradigm, sophisticated, efficient and for having a large standard library. The development of programs in C++ might be more time expensive than, for example, Python or MATLAB. Nevertheless, it is expected to run faster [Nanz and Furia 2015]. C++ offers standard support for multi-threading, but not for parallel processing using processes [Williams 2012]. A search was conducted to find an available online library that would facilitate the implementation of parallel processing using processes. However, this search did not lead to a solution that would be easily implemented. Therefore, it was decided not to implement C++ programs in parallel processing, having only been implemented in sequential processing.

The algorithms and applications are implemented in the following versions of the pro-

[1]Dynamically typed programming languages do not require the declaration of the type of variables.

[2]Statically typed programming languages require the declaration of the type of variables.

[3]GitHub (`https://github.com`) is a web-based version-control and collaboration platform for software developers.

gramming languages: Python 3.7.0, MATLAB R2015a, Java 10.0.2 with javac compiler and C++ compiled with Intel® C++ Compiler. Furthermore, the implementation of the algorithms in the four programming languages follows an objected-oriented programming paradigm. Moreover, the structure of the algorithms in each programming language is identical. However, differences exist when features for a boost in performance of each language are available.

4.4 Applications

To evaluate the algorithms' and programming languages' performance, four applications are implemented. One application is a composition function benchmark used to test the algorithms' efficiency and robustness. Other three applications are mechanical design problems, consisting of a problem solely evaluated using numerical equations and two problems that require the use of external programs.

4.4.1 Composition Function

In order to assess the robustness and efficiency of optimization methods, tests are frequently carried out using standard benchmark functions from the literature. However, some algorithms take advantage of known properties of the benchmark functions, such as local optima lying along the coordinate axes or global optimum having the same values for different variables. To tackle this advantage, Liang et al. [Liang *et al.* 2005] identified shortcomings associated with the existing benchmark functions and proposed hybrid benchmark functions whose complexity and properties can be easily controlled. According to their experience, some properties encountered in standard benchmark functions are:

1. Global optimum with the same parameter values for different dimensions;
2. Global optimum at the origin;
3. Global optimum lying in the center of the search range;
4. Global optimum on the bounds;
5. Local optima lying along the coordinate axes or no linkage among dimensions.

4.4.1.1 Mathematical Formulation

Based on the previous considerations, Liang *et al.* [Liang *et al.* 2005] proposed a structure to construct challenging composition test functions. These composition functions are built using standard benchmark functions with a randomly located global optimum and considerable randomly located local optima. The composition functions $F(\mathbf{x})$ are obtained according to

$$F(\mathbf{x}) = \sum_{i=1}^{n} w_i \left(f_i(\mathbf{x}) + bias_i \right), \tag{4.1}$$

where n denotes the number of basic functions $f_i(\mathbf{x})$, with $i = 1, 2, ..., n$, w_i represents the weight of each n function and $bias_i$ defines which optima is global optimum. The smallest

value of **bias** corresponds to the global optimum and the bigger n is, the more complex $F(\mathbf{x})$ is. The weight w_i of each n function is given by

$$w_i = \exp\left(-\frac{\sum_{k=1}^{D}(x_k - o_{i,k})^2}{2D\sigma_i^2}\right), \tag{4.2}$$

where D denotes the dimension, $\mathbf{o}_i$ is a vector that defines the global and local optima positions for each n function. σ_i is a parameter used to control the coverage range, where a small σ_i gives a narrow range for each n function. Subsequently the maximum value of the weight $\max(w_i)$ is determined and w_i rearranged according to

$$w_i = \begin{cases} w_i & \text{if } w_i \neq \max(w_i) \\ w_i\left(1 - \max(w_i)^{10}\right) & \text{otherwise} \end{cases}. \tag{4.3}$$

The weight w_i of each function is finally normalized following

$$w_i = \frac{w_i}{\sum_{i=1}^{n} w_i}. \tag{4.4}$$

In the situation of $f_i(\mathbf{x})$ representing different functions and in order to obtain a better mixture, the biggest function value $f_i^{\max}$ is estimated for each function and then $f_i(\mathbf{x})$ is normalized to similar height as

$$f_i(\mathbf{x}) = \frac{C\, fit_i(\mathbf{x})}{|f_i^{\max}|}, \tag{4.5}$$

where C is a pre-defined constant, $f_i^{\max}$ and $fit_i(\mathbf{x})$ are given by

$$f_i^{\max} = f_i\left(\frac{z}{\boldsymbol{\lambda}} \cdot \mathbf{M}_i\right) \quad \text{and} \tag{4.6}$$

$$fit_i(\mathbf{x}) = f_i\left(\frac{\mathbf{x} - \mathbf{o}_i}{\boldsymbol{\lambda}} \cdot \mathbf{M}_i\right), \tag{4.7}$$

where z corresponds to the upper boundary of the composition function, $\boldsymbol{\lambda}$ is used to stretch or compress the function, to which $\lambda_i > 1$ means stretch, $\lambda_i < 1$ means compress. $\mathbf{M}_i$ is an orthogonal rotation matrix of each function, with size $D \times D$. In Figure 4.2, the pseudo-code for the construction of the composition functions is presented.

4.4.1.2 Construction of Composition Function

From the procedure previously described it is possible to construct different composition functions by modifying the parameters and using different basic functions. Liang *et al.* [Liang *et al.* 2005] defined and constructed six different composition functions. The parameters used in the construction of the composition functions are as follows:

- Number of basic functions, $n = 10$;
- Dimension, $D = 10$;
- $C = 2000$;
- Design space, $[-5, 5]^D$;

Define f_i, $\boldsymbol{\sigma}$, $\boldsymbol{\lambda}$, $\boldsymbol{bias}$, $\mathbf{o}_i$, $\mathbf{M}_i$, n and constants C and z
for $i = 1, 2, ..., n$ **do**
  $w_i = \exp\left(-\left[\sum_{k=1}^{D}(x_k - o_{i,k})^2\right] / \left(2D\sigma_i^2\right)\right)$
  $f_i^{\max} = f_i\left((z/\boldsymbol{\lambda}) \cdot \mathbf{M}_i\right)$
  $fit_i(\mathbf{x}) = f_i\left(\left[\left(\mathbf{x} - \mathbf{o}_i\right)/\boldsymbol{\lambda}\right] \cdot \mathbf{M}_i\right)$
  $f_i(\mathbf{x}) = \left(C\, fit_i(\mathbf{x})\right)/|f_i^{\max}|$
end
$SumW = \sum_{i=1}^{n} w_i$
$MaxW = \max(w_i)$
for $i = 1, 2, ..., n$ **do**
  if $w_i = MaxW$ **then**
    $w_i = w_i$
  else
    $w_i = w_i\left(1 - MaxW^{10}\right)$
  end
  $w_i = w_i / SumW$
end
$F(\mathbf{x}) = \sum_{i=1}^{n} w_i\left(f_i(\mathbf{x}) + bias_i\right)$

Figure 4.2: Pseudo-code for the construction of composition functions.

- **bias** $= [0, 100, 200, 300, 400, 500, 600, 700, 800, 900]$.

From the values set for **bias**, the first basic function is the one with the global optimum, as its *bias* is always 0. $\mathbf{o}_1$, $\mathbf{o}_2$,...,$\mathbf{o}_9$ are generated randomly in the design space, except for $\mathbf{o}_{10}$ which is set as $[0, 0, ..., 0]$ in order to trap algorithms that have a potential to converge at the center of the design space. $\mathbf{M_1}$, $\mathbf{M_2}$, ..., $\mathbf{M_{10}}$ are $D \times D$ orthogonal rotation matrices obtained using Salomon's method [Salomon 1996].

The basic functions composed to construct the composition functions are five, although not all are necessarily used at the same time, as one basic function can be repeated as many times as necessary. These functions are described as follows:

- Sphere Function:

$$f(\mathbf{x}) = \sum_{i=1}^{D} x_i^2 \,; \tag{4.8}$$

- Rastrigin's Function:

$$f(\mathbf{x}) = \sum_{i=1}^{D}\left\{\sum_{k=0}^{k_{\max}}\left[a^k \cos\left(2\pi b^k(x_i + 0.5)\right)\right]\right\} - D\sum_{k=0}^{k_{\max}}\left(a^k \cos(\pi b^k)\right), \tag{4.9}$$

$$a = 0.5, b = 3, k_{\max} = 20 \,;$$

- Weierstrass's Function:

$$f(\mathbf{x}) = \sum_{i=1}^{D} \frac{x_i^2}{4000} - \prod_{i=1}^{D} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1\,; \tag{4.10}$$

- Griewank's Function:

$$f(\mathbf{x}) = \sum_{i=1}^{D} \frac{x_i^2}{4000} - \prod_{i=1}^{D} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1\,; \tag{4.11}$$

- Ackley's Function:

$$f(\mathbf{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} x_i^2}\right) - \exp\left(\frac{1}{D}\sum_{i=1}^{D}\cos(2\pi x_i)\right) + 20 + e\,. \tag{4.12}$$

Although six composition functions were introduced, in this work only one is implemented, corresponding to composition function 5 in the works of Liang *et al.* [Liang *et al.* 2005] and Caseiro [Caseiro 2009]. This composition function is composed of all five basic functions, each repeated once according to Table 4.1. The parameters $\boldsymbol{\sigma}$ and $\boldsymbol{\lambda}$ are set as:

- $\boldsymbol{\sigma} = [\sigma_1, \sigma_2, ..., \sigma_{10}] = [1, 1, ..., 1]$;
- $\boldsymbol{\lambda} = [\lambda_1, \lambda_2, ..., \lambda_{10}] = \left[\frac{1}{5}, \frac{1}{5}, 10, 10, \frac{5}{100}, \frac{5}{100}, \frac{5}{32}, \frac{5}{32}, \frac{5}{100}, \frac{5}{100}\right]$,

where λ_1 and λ_2 are set $1/5$ in order to achieve a more complex landscape for the global optimum's area. In Figure 4.3, a three-dimensional (corresponding to $D = 2$) representation of the composition function is illustrated.

Table 4.1: Basic functions composing the implemented composition function.

$f_i(\mathbf{x})$	Basic Function
$f_{1-2}(\mathbf{x})$	Rastrigin's
$f_{3-4}(\mathbf{x})$	Weierstrass
$f_{5-6}(\mathbf{x})$	Griewank's
$f_{7-8}(\mathbf{x})$	Ackley's
$f_{9-10}(\mathbf{x})$	Sphere

To guarantee that the algorithms are tested for the same composition function, that is with the same parameters and global optimum, the orthogonal rotation matrix $\mathbf{M}_i$ and the vectors containing the global and local optima $\mathbf{o}_i$ were previously generated and maintained through all tests. The generated values correspond to a global optimum of $F(\mathbf{x}) = 0.0000$ located at $\mathbf{x}$ = [1.5953, 2.6440, 1.8047, 0.9389, -3.0486, -1.1571, 3.5582, 2.4246, -0.3767, 4.4637]. Furthermore, the number of function evaluations used as a stopping criterion is 10^5.

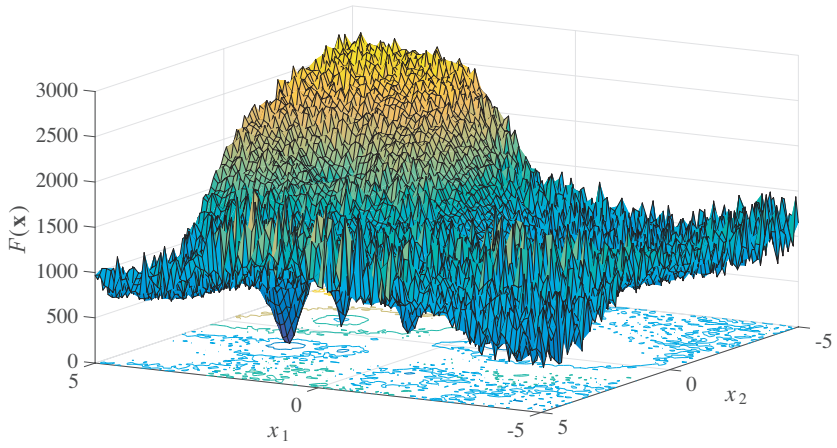


Figure 4.3: Three-dimensional ($D = 2$) representation of the composition function.

4.4.2 Speed Reducer

The weight of a speed reducer [Golinski 1970] is to be minimized subject to constraints on bending stress of the gear teeth, surface stress, transverse deflections of the shafts and stresses in the shafts. This design problem involves seven design variables as shown in Figure 4.4, which are the face width x_1, module of teeth x_2, number of teeth on the pinion x_3, length of the first shaft between bearings x_4, length of the second shaft between bearings x_5, diameter of the first shaft x_6 and diameter of the second shaft x_7. The third variable x_3 is an integer, while the rest are continuous. With eleven constraints, this is a constrained optimization problem that could be transformed into an unconstrained problem using a penalty function.

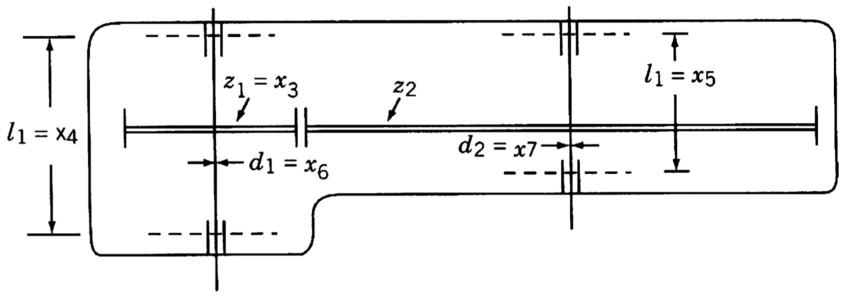


Figure 4.4: Illustrative representation of the speed reducer design geometry [Rao and Savsani 2012].

The problem is formulated as a constrained nonlinear mathematical programming to minimize the objective function $f(\mathbf{x})$, subject to the inequality constraints $g_j(\mathbf{x})$, with $j = 1, 2, ..., 11$, stated as

$$\begin{aligned}\text{minimize} \quad & f(\mathbf{x}) = 0.7854x_1x_2^2\left(3.3333x_3^2 + 14.9334x_3 - 43.0934\right) - 1.508x_1\left(x_6^2 + x_7^2\right) \\ & + 7.4777\left(x_6^3 + x_7^3\right) + 0.7854\left(x_4x_6^2 + x_5x_7^2\right), \end{aligned} \tag{4.13}$$

$$\text{subject to} \quad g_1(\mathbf{x}) = \frac{27}{x_1x_2^2x_3} - 1 \leq 0\,, \tag{4.14}$$

$$g_2(\mathbf{x}) = \frac{397.5}{x_1x_2^2x_3^2} - 1 \leq 0\,, \tag{4.15}$$

$$g_3(\mathbf{x}) = \frac{1.93x_4^3}{x_2x_3x_6^4} - 1 \leq 0\,, \tag{4.16}$$

$$g_4(\mathbf{x}) = \frac{1.93x_5^3}{x_2x_3x_7^4} - 1 \leq 0\,, \tag{4.17}$$

$$g_5(\mathbf{x}) = \frac{\sqrt{\left(\frac{745x_4}{x_2x_3}\right)^2 + 16.9e6}}{110x_6^3} - 1 \leq 0\,, \tag{4.18}$$

$$g_6(\mathbf{x}) = \frac{\sqrt{\left(\frac{745x_5}{x_2x_3}\right)^2 + 157.5e6}}{85x_7^3} - 1 \leq 0\,, \tag{4.19}$$

$$g_7(\mathbf{x}) = \frac{x_2x_3}{40} - 1 \leq 0\,, \tag{4.20}$$

$$g_8(\mathbf{x}) = \frac{5x_2}{x_1} - 1 \leq 0\,, \tag{4.21}$$

$$g_9(\mathbf{x}) = \frac{x_1}{12x_2} - 1 \leq 0\,, \tag{4.22}$$

$$g_{10}(\mathbf{x}) = \frac{1.5x_6 + 1.9}{x_4} - 1 \leq 0\,, \tag{4.23}$$

$$g_{11}(\mathbf{x}) = \frac{1.1x_7 + 1.9}{x_5} - 1 \leq 0\,, \tag{4.24}$$

$$2.6 \leq x_1 \leq 3.6,\ 0.7 \leq x_2 \leq 0.8,\ 17 \leq x_3 \leq 28,\ 7.3 \leq x_4 \leq 8.3,$$
$$7.8 \leq x_5 \leq 8.3,\ 2.9 \leq x_6 \leq 3.9,\ 5.0 \leq x_6 \leq 5.5\,.$$

However, as previously stated, the problem does not take into account unfeasible solutions (which are not desired). In order to handle the inequality constraints presented in the design problem, a dynamic penalty function (cf. Section 2.1.2) is used with the parameters $C = 60$, $\alpha = 2$ and $\beta = 1$.

As the problem is solved by many authors [Baykasoglu 2012, Guedria 2015, Rao and Waghmare 2017] using different algorithms, the best reported result is $f(\mathbf{x}) = 2996.348165$ located at $\mathbf{x} = [3.499999, 0.7, 17, 7.3, 7.8, 3.350215, 5.286683]$. The reported result serves as a reference to the global optimum in the analysis of results. The selected number of function evaluations as a stopping criterion is 10^5.

4.4.3 Three-Bar Truss

The problem consists of a three-bar truss, represented in Figure 4.5, subject to three different loading conditions, where the goal is to minimize the structure volume without

compromising its structural integrity or without presenting inadmissible strain energy. Each bar is characterized by having equal length and different cross sections, with volume x_i, and the problem can be mathematically defined as

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{x}) = x_1 + x_2 + x_3 \,, & & (4.25) \\ \text{subject to} \quad & g_j(\mathbf{x}) = \mathbf{p}_j^{\mathrm{T}} \mathbf{u}_j(\mathbf{x}) - c_{\max} \leq 0 \,, \qquad j = 1,2,3 \,, & (4.26) \\ & 0.01 \leq x_i \leq 2 \,, \qquad i = 1,2,3 \,, & (4.27) \end{aligned}$$

where $\mathbf{p}_j$ and $\mathbf{u}_j$ represent, respectively, the load vector and the displacement vector of node 4 for each load $j = 1, 2, 3$. The constraints $g_j(\mathbf{x})$ define the strain energy of the truss for each load $\mathbf{p}_j$ applied, where $c_{\max} = 1$ is the critical value of admissible strain energy.

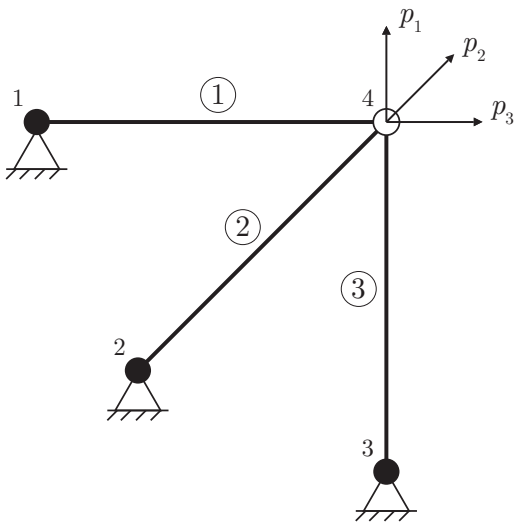


Figure 4.5: Illustrative representation of the three-bar truss design problem.

The displacements $\mathbf{u}_j$ are calculated as $\mathbf{k}(\mathbf{x})\mathbf{u}_j = \mathbf{p}_j$, defined by the finite element method, where $\mathbf{k}(\mathbf{x})$ is the stiffness matrix. The coordinates of the nodes 1, 2, 3, 4 are, respectively, $(-1, 0)$, $\left(-1/\sqrt{2}, -1/\sqrt{2}\right)$, $(0, -1)$ and $(0, 0)$. The three loads are defined by the vectors, $p_1 = (1, 0)$, $p_2 = (1, 1)$ and $p_3 = (0, 1)$.

To avoid unfeasible solutions, a dynamic penalty function (cf. Section 2.1.2) is used with the parameters $C = 2$, $\alpha = 1$ and $\beta = 1$. The global optimum, corresponds to $f(\mathbf{x}) = 2.666667$, located at $\mathbf{x} = [0.666667, 1.333333, 0.666667]$. Additionally, the number of function evaluations used as a stopping criterion is 10^4.

4.4.3.1 Computational Implementation

To compute the displacement of node 4, the program FRAN[4] is used. FRAN analyses structures composed of rod elements with different properties. The program receives as input the node coordinates, element connectivity, loading conditions and prescribed displacements. FRAN assembles the stiffness matrix and computes the equation system, returning as output the axial forces in each element and the displacements of each node.

[4]FRAN is an academic program to analyze articulated and reticulated structures.

The program also exports a file recognized by GiD[5] software, allowing the visualization and post-processing of results.

To establish a connection between the optimization algorithm and FRAN, an interface program is implemented. The interface receives the design variables generated by the algorithm and transmits this information to FRAN, writing the area of each bar to a file, that is subsequently read by FRAN. After the execution of the program, this interface reads the results from a generated file and computes the objective function, including its constraints and associated penalty.

On the sequential implementation of the algorithms, only one instance of the program is necessary at the same time. However, on the parallel processing implementation, several instances are needed simultaneously. To avoid different processes accessing the same files and the same executable of the program, risking a conflict of access, different directories are created. For p processes used in the computation, a directory containing the executable and associated files is replicated p times. The newly replicated directories are associated with the process identifier[6] of each process p, which is used by the interface to know which directory it should access. The described implementation is represented in the flowchart of Figure 4.6.

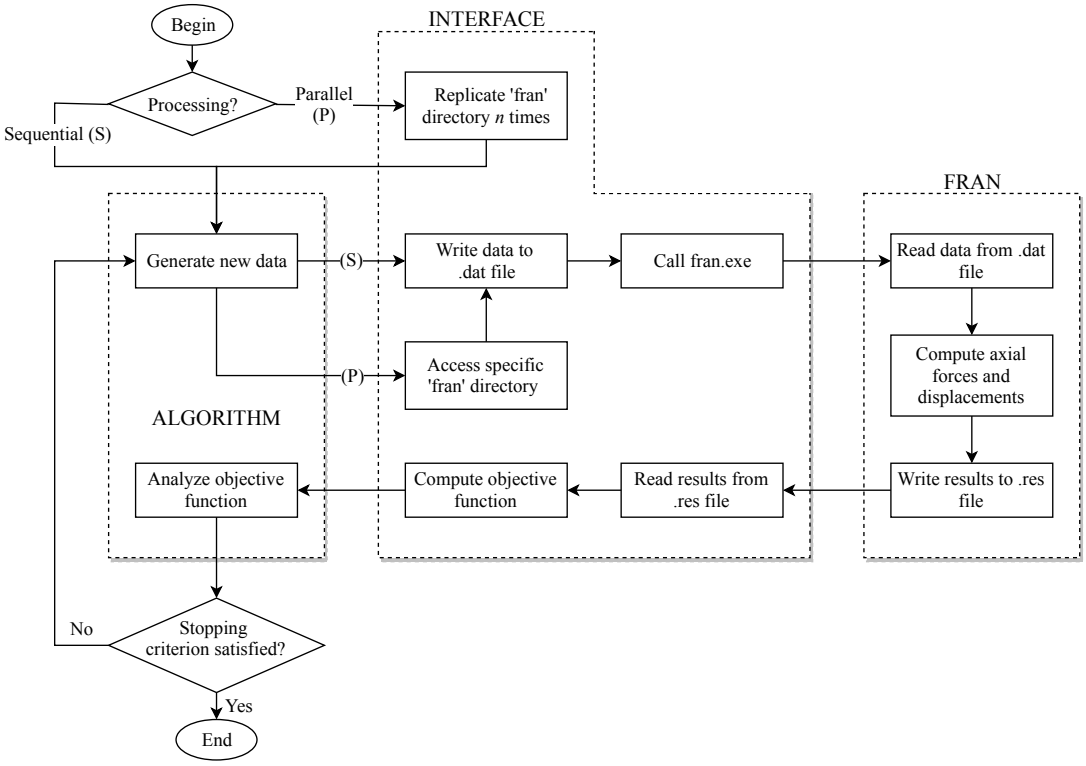
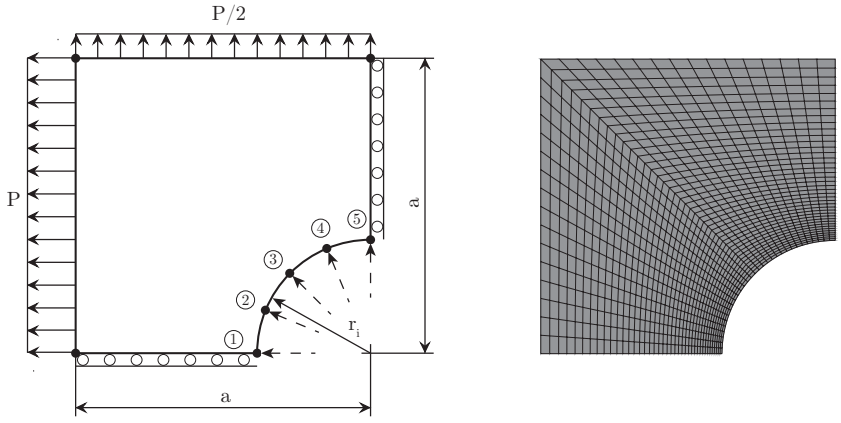


Figure 4.6: Flow diagram of the computational implementation of the three-bar truss design problem using FRAN.

[5]GiD (`https://www.gidhome.com`) is a pre and post-processing software for numerical simulations.
[6]Number used to uniquely identify an active process, commonly referred as PID.

4.4.4 Square Plate

The square plate with central cut-out hole [Papadrakakis and Lagaros 2003] is a structural application, where the goal is to minimize the area of the structure when subject to distributed loads and a threshold to the equivalent stress of $\sigma_{\max} = 7$ MPa. Plane stress conditions and isotropic material properties are assumed, with Poisson's ratio $\nu = 0.3$ and elastic modulus $E = 210$ GPa. The problem definition is given schematically in Figure 4.7, where due to double symmetry only a quarter of the plate is modeled. The initial shape of the structure is of dimensions $a = 650$ mm, with the two exterior sides of the plate loaded with a distributed loading of $P = 0.65$ MPa.



(a) (b)

Figure 4.7: Representation of (a) the geometry, loading and boundary conditions, and (b) domain discretization of the square plate design problem.

The problem consists of five design variables x_i, with $i = 1, 2, ...5$, corresponding to the radius r_1, r_2, r_3, r_4 and r_5 which can move along radial lines and are used to generate the shape of the central hole. A mathematical formulation of the problem is stated as

$$\text{minimize} \quad f(\mathbf{x}) = A_{\text{t}} - A_{\text{f}}(\mathbf{x}) \,, \tag{4.28}$$

$$\text{subject to} \quad g_j(\mathbf{x}) = \sigma_j - \sigma_{\max} \leq 0 \,, \qquad j = 1, 2, ..., m \,, \tag{4.29}$$

$$250 \leq x_i \leq 649 \,, \qquad i = 1, 2, ..., 5 \,, \tag{4.30}$$

where A_{t} is the total area of a quarter of the initial geometry and $A_{\text{f}}(\mathbf{x})$ the area of the central hole. The constraints $g_j(\mathbf{x})$ define the stress constraints imposed for all the model which is generated using 1280 linear quadrilateral elements with complete integration, corresponding to $m = 4 \times 1280$ constraints.

A fourth-degree polynomial function (in polar coordinates) is calculated for each set of design variables by the least square method. The polynomial, that passes through the five design variables, gives the coordinates for the central hole boundary.

To take only into account feasible solutions, a generic penalty function is used with the parameters $r_g = 10000$ and $\beta = 2$. The global optimum of the square plate design problem

is not known and the selected number of function evaluations as a stopping criterion is 10^3.

4.4.4.1 Computational Implementation

To compute the equivalent stresses in the model, the numerical simulation software Abaqus[7] is used. To establish a connection between the optimization algorithm and Abaqus, an interface program is implemented. The interface receives the design variables generated by the algorithm and generates the fourth-degree polynomial function which is afterwards used to calculate the Cartesian coordinates of the nodes composing the central hole. Subsequently, the interface writes the new coordinates to an input file to be read by Abaqus. After Abaqus concludes the simulation, it exports the results to a file, which the interface reads and uses to compute the objective function, constraints and associated penalty.

Both, sequential and parallel processing implementations, use the same approach as described for the computational implementation of the three bar truss design problem. The implementation of the square plate design problem is represented in the flowchart of Figure 4.8.

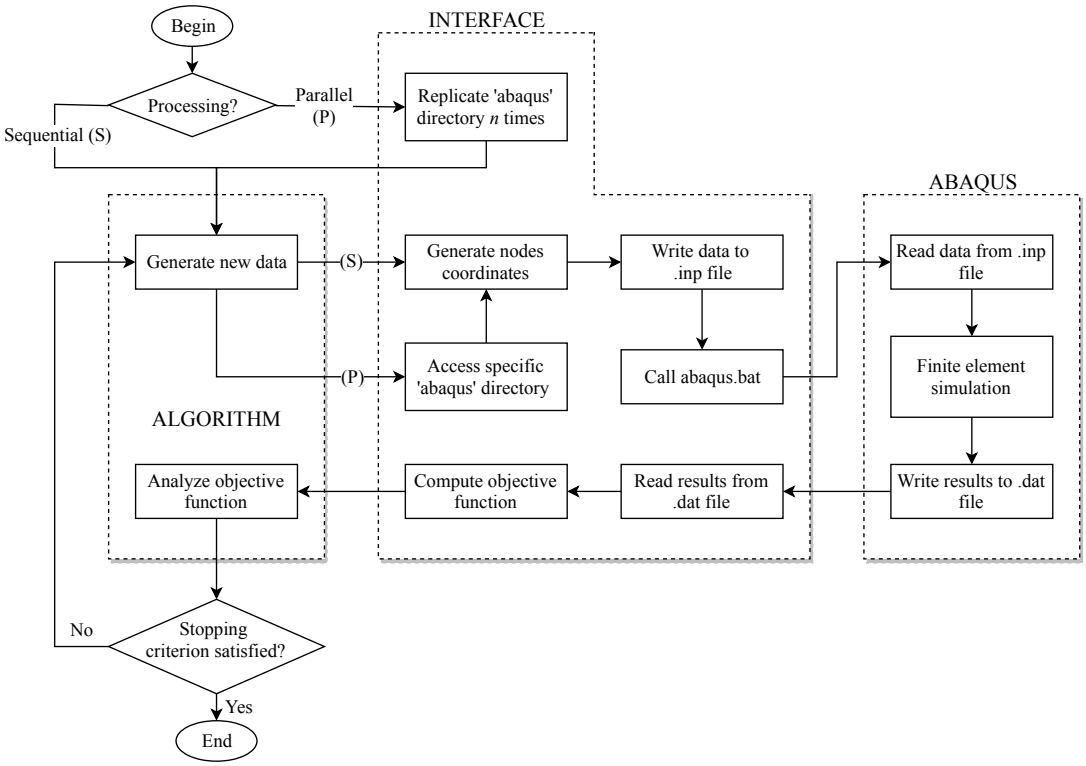


Figure 4.8: Flow diagram of the computational implementation of the square plate design problem using Abaqus.

[7]Abaqus is a software for finite element analysis and computer-aided engineering [Smith 2009].

4.5 Flow of Implementation

In summary, the implementation of the present work can be divided into four levels: the optimization algorithms, the programming languages, the type of processing and the applications. In general, all levels are interconnected. The three optimization algorithms are all implemented in the four selected programming languages. However, DE and TLBO are only implemented with sequential processing, in opposition to PSO which is implemented in both sequential and parallel processing. The implementations in C++ only include sequential processing, in opposition to Python, MATLAB and Java, which are all implemented in sequential and parallel processing. Finally, the four applications are implemented in all the presented configurations. The flow of inter-connectivity between the four levels is described in Figure 4.9.

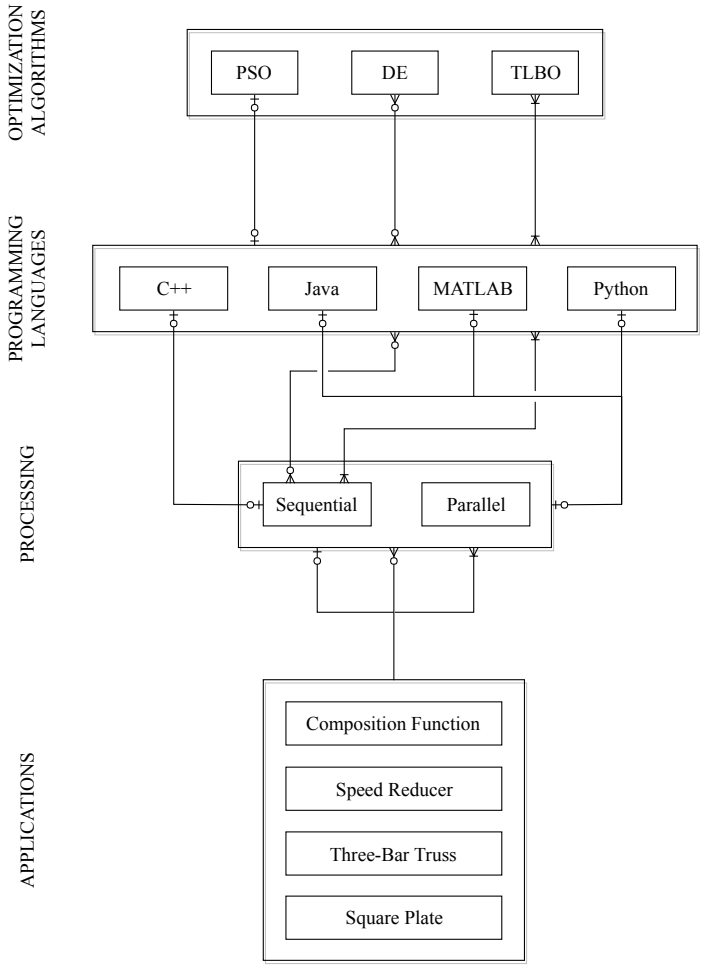


Figure 4.9: Flow diagram illustrating the inter-connectivity of the computational implementations of the algorithms, programming languages, type of processing and applications. The line symbols ⊣○, ⋊○ and ⇥⊢ represent, respectively, the inter-connectivity of the Particle Swarm Optimization, Differential Evolution and Teaching-Learning-Based Optimization with other levels.

Chapter 5

Results and Analysis

The results of a study on the size of the population and the analyses of the algorithms best solutions are presented. The results of the computational performance of the algorithms is also presented and analyzed.

In this chapter, the results for the efficiency of the advanced optimization methods and performance of programming languages, both in sequential and parallel processing are presented. Results are obtained with the same computer using a Intel® Core™ i7-4790 @ 3.60 GHz Quad-Core processor and 8 GB of RAM[1]. Moreover, the computer presents hyper-threading technology that enables each physical core to appear as two logical cores for the operating system. Thereby, from the perspective of the operating system, the processor presents eight logical cores. Additionally, Windows 10 is used as the operating system.

As a consequence of the four levels of implementation, the amount of results obtained is considerably big. Consequently, different strategies and ways of presenting the results are possible. Following this consideration, the selected strategy in which results are presented is to divide them according to the target of analysis, each including an individual analysis for each application. This way it is given emphasis to the analysis, rather than to the application itself, as different problems could have been selected for this work.

5.1 Analysis of Advanced Optimization Methods

5.1.1 Study on the Size of the Population

The implemented advanced optimization methods require the selection of operational parameters, such as the size of the population. Some parameters have already been selected (cf. Section 4.1) and are maintained throughout the different applications. However, the size of the population should be dependent on the problem's domain and constitutes an important parameter in the optimization process. If the size of the population is too small, the algorithm might not be able to carry out a detailed exploitation of the design space

[1]Random-access memory (RAM) is a form of computer data storage that stores data and machine code currently being used.

while, on the other hand, a size of the population too large requires more computational time.

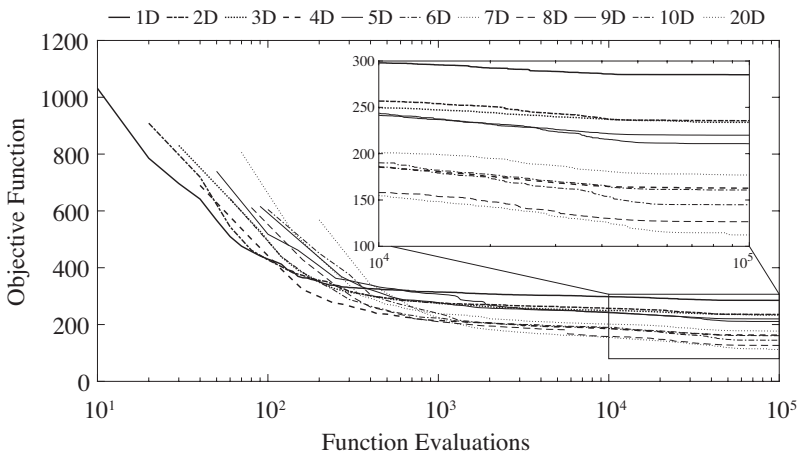
In order to achieve the best results, a parametric study on the size of the population for each algorithm is carried out for all applications. In the selection of the size of the population, it is taken into consideration that the study is not excessively extensive and covers a broad range of options. Following these considerations, the size n of the population is given by the multiplication of an integer $i = 1, 2, ..., 10$ by the dimension D of the problem. To analyze if the results of the algorithms improve with a constant increase in the size of the population, an additional value is studied, with $n = 20D$, although this value is only used for comparison purposes between the different size of the population. For all studies, twenty independent runs are computed, with its analysis focusing on the results of best (Best), mean (Mean), standard deviation (SD) and worst (Worst) values of the objective function for the twenty runs, as well as the success rate (SR) and the function evaluations (FE), required to converge on the best solution. The evolution of the mean objective function values is also analyzed, which is stored at every iteration of the optimization process. In order to select the size of the population for further analysis and simulations, it is given preference to the best solution found and success rate of all runs.

5.1.1.1 Composition Function

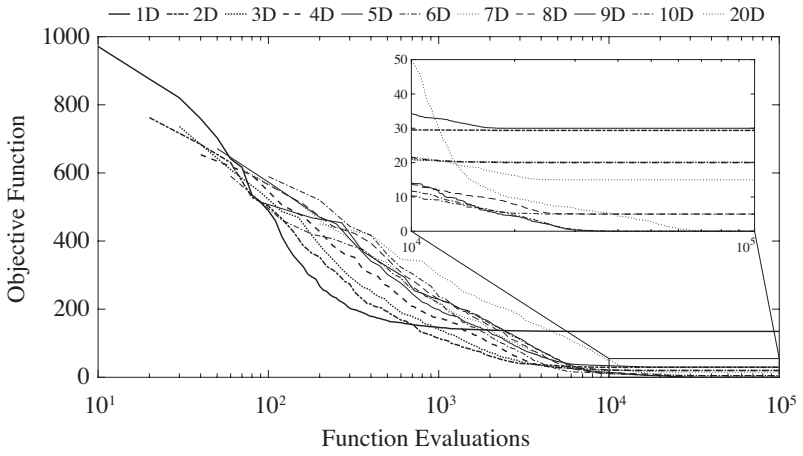
The first analyzed application is the composition function (cf. Section 4.4.1). It is presented the results of the evolution of the mean objective function for all sizes of the population (cf. Figure 5.1) and in Table 5.1 the numerical results obtained for each algorithm. In general, the three algorithms (PSO, DE and TLBO), start converging after 10^4 function evaluations, thus indicating that the selected number of function evaluations is sufficient for the algorithms to converge to a feasible solution.

In the case of the PSO, the algorithm is not able to find the global optimum in any of the runs. The results of the evolution of the mean objective function (cf. Figure 5.1) suggest that, in general, for $n \leq 5D$ the algorithm is not able to find solutions as good as the results obtained with a bigger size of the population. Nevertheless, it is observed in the results of Table 5.1 that for values such as $n = 4D$ and $5D$ the algorithm finds a solution similar to those found for bigger values. These results might be considered as a strike of 'luck', as the corresponding values for the mean and standard deviation are significantly higher than results for bigger values of n. The mean and worst results of the PSO are relatively high when compared to the DE, which is an indication that the algorithm is, sometimes, not able to escape local optima. Even though for $n = 10$ the PSO finds the best solution of all simulations, for $n = 20D$ the algorithm achieves the lowest results of mean, standard deviation and worst parameters, presenting an indication that, for this specific application, increasing the size of population for values bigger than $10D$ is a good strategy.

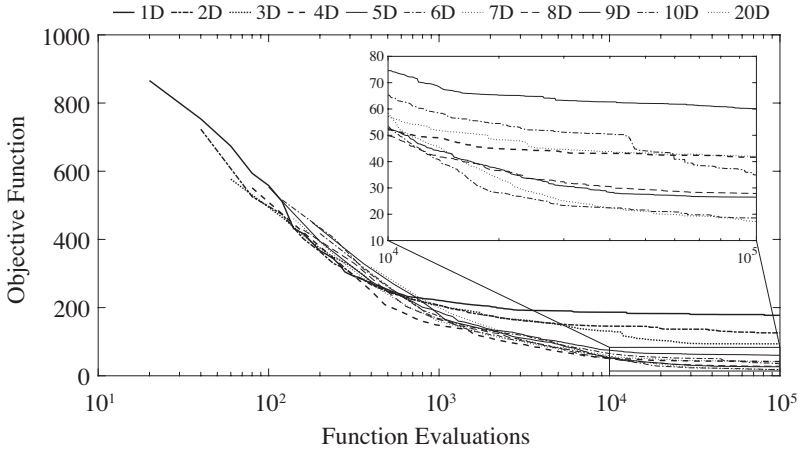
Analyzing Figure 5.1, it seems the DE algorithm performs overall better than the PSO. Except for $n = D$, the DE converges very close to the global optimum for all values of n. Observing Table 5.1, results for $n > D$ show that the algorithm is able to find the global optimum for all values of n, achieving the best results at $n = 9D$, $10D$ and $20D$. For these values of n, the DE achieves a success rate of 100%, indicating that it is able to find the global optimum in every attempt. For values of n between $2D$ and $8D$, the success rate of the algorithm is high, while mean and standard deviation parameters are relatively



(a)



(b)



(c)

Figure 5.1: Evolution of the mean objective function in relation to the number of function evaluations for different sizes of the population, obtained by (a) Particle Swarm Optimization, (b) Differential Evolution and (c) Teaching-Learning-Based Optimization for the composition function.

small. However, as observed in the worst values parameter, the algorithm seems to be stuck in local optima, at least once. From a computational effort perspective, selecting smaller values of n would be ideal, as the necessary number of function evaluations to find the global optimum is comparatively smaller than bigger values of n. However, selecting smaller values for the size of the population does not guarantee that the DE finds the global optimum, in contrast to the top three values of n that guarantee 100% probability of success. Consequently, the size of the population selected is of $n = 9D$, which provides the best solution, but it is at the same time the one with the least computational effort (least function evaluations to find global optimum) from the top three values.

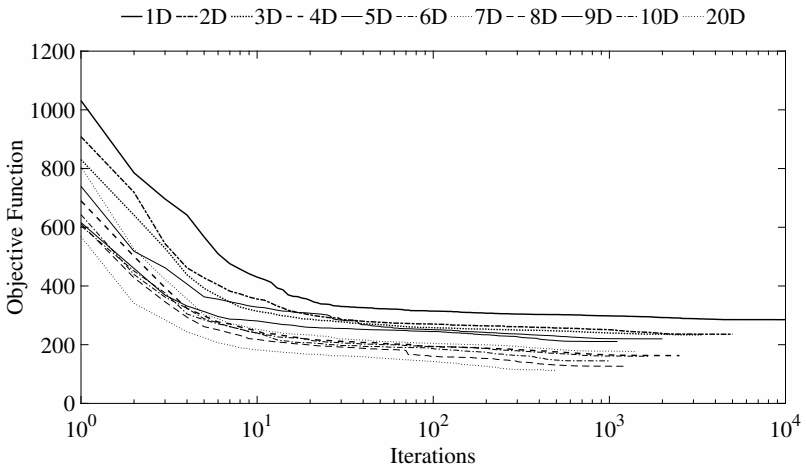
The results of the evolution of the mean objective function for the TLBO are similar to those of the PSO and the DE. For values of $n \leq 3D$ the algorithm presents difficulties converging to a good solution, in opposition to values of $n \geq 4D$ where results converge closer to the global optimum. Results presented in Table 5.1 show that, comparatively to the best solution found for $n = D$, solutions found for $n > D$ are closer to the global optimum. However, for none of the values of n is the TLBO able to reach the global optimum. The value of n that provides the best solution corresponds to $n = 20D$, while having at the same time the best result of mean value, demonstrating that an increase in the size of the population might return better results. From the other values of n, the one with the best results is $n = 10D$, which finds a solution close to the one found with $n = 20D$ and closer to the global optimum, as well as presenting the best results of standard deviation and worst parameters.

Additionally, in Figure 5.2 is presented the same results of Figure 5.1, but in relation to the number of iterations. Comparing the results of the evolution of the mean objective function this way might suggest that using bigger sizes of the population returns better solutions with fewer iterations than lower sizes of the population. However, the number of function evaluations per iteration depends on the size of the population. For example, using a $n = 20D$ comparatively to $n = D$, means that in the first scenario the algorithms evaluate the objective function twenty times more than in the latter, thus not representing a good indicator for comparison between different sizes of the population or between algorithms [Črepinšek *et al.* 2012]. Nevertheless, for this specific application results for bigger sizes of the population returned the best solutions, but that is not necessarily true for others as it will be analyzed then.

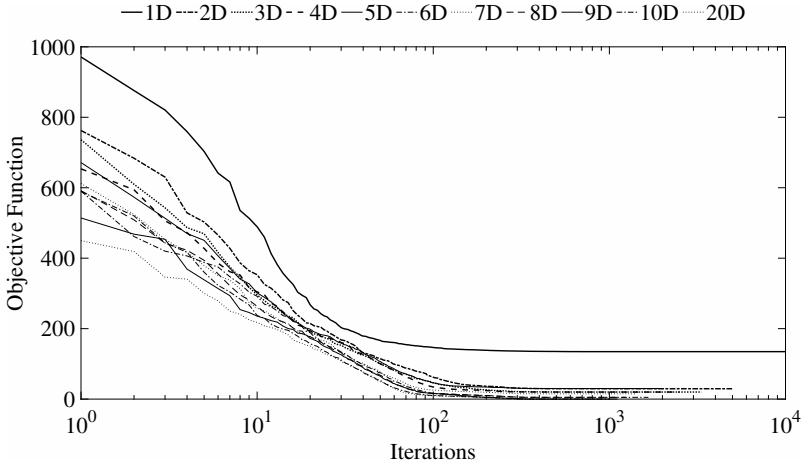
5.1.1.2 Speed Reducer

Overall, the PSO, DE and TLBO perform well for the speed reducer design problem (cf. Section 4.4.2), as results in Figure 5.3 and Table 5.2 suggest. In general, after 10^3 function evaluations the algorithms start converging towards a final solution and the evolution of the mean objective function stagnates after 10^4 function evaluations. Similar to the composition function, these results suggest that the number of function evaluations selected as the stopping criteria is adequate.

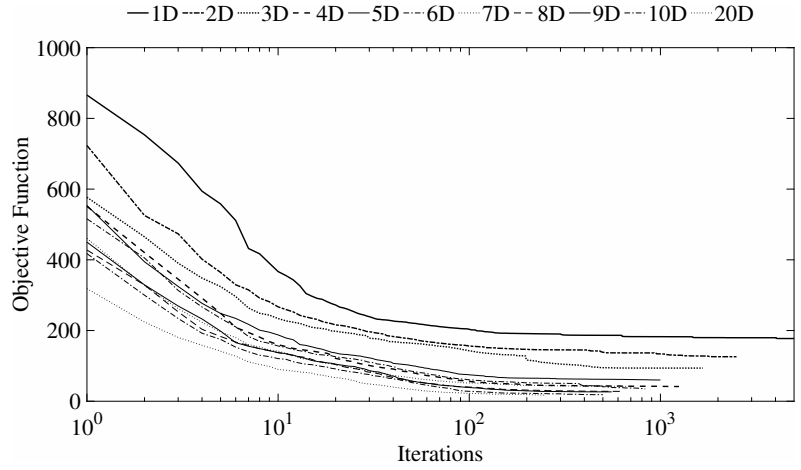
The PSO presents good results, as for all values of n it is able to find the global optimum at least three times. For $n = D$ the algorithm finds the global optimum with the least computational effort. However, the success rate is considerably low. With the increase of n, the success rate increases almost linearly, but the number of function evaluations required to find the global optimum also increases. When $n = 20D$, the best results are observed, as it presents the biggest success rate and the lowest mean and standard deviation parameters.



(a)



(b)



(c)

Figure 5.2: Evolution of the mean objective function in relation to the number of iterations for different sizes of the population, obtained by (a) Particle Swarm Optimization, (b) Differential Evolution and (c) Teaching-Learning-Based Optimization for the composition function.

Table 5.1: Results of the study of the size of the population for the composition function after 10^5 objective function evaluations.

	$D = 10$	$2D$	$3D$	$4D$	$5D$	$6D$	$7D$	$8D$	$9D$	$10D$	$20D$
Particle Swarm Optimization (PSO)											
Best	68.1040	18.8609	67.5503	21.4803	20.2354	18.8623	19.9630	21.4123	19.1741	**5.5382**	19.4867
Mean	285.1967	235.5803	233.8069	162.7580	219.9371	160.8046	176.9797	126.5760	210.7282	144.8705	**112.2948**
SD	175.1655	158.9303	124.2532	114.7194	132.6972	140.9442	96.4383	79.3632	178.0633	143.4580	**75.2282**
Worst	608.1888	561.2029	535.2635	533.6388	546.8714	538.2419	440.8937	288.3453	555.5497	536.6361	**214.6942**
SR [%]						0					
FE						-					
Differential Evolution (DE)											
Best	8.5809	**0.0000**	**0.0000**	**0.0000**	**0.0000**	**0.0000**	**0.0000**	**0.0000**	**0.0000**	**0.0000**	**0.0000**
Mean	134.6974	29.3738	20.0648	20.0000	30.0000	5.0000	15.0000	5.0000	**0.0000**	**0.0000**	**0.0000**
SD	124.7878	58.2055	39.9686	40.0000	45.8258	21.7945	35.7071	21.7945	**0.0000**	**0.0000**	**0.0000**
Worst	522.1056	224.8740	100.0000	100.0000	100.0000	100.0000	100.0000	100.0000	**0.0000**	**0.0000**	**0.0000**
SR [%]	0	45	75	80	70	95	85	95	**100**	**100**	**100**
FE	-	**5116**	8675	12634	14179	17777	19036	23673	28220	29815	64802
Teaching-Learning-Based Optimization (TLBO)											
Best	28.3274	8.8928	3.5430	4.2592	4.4064	4.1411	4.8029	2.6066	4.1411	2.2724	**2.1277**
Mean	177.3843	125.8554	93.7670	41.6808	60.0925	34.9474	42.0471	27.9076	26.5021	18.5532	**17.2895**
SD	139.4372	170.9990	110.4328	63.4592	66.5996	39.5137	62.5035	49.9435	38.0823	**28.2106**	35.1329
Worst	510.0844	513.3866	408.8325	225.2778	221.2254	108.5469	223.8318	174.7983	102.7808	**100.0000**	154.5555
SR [%]						0					
FE						-					

SD - Standard Deviation SR - Success Rate FE - Function Evaluations

This observation indicates that, for this application, the PSO benefits from an increase in the size of the population. Observing results for $n \leq 10D$, the size of population with the best results is $n = 10D$.

In the case of the DE, results demonstrate to improve with the increase of n. Except for $n = D$, the algorithm is able to find the global optimum for all values of n. Between $n = 2D$ and $n = 5D$, the results of mean, standard deviation and success rate improve with the increase of n, and when $n \geq 6D$ the algorithm finds the global optimum on 100% of the runs while presenting the least computational effort for $n = 6D$.

Results of the TLBO demonstrate that the algorithm performs relatively well for all values of n except $n = D$ and $n = 20D$. When $n = D$, the TLBO is able to find the global optimum but presents a low success rate. On the other hand, for $n = 20D$ the algorithm does not find the global optimum and the result of the mean solution is close to the global optimum and standard deviation is almost zero, indicating that for this size of the population the algoritghm might require more function evaluations. Between $n = 4D$ and $n = 10D$, high rates of success are observed, but in some situations, such as $n = 4D, 5D, 8D, 10D$, it is not able to converge to the global optimum or is trapped in local optima. When the size of the population is equal to $6D$ or $7D$, the success rate is of 100%, while for $n = 6D$ the algorithm finds the global optimum with fewer function evaluations.

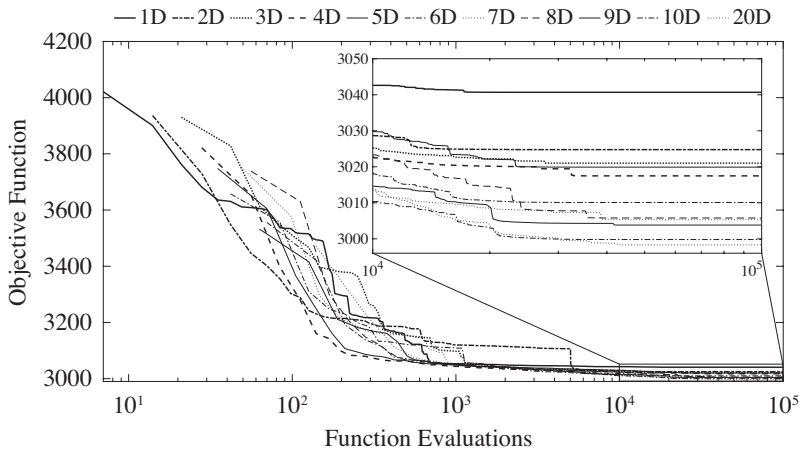
In general, the speed reducer design problem seems to present several local optima in the search range. This observation is reasoned by the analysis of results in Table 5.2, where similar solutions of the worst parameter are repeated throughout the PSO, DE and TLBO.

5.1.1.3 Three-Bar Truss

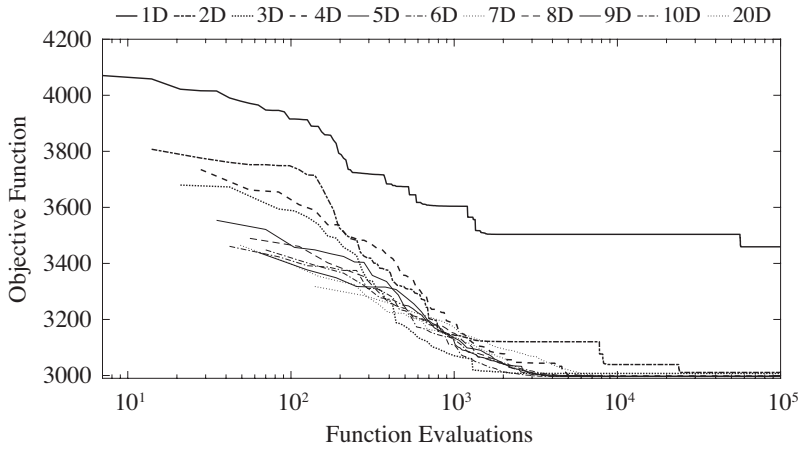
The two applications previously presented are solely implemented with numerical operations, thereby allowing the selection of a large number of function evaluations as the stopping criteria. However, the three-bar truss design problem (cf. Section 4.4.3) uses an external program in the operations leading to the evaluation of the objective function, thus requiring more computational time. Because of this characteristic, it was necessary to decrease the number of function evaluations as the stopping criteria to 10^4. Nevertheless, from the results of the evolution of the mean objective function presented in Figure 5.4, it is observed that the algorithms easily converge until 10^3 function evaluations and begin stagnating after it. These results indicate that the selected number of function evaluations as the stopping criteria are sufficient for the algorithms to find the global optimum.

In the case of the PSO, for all values of n the solution seems to gradually converge to the global optimum. In the range of $D \leq n \leq 10D$, the algorithm finds the global optimum at least four times, achieving a 100% success rate for $n = 3D, 5D, 6D$ and $7D$. For $n = 20D$, the PSO does not have the capability to converge to the global optimum, stagnating close to it in all runs, as results of mean and standard deviation demonstrate. Overall, the algorithm presents high rates of success, while for $n = 3D$ it requires the least computational effort from the values of n with 100% success rate.

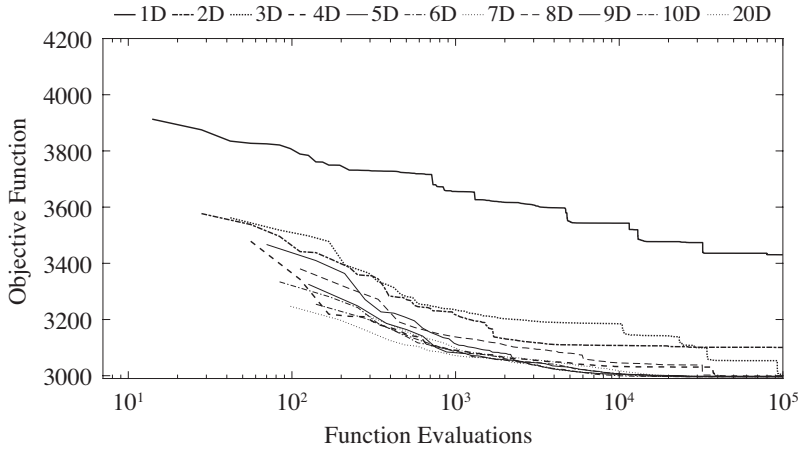
For this application, the DE has the particularity of not having been tested for $n = D$, as the size of the population is less than the minimum required by the algorithm. For this reason, the lowest value of n is equal to $2D$, with results not demonstrating the algorithms ability to find the global optimum. Beginning with $n = 3D$, the DE is able to find the global optimum using 10^5 function evaluations, although it only presents a high success



(a)



(b)



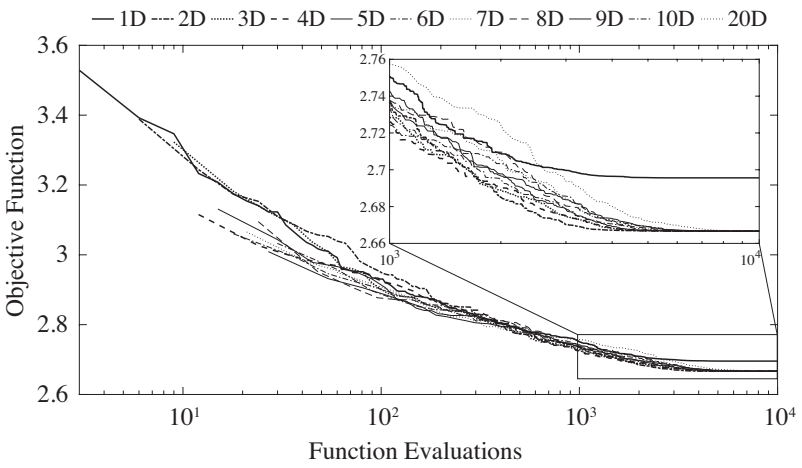
(c)

Figure 5.3: Evolution of the mean objective function for different sizes of the population, obtained by (a) Particle Swarm Optimization, (b) Differential Evolution and (c) Teaching-Learning-Based Optimization for the speed reducer.

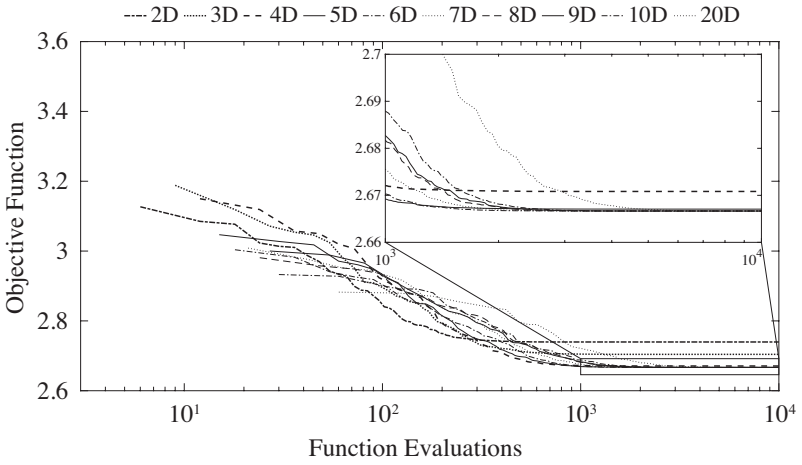
Table 5.2: Results of the study of the size of the population for the speed reducer after 10^5 objective function evaluations.

	$D = 7$	$2D$	$3D$	$4D$	$5D$	$6D$	$7D$	$8D$	$9D$	$10D$	$20D$
					Particle Swarm Optimization (PSO)						
Best	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**
Mean	3040.681890	3024.766060	3021.013933	3017.474574	3019.905022	3010.085783	3005.303010	3005.779178	3003.815207	2999.799825	**2998.312036**
SD	41.704526	18.250183	21.195641	19.404722	20.727679	18.447464	15.156840	15.370436	15.131191	11.119010	**8.560314**
Worst	3204.604187	3044.957401	3056.045507	3044.957401	3046.713685	3044.957401	3044.957401	**3035.625579**	3044.957401	3046.713685	**3035.625579**
SR [%]	15	20	35	40	40	60	65	65	75	85	**95**
FE	**30794**	37180	35022	41570	45313	47804	49237	48541	54394	55771	70669
					Differential Evolution (DE)						
Best	3000.720915	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**
Mean	3459.293814	3010.897000	3007.370880	2998.312036	2998.312036	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**
SD	576.490247	17.425219	17.121336	8.560314	8.560314	**0.000000**	**0.000000**	**0.000000**	**0.000000**	**0.000000**	**0.000000**
Worst	4799.552364	3037.327029	3035.625579	3035.625579	3035.625579	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**
SR [%]	0	35	70	95	95	**100**	**100**	**100**	**100**	**100**	**100**
FE	-	**2798**	4136	5638	7339	8918	10570	12420	14181	15589	30411
					Teaching-Learning-Based Optimization (TLBO)						
Best	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	**2996.348165**	2996.348166
Mean	3430.429860	3100.640799	3004.263829	2998.312036	2998.312036	**2996.348165**	**2996.348165**	2996.348166	**2996.348165**	2996.348166	2996.348268
SD	511.120560	301.778315	15.832989	8.560314	8.560314	**0.000000**	**0.000000**	0.000001	**0.000000**	0.000001	0.000326
Worst	4443.722297	4103.048311	3036.814626	3035.625579	3035.625579	**2996.348165**	**2996.348165**	2996.348184	**2996.348165**	2996.348167	2996.349678
SR [%]	15	75	80	95	95	**100**	**100**	95	**100**	90	0
FE	15769	**9786**	15295	20462	27454	33138	39915	42350	45363	49332	-

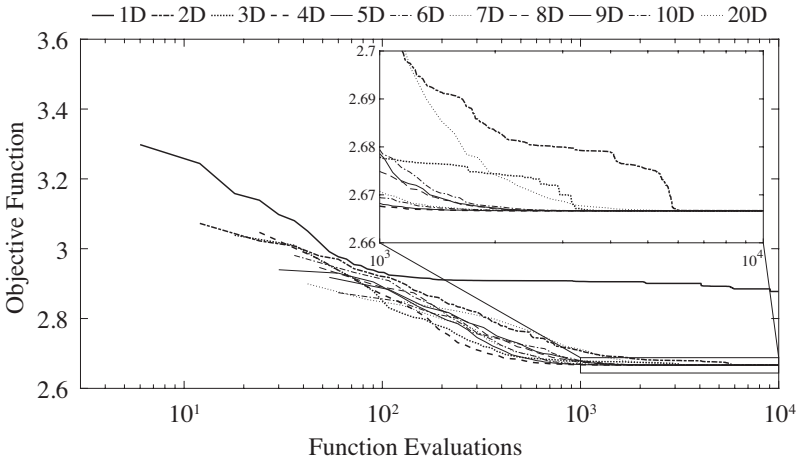
SD - Standard Deviation SR - Success Rate FE - Function Evaluations



(a)



(b)



(c)

Figure 5.4: Evolution of the mean objective function for different sizes of the population, obtained by (a) Particle Swarm Optimization, (b) Differential Evolution and (c) Teaching-Learning-Based Optimization for the three-bar truss.

rate for $n \geq 6D$. For $n = 7D, 8D, 9D$ and $10D$, the algorithm finds the global optimum on 100% of the runs with relative ease, as the required number of function evaluations demonstrate. When $n = 20D$ the global optimum is achieved. However, on approximately half of the runs, the algorithm fails to converge to the global optimum. Overall, for $n = 7D$ is when the DE presents the least computational effort from values of n with perfect success rate.

The TLBO is the algorithm that presents the best results, as it achieved a 100% success rate for seven values of n. Except for $n = D$, the algorithm is able to find the global optimum with a low number of function evaluations. For $n = 20D$ the algorithm requires a lot more computation effort compared to lower values of n and does not present a perfect success rate. From the values of n that present 100% success rate, namely $n = 4D$ to $10D$, the one where the algorithm requires less function evaluations is for $n = 4D$.

From all the implemented applications, the three-bar truss design problem is the one that presents the lowest dimension ($D = 3$), as well as having side constraints with a small range compared to the other application. These characteristics might explain the fact that good results are found for lower values of n.

5.1.1.4 Square Plate

Similarly to the three-bar truss, the optimization process for the square plate design problem was *a priori* limited by the little number of function evaluations defined as the stopping criteria (cf. Section 4.4.4), as a consequence of the operations required in the evaluation of the objective function using an external program (Abaqus). Consequently, observing the results of the evolution of the mean objective function for the square plate design problem (cf. Figure 5.9) it is not evident if the algorithms are able to converge to a solution that is maybe close to the global optimum or not. Results show a slight stagnation of the objective function values close to the end of the optimization process. However, the observed stagnation is not sufficiently long to conclude whether the selected number of function evaluations is adequate or not for this application. Furthermore, results presented in Figure 5.5 for the PSO, DE and TLBO algorithms show considerable differences between them in the values of the objective function in the beginning of the optimization process. In the case of the PSO and the DE, the high values observed are explained by the addition of penalty values to the objective function, a consequence of several constraints being violated and the algorithms not finding feasible solutions. On the other hand, the lower results of the TLBO are explained by the fact that the algorithm evaluates the objective function two times more than the PSO or the DE for the same number of iterations.

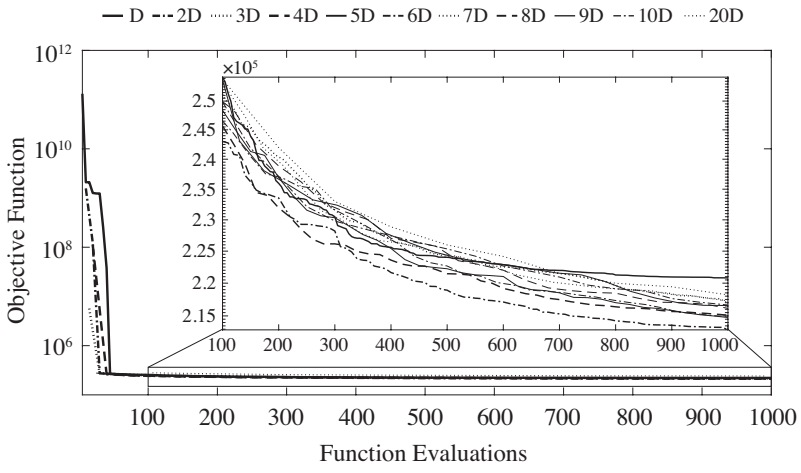
By first observing the results of the PSO for $n = D$ to $4D$ at the beginning of the optimization process, the algorithm presents solutions of lower quality. Approximately after 50 function evaluations, values of the mean objective function evolve into an acceptable range – compared to the reported best solution (cf. Table 5.8). Analyzing the results presented in Table 5.8 the best solution is found for $n = 2D$, while presenting at the same time the best result of the mean parameter. For $n > 2D$, reported solutions are worst and the mean solution tends to be higher with the increase of n, thereby not benefiting the algorithm.

Similarly to the PSO, the DE presents solutions of lower quality in the beginning of the optimization process for $n = D, 2D, 3D$ and $4D$. For $n = 4D$ the algorithm presents the best results of mean and worst parameters. Furthermore, the value of standard deviation

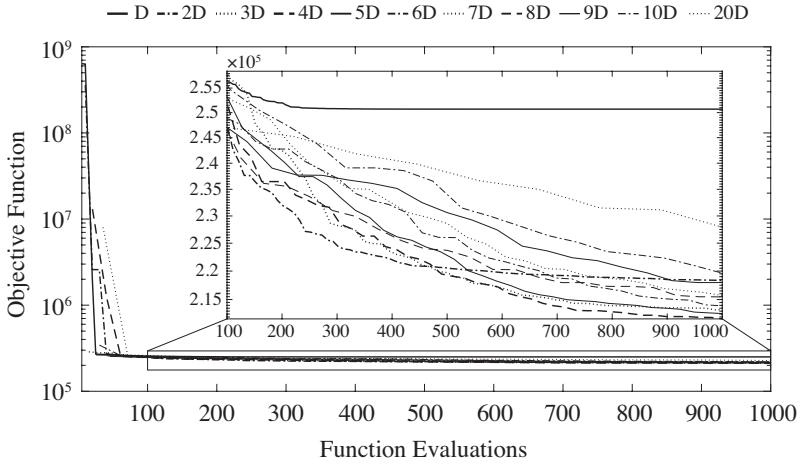
Table 5.3: Results of the study of the size of the population for the three-bar truss after 10^4 objective function evaluations.

	$D = 3$	$2D$	$3D$	$4D$	$5D$	$6D$	$7D$	$8D$	$9D$	$10D$	$20D$
Particle Swarm Optimization (PSO)											
Best	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	2.666669
Mean	2.695602	2.666673	**2.666667**	2.666668	**2.666667**	**2.666667**	**2.666667**	2.666668	2.666668	2.666668	2.666691
SD	0.124000	0.000018	**0.000000**	0.000002	**0.000000**	**0.000000**	**0.000000**	0.000002	0.000001	0.000001	0.000022
Worst	3.236068	2.666744	**2.666667**	2.666676	**2.666667**	**2.666667**	**2.666667**	2.666673	2.666670	2.666669	2.666752
SR [%]	20	70	**100**	95	**100**	**100**	**100**	80	90	90	0
FE	**4788**	4930	5932	5833	6753	6742	6697	7977	7593	8334	-
Differential Evolution (DE)											
Best	-	2.666830	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**
Mean	-	2.739598	2.704157	2.670826	2.667065	2.666676	**2.666667**	**2.666667**	**2.666667**	**2.666667**	2.666668
SD	-	0.084165	0.071833	0.008421	0.001432	0.000044	**0.000000**	**0.000000**	**0.000000**	**0.000000**	0.000001
Worst	-	2.952044	2.995049	2.694800	2.673162	2.666868	**2.666667**	**2.666667**	**2.666667**	**2.666667**	2.666669
SR [%]	-	0	10	55	75	95	**100**	**100**	**100**	**100**	45
FE	-	-	**1183**	1668	2084	2304	3010	3561	3917	4347	8711
Teaching-Learning-Based Optimization (TLBO)											
Best	2.670512	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**
Mean	2.877485	2.666707	2.666668	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	2.666668
SD	0.252081	0.000145	0.000007	**0.000000**	**0.000000**	**0.000000**	**0.000000**	**0.000000**	**0.000000**	**0.000000**	0.000001
Worst	3.582162	2.667329	2.666699	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	**2.666667**	2.666671
SR [%]	0	70	90	**100**	**100**	**100**	**100**	**100**	**100**	**100**	65
FE	-	**1030**	1367	1789	1976	2301	2566	3240	3442	3837	8306

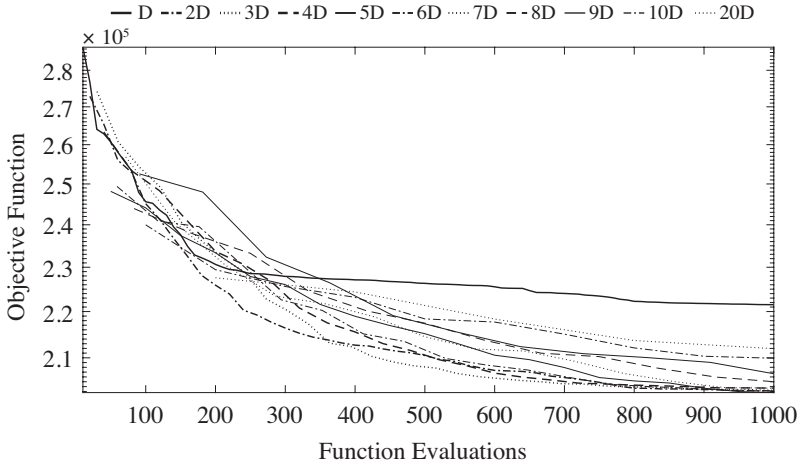
SD - Standard Deviation SR - Success Rate FE - Function Evaluations



(a)



(b)



(c)

Figure 5.5: Evolution of the mean objective function for different sizes of the population, obtained by (a) Particle Swarm Optimization, (b) Differential Evolution and (c) Teaching-Learning-Based Optimization for the square plate.

is comparatively smaller than for other values of n. However, it is for $n = 2D$ that the DE presents the best solution, even though the standard deviation parameter is relatively high and the worst parameter far from the best solution. Following these observations, it would be wise to select a size of the population equal to $4D$, as it returns good solutions more often. However, as the preferred criterion is the best solution found, the selected size of the population is $n = 2D$.

The TLBO is the only algorithm that presents for all values of n solutions that are not excessively penalized from the beginning of the optimization process. Similarly to the PSO and the DE, the best solution in the TLBO is found when $n = 2D$. Nevertheless, this value of n does not present the best results of mean, standard deviation and worst parameters.

In general, the algorithms are not able to converge to similar solutions, as the standard deviation is relatively high for all values of n. Furthermore, it is observed that the number of function evaluations required to find the best solution is, in general, very close to the total number of function evaluations. These results help to conclude that selecting a bigger number of function evaluations as the stopping criteria would be beneficial to reach better conclusions on the most adequate size of the population. Moreover, these results do not necessarily mean that with a bigger size of the population the algorithms do not perform well for the application, only that for the selected number of function evaluations as the stopping criteria results are not ideal and if possible should have been higher. Nevertheless, from an engineering perspective, where quality solutions are desired within an acceptable time, these algorithms show better results for lower sizes of the population.

5.1.1.5 Global Analysis

Overall, results demonstrate that the selection of a proper size of the population that is able to find the global optimum and give guarantees to find the best possible solution is not a trivial decision. Throughout the different applications, the size of population for each algorithm presents great variation, as for the composition function high values of the size of the population returned the best results, but for the square plate, the best results are found for low values. Additionally, for the speed reducer, the best results are reported for intermediate values of the size of the population and for the three-bar truss lower values are preferred. These results seem to demonstrate that for applications of higher dimension, it is preferred higher values of the size of the population, while for applications of lower dimension the opposite is favored. This observation is a consequence of the design space being larger for problems of higher dimension, therefore requiring that a bigger number of solutions simultaneously explore the design space. Furthermore, if the computational time of the application does not allow for a large number of function evaluations as it happens with the square plate, lower values of the size of the population are preferred. Following the study described in this section, the selected sizes of the population for each algorithm and application that returned the best solutions are:

- Composition Function – $10D$ (PSO), $9D$ (DE) and $10D$ (TLBO);
- Speed Reducer – $10D$ (PSO), $6D$ (DE) and $6D$ (TLBO);
- Three-Bar Truss – $3D$ (PSO), $7D$ (DE) and $4D$ (TLBO);
- Square Plate – $2D$ (PSO), $2D$ (DE) and $2D$ (TLBO).

Table 5.4: Results of the study of the size of the population for the square plate after 1000 objective function evaluations.

	$D = 5$	$2D$	$3D$	$4D$	$5D$	$6D$	$7D$	$8D$	$9D$	$10D$	$20D$
					Particle Swarm Optimization						
Best	211229.5438	**210209.5512**	210738.5654	211589.3167	210455.6825	210369.6080	210724.2962	211697.9548	211841.7692	212846.3028	212318.7380
Mean	220813.0501	**212901.9901**	216942.7795	215116.3360	214757.9389	214927.8259	217263.9956	216144.1706	216574.2161	216650.2116	218195.0716
SD	13975.0456	3410.9547	6686.6539	4217.2493	**3154.1935**	4062.4400	4066.0561	4305.8748	3293.3774	3511.2528	3204.9476
Worst	261623.5505	220849.6972	238447.0262	223296.7617	**220844.7234**	224222.1465	221973.3890	230128.2763	221963.7194	224796.8673	225104.7827
FE	886	936	982	871	883	983	986	990	936	992	644
					Differential Evolution						
Best	220608.9802	**209846.2641**	210503.1741	210551.6806	210391.0549	210816.0997	211936.0204	214097.5134	214258.2648	214553.5984	219177.5246
Mean	250704.1744	218342.3776	213201.8772	**211669.7877**	212585.7550	213914.1849	215697.5373	215461.0913	217965.4126	219629.0658	227934.9286
SD	29515.6513	11860.4069	5481.3106	**799.7194**	1427.3761	2485.5009	2318.7278	958.5623	2955.8748	4644.0089	5815.5946
Worst	290789.2691	249477.7440	228673.3140	**212918.0265**	214747.6153	217582.7808	220331.2160	216672.3242	223314.2987	227786.4017	235514.1821
FE	980	996	986	900	926	990	951	995	945	948	800
					Teaching-Learning-Based Optimization						
Best	210861.4829	**209791.7287**	210283.2418	210360.5282	210424.6539	211650.5782	212202.2012	212479.9736	213135.1490	215906.4608	215856.6477
Mean	231521.3448	213128.3322	213215.0398	213696.6814	**212771.5137**	213117.4415	213670.1594	215002.8987	217053.1562	219913.7880	221940.6905
SD	20577.3220	3435.5616	3614.4608	4151.6586	3049.3096	**929.3933**	1728.9387	2023.7672	3373.6517	3325.1304	4366.6694
Worst	270540.2051	220265.1315	222404.5604	220067.5149	218766.5611	**214279.9579**	218035.3803	219110.3461	222944.2539	226626.2019	228907.3521
FE	981	971	781	883	997	934	908	847	900	887	772

SD - Standard Deviation FE - Function Evaluations

5.1.2 Comparison of the Algorithms Best Solution

The previous study allowed for a more thoughtful selection of the size of the population to use in each algorithm and application. However, in the previous study, the analysis did not focus on the comparison of results between algorithms, but only on the results within each algorithm. In this section, the results obtained for the selected sizes of the population (cf. Section 5.1.1.5) are compared for each application.

5.1.2.1 Composition Function

In general, the composition function represents a difficult optimization problem for the algorithms, as the DE is the only one able to find the global optimum (cf. Table 5.5). Results show that the DE excels in every parameter, as it finds the global optimum in every attempt and requires on average less than half of the total number of function evaluations. Furthermore, the number of function evaluations in the run which the global optimum was found with the least function evaluations are not farther away from the mean number of function evaluations of all runs, as the corresponding standard deviation is relatively small. Comparing the PSO and the TLBO, the latter present better results as it is able to find a better solution while presenting the mean, standard deviation and worst parameters lower than the PSO.

Comparing the evolution of the mean objective function for each algorithm as presented in Figure 5.6, it is observed that the PSO is the algorithm with the most difficulty in converging for solutions of higher quality. At the beginning of the optimization process, the fact that the TLBO succeeds in finding better solutions than the PSO and DE might be because, in each iteration, the TLBO evaluates the objective function twice as much as the other two algorithms. The PSO begins the optimization process with higher solutions, but quickly converge to lower solutions. However, at approximately 10^3 function evaluations, its evolution begins stagnating. On the other hand, solutions of the DE and TLBO slightly evolve more slowly, delaying the beginning of stagnation for approximately 10^4 function evaluations. Nevertheless, DE presents better solutions than PSO approximately from 10^3 function evaluations and better than TLBO slightly after. Additionally, the selected size of the population are the same for the PSO and TLBO ($10D$), but results demonstrate the TLBO performs better than the PSO. This difference in the performance of the two algorithms is well represented in the evolution of the mean objective function as TLBO presents at 10^3 function evaluations the same values the PSO presents at 10^5 function evaluations.

5.1.2.2 Speed Reducer

The speed reducer proves to be a relatively easy problem for the three algorithms with the selected size of the population, as all of them are able to find the global optimum within the total number of function evaluations. In Table 5.6 the results are summarized for each algorithm, where it is observed that the DE and TLBO succeed in finding the global optimum 100% of the runs, in opposition to the PSO that presents an 85% success rate. Another difference in the three algorithms is in the number of function evaluations required to find the global optimum, as the DE requires on average less than 10% of the total number of function evaluations, while the PSO and TLBO require approximately 50%. It should be noted that, even though the TLBO presents a lower value of the mean number

Table 5.5: Results of the selected size of the population for the Particle Swarm Optimization, Differential Evolution and Teaching-Learning-Based Optimization for the composition function.

	PSO	DE	TLBO
Best	5.5382	**0.0000**	2.2724
Mean	144.8705	**0.0000**	18.5532
SD	143.4580	**0.0000**	28.2106
Worst	536.6361	**0.0000**	100.0000
SR (%)	0	**100**	0
FE	-	**28220**	-
Mean FE	-	**33987**	-
SD FE	-	**3710**	-
n	$10D$	$9D$	$10D$
G	1000	1112	500
Design Variables			
x_1	1.5832	1.5953	1.6122
x_2	2.6296	2.6440	2.6659
x_3	1.8876	1.8047	2.6659
x_4	1.0234	0.9389	1.0246
x_5	−3.1849	−3.0486	−3.0995
x_6	−1.2192	−1.1571	−1.1036
x_7	3.6541	3.5582	3.6734
x_8	2.3576	2.4246	2.4753
x_9	−0.4575	−0.3767	−0.3660
x_{10}	4.6053	4.4637	4.4981

SD - Standard Deviation SR - Success Rate
FE - Function Evaluations G - Generations

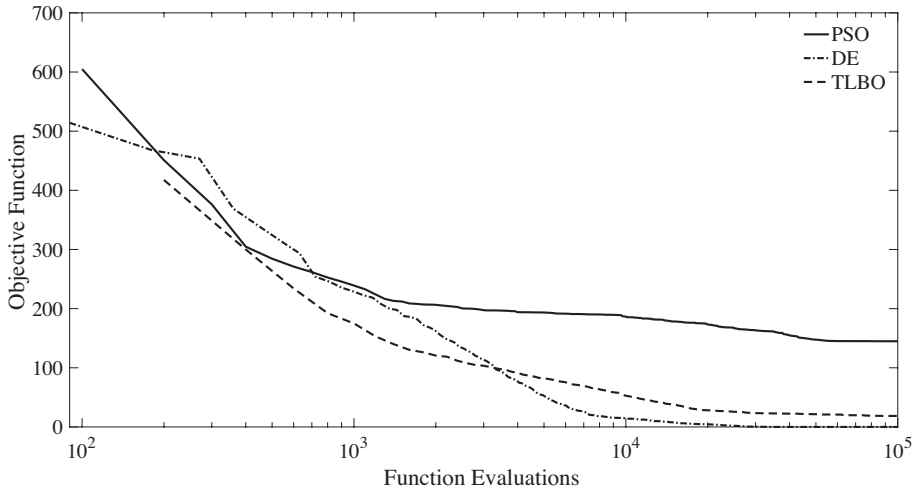


Figure 5.6: Evolution of the mean objective function obtained by the Particle Swarm Optimization, Differential Evolution and Teaching-Learning-Based Optimization for the composition function.

of function evaluations than the PSO, its corresponding standard deviation is significantly higher, demonstrating that on some runs it required more computational effort.

As observed in Figure 5.7, the evolution of the mean penalty function tends, for the three algorithms, to decrease with the number of function evaluations. Approximately at 10^3 function evaluations, the value of penalty function is zero which means that solutions are not violating imposed constraints. However, at the beginning of the optimization process the algorithms are not able to find feasible solutions (penalty function value equal to zero), while the PSO is the quickest algorithm to converge to feasible solutions and the DE the slowest. Furthermore, the evolution of the mean objective function in the PSO tends to quickly converge to better solutions at the beginning of the optimization process, but the convergence rate decreases afterwards, in opposition to DE where the convergence rate is more constant throughout the optimization process. Finally, the values of the design variables corresponding to the best solution found for each algorithm corresponds to those of the global optimum, therefore validating the optimization using the PSO, DE and TLBO.

Table 5.6: Results of the selected size of the population for the Particle Swarm Optimization, Differential Evolution and Teaching-Learning-Based Optimization for the speed reducer.

	PSO	DE	TLBO
Best	**2996.348165**	**2996.348165**	**2996.348165**
Mean	2999.799825	**2996.348165**	**2996.348165**
SD	11.119010	**0.000000**	**0.000000**
Worst	3046.713685	**2996.348165**	**2996.348165**
SR (%)	85	**100**	**100**
FE	55771	**8918**	33138
Mean FE	58216	**9780**	50482
SD FE	1095	**535**	11519
n	$10D$	$6D$	$6D$
G	1429	2381	1191
Design Variables			
x_1	3.499 999	3.499 999	3.499 999
x_2	0.700 000	0.700 000	0.700 000
x_3	17	17	17
x_4	7.300 000	7.300 000	7.300 000
x_5	7.800 000	7.800 000	7.800 000
x_6	3.350 215	3.350 215	3.350 215
x_7	5.286 683	5.286 683	5.286 683

SD - Standard Deviation SR - Success Rate FE - Function Evaluations
G - Generations

5.1.2.3 Three-Bar Truss

As demonstrated by the results of Table 5.7, the three-bar truss design problem presents itself an easier optimization problem for the three algorithms, as all of them are able to find the global optimum 100% of the runs. Consequently, the results for the best, mean, standard deviation and worst parameters are identical for the three algorithms. Differences

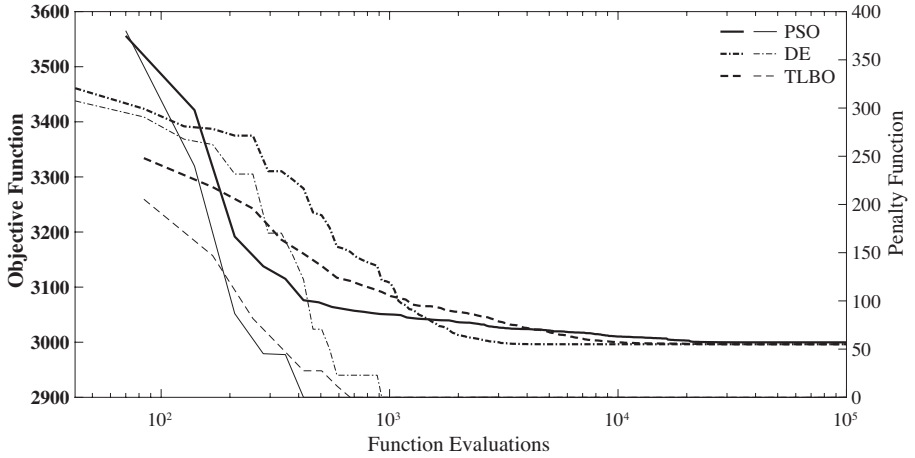


Figure 5.7: Evolution of the mean objective function (—) and penalty function (—) obtained by the Particle Swarm Optimization, Differential Evolution and Teaching-Learning-Based Optimization for the speed reducer.

between the algorithms are found in the computational effort each one requires to reach the global optimum. The TLBO is the algorithm that finds the global optimum with the least computation effort, as the DE takes, on average, almost twice and the PSO three times number of function evaluations.

Table 5.7: Results of the selected size of the population for the Particle Swarm Optimization, Differential Evolution and Teaching-Learning-Based Optimization for the three-bar truss.

	PSO	DE	TLBO
Best	**2.666667**	**2.666667**	**2.666667**
Mean	**2.666667**	**2.666667**	**2.666667**
SD	**0.000000**	**0.000000**	**0.000000**
Worst	**2.666667**	**2.666667**	**2.666667**
SR (%)	**100**	**100**	**100**
FE	5932	3010	**1789**
Mean FE	7056	3740	**2285**
SD FE	863	834	**350**
n	$3D$	$7D$	$4D$
G	1112	477	417
Design Variables			
x_1	0.666 667	0.666 667	0.666 667
x_2	1.333 333	1.333 333	1.333 333
x_3	0.666 667	0.666 667	0.666 667

SD - Standard Deviation SR - Success Rate FE - Function Evaluations
G - Generations

Regarding the evolution of the mean objective and penalty function presented in Figure 5.8, it is observed that the PSO presents solutions of lower quality at the beginning of the optimization process, derived from the reduced size of the population used (cf. Table 5.7).

However, it is able to progressively converge to solutions of more quality in an advanced stage of the optimization process. The DE and TLBO algorithms present better solutions in the beginning and start converging to the global optimum sooner than PSO. The mean evolution of penalty function for each algorithm demonstrates that TLBO is the first algorithm to find feasible solutions, approximately at 10^2 function evaluations, followed by the DE and lastly the PSO, that rapidly finds solutions close to the feasible boundary, even though it is only able to find feasible ones after 10^3 function evaluations.

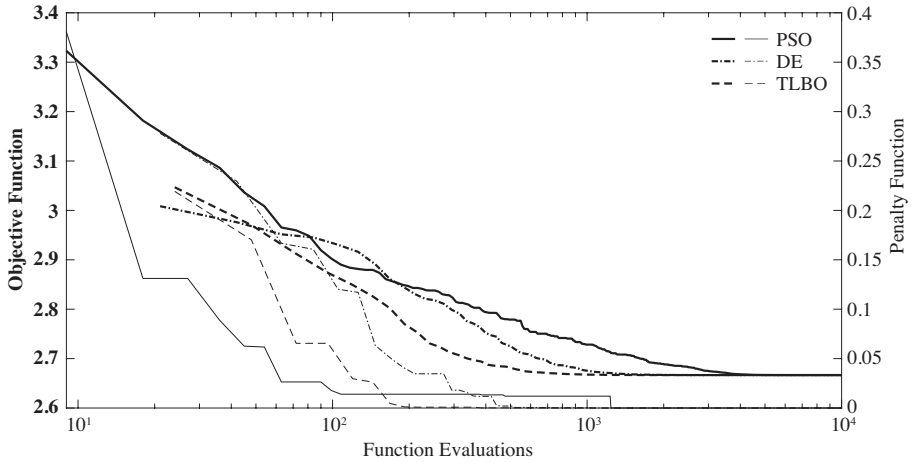


Figure 5.8: Evolution of the mean objective function (—) and penalty function (—) obtained by the Particle Swarm Optimization, Differential Evolution and Teaching-Learning-Based Optimization for the three-bar truss.

5.1.2.4 Square Plate

The square plate results are presented in Table 5.8, where it is observed that the TLBO is the algorithm that returns the best solution. The reported best solution of the DE is close to the one found in the TLBO, but slightly higher while the PSO returns the worst solution out of the three algorithms. The correspondent design variables of each solution are also presented in Table 5.8, as it is observed that the values of each design variable are similar for the three algorithms. These results are better illustrated in Figure 5.10, where the design geometry of the square plate obtained for each algorithm is presented. Moreover, it is observed that the solutions obtained by the PSO, the DE or the TLBO are different from the one presented by Valente *et al.* [Valente *et al.* 2011]. The solution reported by Valente *et al.* presents less area in the top zone of the plate, but more in the middle. However, the authors do not report a numerical solution to compare with those of the PSO, DE and TLBO. Additionally, in Figure 5.11 the deformed shape of the square plate for each solution is presented, where the maximum values of the equivalent stress are located in the exterior border and in the interior hole border, similarly for all solutions.

Regarding the evolution of the mean objective and penalty functions represented in Figure 5.9, the PSO and DE present penalized solutions in the first iterations, while the TLBO is able to find solutions that are not penalized from the first iteration. Looking at the results from 100 function evaluations onwards, the TLBO presents higher objective function values than the other two algorithms up until 500 function evaluations while

Table 5.8: Results of the selected size of the population for the Particle Swarm Optimization, Differential Evolution and Teaching-Learning-Based Optimization for the square plate.

	PSO	DE	TLBO
Best	210209.5512	209846.2641	**209791.7287**
Mean	**212901.9901**	218342.3776	213128.3322
SD	**3410.9547**	11860.40689	3435.5616
Worst	220849.6972	249477.744	**220265.1315**
FE	**936**	996	971
Mean FE	976	977	**919**
SD FE	**19**	36	91
n	$2D$	$2D$	$2D$
G	100	100	50
Design Variables			
x_1	360.2695	364.1433	354.1008
x_2	564.2672	563.6865	560.2531
x_3	572.0206	580.3878	578.9840
x_4	495.5253	492.4464	497.6825
x_5	465.7822	464.2892	470.8945

SD - Standard Deviation FE - Function Evaluations G - Generations

at this point the DE starts stagnating and is not able to converge to better solutions. Furthermore, it is observed that the mean evolution of the TLBO is very similar to the PSO until the end of the optimization process.

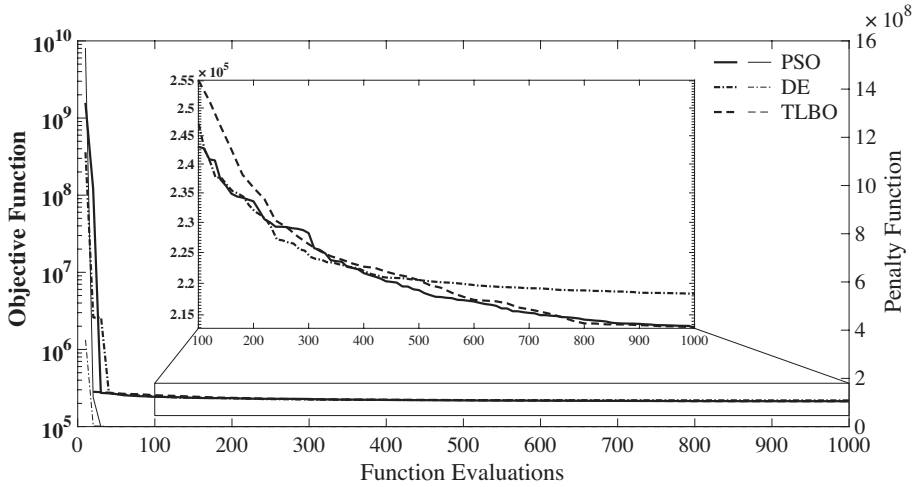


Figure 5.9: Evolution of the mean objective function (—) and penalty function (—) obtained by the Particle Swarm Optimization, Differential Evolution and Teaching-Learning-Based Optimization for the square.

On one hand, the results presented in Table 5.8 show that even though the PSO returns the highest solution, its mean and standard deviation parameters are the lowest of the three algorithms. On the other hand, the DE presents higher values of mean and

standard deviation parameters, indicating that the reported best solution was possibly found only once and that on average the reported solutions are relatively higher. Although TLBO presents a mean result slightly higher than the PSO the worst solution found is comparatively better, thereby demonstrating to be the most reliable algorithm for this application.

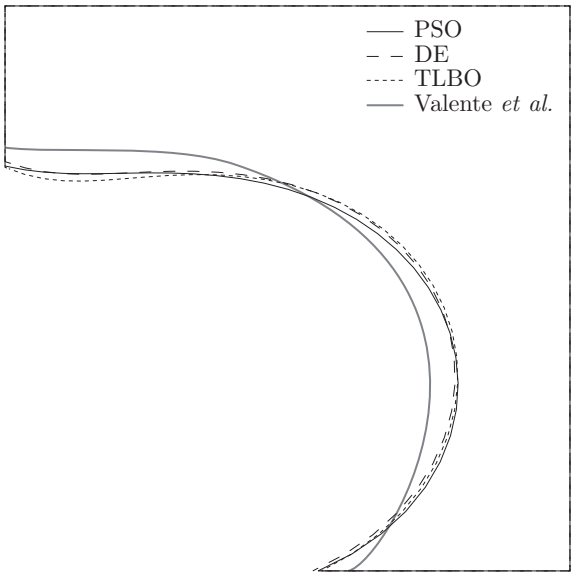


Figure 5.10: Representation of the best solution for the square plate design geometry obtained by the implemented algorithms and reported by Valente *et al.* [Valente *et al.* 2011].

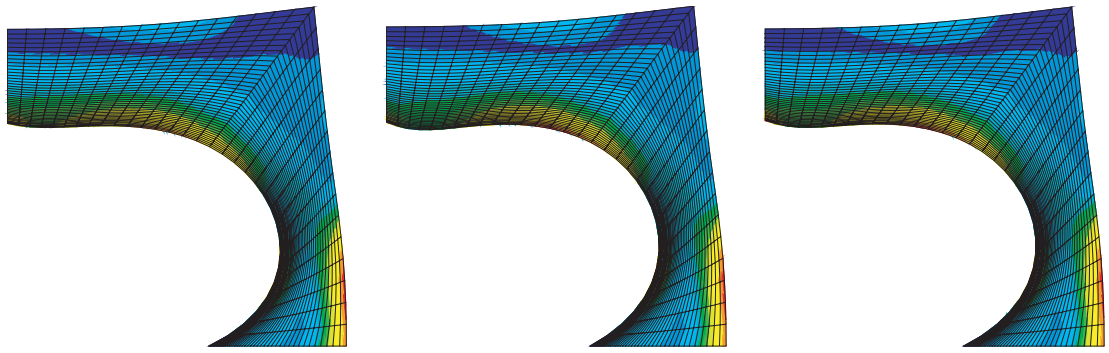


Figure 5.11: Representation of best solution for the square plate deformed geometry obtained by (a) Particle Swarm Optimization, (b) Differential Evolution and (c) Teaching-Learning-Based Optimization.

5.1.2.5 Global Analysis

In Table 5.9 are summarized the results of the best solution, success rate and the least number of function evaluations required to find the best solution obtained by each algorithm for all applications. The comparison between the results of the algorithms best solution,

demonstrated that the DE is the only algorithm that is able to find the global optimum for the composition function, speed reducer and the three-bar truss. Moreover, it is the only algorithm that presents a 100% success rate for the same applications. The TLBO also performs significantly well, presenting the same results as the DE for the speed reducer and the three-bar truss, while outperforming the DE and PSO in the best solution found for the square plate. Consequently, PSO is, in general, the most inefficient algorithm as it only outperforms the DE and TLBO in the mean and standard deviation parameters for the square plate.

From a computational effort perspective, the DE outperforms by a large margin PSO and TLBO for the speed reducer application. For the same application, the DE requires, on average, approximately six times fewer function evaluations than PSO and almost four times less than TLBO. However, for the three-bar truss the TLBO is more efficient than the PSO and DE and results regarding the computational effort for the square plate are not conclusive as for the number of function evaluations required to find the reported best solutions are close to the total number of function evaluations. Overall, PSO demonstrates to require more computational effort than the other two algorithms.

Table 5.9: Summary of the results of best solution, success rate and function evaluations obtained by Particle Swarm Optimization, Differential Evolution and Teaching-Learning-Based Optimization for all aplications.

		PSO	DE	TLBO
Composition Function	Best	5.5382	**0.0000**	2.2724
	SR [%]	0	**100**	0
	FE	-	**28220**	-
Speed Reducer	Best	**2996.348165**	**2996.348165**	**2996.348165**
	SR [%]	85	**100**	**100**
	FE	55771	**8918**	33138
Three-Bar Truss	Best	**2.666667**	**2.666667**	**2.666667**
	SR [%]	**100**	**100**	**100**
	FE	5932	3010	**1789**
Square Plate	Best	210209.5512	209846.2641	**209791.7285**
	SR [%]	-	-	-
	FE	**936**	996	971

SR - Success Rate FE - Function Evaluations

In can be concluded that the most reliable and robust algorithm seems to be the DE, as it performs overall better than the PSO and TLBO. This conclusion might be related to the structure of the DE implementation (cf. Section 3.2), as at every iteration the new solution obtained after the operations of mutation, crossover and selection replaces the current solution and influences the newly generated solutions at the same iteration. The same is also observed in the TLBO implementation at the learner phase (cf. Section 3.3), as the new solution is placed in the current population and is able to influence newly generated solutions. The opposite is observed in PSO as at each iteration all operations are carried out for all solutions in the population, but new solutions only affect the next iteration. Furthermore, the type of operations in the DE implementation differ from those of the PSO and TLBO implementations in relation to the strategies used. Both PSO and TLBO rely on information and statistics of the population while the DE only uses

probabilistic rules and other solutions from the population. The PSO relies on the best position of the population and the best individual solution while the TLBO also relies on the best position of the population as well as the mean result of the population in each design variable. Without knowing the results obtained by the algorithms, it would be possible to assume that the PSO or TLBO could perform better than the DE as its operations are more complex and rely on information of the population. However, as already analyzed that is not the case as the simpler operations in the DE return better results. Between PSO and TLBO, the latter present more complex operations and uses more information of the population, probably explaining why it presented better results than the PSO. Additionally, even though the TLBO is more complex than the other two algorithms, it presents the advantage of not requiring the definition of an operational parameter aside from the size of the population.

5.2 Performance of Computational Processing

In this section, the performance of computational processing of algorithms and programming languages is analyzed. For both analyses, sequential and parallel, only one run is performed for each algorithm, rather than the twenty runs performed in the previous section. However, as independent runs might present variations in the measured parameters, three independent simulations are performed and the mean results presented. Additionally, the analyses were carried out using the selected size of the population (cf. Section 5.1.1.5) for each algorithm and application. In each run, both for the sequential and parallel analyses, the measured parameters are:

- Evaluation Time: time range the algorithm is computing the required operations to evaluate the objective function, in particular, the computation of constraints, penalty function or specific operations depending on the application (e.g. computation of fourth-degree polynomial function in the square plate design problem);
- Additional Time: time range the algorithm is performing operations aside from the evaluation of the objective function, such as the position and velocity updating in the PSO, mutation, crossover and selection in the DE or the teacher and learner phase in the TLBO;
- Other Time: time range of background operations, post-processing operations or initialization and termination of processes in the parallel processing implementation;
- Total Time: time range of the entire simulation.

The described parameters are measured using the elapsed real time or wall-clock time of the computer with a precision of milliseconds. Moreover, it was attempted to maintain the same conditions in the computer throughout all simulations.

Section 5.2.1 refers to the analysis of the sequential processing implementations, where PSO, DE and TLBO algorithms are analyzed and implemented in Python, MATLAB, Java and C++. Section 5.2.2 refers to the analysis of the parallel processing implementations, where PSO is the only algorithm analyzed and implemented in Python, Java and MATLAB. Furthermore, the parallel processing analysis is performed using one to eight processes.

5.2.1 Performance in Sequential Processing

5.2.1.1 Composition Function

In terms of numerical operations of the evaluation of the objective function, the composition function is particularly more complex than the other applications. This complexity is related to the several operations required to build the composition function, as these are mainly based on vectors and matrices, in particular, ten matrices of size 10×10.

Following these considerations, the sequential processing results of the measured times for the composition function are presented in Figure 5.12, where for the evaluation time it is observed that the three algorithms present similar results through all programming languages. Nevertheless, these results are expected, as each algorithm evaluates the objective function exactly the same number of times. Comparing the results of the programming languages for the three algorithms, Python is the one that takes more time evaluating the objective function, while Java and C++ outperform Python and MATLAB by a large difference. Between Python and MATLAB, differences might relate to the fact that MATLAB is a matrix-enhanced programming language, providing operations to easily manipulate vectors and matrices. On the other hand, the basic operations of Python do not allow for an easy manipulation of matrices without using for loops or a package (e.g. NumPy[2]). Moreover, even though the results of evaluation time for C++ and Java are much better than those of Python and MATLAB, its implementation is more complex as both, C++ and Java, do not present standard features do deal with matrices while advanced packages were not used. Thereby, for this specific application, it is important to consider the trade-off between computational time and development time, as the cost of development in Python and MATLAB is lower than in C++ and Java. Although the computational time in Python and MATLAB is higher, it might not be worth in terms of development compared to C++ and Java.

Further observing the results of additional time, in the case of Python and MATLAB the differences between algorithms are significant, as the TLBO is the algorithm that takes more time performing additional operations, followed by the DE and lastly the PSO. The measured additional time for Java and C++ is very small when compared to the other programming languages and differences between the two are almost insignificant. Results of the other time, are almost insignificant compared to the evaluation and additional time. Results of the total time are relatively proportional to those of the evaluation time while Python demonstrates to be the programming language that, overall, requires more computational time, followed by MATLAB, Java and C++, in that order. This observation is well illustrated in Figure 5.13, where the fractions of total time are represented. For all situations, the fraction of the evaluation time is above 0.6, as the additional time represents the great majority of the remaining total time. For the three algorithms, MATLAB is the programming language in which the fraction of the evaluation time has less impact, as the additional time represents a bigger fraction when compared to the other programming languages. The fact that MATLAB presents higher values of additional time compared to the other programming languages and the higher impact of this parameter in the total time might be related to the observations made above of the language matrix capabilities as well as the algorithms being implemented using an object-oriented structure (cf. Section 4.3). The object-oriented structure might be directly related to the worst results of

[2]NumPy (`http://www.numpy.org`) is a package for scientific computing with Python.

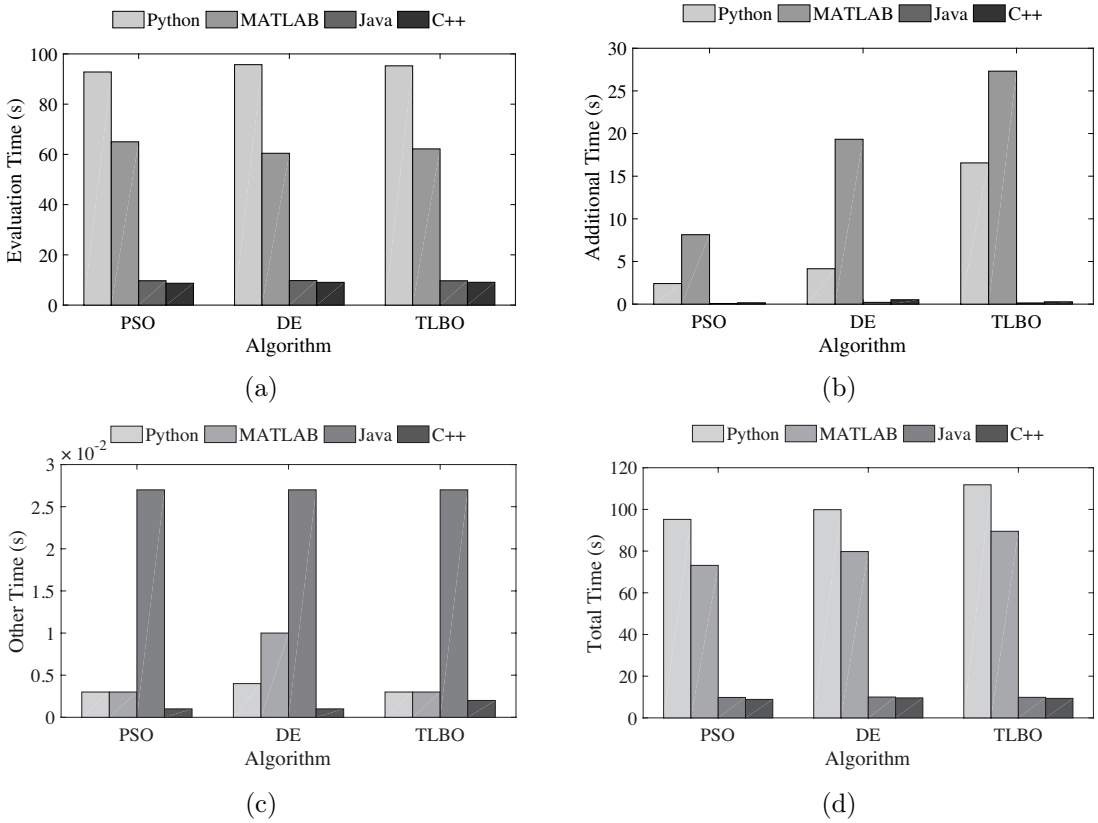


Figure 5.12: Sequential processing results of (a) evaluation time, (b) additional time, (c) other time and (d) total time for the composition function.

MATLAB regarding the evaluation time, as a vector-based structure would possibly be favorable for MATLAB. The total time for Java and C++ is almost entirely represented by the evaluation time, as the additional time stands as a small fraction. The other time represents for all situations an insignificant fraction of the total time and is barely noticed. Comparing the fraction of the additional time between algorithms is observed that the PSO is the algorithm where this parameter has less impact. The additional time of the TLBO demonstrates to have more impact of the three algorithms in Python and MATLAB. However, in Java and C++ the additional time of the DE presents slightly more impact than in the TLBO.

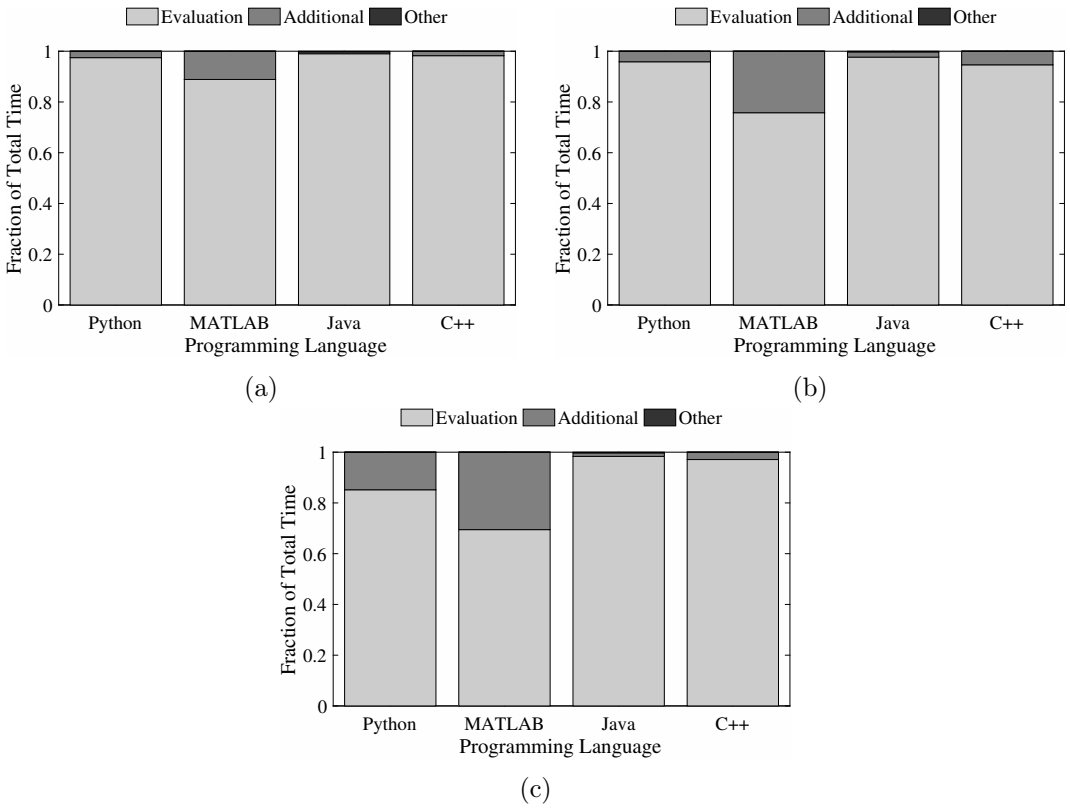


Figure 5.13: Sequential processing results of fractions of total time obtained by the (a) Particle Swarm Optimization, (b) Differential Evolution and (c) Teaching-Learning-Based Optimization for the composition function.

Overall, C++ is the fastest programming language, with Python being the slowest, for the three algorithms. As for the fraction of evaluation time, MATLAB is the programming language where it has less impact and Java the most.

5.2.1.2 Speed Reducer

The speed reducer design problem demonstrates to require lower computational time than the composition function, as the results of the evaluation time presented in Figure 5.14 demonstrate. For the three algorithms, MATLAB is the programming language that takes more time evaluating the objective function and Python being the second slowest. Java and C++ demonstrate to be faster than the other two programming languages, with C++

slightly outperforming Java. In general, the evaluation time is very similar between algorithms, as the number of function evaluations is the same between them. Regarding the results of the additional time, Java and C++ are faster than the other two programming languages, presenting additional times lower than 1 second. On the other hand, MATLAB is the slowest programming language computing the algorithm operations, taking more than twice the time of Python. It is particularly interesting to note that in the case of MATLAB, the DE requires more computational effort than the TLBO, in opposition to Python, where the TLBO takes more time computing the operations than the DE. Measured results for the other time present once again very small values and the differences between the programming languages are almost insignificant (order of milliseconds).

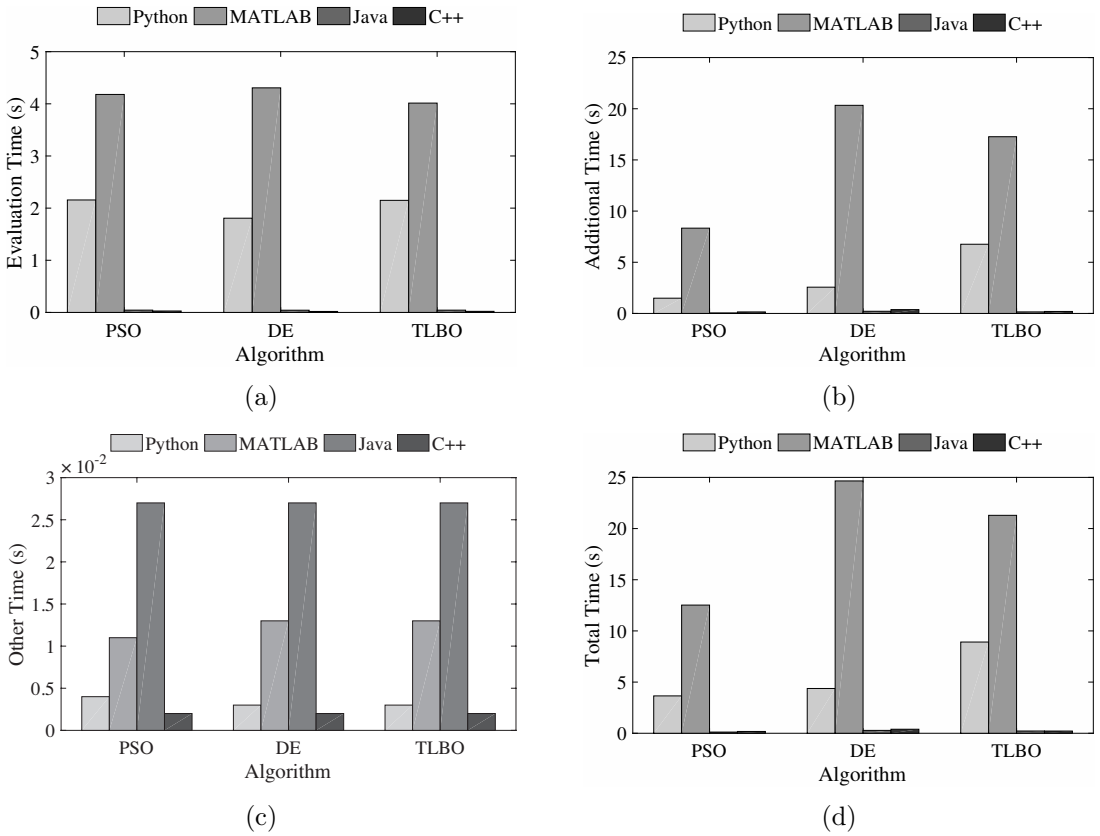


Figure 5.14: Sequential processing results of (a) evaluation time, (b) additional time, (c) other time and (d) total time for the speed reducer.

The total time for the speed reducer implementations is strongly influenced by the additional time, as results are almost identical. MATLAB is the programming language that, in overall, requires more computational effort, as Python is more than two times faster, while Java and C++ are at least twelve times faster. Results for Java and C++ are similar, with insignificant differences between the two. Python presents intermediate results. However, these are closer to Java and C++ than to MATLAB. The fact that Java and C++ are faster than Python and MATLAB is directly related to the fact that first two programming languages are compiled and statically typed, in opposition to Python and MATLAB that are interpreted and dynamically typed. Programming languages that are compiled and statically typed benefit from having the source code (code written by

the user) translated to machine code while the variables' type is checked before execution allowing for machine code optimization. On the other hand, in interpreted and dynamic programming languages the source code is not previously compiled and is interpreted during execution, meaning the variables' type are checked during execution.

Analyzing the fractions of total time presented in Figure 5.15 for the PSO, DE and TLBO, it is observed a big difference when compared to the results for the composition function. Except for the implementation of the PSO in Python, the bigger fraction of the total time corresponds to the additional time. In all C++ implementations, the fraction of the evaluation time is below 0.2 while in MATLAB and Java is below 0.4. The additional time in Python demonstrates to have less impact than in other programming languages. Overall, the impact of the evaluation of the objective function is not significant compared to the impact of the additional time in the total time. This observation might anticipate that a possible boost in the performance of the evaluation time might not improve significantly the results of total time.

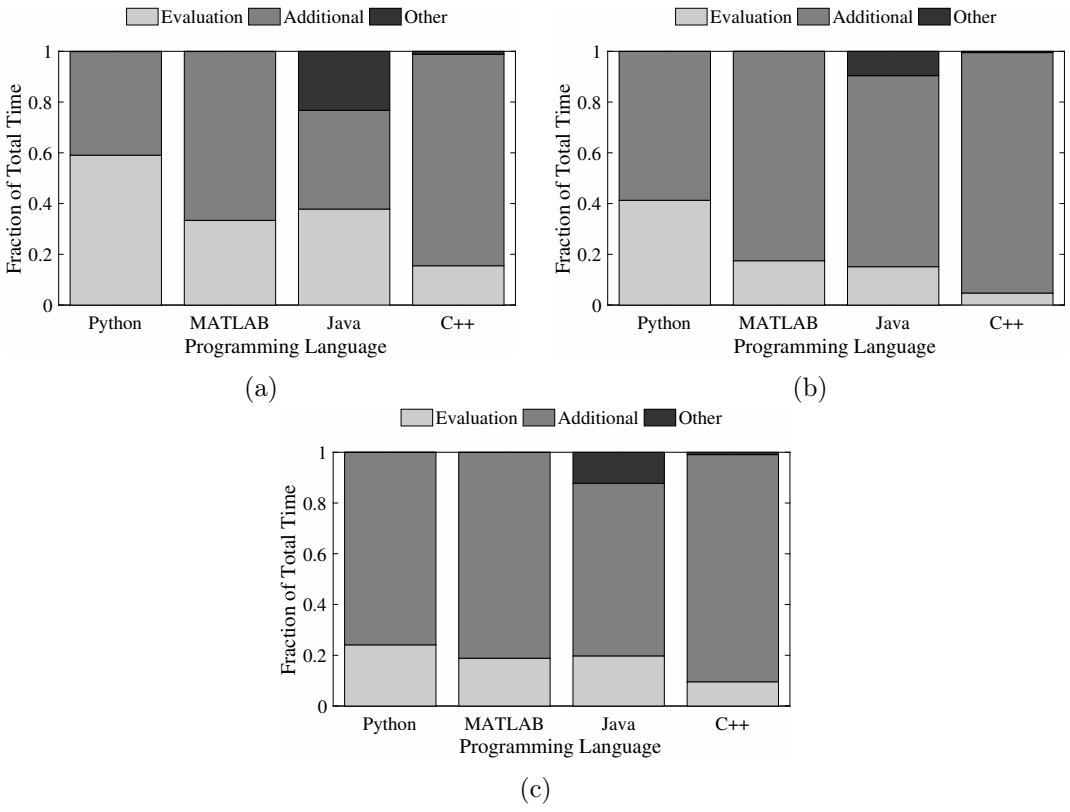


Figure 5.15: Sequential processing results of the fractions of total time obtained by (a) Particle Swarm Optimization, (b) Differential Evolution and (c) Teaching-Learning-Based Optimization for the speed reducer.

5.2.1.3 Three-Bar Truss

The three-bar truss design problem differs from the two previous applications, as it requires the use of an external program. Therefore, a fraction of the computational time is independent of the programming languages in which it is implemented. The results of the

measured times are presented in Figure 5.16, where the evaluation time is in the order of ten thousand seconds. As expected, the evaluation time for the same programming language is very similar between algorithms, presenting insignificant differences. Observing the results of evaluation time, it is observed that MATLAB stands out as the evaluation time is comparatively superior to other programming languages. In opposition to the two previous applications, Java presents higher evaluation time when compared to Python, where results are very similar to C++. Even though this application relies on an external program, the evaluation time is significantly different between programming languages. Moreover, the fact that Python presents better results than Java would not be expected from the analysis of the previous applications. Major differences in the implementation of this application compared to the composition function and the speed reducer are the external program used to compute the nodes displacement and the use of files to write new input and read output. This way the results might indicate that the impact in the evaluation time of calling the external program is not large enough, allowing differences between programming languages to be observed in the other operations. The other significant operations are related to the reading and writing to files, which might be the cause of the observed differences. However, to have certainties regarding this analysis it would be best to analyze with more detail the operations involved in the evaluation of the objective function.

The results of the additional time are consistent with those in the composition function and the speed reducer, as variations are related to the number of function evaluations and the size of the population in each algorithm. In the case of the other time results, these represent, once again, small values compared to the other parameters while in some situations, for Python and C++, the measured other time is zero. For this reason, the other time parameter is not of much interest in this analysis. Lastly, the results of the total time are even more influenced by the evaluation time when compared to the previous applications, as the fraction of the evaluation time is very high for all implementations. The impact of the evaluation time in the total time is overwhelming, indicating that further improvements in the evaluation of the objective function might make a big difference. Comparing the differences between programming languages in the total time, MATLAB clearly takes more time computing, as it is, approximately, more than three times slower than Java and more than thirty times slower than Python and C++. Java demonstrates to be, approximately, seven times slower than Python and C++ in this application. Finally, C++ stands out as the fastest programming language, being approximately two times faster than Python.

5.2.1.4 Square Plate

Analogously to the three-bar truss, the implementation of the square plate relies on an external program and requires operations with files. Consequently, the results for the square plate demonstrate even more the impact of an external program to calculate the objective function, as the measured evaluation times (cf. Figure 5.17) are very high.

Furthermore, the results of the evaluation time are very similar for all algorithms and programming languages. This observation is an indicator that the weight of running the external program is much bigger than other operations in the evaluation of the objective function, thereby showing insignificant differences in the evaluation time between programming languages. In spite of these observed similarities, MATLAB stands out for being

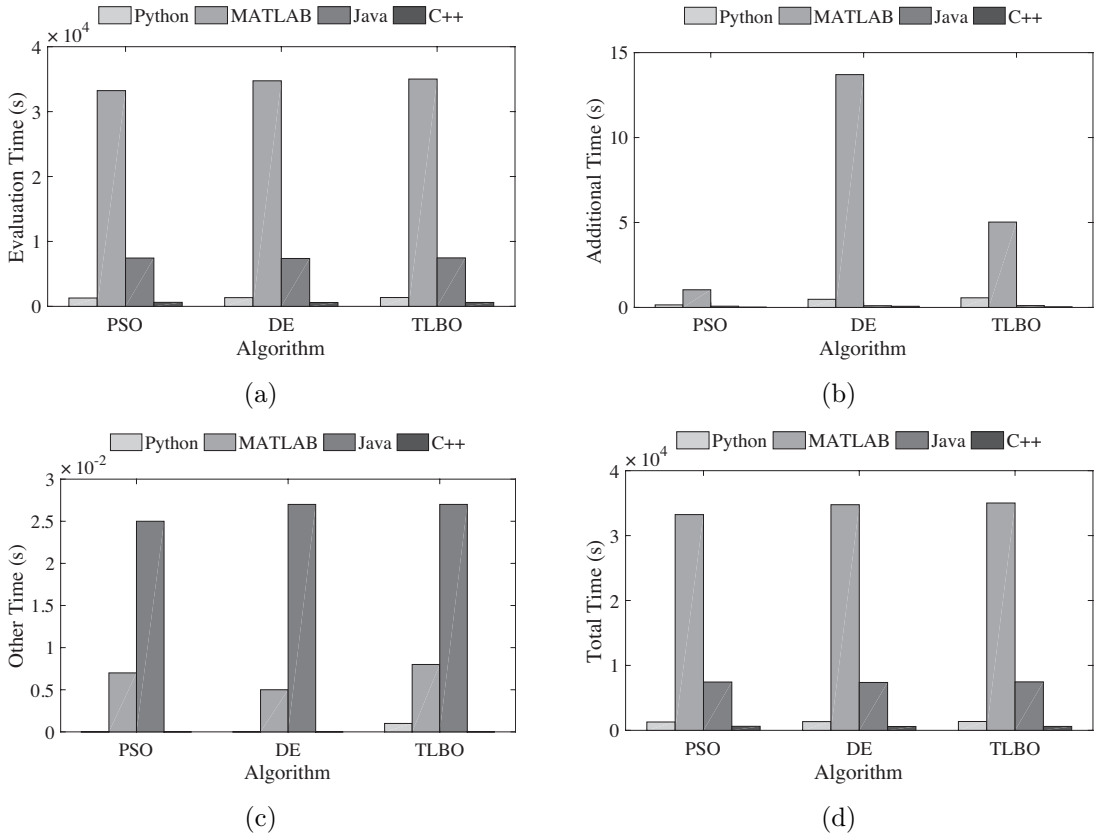


Figure 5.16: Sequential processing results of (a) evaluation time, (b) additional time, (c) other time and (d) total time for the three-bar truss.

slightly slower than the other three programming languages. The results of additional time are significantly low, as compared to previous applications the number of function evaluations is lower. The results of other time are, once again, insignificant compared to the evaluation time. Consequently, for all situations, the measured total time is almost identical to the evaluation time. Furthermore, as a consequence of the high values registered in evaluation time, the impact additional and other time have on the total time is insignificant for all algorithms and programming languages.

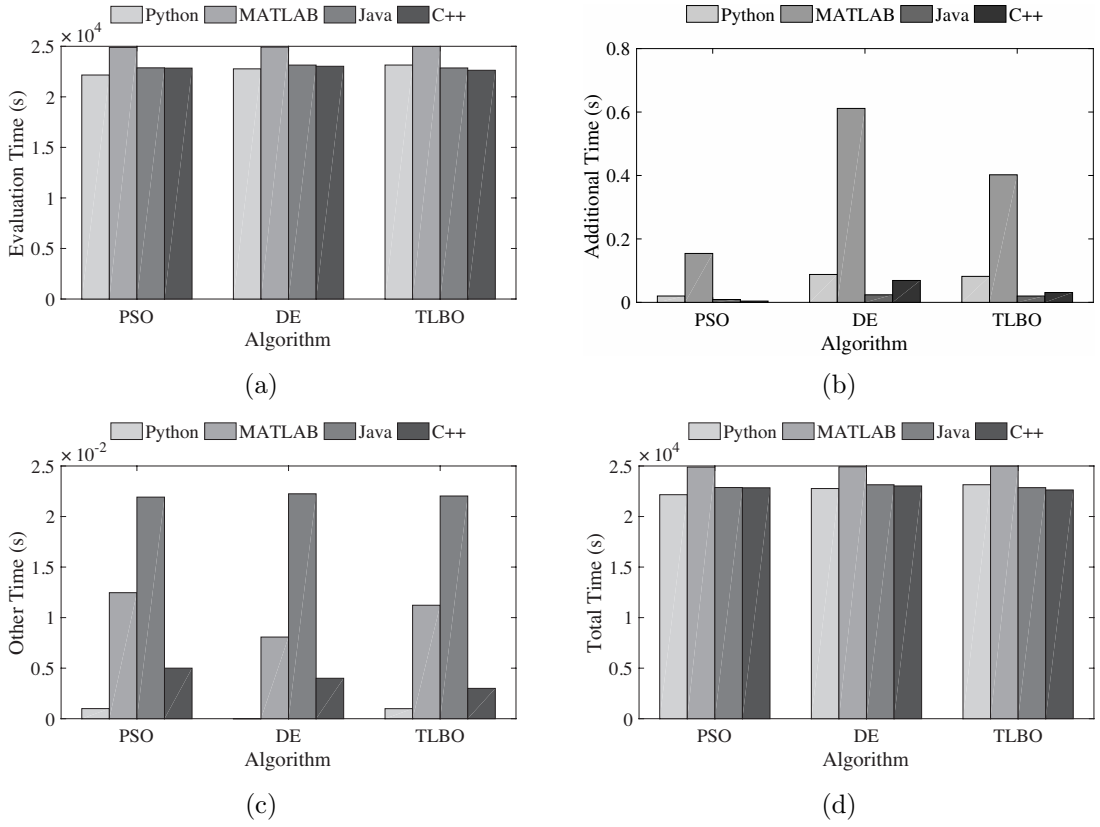


Figure 5.17: Sequential processing results of (a) evaluation time, (b) additional time, (c) other time and (d) total time for the square plate.

These results might suggest that when the application is computationally heavy, specifically when it requires the use of an external program with great impact on the evaluation time, the selection of the programming language in terms of computational capabilities is less relevant while the cost of developing the application becomes a more important parameter. From this perspective, it can be said that C++ and Java present higher costs of development comparatively to Python and MATLAB. For example, the computation of the fourth-degree polynomial function requires two lines of code for Python and MATLAB, using methods available in NumPy and in MATLAB it was achieved using standard functions while for Java and C++ it was necessary to download and use external libraries.

5.2.2 Performance of the PSO in Parallel Processing

5.2.2.1 Composition Function

The sequential processing results for the composition function anticipated that a parallel processing implementation could potentially benefit the performance, as the evaluation time demonstrated to be the larger fraction of the total time. Although this observation is true for some situations, it does not stand for others, as results in Figure 5.18 and 5.19 demonstrate.

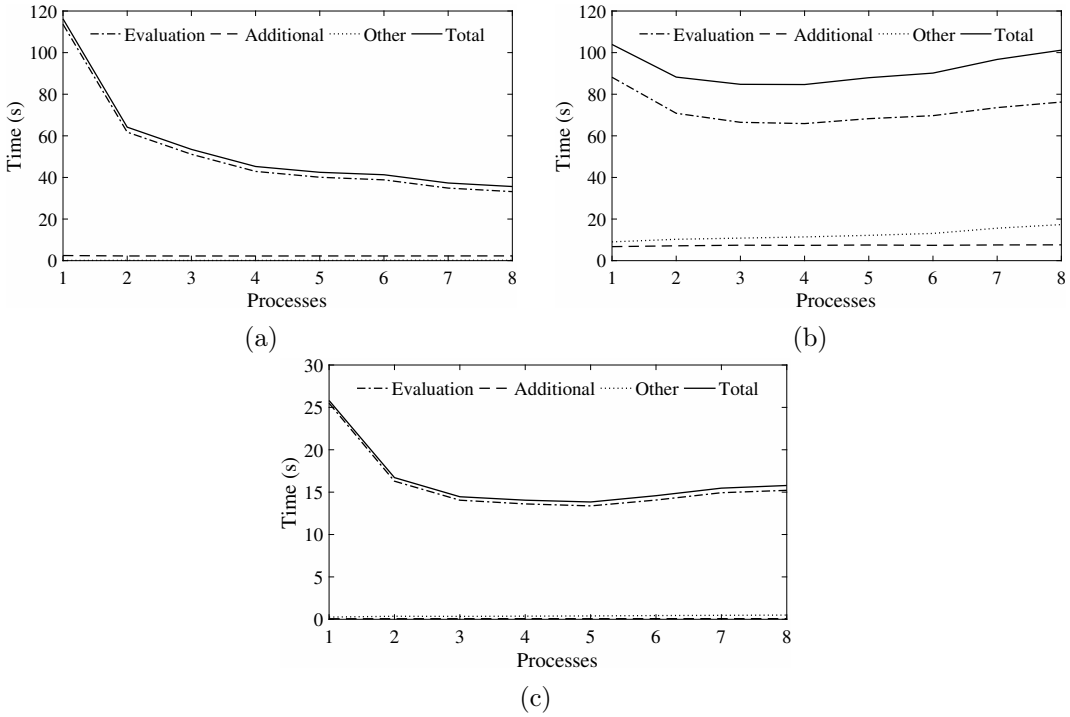


Figure 5.18: Parallel processing results of measured parameters obtained by (a) Python, (b) MATLAB and (c) Java for the composition function.

By first analyzing the measured times for Python, it is observed that the evaluation and total times are almost identical through the number of processes, as the additional and other times do not present a significant weight on the total time, but as the number of processes increases, the weight of additional time in the total time slightly increases. It is also demonstrated that an increase in the number of processes significantly benefits the performance of the evaluation time, consequently reducing the total time. This benefit is more evident from an increase between one and four processes, while onwards the measured evaluation and total times present low improvements. While the evaluation time improves, derived from the parallel implementation, the additional and other times are implemented in sequential processing, presenting similar results through the number of processes. The amount of improvement is explicitly given by the speedup, where results of total time for two processes are close to the linear speedup of 2. However, as the number of processes increases the speedup of total time distances itself from the linear values and it is never higher than 4. Although the parallel processing implementation benefits the overall performance compared to results of sequential processing, the number of benefits is lim-

ited. In this context, it is relevant to consider that the implementation was achieved using available methods in which the user only has control over the number of processes used while the control over communication and distribution of data by each process is performed automatically by the used methods. In this scenario the achieved boost in performance is relatively good compared to the cost of implementation.

Results for MATLAB differ significantly from the ones in Python, as the additional time plays a slightly bigger role in the total time, although it demonstrates to be similar for all the number of processes. It is of interest to observe that values of the other time are relatively significant, as it represents a bigger fraction of total time than the additional time. Furthermore, it is observed that the other time increases with the increase in the number of processes, thereby not benefiting the performance of the total time. Even though the other time impacts the overall performance, the tendency of total time is significantly influenced by the evaluation time, as the parallel performance slightly demonstrates to improve for two and three processes. From four processes onwards the evaluation time is worse while for eight processes it is almost similar to one process. Analyzing the results of speedup and efficiency it becomes clear that a parallel processing implementation results in a poor performance compared to the results of sequential processing, as the speedup and efficiency of evaluation and total time are similar to results of additional and other time.

Java presents a similar tendency as Python for the measured parameters. Results of additional and other time show values almost constant through the number of processes while the total time is mostly influenced by the evaluation time. Using two to four processes results show an improvement in the evaluation time comparatively to one process. However, from two processes onwards these improvements are low or even non-existent and after four processes the evaluation time stabilizes close to 15 seconds. Regarding the fractions of total time, small changes are observed for all processes, whereas the weight of other time slightly increases, but does not represent a significant change. Finally, by observing the results of speedup and efficiency it is notorious that the benefits of increasing the number of processes are limited, with low speedups and values of efficiency moving towards zero.

Overall, the improvements obtained with the parallelization of the evaluation of the objective function are affected by the sequential tasks, as the values for speedup of the total time are lower than those of the evaluation time. When compared to MATLAB and Java an increase in the number of processes in Python benefits the overall performance, although demonstrating to be limited.

5.2.2.2 Speed Reducer

The sequential processing implementation in the speed reducer demonstrated the evaluation time to be lower than the additional time, possibly indicating that a parallel processing implementation would not bring significant improvements. Observing the results of the parallel processing implementations presented in Figures 5.20 and 5.21, it is demonstrated that the parallel processing does not benefit the overall performance in all programming languages.

Results for Python show little improvement in the evaluation and total time, as for only two processes a decrease in the reported values is observed compared to one process. Moreover, from two to eight processes the evaluation time presents a tendency to increase, in opposition to what would be desired. The evolution of total time is affected by the

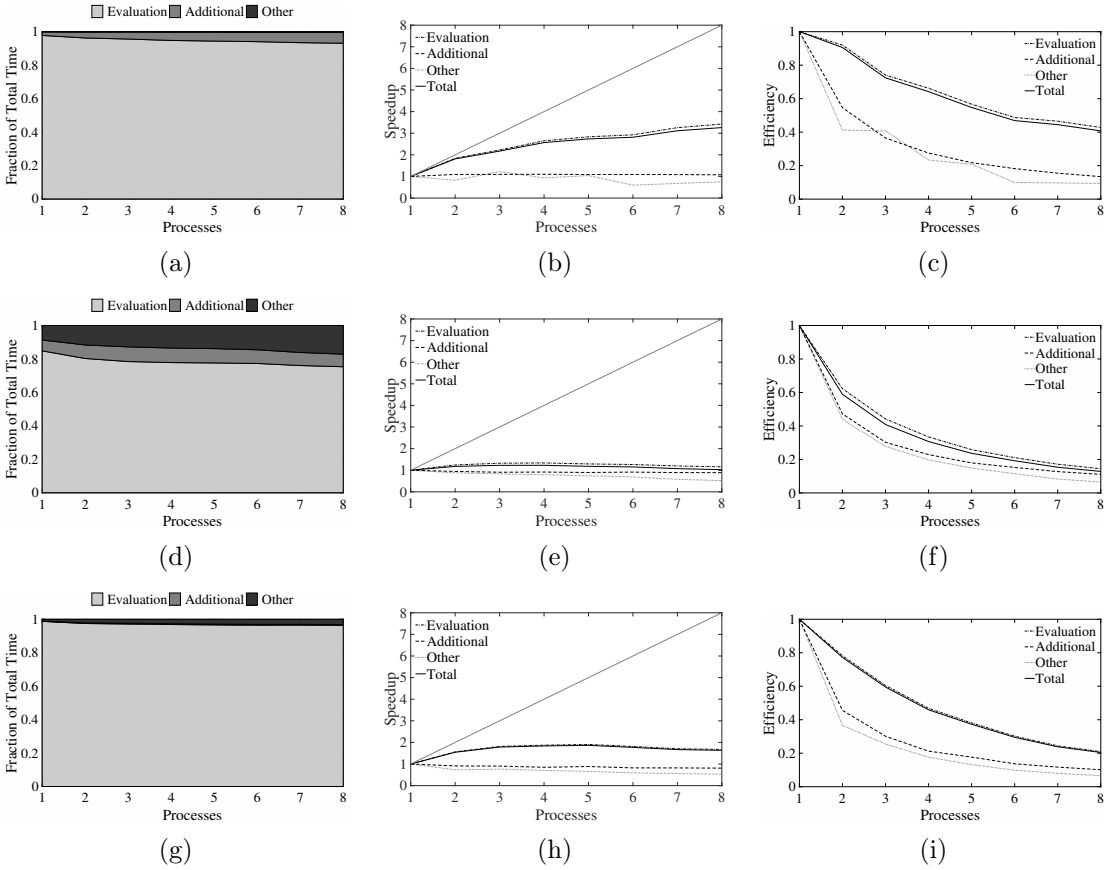


Figure 5.19: Parallel processing results of the fractions of total time, speedup and efficiency obtained by (a), (b) and (c) Python, (d), (e) and (f) MATLAB, and (g), (h) and (i) Java for the composition function.

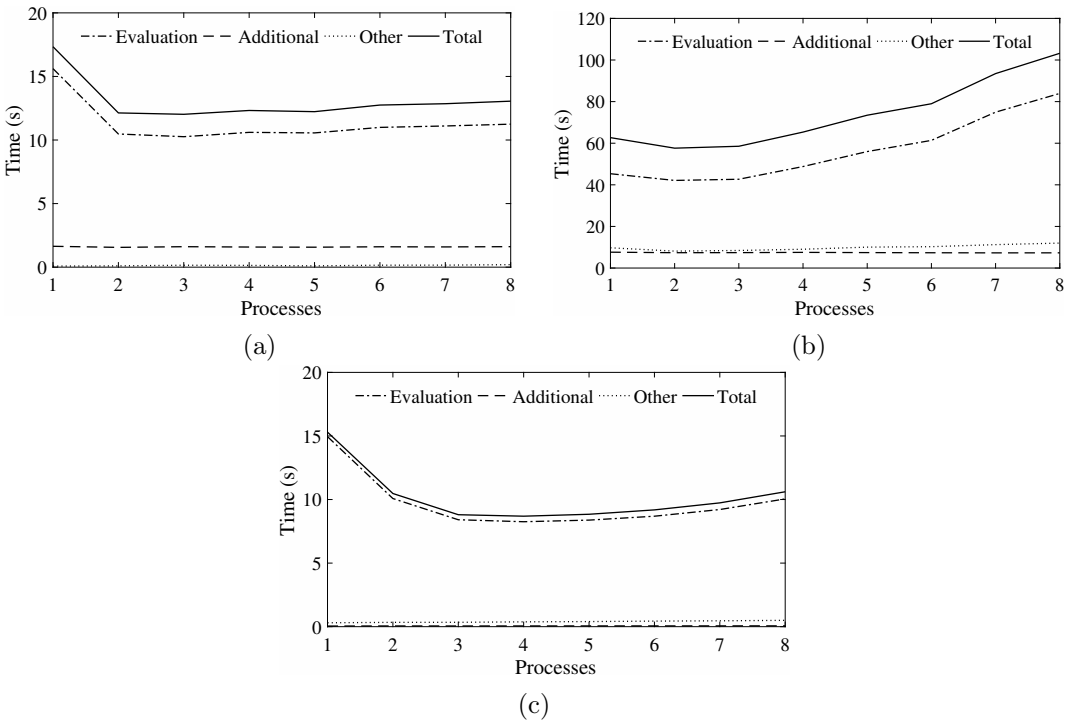


Figure 5.20: Parallel processing results of measured parameters obtained by (a) Python, (b) MATLAB and (c) Java for the speed reducer.

additional time, which is processed sequentially and represents a great fraction of the total time. Furthermore, the results of speedup and efficiency illustrate that the parallel processing implementation does present benefits. Speedups of additional and total time are below 2 for all the number of processes and efficiency results tend to zero with the increase in the number of processes. Comparatively to the sequential processing results, the evaluation time increased almost six times, probably meaning that communication operations between processes take more time than the evaluation of the objective function.

The results for MATLAB are even worse than those reported for Python, as the evaluation time shows a slight improvement for two processes, but afterwards tends to largely increase. Similarly to the results of the composition function, the other time tends to increase with the number of processes, demonstrating to have more impact on the total time than the additional time. Comparatively to Python and Java, MATLAB demonstrates to be the programming language where the total time is more influenced by the additional and other time. Results of speedup and efficiency confirm that this application does not benefit from the parallel processing, as speedups of evaluation and total time tend to values below 1 and efficiency close to zero.

Analogous to the results reported for Python and MATLAB, results for Java demonstrate that this implementation does not benefit the performance. For only two and three processes, the evaluation time decreases relatively to one process, as afterwards the tendency is to increase. However, the fraction of evaluation time decreases with the number of processes while the other time presents an increasing impact, even greater than the additional time. Additionally, results of speedup for the evaluation and total time are lower than 2 for all the number of processes, while efficiency for these parameters demonstrates

to continuously decrease.

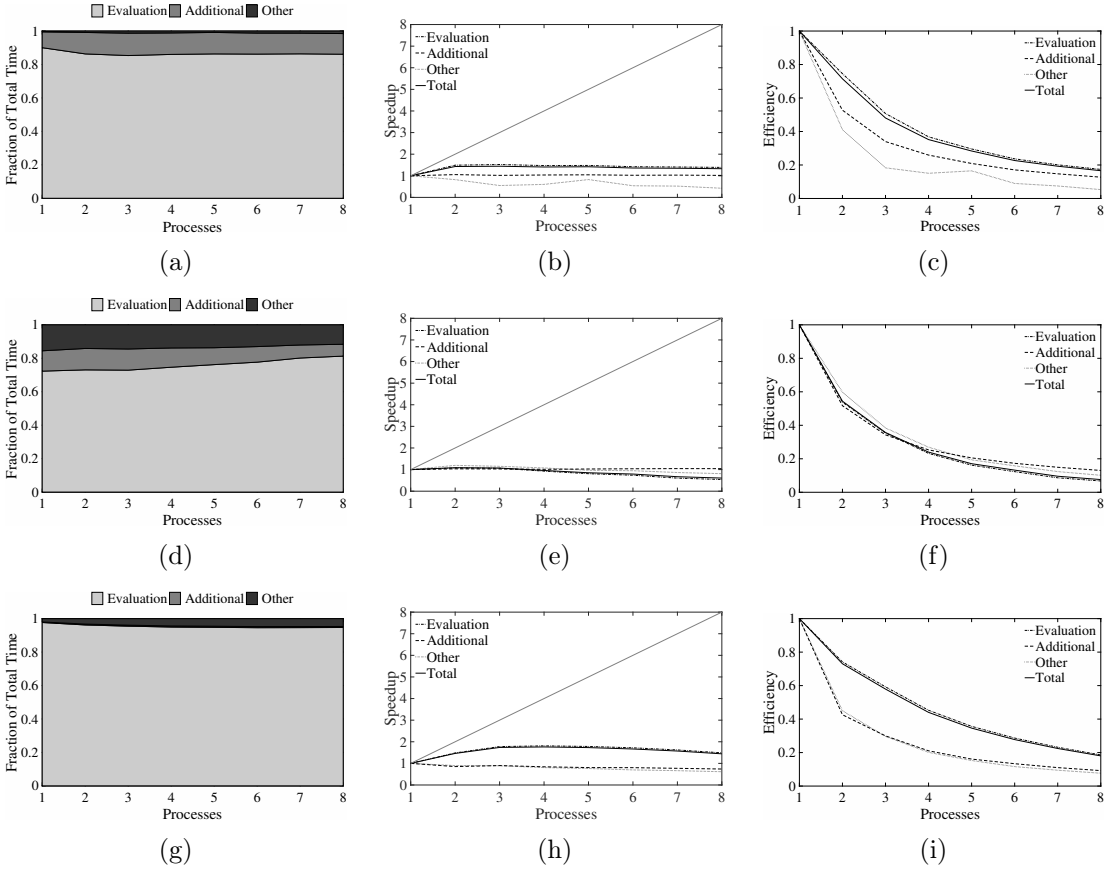


Figure 5.21: Parallel processing results of the fractions of total time, speedup and efficiency obtained by (a), (b) and (c) Python, (d), (e) and (f) MATLAB, and (g), (h) and (i) Java for the speed reducer.

Overall, results for the speed reducer are not satisfying, as it could be anticipated from the analysis of the sequential processing. In opposition to the other applications, the speed reducer problem demonstrated that the weight in total time of the evaluation time is lower than the additional time. This observation might represent an indicator when considering the implementation of parallel processing in these type of application.

5.2.2.3 Three-Bar Truss

The three-bar truss design problem demonstrated in the results of sequential processing to be more computationally demanding than the composition function and the speed reducer. In the sequential processing results, the evaluation time presented as the bigger fraction of the total time, representing a good indicator of the benefits of a parallel processing implementation. Results relative to this application are presented in Figure 5.22 and 5.23.

Analyzing the evolution of the measured times for Python, results appear to be satisfying, as the total time is almost similar to the evaluation time while the initial tendency, with the increase in the number of processes, is to benefit the performance. However, from five processes onwards no improvements in the evaluation and total time are observed, as

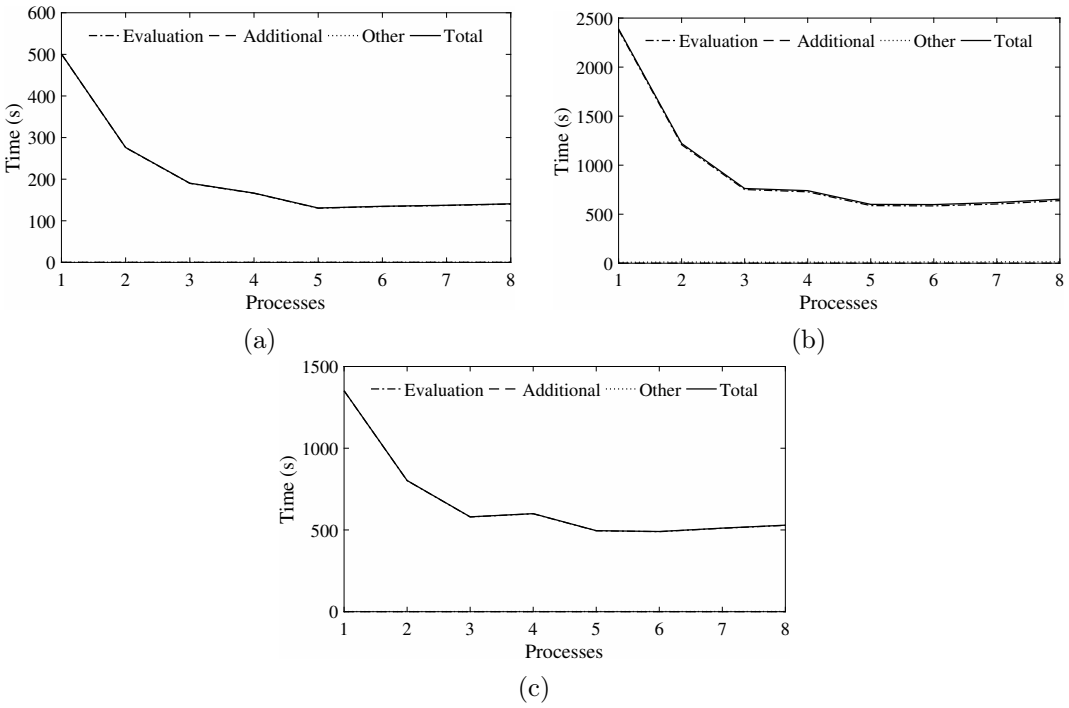


Figure 5.22: Parallel processing results of measured parameters obtained by (a) Python, (b) MATLAB and (c) Java for the three-bar truss.

the measured times present an increase. Observing the evolution of the fractions of total time, it is evident that the evaluation time almost entirely represents the whole time, even though a slight decrease is observed with the increase in the number of processes. The correspondent results of speedup and efficiency illustrate well these results, as until five processes the speedup tendency is not far distanced from the linear speedup and efficiency values are not far below 1. However, from there onwards the tendency of speedup is to slightly decrease and efficiency presents an abrupt decrease.

Similar results are observed for MATLAB, as the overall performance seems to improve until five processes, but afterwards, the same pattern as in Python is observed. However, significant differences compared to Python are shown in the speedup, where from one to two processes the speedup appears to be almost linear, while from two to three processes results demonstrate higher speedup than the linear value. Nevertheless, these good results do not stand onwards as the tendency in speedup is to stagnate and decrease. Additionally, results of efficiency are of interest, as with three processes the parallel processing is more efficient than what is expected in theory.

Java results are in sync with the ones of Python and MATLAB, as the increase in the number of processes benefits the computational performance. Even though Java presents a slight increase in the evaluation time for four processes, the same is not observed for five and six processes, which demonstrate to benefit the computational time. Speedup results present a peak value of approximately three for six processes but demonstrate that, in general, the parallel processing implementation in Java is not as much efficient as it is in Python and MATLAB.

Comparing the results for the three programming languages, it is interesting to ob-

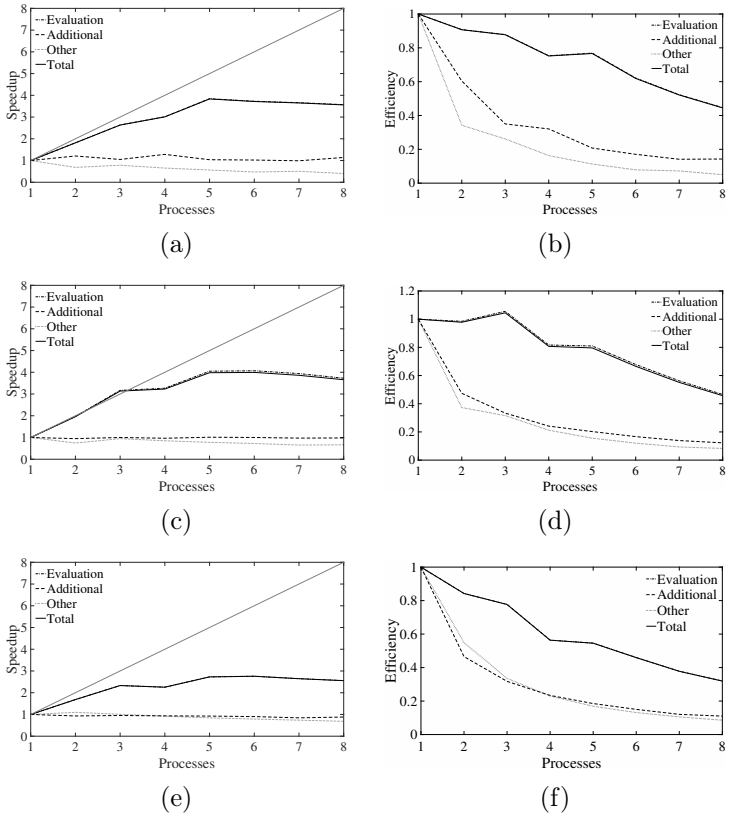


Figure 5.23: Parallel processing results of the speedup and efficiency obtained by (a) and (b) Python, (c) and (d) MATLAB, (e) and (f) Java for the three-bar truss.

serve that all of them present better results with one process in the parallel processing implementation than in the sequential processing implementation. To note that both implementations only differ in the calling of the subroutine to evaluate the objective function and in the folder, the process needs to access to call the external program. Aside from these differences, both implementations are similar, thereby a logical explanation for the decrease in the evaluation time is not provided. In this context, a detailed analysis of the operations involved in the evaluation of the objective function would be interesting to find out the source of the observed differences.

5.2.2.4 Square Plate

This application stands out from the other applications because the results for the sequential processing demonstrate great similarities between programming languages. Additionally, values of the evaluation time reported for the sequential processing are much higher than for other applications and significantly higher than the additional time. Thereby, results of the parallel processing implementations demonstrate to be very satisfying as it is observed in Figure 5.24 and 5.25. It is of interest to mention that, in the presented results, the evolution of the evaluation time is not easily identified, as its values are overlapped by those of total time. According to this observation, the values of the additional and other time are insignificant in the total time.

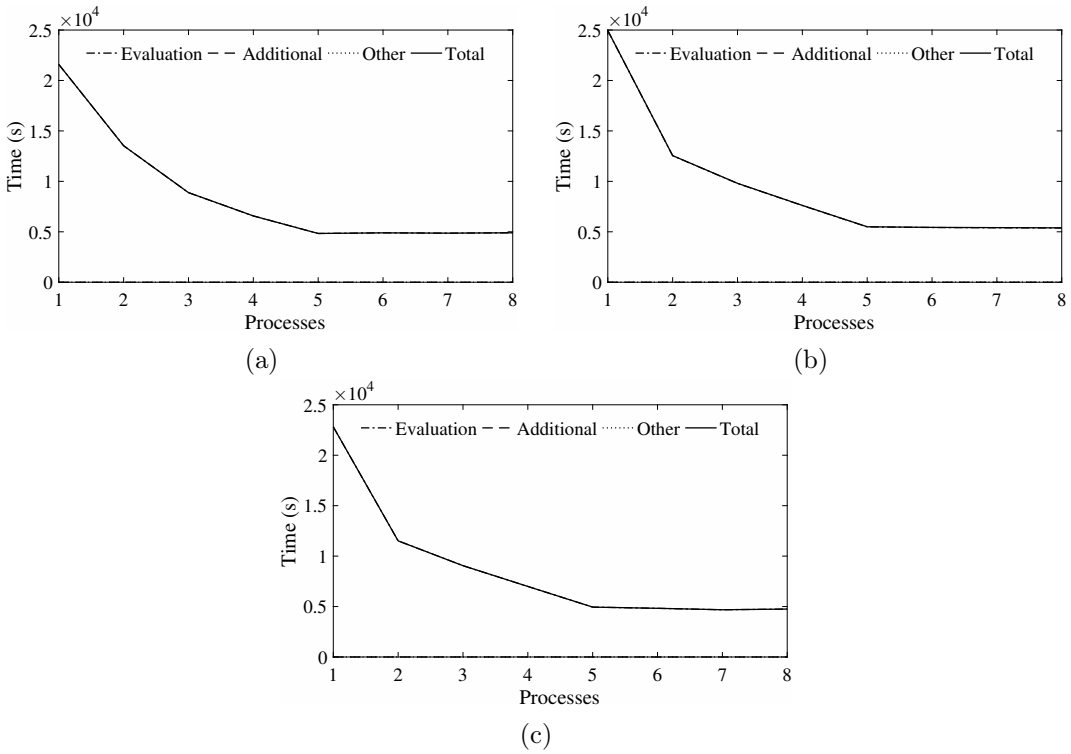


Figure 5.24: Parallel processing results of measured parameters obtained by (a) Python, (b) MATLAB and (c) Java for the square plate.

The evolution of the measured parameters for Python demonstrates a large decrease in the evaluation and total time as the number of processes increase. This tendency prevails

until five processes, as afterwards the total time stagnates and little or no improvements are observed similarly to what was observed for the three-bar truss. The evolution of speedup for the total time is at first below, but close to the linear speedup while presenting a tendency to increase until five processes, where the value of speedup is almost 5. Although the results for a higher number of processes do not demonstrate improvements in performance, the obtained gains for two to five processes are very satisfying relatively to the effort of its implementation.

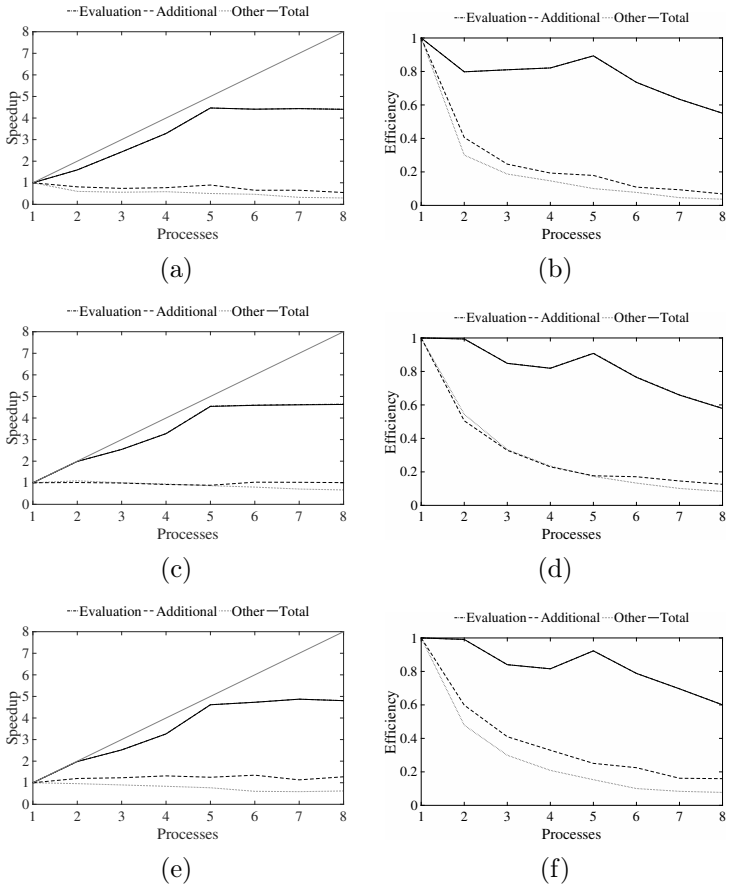


Figure 5.25: Parallel processing results of the speedup and efficiency obtained by (a) and (b) Python, (c) and (d) MATLAB, (e) and (f) Java for the square plate.

The tendency of measured times for MATLAB and Java are very similar, as the following observations apply to both programming languages. Both present a big decrease in the total time from one to two processes while the corresponding speedup of total time is almost linear and efficiency is slightly below 1. From there onwards the tendency of total time to decrease continues, but with less emphasis than before, as values of speedup are farther relative to linear speedup. For five processes and similarly to Python, it is observed a peak in efficiency while the speedup presents a slight approximation to the linear speedup. Once again, after five processes the computational total time stagnates and improvements are of little significance.

Overall, the three programming languages present similar results, either relative to the total time or obtained speedups. However, Java stands as the programming language

with the biggest peak in speedup and Python the lowest. Nevertheless, the cost of the parallel processing implementation in Python and MATLAB are significantly lower than in Java, meaning that with less effort it is possible to achieve similar results using Python or MATLAB.

5.2.3 Global Analysis

Concerning the results of sequential processing, it is interesting to observe that the results are not uniform for all applications. Comparing the results for the composition function and speed reducer, that are two applications solely computed using numerical operations, it is observed that for the first, MATLAB outperforms Python and for the second Python outperforms MATLAB as possible explained by the nature of the applications (vector/matrix computations) and characteristics of both languages. As for Java and C++, results are very similar, but both outperform the other two programming languages by a large margin as both programming languages are compiled and statically typed. A different scenario is observed for the square plate application, as results for all programming languages are very similar. An explanation for these results might relate to the time it takes for the external program to perform the simulation for every solution. As this time is very high (approximately 30 seconds) compared to the computation of additional operations in the evaluation of the objective function. Although the observed similarities, MATLAB performs slightly poorly compared to Python, Java and C++. In the three-bar truss application, even though the evaluation of the objective function requires an external program, results are very interesting, as Python considerably outperforms Java. C++ stands as the fastest programming languages, while the measured total time for MATLAB is the highest of them all. These results in the three-bar truss application are an indication of significant differences in the programming languages regarding operations with files.

In what concerns the results of parallel processing, different situations are observed. Firstly, for the implementation of the composition function in Python, it is observed that the total time improved with the increase in the number of processes. On the other hand, the same cannot be said for MATLAB and Java, as little or no improvements are observed. As for the speed reducer, the parallel processing implementations prove to not benefit the computational time as gains in speedup are not observed. Regarding the three-bar truss and square plate implementations, both applications presented significant gains in speedup, even though the observed results of efficiency are not ideal. Moreover, both applications demonstrated that the gains in speedup do not increase significantly or even decrease for more than five processes. These results might be directly related to the size of the population used in these two applications – nine for the three bar truss and ten for the square plate. These sizes of the population can be considered low and with the increase in the number of processes the tendency would be for them to be equal which does not allow to take full advantage of the increase in the number of processes. Technically, the total time of the parallelized operations is as high as the slowest process, which is why from five to eight processes the evaluation time is similar for both applications, as at least one process is reused twice.

Comparing the results obtained for the sequential and parallel processing implementations, it is shown that the implementation of parallel procedures in some applications does not benefit the performance. That is the case for the speed reducer, as for the three programming languages implemented in both situations the total time increases compared

to the sequential implementations. Additionally, in the case of the composition function, the parallel processing only improves the total time in Python, in opposition to MATLAB and Java where the total time increases. Another interesting observation is made regarding the sequential implementations and parallel implementations with 1 process, as for the three-bar truss, a significant boost in the total time is observed for the three programming languages, a fact that is not well understood. At last, it is observed that the other time in parallel implementations increases comparatively to sequential implementations, probably related to the initialization of processes. This increase in other time becomes more relevant for MATLAB, as it can take as much as 15 seconds to initialize and terminate processes.

In general, C++ is the programming language that presents better results, even though the cost of development can be considered to be higher. As for Python, it demonstrates to perform better than MATLAB, except for the composition function in sequential processing. The reason for this result might be referred to the fact that MATLAB is enhanced in matrix computations and the composition function uses matrices in its objective function. Additionally, the implementation in MATLAB using object-oriented programming might not favor its performance, as maybe it would be preferred an implementation based on arrays. Java demonstrates to perform better than Python for applications solely implemented with numerical calculations. However, it was outperformed by Python when computations involving reading from and writing to files are present, as is the case for the three-bar truss.

Objectively, when the application is purely implemented with numerical calculations, C++ is the fastest programming language. Moreover, if the computational time of an external program is not predominant in the evaluation time, C++ also presents as the fastest programming language. However, as C++ is not implemented using parallel procedures no conclusions are made regarding its performance in parallel processing. When the application can potentially benefit from parallel processing, selecting any programming language out of Python, MATLAB and Java for the implementation, can lead to similar results in speedup. Subjectively, if the computational operations of a specific application are complex (e.g. matrices manipulation, polynomials calculation), implementations using standard features of C++ or Java can lead to an increase in development time compared to Python or MATLAB, which present features that are easily implemented. Moreover, implementing parallel procedures in MATLAB can be as simple as changing two lines of code while in Java the task can prove to be more demanding using advanced features.

Chapter 6

Final Considerations

General conclusions and suggestions for further work are presented.

6.1 Conclusions

The main goal of this work was to analyze the use of advanced optimization methods in mechanical design problems, in which three distinct algorithms were selected: Particle Swarm Optimization (PSO), Differential Evolution (DE) and Teaching-Learning-Based Optimization (TLBO). The implemented formulation of each algorithm was similar to the standard algorithms, only with little modifications in order to improve convergence rates and exploration capabilities of the design space.

In a first phase of the work, it was carried out a study on the size of the population for each algorithm and application, in order to understand how the algorithms performed with the variation of this parameter and to select a size of the population that demonstrated the best results. On one hand, it was observed that for applications where a large number of function evaluations as the stopping criterion are defined, the algorithms demonstrate better results for higher values of the size of the population. Nevertheless, it was observed that the algorithms require more computational effort to reach the same solution than for lower values of the size of the population. On the other hand, if the nature of the operations in the application tends to be computationally heavy, as it happens in the three-bar truss and square plate design problem, and consequently a lower number of function evaluations is preferred, the best results were observed for lower values of the size of the population. Later, the algorithms were compared using the results for the selected size of the population. The DE was the algorithm that presented better results, proving to be the most efficient and robust of the three algorithms. Between the PSO and TLBO, the last demonstrated to be able to find better solutions while in what concerns to the computational effort the results are similar.

Regarding the computational performance of the algorithms in the implemented programming languages, different results were observed for each application. In what concerns the sequential processing performance it was observed that for the composition function, MATLAB outperforms Python as the nature of the problem involves several operations

based on matrices. On the other hand, for the speed reducer design problem Python outperformed MATLAB as the operations are computationally simple. For both applications, Java and C++ demonstrated to be computationally faster than Python and MATLAB, as the code only involves numerical operations. On the other hand, for the applications involving operations with files and using external programs, the results were very different. In the case of the square plate, the use of the external program demonstrated to have a great impact on the evaluation time of the objective function, thereby resulting in similar computational times for all programming languages. However, the same pattern was not observed in the three-bar truss design problem, where several differences were observed between programming languages. With the implementation of parallel processing techniques to the PSO, it was shown that, in general, the applications that demonstrated to be computationally fast in the sequential processing implementations do not benefit with the parallel processing. On the other hand, applications computationally heavy, as was the case of the three-bar truss and square plate design problems, presented great benefits from the parallel processing.

Overall, the use of advanced optimization methods in mechanical design problems presents as a viable option as they demonstrate to be efficient in the search of engineering solutions. The DE presents as the more reliable algorithm while its operations are simple to implement. Nevertheless, the DE and PSO require the selection of operational parameters that might be difficult to estimate and for this reason, the TLBO presents as a viable algorithm as it does not require the selection of any operational parameter aside from the size of the population. Furthermore, in the selection of the programming language for their implementation, it requires the consideration of the cost of development with the computational impact. Even though it was demonstrated that Java and C++ are computationally fast for applications solely involving numerical operations their cost of development can be said to be greater than in Python or MATLAB. Moreover, as for applications where the bulk of the computational effort is independent of the programming language, results were very similar between programming languages while in Python and MATLAB the performance can be easily enhanced with simple modifications to the sequential processing implementations.

6.2 Future Work

This work is intended to aid in the selection of algorithms and programming tools applied to mechanical design problems. Even though the analyses carried out in this work is expected to help in this decision, there are still things that can be further studied. With this purpose, it is here suggested guidelines for future work:

- Application and comparison of additional advanced optimization methods to mechanical design applications, in particular, more complex problems, such as multi-objective or with a larger number of design variables;
- Analysis of different programming strategies applied to the advanced optimization methods;
- Detailed analysis of the operations involved in computational processing, in particular, the communication between processes;

- Analysis on the influence of the size of the population in parallel processing implementations.

Bibliography

[Alcántar *et al.* 2017] V. Alcántar, S. Ledesma, S.M. Aceves, E. Ledesma and A. Saldaña. Optimization of type III pressure vessels using genetic algorithm and simulated annealing. *International Journal of Hydrogen Energy*, 42(31):20125 – 20132, 2017.

[Babu and Jehan 2003] B.V. Babu and M.M.L. Jehan. *Differential evolution for multi-objective optimization.* In *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, Vol. 4, pp. 2696–2703, 2003.

[Baykasoglu 2012] A. Baykasoglu. Design optimization with chaos embedded great deluge algorithm. *Applied Soft Computing*, 12(3):1055–1067, 2012.

[Caseiro 2009] J.F.M. Caseiro. *Estratégias Evolucionárias de Optimização de Parâmetros Reais.* Master's thesis in Mechanical Engineering, Departamento de Engenharia Mecânica, Universidade de Aveiro, Portugal, 2009.

[Chen *et al.* 2015] S. Chen, J. Montgomery and A. Bolufé-Röhler. Measuring the curse of dimensionality and its effects on particle swarm optimization and differential evolution. *Applied Intelligence*, 42(3):514–526, 2015.

[Coello 2002] C.A.C. Coello. Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, 191(11):1245–1287, 2002.

[Csomor 2017] V. Csomor. PP4J - A multiprocessing library for Java. Github. Available at: `https://github.com/ViktorC/PP4J` [Accessed May 2018], 2017.

[Cung *et al.* 2002] V.D. Cung, L.M. Simone, C.C. Ribeiro and C. Roucairol. Strategies for the parallel implementation of metaheuristics. In *Essays and Surveys in Metaheuristics*, pp. 263–308. Springer, Boston, MA, 2002.

[Degertekin and Hayalioglu 2013] S.O. Degertekin and M.S. Hayalioglu. Sizing truss structures using teaching-learning-based optimization. *Computers & Structures*, 119:177–188, 2013.

[Duncan 1990] R. Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, 1990.

[Epitropakis *et al.* 2011] M.G. Epitropakis, D.K. Tasoulis, N.G. Pavlidis, V.P. Plagianakos and M.N. Vrahatis. Enhancing differential evolution utilizing proximity-based mutation operators. *IEEE Transactions on Evolutionary Computation*, 15(1):99–119, 2011.

[Golinski 1970] J. Golinski. Optimal synthesis problems solved by means of nonlinear programming and random methods. *Journal of Mechanisms*, 5(3):287–309, 1970.

[Gong and Cai 2013] W. Gong and Z. Cai. Differential evolution with ranking-based mutation operators. *IEEE Transactions on Cybernetics*, 43(6):2066–2081, 2013.

[Guedria 2015] N.B. Guedria. Improved accelerated PSO algorithm for mechanical engineering optimization problems. *Applied Soft Computing*, 40:455–467, 2015.

[He *et al.* 2004] S. He, E. Prempain and Q.H. Wu. An improved particle swarm optimizer for mechanical design optimization problems. *Engineering Optimization*, 36(5):585–605, 2004.

[Huu *et al.* 2016] V.H. Huu, T.N. Thoi, T.V. Duy and T.N. Trang. An adaptive elitist differential evolution for optimization of truss structures with discrete design variables. *Computers & Structures*, 165:59–75, 2016.

[Jie *et al.* 2008] J. Jie, J. Zeng, C. Han and Q. Wang. Knowledge-based cooperative particle swarm optimization. *Applied Mathematics and Computation*, 205(2):861–873, 2008.

[Joines and Houck 1994] J.A. Joines and C.R. Houck. *On the use of non-stationary penalty functions to solve nonlinear constrained optimization problems with GA's.* In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, Vol. 2, pp. 579–584, 1994.

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

[Kiran 2017] M. Kiran. Particle swarm optimization with a new update mechanism. *Applied Soft Computing*, 60:670–678, 2017.

[Liang *et al.* 2005] J.J. Liang, P.N. Suganthan and K. Deb. *Novel composition test functions for numerical global optimization.* In *Proceedings 2005 IEEE Swarm Intelligence Symposium, 2005. SIS 2005*, pp. 68–75, 2005.

[Liu and Abraham 2005] H. Liu and A. Abraham. *Fuzzy adaptive turbulent particle swarm optimization.* In *Fifth International Conference on Hybrid Intelligent Systems (HIS'05)*, pp. 6–, 2005.

[Nanz and Furia 2015] S. Nanz and C.A. Furia. *A comparative study of programming languages in rosetta code.* In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*, Vol. 1, pp. 778–788, 2015.

[Nelder and Mead 1965] J.A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.

[Pacheco 2011] P.S. Pacheco. *An Introduction to Parallel Programming.* Morgan Kaufmann, 2011.

[Papadrakakis and Lagaros 2003] M. Papadrakakis and N.D. Lagaros. Soft computing methodologies for structural optimization. *Applied Soft Computing*, 3(3):283–300, 2003.

[Perez and Behdinan 2007] R.E. Perez and K. Behdinan. Particle swarm approach for structural design optimization. *Computers & Structures*, 85(19):1579–1588, 2007.

[Price *et al.* 2005] K. Price, R. Storn and J.A. Lampinen. *Differential Evolution: A Practical Approach to Global Optimization*. Springer-Verlag, Berlin, Heidelberg, 2005.

[Qu *et al.* 2017] J. Qu, X. Liu, M. Sun and F. Qi. GPU-based parallel particle swarm optimization methods for graph drawing. *Discrete Dynamics in Nature and Society*, pp. 1–15, 2017.

[Rao 2009] S.S. Rao. *Engineering Optimization: Theory and Practice*. John Wiley and Sons, Inc., 4th edition, 2009.

[Rao and Savsani 2012] R.V. Rao and J.V. Savsani. *Mechanical Design Optimization Using Advanced Optimization Techniques*. Springer-Verlag London, 2012.

[Rao and Waghmare 2017] R.V. Rao and G.G. Waghmare. A new optimization algorithm for solving complex constrained design optimization problems. *Engineering Optimization*, 49(1):60–83, 2017.

[Rao *et al.* 2011] R.V. Rao, V.J. Savsani and D.P. Vakharia. Teaching–learning-based optimization: a novel method for constrained mechanical design optimization problems. *Computer-Aided Design*, 43(3):303–315, 2011.

[Rao *et al.* 2012] R.V. Rao, V.J. Savsani and D.P. Vakharia. Teaching–learning-based optimization: an optimization method for continuous non-linear large scale problems. *Information Sciences*, 183(1):1–15, 2012.

[Reghunadh and Jain 2011] J. Reghunadh and N. Jain. Selecting the optimal programming language. IBM. Available at: `https://www.ibm.com/developerworks/library/wa-optimal` [Accessed Feb 2018], 2011.

[Salomon 1996] R. Salomon. Re-evaluating genetic algorithm performance under coordinate rotation of benchmark functions. A survey of some theoretical and practical aspects of genetic algorithms. *BioSystems*, 39(3):263–278, 1996.

[Saruhan 2014] H. Saruhan. Differential evolution and simulated annealing algorithms for mechanical systems design. *Engineering Science and Technology, an International Journal*, 17(3):131–136, 2014.

[Schutte *et al.* 2004] J.F. Schutte, J.A. Reinbolt, B.J. Fregly, R.T. Haftka and A.D. George. Parallel global optimization with the particle swarm algorithm. *International Journal for Numerical Methods in Engineering*, 61(13):2296–2315, 2004.

[Sebesta 2012] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson, 10th edition, 2012.

[Shi and Eberhart 1998a] Y. Shi and R. Eberhart. *A modified particle swarm optimizer.* In *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence*, pp. 69–73, 1998.

[Shi and Eberhart 1998b] Y. Shi and R. Eberhart. *Parameter selection in particle swarm optimization.* In *EP '98 Proceedings of the 7th International Conference on Evolutionary Programming VII*, pp. 591–600, 1998.

[Smith 2009] M. Smith. *ABAQUS/Standard User's Manual, Version 6.9.* Simulia, 2009.

[Storn 1996] R. Storn. *On the usage of differential evolution for function optimization.* In *Proceedings of North American Fuzzy Information Processing*, pp. 519–523, 1996.

[Storn and Price 1997] R. Storn and K. Price. Differential evolution - A simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359, 1997.

[Suganthan 1999] P. N. Suganthan. *Particle swarm optimiser with neighbourhood operator.* In *Proceedings of the 1999 Congress on Evolutionary Computation - CEC99*, Vol. 3, pp. 1958–1962, 1999.

[TIOBE 2018] TIOBE - The Software Quality Company. TIOBE Index. Available at: `https://www.tiobe.com/tiobe-index/` [Accessed Feb. 2018], 2018.

[Trelea 2003] I.C. Trelea. The particle swarm optimization algorithm: convergence analysis and parameter selection. *Information Processing Letters*, 85(6):317–325, 2003.

[Valente *et al.* 2011] R.A.F. Valente, A. Andrade-Campos, J.F. Carvalho and P.S. Cruz. Parameter identification and shape optimization. *Optimization and Engineering*, 12(1):129–152, 2011.

[Veeramachaneni *et al.* 2003] K. Veeramachaneni, T. Peram, C. Mohan and L.A. Osadciw. *Optimization using particle swarms with near neighbor interactions.* In *Genetic and Evolutionary Computation - GECCO 2003*, pp. 110–121, 2003.

[Williams 2012] A. Williams. *C++ Concurrency in Action: Practical Multithreading.* Manning Publications Co., 2012.

[Yang and Deb 2010] X. Yang and S. Deb. Engineering optimisation by cuckoo search. *International Journal of Mathematical Modelling and Numerical Optimisation*, 1(4):330–343, 2010.

[Yang *et al.* 2015] M. Yang, C. Li, Z. Cai and J. Guan. Differential evolution with auto-enhanced population diversity. *IEEE Transactions on Cybernetics*, 45(2):302–315, 2015.

[Yang *et al.* 2016] X. Yang, S. Deb, S. Fong, X. He and Y. Zhao. From swarm intelligence to metaheuristics: nature-inspired optimization algorithms. *Computer*, 49(9):52–59, 2016.

[Yang *et al.* 2018] X. Yang, S. Deb, Y. Zhao, S. Fong and X. He. Swarm intelligence: past, present and future. *Soft Computing*, 22(18):5923–5933, 2018.

[Zhang *et al.* 2018] J. Zhang, X. Qin, C. Xie, H. Chen and L. Jin. Optimization design on dynamic load sharing performance for an in-wheel motor speed reducer based on genetic algorithm. *Mechanism and Machine Theory*, 122:132–147, 2018.

[Zhenyu *et al.* 2006] G. Zhenyu, C. Bo, Y. Min and C. Binggang. *Self-adaptive chaos differential evolution.* In *Advances in Natural Computation. ICNC 2006*, pp. 972–975. Springer, Berlin, Heidelberg, 2006.

[Zhou and Tan 2009] Y. Zhou and Y. Tan. *GPU-based parallel particle swarm optimization.* In *2009 IEEE Congress on Evolutionary Computation*, pp. 1493–1500, 2009.

[Zhou *et al.* 2006] C. Zhou, L. Gao, H.B. Gao and K. Zan. *Particle swarm optimization for simultaneous optimization of design and machining tolerances.* In *Simulated Evolution and Learning. SEAL 2006*, pp. 873–880. Springer, Berlin, Heidelberg, 2006.

[Črepinšek *et al.* 2012] M. Črepinšek, S.H. Liu and L. Mernik. A note on teaching–learning-based optimization algorithm. *Information Sciences*, 212:79–93, 2012.