DM

Nature-Inspired Algorithms for Solving Some Hard Numerical Problems

MASTER DISSERTATION

Diogo Nuno Teixeira Freitas

MASTER IN MATHEMATICS, STATISTICS AND APPLICATIONS

UNIVERSIDADE da MADEIRA

A Nossa Universidade

www.uma.pt

September | 2020

Nature-Inspired Algorithms for Solving Some Hard Numerical Problems

MASTER DISSERTATION

Diogo Nuno Teixeira Freitas
MASTER IN MATHEMATICS, STATISTICS AND APPLICATIONS

SUPERVISOR
Luiz Carlos Guerreiro Lopes

CO-SUPERVISOR
Fernando Manuel Rosmaninho Morgado Ferrão Dias

Nature-Inspired Algorithms for Solving Some Hard Numerical Problems

MASTER DISSERTATION

Diogo Nuno Teixeira Freitas
MASTER IN MATHEMATICS, STATISTICS AND APPLICATIONS

EXAMINATION COMMITTEE
Chairperson: Ana Maria Cortesão Pais Figueira da Silva Abreu
Examiner: Paulo Sérgio Abreu Freitas
Co-Supervisor: Fernando Manuel Rosmaninho Morgado Ferrão Dias

Nature-Inspired Algorithms for Solving Some Hard Numerical Problems

Diogo Nuno Teixeira Freitas

Supervisor:
Prof. Dr. Luiz Carlos Guerreiro Lopes

Co-Supervisor:
Prof. Dr. Fernando Manuel Rosmaninho Morgado Ferrão Dias

A thesis presented to the University of Madeira in fulfillment
of the requirements for the degree of Master of Science in
Mathematics, Statistics and Applications

Funchal – Portugal
September 2020

To my Mother, my Dad
and my Brother.

Nature-Inspired Algorithms for Solving Some Hard Numerical Problems

Abstract

Optimisation is a branch of mathematics that was developed to find the optimal solutions, among all the possible ones, for a given problem. Applications of optimisation techniques are currently employed in engineering, computing, and industrial problems. Therefore, optimisation is a very active research area, leading to the publication of a large number of methods to solve specific problems to its optimality.

This dissertation focuses on the adaptation of two nature inspired algorithms that, based on optimisation techniques, are able to compute approximations for zeros of polynomials and roots of non-linear equations and systems of non-linear equations.

Although many iterative methods for finding all the roots of a given function already exist, they usually require: (a) repeated deflations, that can lead to very inaccurate results due to the problem of accumulating rounding errors, (b) good initial approximations to the roots for the algorithm converge, or (c) the computation of first or second order derivatives, which besides being computationally intensive, it is not always possible.

The drawbacks previously mentioned served as motivation for the use of Particle Swarm Optimisation (PSO) and Artificial Neural Networks (ANNs) for root-finding, since they are known, respectively, for their ability to explore high-dimensional spaces (not requiring good initial approximations) and for their capability to model complex problems. Besides that, both methods do not need repeated deflations, nor derivative information.

The algorithms were described throughout this document and tested using a test suite of hard numerical problems in science and engineering. Results, in turn, were compared with several results available on the literature and with the well-known Durand–Kerner method, depicting that both algorithms are effective to solve the numerical problems considered.

Keywords: optimisation; particle swarm optimisation; artificial neural networks; roots; polynomials; non-linear equations

Nature-Inspired Algorithms for Solving Some Hard Numerical Problems

Resumo

A Optimização é um ramo da matemática desenvolvido para encontrar as soluções óptimas, de entre todas as possíveis, para um determinado problema. Actualmente, são várias as técnicas de optimização aplicadas a problemas de engenharia, de informática e da indústria. Dada a grande panóplia de aplicações, existem inúmeros trabalhos publicados que propõem métodos para resolver, de forma óptima, problemas específicos.

Esta dissertação foca-se na adaptação de dois algoritmos inspirados na natureza que, tendo como base técnicas de optimização, são capazes de calcular aproximações para zeros de polinómios e raízes de equações não lineares e sistemas de equações não lineares.

Embora já existam muitos métodos iterativos para encontrar todas as raízes ou zeros de uma função, eles usualmente exigem: (a) deflações repetidas, que podem levar a resultados muito inexactos, devido ao problema da acumulação de erros de arredondamento a cada iteração; (b) boas aproximações iniciais para as raízes para o algoritmo convergir, ou (c) o cálculo de derivadas de primeira ou de segunda ordem que, além de ser computacionalmente intensivo, para muitas funções é impossível de se calcular.

Estas desvantagens motivaram o uso da Optimização por Enxame de Partículas (PSO) e de Redes Neurais Artificiais (RNAs) para o cálculo de raízes. Estas técnicas são conhecidas, respectivamente, pela sua capacidade de explorar espaços de dimensão superior (não exigindo boas aproximações iniciais) e pela sua capacidade de modelar problemas complexos. Além disto, tais técnicas não necessitam de deflações repetidas, nem do cálculo de derivadas.

Ao longo deste documento, os algoritmos são descritos e testados, usando um conjunto de problemas numéricos com aplicações nas ciências e na engenharia. Os resultados foram comparados com outros disponíveis na literatura e com o método de Durand–Kerner, e sugerem que ambos os algoritmos são capazes de resolver os problemas numéricos considerados.

Palavras-chave: otimização; otimização por enxame de partículas; redes neurais artificiais; raízes; polinómios; equações não-lineares

Acknowledgements

In the first place, I would like to express my sincerest gratitude to my advisors, Professor Luiz Guerreiro Lopes and Professor Fernando Morgado-Dias, who have always been available to help me, to review the contents of this dissertation, to give suggestions and to open doors for my future as a researcher.

My family, Mary, Duarte and Afonso, also played an essential role in developing this work, always showing fundamental support in all the decisions I have been making. I dedicate this dissertation to them.

I thank my colleagues from my undergraduate course, and especially those from the Master's degree: João Gouveia, Joana Correia and Patrícia Freitas.

As it should be, I also thank all my professors from the Faculty of Exact Sciences and Engineering; without these, I certainly would not have the knowledge and methodologies that I have today.

My colleagues from GDG Madeira (Carlos Silva Abreu and Fernando Martins), from the University of Madeira Students' Union (Andreia Micaela Nascimento, Luís Eduardo Nicolau and Gonçalo Martins) and from Asseco PST (especially, Paulo Teixeira), that also contributed directly or indirectly to this work. Mine acknowledge to them.

I thank the University of Madeira, the Interactive Technologies Institute and all the non-teaching staff for providing me with a quality space for work.

Acknowledgement to the Project MITIExcell (Project – UIDB/50009/2020), co-financed by Regional Development European Funds for the "Operational Programme Madeira 14–20"—Priority Axis 1 of the Autonomous Region of Madeira, number M1420-01-0145-FEDER-000002. Also acknowledged is the funding from LARSyS – FCT Plurianual funding 2020–2023.

Finally, to all those who, by mistake, I did not mention previously but contributed to this work, my thanks.

Contents

1 Introduction **1**
1.1 Optimisation . . . 1
1.2 Aim of the Research . . . 3
1.3 Structure of the Dissertation . . . 3

2 State-of-the-Art **5**
2.1 Introduction . . . 5
2.2 Particle Swarm Optimisation . . . 6
2.3 Modifications to the Particle Swarm Optimisation . . . 8
2.3.1 Algorithm Convergence Improvements . . . 8
2.3.2 Neighbourhoods . . . 11
2.3.3 The Stagnation Problem . . . 14
2.4 Particle Swarm Optimisation Variants . . . 15
2.4.1 Cooperative Particle Swarm Optimisation . . . 15
2.4.2 Adaptive Particle Swarm Optimisation . . . 16
2.4.3 Constrained Optimisation Problems . . . 17
2.4.4 Multi-Objective Optimisation . . . 18
2.4.5 Multimodal Function Optimisation . . . 19
2.4.6 The Fully Informed Particle Swarm Optimisation . . . 23
2.4.7 Parallel Implementations of Particle Swarm Optimisation . . . 24
2.5 Connections to Other Artificial Intelligence Tools . . . 26
2.5.1 Hybrid Variants of Particle Swarm Optimisation . . . 26
2.5.2 Artificial Neural Networks with Particle Swarm Optimisation . . . 33
2.6 Conclusion . . . 35

3 On the Use of Particle Swarm Optimisation for Root-Finding **37**
3.1 Introduction . . . 38
3.2 Particle Swarm Optimisation . . . 38
3.2.1 Convergence Improvements . . . 39
3.2.2 Swarm Initialisation . . . 42
3.2.3 Optimisation Cycle . . . 43
3.2.4 Stopping Criterion . . . 43
3.3 Related Work . . . 44
3.4 Root-Finding Particle Swarm Optimisation . . . 46
3.4.1 Adaptation of Particle Swarm Optimisation . . . 47
3.4.2 Parameter Tuning . . . 48

3.4.3 Population Size . . . 52
3.4.4 Velocity Update Equation . . . 54
3.4.5 Communication Structure . . . 58
3.4.6 Summary . . . 62
3.5 Multi Root-Finding Particle Swarm Optimisation . . . 62
3.5.1 Detecting Equal Roots . . . 64
3.5.2 Particle Positioning . . . 64
3.5.3 Examples of Execution . . . 65
3.6 Conclusion . . . 68

4 A Neural Network-Based Approach for Approximating the Roots of Polynomials 71
4.1 Introduction . . . 71
4.2 Feedforward Artificial Neural Networks . . . 73
4.2.1 Neuron Functions . . . 73
4.2.2 Number of Neurons . . . 74
4.2.3 Number of Hidden Layers . . . 75
4.2.4 Training Algorithms . . . 75
4.3 State-of-the-Art . . . 76
4.4 Related Work . . . 79
4.5 Methodology . . . 83
4.5.1 Durand–Kerner Algorithm . . . 84
4.5.2 Artificial Neural Networks . . . 86
4.6 Results and Discussion . . . 88
4.6.1 Polynomials with Only Real Roots . . . 88
4.6.2 Polynomials with Both Real and Complex Roots . . . 90
4.7 Enhancing the Artificial Neural Network for Root-Finding . . . 92
4.8 Initialisation Scheme for the Durand–Kerner Method . . . 94
4.9 Conclusion . . . 96

5 Conclusions and Future Work 97
5.1 Conclusion . . . 97
5.1.1 Parameter Selection for Root-Finding with Particle Swarm Optimisation 97
5.1.2 Multi Root-Finding Particle Swarm Optimisation . . . 98
5.1.3 Neural Network-Based Approach for Approximating Roots of Polynomials . . . 98
5.1.4 Initialisation Scheme for the Durand–Kerner Method . . . 99
5.2 Future Work . . . 99

Bibliography 100

Appendices

A Particle Swarm Optimisation Initialisation Scheme 124

B Particle Swarm Optimisation Algorithm Flowchart 125

C Particle Swarm Optimisation Cycle **126**

D Test Functions **127**
D.1 Ackley Function 127
D.2 Rastrigin Function 128
D.3 Rosenbrock Function 128
D.4 Schaffer Function No. 2 129
D.5 Sphere Function 129

E Swarm Architectures **131**
E.1 All-Connected-To-All 131
E.2 Ring and Pyramid 131
E.3 Random 132
E.4 Mesh and Toroid 132
E.5 Star 133

F MRF–PSO Algorithm Flowchart **134**

G Data Sets **135**
G.1 Input Data Set for Training the ANN 5 for Computing the Real Roots . . . 135
G.2 Output Data Set for Training the ANN 5 for Computing the Real Roots . . 135
G.3 Input Data Set for Training the ANN 5 for Computing the Real and Complex Roots 136
G.4 Output Data Set for Training the ANN 5 For Computing the Real and Complex Roots 136

H Results of the Neural Network-Based Approach for Approximating the Roots of Polynomials Using Particle Swarm Optimisation with the Parameters Found in Chapter 3 **137**

List of Figures

2.1 Summary of the most important convergence improvements developed for PSO. 11
2.2 Summary of the most important architecture strategies developed for PSO. . 14
2.3 Summary of the most important applications of PSO. 35

3.1 PID controller, proposed by Wang et al., to control the inertia term. 45
3.2 The MRF-PSO architecture for sharing information about the roots found during the search process of each swarm. 63
3.3 Example of particle repositioning when three roots were discovered by Swarms 1, 3 and 5. 65

4.1 An example of a shallow ANN with four inputs nodes, five hidden nodes and an output node. 73
4.2 Behaviour of a single artificial neuron for shallow networks. 74
4.3 Flowchart showing the inputs, processing flow and the outputs of the proposed neural approach for polynomial root-finding. 83
4.4 Architecture of the ANN n for root-finding. 87

B.1 Flowchart of the PSO algorithm. 125

D.1 A 2D graphical representation of the Ackley function. 127
D.2 A 2D graphical representation of the Rastrigin function. 128
D.3 A 2D graphical representation of the Rosenbrock function. 128
D.4 A 2D graphical representation of a Schaffer function no. 2. 129
D.5 A 2D graphical representation of the sphere function. 130

E.1 An example of a graphical representation of an all-connected-to-all communication structure, also known as gbest model, with six particles. 131
E.2 An example of the mesh and toroid communication structures. 131
E.3 An example of a graphical representation of a random communication structure, with five particles. 132
E.4 An example of the mesh and toroid communication structures. 132
E.5 An example of a graphical representation of a star communication structure, also knows as wheel architecture, with ten particles. 133

F.1 Flowchart of the MRF-PSO algorithm. 134

List of Tables

3.1 Independent variables and their respective levels. 50
3.2 Auxiliary table used to compare the effect of the different number of particles in the swarm on the PSO algorithm. 53
3.3 Comparison of the different number of particles in the swarm (X means performed better.) 54
3.4 Inertia weights variations and the constriction term used in the tests, and the values of the parameters used. 55
3.5 Auxiliary table used to compare the effect of the different particle's velocity update equations on the PSO algorithm. 56
3.6 Comparison of the different particle's velocity update equations (X means performed better.) 58
3.7 Auxiliary table used to compare the effect of the different swarm's communication structures on the PSO algorithm. 60
3.8 Comparison of the different swarm's communication structures (X means performed better.) 62
3.9 Comparison of the MRF-PSO with some of the results available on the literature for Example 3.5.1. 68
3.10 Comparison of the MRF-PSO with some of the results available on the literature for Example 3.5.2. 69
3.11 Comparison of the MRF-PSO with some of the results available on the literature for Example 3.5.3. 69
3.12 Comparison of the MRF-PSO with some of the results available on the literature for Example 3.5.4 69

4.1 Example of execution of the D–K algorithm. 84
4.2 Comparison between the different initialisation values assigned to $\alpha_1^{(0)}$, $\alpha_2^{(0)}$ and $\alpha_3^{(0)}$. 85
4.3 Comparison between ANN and D–K methods in terms of the MSE for polynomials with only real roots. 89
4.4 Comparison between ANN and D–K methods in terms of average execution time (in seconds) for polynomials with only real roots. 89
4.5 Comparison between ANN and D–K methods in terms of the MSE for polynomials with both real and complex roots. 91
4.6 Comparison between ANN and D–K methods in terms of the average execution time (in seconds) for polynomials with both real and complex roots. . . 91
4.7 Comparison of the MSE between complex roots in polar coordinates, and in real and imaginary parts for ANN-based approach. 91

4.8 Comparison of the capacity of the networks to generalise the outputs between the D–K method, ANN trained with LMA and with the PSO algorithm for the case with real roots. 93
4.9 Comparison of the capacity of the networks to generalise the outputs between the D–K method, ANN trained with LMA and with the PSO algorithm for the case with complex roots. 93
4.10 Comparison of the capacity of the networks to generalise the outputs between ANN trained with LMA and with the PSO algorithm for the case with complex roots in polar coordinates. 94
4.11 Comparison between the Cauchy's upper bound and ANN-based approach to provide the initial approximation for each root to the D–K method for the case with real roots. 95
4.12 Comparison between the Cauchy's upper bound and ANN-based approach to provide the initial approximation for each root to the D–K method for the case with complex roots. 96

G.1 Head of the input data set for training the ANN 5 for computing only the real roots. 135
G.2 Head of the output data set for training the ANN 5 for computing only the real roots. 135
G.3 Head of the input data set for training the ANN 5 for computing both real and complex roots. 136
G.4 Head of the output data set for training the ANN 5 for computing both real and complex roots. 136

H.1 Capacity of the networks to generalise the outputs using PSO with the parameters found in Chapter 3. Results are given in terms of MSE. 137

Abbreviations

ABC	Artificial Bee Colony.
ACO	Ant Colony Optimisation.
ANN	Artificial Neural Network.
ANOVA	Analysis of Variance.
APSO	Adaptive Particle Swarm Optimisation.
BFGS	Broyden–Fletcher–Goldfarb–Shanno.
BFO	Bacterial Foraging Optimisation.
CLA	Constrained Learning Algorithm.
COP	Constrained Optimisation Problem.
CPSO	Cooperative Particle Swarm Optimisation.
CPU	Central Processing Unit.
D–K	Durand–Kerner.
DE	Differential Evolution.
DEEPSO	Differential Evolutionary Particle Swarm Optimisation.
DNSPSO	Diversity Enhanced Particle Swarm Optimisation with Neighbourhood Search.
EA	Evolutionary Algorithm.
EPSO	Evolutionary Particle Swarm Optimisation.
FDR	Fitness-Distance-Ratio.
FIPS	Fully Informed Particle Swarm.
FNN	Feedforward Artificial Neural Network.
FSM	Preservation of Feasible Solutions Method.
GA	Genetic Algorithm.
GCPSO	Guaranteed Convergence Particle Swarm Optimisation.
GPU	Graphics Processing Unit.
LMA	Levenberg–Marquardt Algorithm.
LSTM	Long Short-Term Memory.

MOPSO	Multi-Objective Particle Swarm Optimisation.
MPPSO	Multi-Phase Particle Swarm Optimisation.
MRF-PSO	Multiple Root-Finding Particle Swarm Optimisation.
MSE	Mean Squared Error.
OBS	Optimal Brain Surgeon.
PI	Proportional Integral.
PID	Proportional Integral Derivative.
PPSO	Parallelised Particle Swarm Optimisation.
PSO	Particle Swarm Optimisation.
ReLU	Rectified Linear Unit.
SA	Simulated Annealing.
SCD	Special Crowding Distance.
SPSO	Standard Particle Swarm Optimisation.
STPSO	Stretched Particle Swarm Optimisation.

Mathematical Notation

D	Mean distance of each particle to other particles.
$F(\cdot)$	Penalty function to be minimised or maximised.
$H(\cdot)$	Function stretching for multimodal function optimisation.
$H_p(\cdot)$	Penalty factor in a penalty function.
I	Identity matrix.
I_{ij}	i-th input of the j-th neuron in an ANN.
J	Jacobian matrix.
$K(\cdot)$	Constriction factor or the number of populations in an ANOVA procedure.
N_i	Neighbourhood of the particle i.
O	Output of an ANN or from a neuron unit.
$P(\cdot)$	Real univariate polynomial.
R_1	Cognitive uniformly distributed random vector used to compute the particle's velocity.
R_2	Social uniformly distributed random vector used to compute the particle's velocity.
S	Search space, defined by the domain of the function to be optimised, that contains all the feasible solutions for the problem.
S_δ	δ order root moment of a given polynomial.
$T(\cdot)$	Transfer function applied in every neuron in an ANN.
α	Diagonal matrix whose diagonal values are within the range of $[0, 1]$, or a real or complex root of a given real univariate polynomial.
β	Dilation factor.
ϵ	Absolute difference between the last and the current best fitness value, or the algorithm accuracy.
$\gamma(\cdot)$	Power of a penalty function.
$\hat{y}$	Position of the best particle in the swarm or in the neighbourhood (target particle).
μ	Index of the global best particle in the swarm.

$\omega(\cdot)$	Inertia weight parameter used to compute the velocity of each particle.
ϕ	Activation function of an ANN.
ρ	Diagonal matrix that represents the architecture of the swarm.
σ	Scale parameter of the Cauchy mutation.
τ_k	Average of the population k.
$\theta(\cdot)$	Multi-stage assignment function in a penalty function.
υ	Non-linear modulation index.
φ_1	Cognitive real acceleration coefficient used to compute the particle's velocity.
φ_2	Social real acceleration coefficient used to compute the particle's velocity.
φ_3	Deviation real acceleration coefficient used to compute the particle's velocity.
$\vec{P}_t^{\,j}$	Prior best position of the particle j (nbest particle) that maximises the FDR measure at iteration t.
$\vec{V}$	Particle's velocity.
$\vec{c}^{\,j}$	Position of the centroid of the group j.
$\vec{g}$	Global best position of a particle in the swarm.
$\vec{l}_t^{\,i}$	Local best position of the best particle in the particle i's neighbourhood at iteration t.
$\vec{p}_t^{\,i}$	Personal best position of the particle i at iteration t.
$\vec{x}$	Position vector of a solution found in the search space.
$\vec{x}_{\max}$	Upper limit of the search space.
$\vec{x}_{\min}$	Lower limit of the search space.
$\vec{y^*}$	Set of feasible solutions that forms the Pareto front.
ξ	Random variable drawn from the uniform distribution over the closed interval of 0 to 1.
a	Coefficient of a given real univariate polynomial.
d	Number of dimensions of the search space.
e	Squared error between the target and the estimated values or absolute tolerance parameter used in the MRF-PSO algorithm.
e_f	Evolutionary factor used in the APSO.
$f(\cdot)$	Objective function to be minimised or maximised.
g	Set of inequality function constraints.
h	Set of equality function constraints.

$h_p(\cdot)$	Dynamic modified penalty value in a penalty function.
k	Number of independent populations in an ANOVA procedure.
l	Number of particles in the swarm or in the neighbourhood.
m	Number of inequality constraints.
n	Number of inputs of a neuron in an ANN or the degree of a real univariate polynomial.
$nr(\cdot)$	Function to detect equal roots in the MRF-PSO algorithm.
p	Number of equality constraints.
$q_i(\cdot)$	Relative violated function of the i-th constraint in a penalty function.
r	Set of roots found by the MRF-PSO algorithm.
s	Number of particles in the swarm.
t	Number of the current iteration.
w_t	Parameter, in the form of a diagonal matrix, to add variability to the best position in the swarm at iteration t.
w_{ij}	Weight of the i-th input of the j-th neuron in an ANN.
z	Number of roots found by the MRF-PSO algorithm.

Chapter 1

Introduction

— If I have seen further it is by standing on the shoulders of Giants.

Isaac Newton (1643–1727)

1.1 Optimisation

Optimisation is one of the oldest branches of mathematics and aims to obtain the optimal solution(s) for a given problem out of all possible candidate solutions that fulfil the problem's constraints, that is, to do things best under the given circumstances [1].

This term is used frequently in different contexts, e.g., in mathematics, engineering, computing or business industry, with different applications such as agricultural planning, data analysis, military operations, risk management and other decision-making systems.

Optimisation is, therefore, present in our daily life, but the task of finding the optimal solution(s), according to multiple criteria, is not always straightforward. Thus, many techniques and mathematical models have emerged for tackling different kinds of optimisation problems. Interestingly, some mathematical models of optimisation take as inspiration the biological behaviour of living beings, using individual and collective behaviour to accomplish some common goal.

The first step to find the optimal solution(s) for a problem is to model it, that is, to create a mathematical model considering the objective to optimise and all the underlying conditions. The result of this phase is an objective function, which will be used to assess the quality of the candidate solutions found, and a set of constraints [2]. (Some authors may also refer to the objective function as fitness function, loss function or cost function, according to the context of the problem.)

The objective function will be then minimised or maximised, according to the problem at hands (e.g., minimisation of cost or maximisation of profit), and subject to the defined constraints.

In this work, it will be considered, without any loss of generality, that a single objective optimisation problem is modelled as (see [3]):

Find $\vec{x} \in S \subseteq \mathbb{R}^d$ such that

$$\forall\, \vec{y} \in S,\ \ f(\vec{x}) \leq f(\vec{y}), \tag{1.1}$$

subject to

$$\begin{aligned} g_i(\vec{x}) &\leq 0,\ i = 1, ..., m, \\ h_j(\vec{x}) &= 0,\ j = 1, ..., p, \end{aligned} \tag{1.2}$$

where $f(\vec{x})$ is the function to be optimised, $\vec{x} = \left[x_1, x_2, ..., x_d\right]$ is the position vector of a solution found in the search space S with d dimensions. $g(\vec{x})$ and $h(\vec{x})$ are sets of inequality and equality constraints with m and p functions, respectively. (When $m > 0$ or $p > 0$, the problem is called a Constrained Optimisation Problem [COP].)

The above formulation models any single objective minimisation optimisation problem; however, it can be translated into a maximisation problem by minimising $-f(\vec{x})$.

Thus, three components must be set before using an optimisation technique: (a) a set of variables, (b) a fitness function to be optimised, and (c) a set of constraints that specify the feasible space of each variable.

One of the first techniques developed to solve optimisation problems was the linear programming method (sometimes referred to it as linear optimisation). This method requires that all $f(\vec{x})$, $g_i(\vec{x})$ and $h_j(\vec{x})$ to be linear functions.

Although it was used for many business, economics and engineering problems, in real-life problems $f(\vec{x})$, $g_i(\vec{x})$ or $h_j(\vec{x})$ may not be linear. Then, other techniques were developed to solve both linear and non-linear problems, raising the concepts of non-linear programming, dynamic programming, and computational intelligence-based techniques, such as Particle Swarm Optimisation (PSO).

These techniques can be subdivided into two categories: exact algorithms, which always solve an optimisation problem to optimality (e.g., analytical derivation), and stochastic algorithms, which rest on the presence of randomness to minimise or maximise a function for finding quasi-optimal solutions.

Because conventional mathematical techniques for optimisation are not sufficient or sometimes even impossible to use, especially when they do not meet all the required mathematical assumptions or when the problem's search space is too large and complex, stochastic techniques have received more attention by researchers in recent years.

Nevertheless, traditional mathematical optimisation techniques are still being employed as, e.g., training algorithms for finding the optimal weights and biases of an Artificial Neural Network (ANN). Examples of this type of algorithm include the Newton's method, the Levenberg–Marquardt Algorithm (LMA) and also the well known Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm.

Inside the stochastic category, there are the so-called metaheuristic optimisation strategies. These evolution-based strategies were studied to solve complex computational optimisation problems. To do that, researchers have been looking into nature for years (as a model and as a metaphor) in search of inspiration (such as the Darwinian evolution [4]), and tried to mimic the behaviour of elements inside a population.

In these population-based strategies, each element is a candidate solution to the problem. Some biological evolution operations (e.g., selection, reproduction, recombination, and mutation) are then applied to each population's element to generate new elements [5], and thus new potential candidate solutions for the problem.

Many variants of that strategy have emerged, but the general steps of any evolutionary method include: (1) initialisation of population, (2) evaluation of the fitness value of each

element, and (3) generation of a new population. Examples of such strategies are Genetic Algorithm (GA), Differential Evolution (DE) and PSO.

1.2 Aim of the Research

Finding approximations for the roots, or zeros, of functions is a task that is often required in various areas, and is still a research area with highly essential applications, e.g., signal processing, filter design, speech processing, communication, cryptography and many other applications of science and technology [6, 7].

Although many iterative methods for finding all the roots of a function sequentially or simultaneously exist, there are some drawbacks, such as the necessity of repeated deflations, which leads to the accumulation of rounding errors and inaccurate results, or the need for good initial approximations for all the roots to the algorithm converge [8].

One of the main objectives of this research is to study how the PSO algorithm can be used to find, at the first stage, one root, and the effect of the choice of parameters. Then, how the algorithm can be generalised to be used to find the multiple roots of non-linear equations or the solutions of a non-linear system of equations simultaneously.

On the other hand, since traditional ANNs are well known for their capability to recognise numerical patterns and thus find good approximations for complex mathematical problems, it will also be studied its potentialities and limitations to find the arbitrary (real or complex) roots of a given polynomial.

Both approaches will be compared, in terms of effectiveness and efficiency, with other results available in the literature and with a traditional iterative method for polynomial root-finding.

1.3 Structure of the Dissertation

This dissertation is structured as follows. After the introduction in Chapter 1, Chapter 2 presents the state-of-the-art of the PSO algorithm and its variants. From this chapter, the following article was extracted and published in an open-access journal (ranked Q1 in Mathematical Physics according to Scopus's data – impact factor: 2.494):

- [9] D. Freitas, L. G. Lopes, and F. Morgado-Dias, "Particle swarm optimisation: A historical review up to the current developments," *Entropy*, vol. 22, p. 362, Mar. 2020

Chapter 3 describes the PSO algorithm and its implementation. A comparison between the effect of the different PSO's parameters in the task of finding a single root is made by conducting a set of statistical tests. In this chapter, the Multiple Root-Finding Particle Swarm Optimisation (MRF-PSO) algorithm is presented and tested. (The MRF-PSO is a PSO variant that will be used as a mean for finding approximations to the multiple roots of a given function simultaneously.)

On the other hand, ANNs are used in Chapter 4 to find the real and complex roots of a polynomial. Besides that, from this chapter, the following conference paper was extracted and published (distinguished with the Best Student Paper Award):

- [8] D. Freitas, L. G. Lopes, and F. Morgado-Dias, "A neural network based approach for approximating real roots of polynomials," in *Proc. of the International Conference on Mathematical Applications (ICMA)*, (Funchal, Portugal), pp. 44–47, July 2018

Finally, the conclusions and future work will be outlined in Chapter 5.

Chapter 2

State-of-the-Art

> — What we know is a drop, what we don't know is an ocean.
>
> *Isaac Newton* (1643–1727)

Abstract – The Particle Swarm Optimisation (PSO) algorithm was inspired by the social and biological behaviour of bird flocks searching for food sources. In this nature-based algorithm, individuals are referred to as particles and fly through the search space seeking for the global best position that minimises (or maximises) a given problem. Today, PSO is one of the most well known and widely used swarm intelligence algorithms and metaheuristic techniques, because of its simplicity and ability to be used in a wide range of applications. However, in-depth studies of the algorithm have led to the detection and identification of a number of problems with it, especially convergence problems and performance issues. Consequently, a myriad of variants, enhancements and extensions to the original version of the algorithm, developed and introduced in the mid-1990s, have been proposed, especially in the last two decades. In this chapter, a systematic literature review about those variants and improvements is made, which also covers the hybridisation and parallelisation of the algorithm and its extensions to other classes of optimisation problems, taking into consideration the most important ones. These approaches and improvements are appropriately summarised, organised and presented, in order to allow and facilitate the identification of the most appropriate PSO variant for a particular application.

2.1 Introduction

The Particle Swarm Optimisation (PSO) technique was proposed and initially developed by the electrical engineer Dr Russell C. Eberhart and the social psychologist Dr James Kennedy. The method was described in two papers [10, 11] co-authored by those two authors and published in 1995, one of them having as its title the exact name of the technique they proposed.

This technique had (and still has) a deep connection with some social relations, concepts and behaviours that emerged from a computational study and simulation of a simplified social model of a bird flock seeking for food conducted by those authors, and it belongs to

the so-called swarm intelligence, an important and extensive research area within natural computing.

The PSO method is based on the premise that the knowledge lies not only in the social sharing of information among generations but also between elements of the same generation. Although PSO has some characteristics that, in some sense and to a certain extent, have some similarity to those found in other population-based computational models, such as Genetic Algorithm (GA) and other evolutionary computing techniques, it has the benefit of being relatively simple, and its algorithm is comparatively easy to describe and implement.

In fact, its simplicity and apparent competence in finding optimal solutions in complex search spaces led the PSO algorithm to become well known among the scientific community, which contributed to its study and improvement. Thus, many approaches were suggested and different applications were tested with it, especially over the past decade. This review is intended to summarise all the main developments related to the PSO algorithm, from its original formulation up to current developments.

This review is organised as follows: Section 2.2 introduces the original PSO approach suggested by Eberhart and Kennedy [10, 11]. Section 2.3 presents the most important parameter modifications and the main topological neighbourhood structures used with PSO. In Section 2.4, several PSO variants and its applications are presented. Subsequently, Section 2.5 introduces a number of hybrid algorithms resulting from combinations of PSO with other artificial intelligence tools. Finally, the last section presents some concluding remarks.

2.2 Particle Swarm Optimisation

The PSO computational method aims to optimise a problem iteratively, starting with a set, or population, of candidate solutions, called in this context a swarm of particles, in which each particle knows the global best position within the swarm (and its corresponding value in the context of the problem), along with its individual best position (and its fitness value) found so far during the search process in the problem's solution space.

At each iteration, the velocity and position of each particle in the swarm, represented by d-dimensional vectors, are influenced by the individual and the collective knowledge, which directs the repeated flights of the particles over the space of possible solutions to the problem in search of the optimum, until a suitable stopping criterion is satisfied.

The velocity of each particle i in the swarm, at every iteration t, is updated according to the following equation [3]:

$$\vec{V}\,{}^{i}_{t+1} = \vec{V}\,{}^{i}_{t} + \varphi_1 R_{1t}^{i}(\vec{p}\,{}^{i}_{t} - \vec{x}\,{}^{i}_{t}) + \varphi_2 R_{2t}^{i}(\vec{g}_{\,t} - \vec{x}\,{}^{i}_{t}), \tag{2.1}$$

where φ_1 and φ_2 are real acceleration coefficients known respectively as cognitive and social weights, which control how much the global and individual best positions should influence the particle's velocity and trajectory.

In the original PSO algorithm [11], both φ_1 and φ_2 are equal to 2, making the weights for the social and cognition parts, on average, equal to 1.

In multimodal problems, where multiple areas of the search space are promising regions, the fine-tuning of these parameters is even more critical to avoid premature convergence. Premature convergence happens when some particle finds a local best position in the swarm that is not the global best solution to the problem. Other particles will then mistakenly

fly towards it, without exploring other regions of the search space. In consequence, the algorithm will be trapped into that local optimum and will converge prematurely.

R_1 and R_2 are uniformly distributed d-dimensional random vectors, which are used to maintain an adequate level of diversity in the swarm population. Finally, $\vec{p}\,{}_t^{\,i}$ and $\vec{g}_{\,t}$ are, respectively, the personal or individual best position of particle i at iteration t, and the current global best position of the swarm.

In turn, the position of each particle i, at every iteration t, varies according to the following equation [3]:

$$\vec{x}\,{}_{t+1}^{\,i} = \vec{x}\,{}_t^{\,i} + \vec{V}\,{}_{t+1}^{\,i}. \tag{2.2}$$

Note that $\vec{x}\,{}_0^{\,i}$ and $\vec{V}\,{}_0^{\,i}$ can be generated using a uniformly distributed random vector, whereas the particle's best personal position should be initialised by its initial position, i.e., $\vec{p}\,{}_0^{\,i} = \vec{x}\,{}_0^{\,i}$.

The information about the best personal position (and its fitness value) then flows through the imaginary connections among the swarm of particles, making them move around in the d-dimensional search space until they find the best position that fulfils all the problem's constraints.

These stochastic changes towards the $\vec{p}\,{}^{\,i}$ and $\vec{g}$ positions are conceptually similar to the crossover (or recombination) operation, which is the main exploration operation used by GA. However, in PSO, this operation is not necessarily applied by using a random probability.

The PSO algorithm has some advantages when compared to other continuous optimisation techniques; for instance: (i) it does not make assumptions on the continuity and differentiability of the objective function to be optimised; (ii) it does not need to compute the gradient of the error function; and (iii) it does not need good initial starting points or deep a priori knowledge about the most promising areas of the search space.

Besides that, PSO is a problem-independent algorithm; i.e., it can be used in a wide range of applications, since the only information that is needed to know to run the algorithm is the fitness evaluation of each candidate solution (and possibly the set of constraints of the problem).

The PSO algorithm has become better known over time, leading to other studies that extended its original formulation. Many variants have been suggested, such as the adoption of different communication structures (such as the use of ring and star topologies, often referred to as lbest models) as alternatives to the original approach (gbest model), wherein all particles are connected with each other [12–14].

The Gbest and Lbest Models

A gbest model swarm, with s particles, is formally defined as:

$$\hat{y}_{\,t} \in \left\{\vec{p}\,{}_t^{\,1}, \vec{p}\,{}_t^{\,2}, \ldots, \vec{p}\,{}_t^{\,s}\right\} \mid f(\hat{y}_{\,t}) = \min\Big(\left\{f(\vec{p}\,{}_t^{\,1}), f(\vec{p}\,{}_t^{\,2}), \ldots, f(\vec{p}\,{}_t^{\,s})\right\}\Big), \tag{2.3}$$

where $\hat{y}$ denotes the position of the best particle in the entire swarm in a d-dimensional search space, also known as the target particle.

In this model, the information about the new positions found by any particle in the swarm is shared among all the others particles, which turns $\hat{y}$ into a kind of magnet, making all the particles converge to its position.

On the other hand, in a lbest model, a neighbourhood of size l is created for each particle in the swarm. In this view, the lbest model is formulated as below:

$$\hat{y}_t \in N_i \mid f(\hat{y}_t) = \min\Big(\{f(\vec{a})\}\Big),\ \forall \vec{a} \in N_i, \tag{2.4}$$

where N_i is the set of particles neighboring particle i (in which particle i may or may not be included).

This means that, instead of sharing the information among all the particles in the swarm, the lbest model restricts the knowledge to the particles that are neighbouring each other. When l is set to be equal to s, the lbest model is equivalent to the gbest model.

The selection of the neighbourhood of each particle can be defined by each index i; however, it can also be defined by the distance between them. In this case, the set N_i can be time-varying.

2.3 Modifications to the Particle Swarm Optimisation

Other different aspects of the original version of PSO have also been modified, and many variants have been proposed to address different kinds of problems; e.g., a discrete binary version of PSO [15] that is useful for combinatorial optimisation problems, such as the travelling salesman problem [16] and task scheduling problems [17, 18].

Over time, PSO gained even more attention, and thus, more research was being done on it (see, e.g., [19, 20] for an analysis of the trajectory and velocity of each particle during the execution of the PSO algorithm). This led many researchers to begin noticing problems with the original version of PSO, such as premature convergence (especially in multimodal domains) or performance issues (see, e.g., [21], wherein the number of fitness evaluations is reduced by using an estimated fitness value for each particle).

Many different approaches were suggested, and some were proven to be equivalent to the original PSO algorithm, leading to the same results. These changes were mainly in the population architecture and in the way of computing the next velocity of each particle in order to improve the efficacy and effectiveness of the search process and reduce the loss of diversity. In-depth studies were done to tune the parameters and to control velocity explosion (since the motion update equations usually tend towards infinity), stability and convergence [22].

2.3.1 Algorithm Convergence Improvements

The Inertia Weight Parameter

In 1998, Shi and Eberhart [23] introduced the notion of the inertia weight, ω, of a particle. This coefficient controls the local and global search ability, determining how much influence the previous velocity should have on the current particle's movement.

With this parameter, the velocity update equation (Equation (2.1)) is changed to:

$$\vec{V}^{\,i}_{t+1} = \omega\vec{V}^{\,i}_{t} + \varphi_1 R_1{}^{i}_{t}(\vec{p}^{\,i}_{t} - \vec{x}^{\,i}_{t}) + \varphi_2 R_2{}^{i}_{t}(\vec{g}_t - \vec{x}^{\,i}_{t}). \tag{2.5}$$

Most of the PSO algorithm variants developed since then include this coefficient. This is why the algorithm with this improvement is commonly referred to as the Standard Particle Swarm Optimisation (SPSO).

Note that the original PSO velocity update equation can be obtained when $\omega = 1$.

Van den Bergh [24] stated a strong relationship between φ_1, φ_2 and ω, which can be modelled by the following inequality:

$$\omega > \frac{1}{2}(\varphi_1 + \varphi_2) - 1. \tag{2.6}$$

When a high value is set for ω, the algorithm gives more importance to the particles' self-knowledge, rather than the swarm's knowledge (i.e., the other particles' knowledge). On the other hand, a small inertia weight prevents the algorithm from converging to a local optimum, acting as a jumping out function. However, too many jumps will progressively worsen the algorithm's properties, making it similar to a stochastic search [25].

As stated in [23], ω can be a positive constant (within the range $[0.9, 1.2]$), but also a function of time (where time corresponds to the iteration number, t), or even a random number [26].

Unfortunately, and due to the lack of knowledge of the search process, it is difficult or impossible to develop a mathematical model to adjust the inertia weight dynamically [27]. Therefore, typically, to better control exploration and exploitation of the search space, ω is changed from 0.9 ($\omega_{\max}$) to 0.4 ($\omega_{\min}$) using a negative linear function of time [28, 29], such as:

$$\omega(t) = \omega_{\max} - \frac{\omega_{\max} - \omega_{\min}}{t_{\max}} \times t. \tag{2.7}$$

As Chatterjee and Siarry suggested [30], the inertia weight parameter can also be changed using a non-linear time-dependent function, such as:

$$\omega(t) = \left(\frac{t_{\max} - t}{t_{\max}}\right)^{\upsilon} (\omega_{\max} - \omega_{\min}) + \omega_{\min}, \tag{2.8}$$

where $t_{\max}$ is the maximum number of iterations and υ is the non-linear modulation index chosen by the user/researcher. According to those authors, $\upsilon \in [0.9, 1.3]$ is usually satisfactory.

Changing the particles' momentum using a linear or a non-linear time-varying approach was proven to be the best rule of thumb in several applications, since the compromise between global and local searching throughout the course of the search process is critical to the success of the algorithm. That is, on its initials stages, the algorithm performs a fast initial exploration of the search space, but gradually becomes more focused around the best solution found until that point. This type of strategy is similar to the cooling schedule used in the Simulated Annealing (SA) algorithm.

Shi and Eberhart then suggested a fuzzy adaptive PSO algorithm [27] to better adapt the inertia weight to the search process. As the name suggests, a fuzzy system was implemented to improve the performance of the PSO by dynamically adjusting the inertia weight based on the global best position's distance from an optimum.

In their benchmark tests, the fuzzy adaptive strategy was able to improve the performance of the PSO algorithm when compared to the use of a time-varying inertia weight parameter.

The PSO with inertia weight is considered a canonical PSO algorithm, since the search process runs iteratively in a region that is defined by each particle's previous best position and velocity, the best previous successful positions of any of its neighbours and the particle's current position.

The Constriction Factor

In 1999, Maurice Clerc suggested the use of a constriction factor [14] to help the PSO algorithm solve optimisation problems faster, ensuring the convergence of the algorithm by making a trade-off between exploration and exploitation, affecting with this the particles' trajectories around possible candidate solutions in the search space [14, 31].

This constriction factor is given by:

$$K = \frac{2}{\left|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}\right|}, \tag{2.9}$$

where $\varphi = \varphi_1 + \varphi_2$ and $\varphi > 4$. Thus, Equation (2.1) may be written as:

$$\vec{V}\,^{i}_{t+1} = K\Big[\vec{V}\,^{i}_{t} + \varphi_1 R_1\,^{i}_{t}(\vec{p}\,^{i}_{t} - \vec{x}\,^{i}_{t}) + \varphi_2 R_2\,^{i}_{t}(\vec{g}_{\,t} - \vec{x}\,^{i}_{t})\Big]. \tag{2.10}$$

When the constriction factor is used with PSO, typically $\varphi = 4.1$, and thus $K \approx 0.7298$.

Eberhart and Shi [28] compared the constriction factor with the inertia weight. These authors concluded that better quality solutions could be obtained with the constriction factor method, although mathematically the constriction factor and the inertia weight are equivalent.

On the other hand, Eberhart and Shi [28] used the constriction factor while limiting the maximum velocity, since, when running a PSO algorithm without imposing restrictions to the velocities, these may rapidly increase within a few iterations to unacceptable levels, tending towards infinity. Basically, if $\vec{V}\,^{i}_{t+1}$ exceeds $\vec{V}_{\max}$ (defined by the user/researcher) in (2.10), then $\vec{V}\,^{i}_{t+1} = \vec{V}_{\max}$.

$\vec{V}_{\max}$ controls the global exploration ability of the swarm's particles. Thus, if $\vec{V}_{\max}$ is too high, particles might overfly reasonable potential solutions (prioritising, in this way, the global exploration of the search space). However, if $\vec{V}_{\max}$ is too small, there will be diversity loss problems; that is, particles may not explore sufficiently the search space, and can be stuck in a local optimum.

Using five non-linear benchmark functions, those authors found that, when $\vec{V}_{\max} = \vec{x}_{\max}$, the results improved significantly when compared to Clerc's constriction factor K. However, there is a drawback: the need to know beforehand an approximation for the location of the global best position in order to limit $\vec{V}$.

Kar and his collaborators [32] combined the inertia weight parameter and the constriction factor to overcome the premature convergence and the stagnation problem (refer to Section 2.3.3), and thus improve the effectiveness and efficacy of the algorithm in a multidimensional search space. With this, the velocity is updated as follows:

$$\vec{V}\,^{i}_{t+1} = K\Big[\omega\vec{V}\,^{i}_{t} + \varphi_1 R_1\,^{i}_{t}(\vec{p}\,^{i}_{t} - \vec{x}\,^{i}_{t}) + \varphi_2 R_2\,^{i}_{t}(\vec{g}_{\,t} - \vec{x}\,^{i}_{t})\Big]. \tag{2.11}$$

It was reported by those authors that updating each particle's velocity according to (2.11) produced better exploration and exploitation of the search space, along with faster convergence, for the test suite used.

Convergence issues were the most reported problem related to the PSO algorithm. In order to lessen this problem, new parameters were introduced into PSO and different variants

were suggested, including hybrid variants, as can be seen in Figure 2.1. Although some strategies to prevent premature convergence have not yet been mentioned, they were included in this figure for completeness. The reader is referred to the next sections for a description of the remaining approaches.

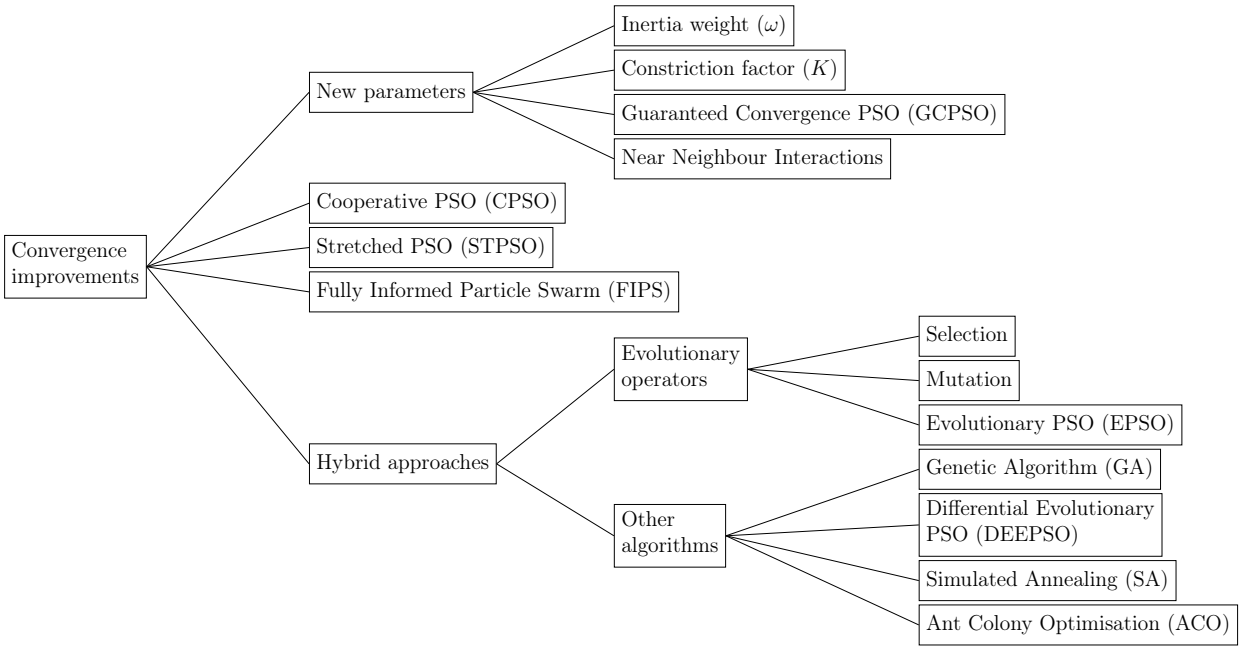


Figure 2.1: Summary of the most important convergence improvements developed for PSO.

2.3.2 Neighbourhoods

Static Neighbourhood

Simultaneously with the previously mentioned improvements in the PSO algorithm, some other different neighbourhood architectures were developed, in order to mimic the sociological phenomenon that an individual indirectly shares information with other people located around her/him.

In 1999, Kennedy reviewed and tested some of them [13], including circle/ring, star, all-connected-to-all and random architectures. These are known as static architectures, because the neighbourhood does not change throughout the algorithm's execution.

Because neighbourhood architectures produced different results when they were tested with different functions, the optimal pattern of connectivity between particles depended on the problem to be solved. For example, with a multimodal function, the star topology produced the best results, although the all-connected-to-all architecture performed better with unimodal functions.

Besides that, Kennedy [13] also concluded that the PSO with a large neighbourhood would perform better for simple problems, whereas small neighbourhoods should be used on complex problems.

Later on, in 2002, Kennedy and Mendes [33] synthesised all the population architectures developed so far: all-connected-to-all, pyramid, toroid, ring and star. They found that the

best and the worst population architectures (based on consistency, performance, number of iterations and standard deviations from the known global best position) were, respectively, the toroid and the all-connected-to-all topologies (the last being the topology of the original PSO algorithm).

Dynamic Neighbourhood

Meanwhile, Suganthan [34] proposed some improvements to the PSO algorithm, such as gradually increasing the local neighbourhood based on a computed radius for each particle.

If any particle is within the radius of another one, then they become neighbours and exchange information between them. As time goes by, this radius gradually becomes wider, until the swarm is fully connected.

The selection of the neighbourhood is, thus, based on the distance to each particle, rather than its indices, as occurs in the static neighbourhood topologies. These forms of neighbourhood organisation are called spatial topologies.

Suganthan [34] also suggested a gradual adjustment of the magnitude of the search in the search space by changing the values of the acceleration coefficients and the inertia weight during the course of the algorithm. Therefore, the parameters' values are changed using the following equations:

$$\begin{aligned} \omega &= \omega^{\infty} + (\omega^{0} - \omega^{\infty})\,(1 - t/t_{\max}), \\ \varphi_1 &= {\varphi_1}^{\infty} + ({\varphi_1}^{0} - {\varphi_1}^{\infty})\,(1 - t/t_{\max}), \\ \varphi_2 &= {\varphi_2}^{\infty} + ({\varphi_2}^{0} - {\varphi_2}^{\infty})\,(1 - t/t_{\max}), \end{aligned} \tag{2.12}$$

where the superscripts ∞ and 0 denote the final and the initial values of the parameters, respectively. In the tests carried out by this author, the initial value for ω was 0.95 and the final 0.2, whereas φ_1 and φ_2 had their values changed from 3 to 0.25 [34].

Suganthan [34] compared, for a set of test functions, his approach with the time-varying inertia SPSO algorithm (φ_1 and φ_2 were kept constant) and reported an improved performance when the parameters were changed according to (2.12).

In 2000, Kennedy proposed another approach for the lbest PSO, based on the spatial neighbourhood and on the ring neighbourhood topology, called social stereotyping [35].

The designation of this approach emerged, again, from social-psychological concepts, in this case, the concept of stereotyping, where people are grouped according to, among other things, their social and physical characteristics, qualities, beliefs and opinions.

This social process often happens when people frequently interact with each other, becoming more and more similar, forming their opinions and making decisions based on the groups that they identify with.

As humans converge to the stereotypical behaviours and beliefs of the groups that they belong to, particles' trajectories will be changed based on the region of the search space that they are in.

Each restricted search region of the search space is called a cluster. To constitute clusters in the search space, several particles are chosen as group leaders, called cluster centres or centroids. Then, the rest of the particles are grouped in a cluster based on the distance to each centre.

The PSO algorithm is modified so that the cognitive component (i.e., the previous individual particle's best position) or the social component (i.e., the best previous position in

the neighbourhood), or both, are replaced by the appropriate cluster centroid [24]. Thus, Kennedy [35] proposed three strategies to calculate the new velocity of each particle:

$$\begin{aligned}
\vec{V}\,^{i}_{t+1} &= \omega\vec{V}\,^{i}_{t} + \varphi_1 R_1\,^{i}_{t}(\vec{c}\,^{j}_{t} - \vec{x}\,^{i}_{t}) + \varphi_2 R_2\,^{i}_{t}(\vec{g}_{t} - \vec{x}\,^{i}_{t}),\\
\vec{V}\,^{i}_{t+1} &= \omega\vec{V}\,^{i}_{t} + \varphi_1 R_1\,^{i}_{t}(\vec{p}\,^{i}_{t} - \vec{x}\,^{i}_{t}) + \varphi_2 R_2\,^{i}_{t}(\vec{c}_{t} - \vec{x}\,^{i}_{t}),\\
\vec{V}\,^{i}_{t+1} &= \omega\vec{V}\,^{i}_{t} + \varphi_1 R_1\,^{i}_{t}(\vec{c}\,^{i}_{t} - \vec{x}\,^{i}_{t}) + \varphi_2 R_2\,^{i}_{t}(\vec{c}_{t} - \vec{x}\,^{i}_{t}),
\end{aligned} \tag{2.13}$$

where $\vec{c}\,^{j}_{t}$ is the position of the centroid of the cluster j at the iteration t, and $\vec{c}_t$ is the centroid of the best particle selected from the neighbourhood.

Although it has a higher computational cost, and therefore, a longer execution time when compared to the original PSO, the first equation of (2.13) performed better than the standard velocity update equation.

Near Neighbour Interactions

Veeramachaneni and his collaborators [36, 37] proposed a simple, effective way to update each particle's velocity dimension, motivated by the convergence behaviour issues detected in the PSO algorithm, especially in multimodal optimisation problems.

They developed an expression named Fitness-Distance-Ratio (FDR) that chooses the neighbourhood of each particle dimension based on the relative fitnesses of other particles in the neighbourhood:

$$FDR = \frac{f(\vec{P}\,^{j}_{t}) - f(\vec{x}\,^{i}_{t})}{|(\vec{P}\,^{j}_{t})_d - (\vec{x}\,^{i}_{t})_d|}, \tag{2.14}$$

where $\vec{P}\,^{j}_{t}$ is the prior best position of a particle called the nbest particle that maximises the FDR measure. Then, for each particle i, at every iteration t, each velocity dimension d is changed according to the following equation:

$$\begin{aligned}
(\vec{V}\,^{i}_{t+1})_d = \omega(\vec{V}\,^{i}_{t})_d + \varphi_1 R_1\,^{i}_{t}\left((\vec{p}\,^{i}_{t})_d - (\vec{x}\,^{i}_{t})_d\right) + \varphi_2 R_2\,^{i}_{t}\left((\vec{g}\,^{i}_{t})_d - (\vec{x}\,^{i}_{t})_d\right)\\
+ \varphi_3 R_2\,^{i}_{t}\left((\vec{P}\,^{i}_{t})_d - (\vec{x}\,^{i}_{t})_d\right),
\end{aligned} \tag{2.15}$$

where φ_3 is the deviation acceleration coefficient that corresponds to the importance, given by the particle, to the best experience of the best nearest neighbour.

Using this approach, besides the best position discovered so far, the velocity of each particle is also influenced by the previous positions visited by its neighbours.

Veeramachaneni et al. [36] reported that, although PSO performed well in the initial iterations of the benchmark test functions considered, overall results indicate that the FDR approach performed better in terms of convergence and thus in terms of the number of iterations.

The different PSO architectures can be grouped into static neighbourhoods (in which the neighbourhood does not change during the execution of PSO) and dynamic neighbourhoods (where the neighbourhood changes according to, e.g., the number of iterations or the distance among particles in the search space), as shown in Figure 2.2. The reader is referred to Subsection 2.4.5 for a description of the niching and speciation strategies.

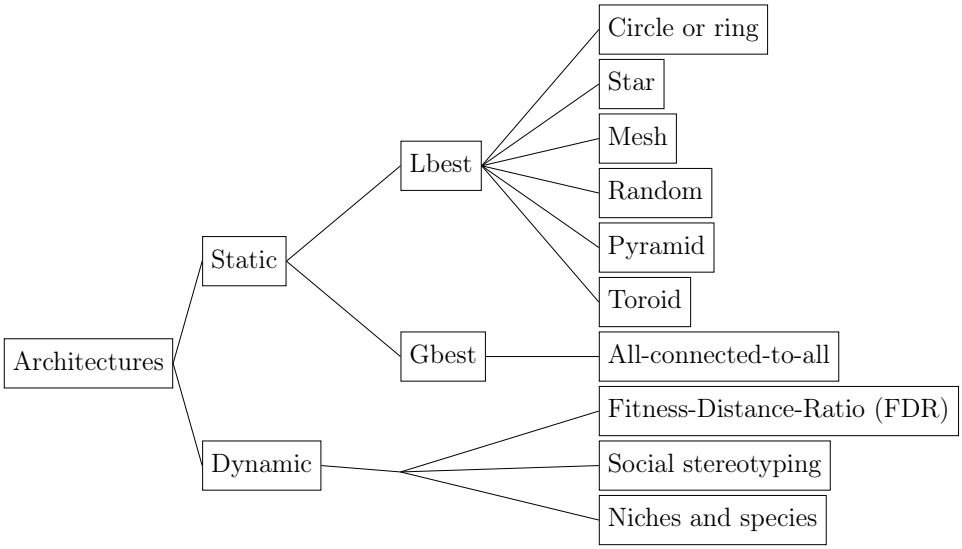


Figure 2.2: Summary of the most important architecture strategies developed for PSO.

2.3.3 The Stagnation Problem

Van den Bergh [38] noticed a property that affected all gbest variants of the SPSO algorithm developed until then.

If a particle's position is the same as the global best position, i.e., if $\vec{x}_t^{\,i} = \vec{p}_t^{\,i} = \vec{g}_t$, then the velocity in Equation (2.5) will only depend on $\omega\vec{V}_t^{\,i}$. This means that the particle will only leave this point if its previous velocity and ω are non-zero.

Otherwise, eventually, all particles will stop moving, leading to premature convergence of the algorithm to a position that is not guaranteed to be the global best position or a local optimum, but only the best position so far found by the particles in the swarm. This problem is known as the stagnation problem.

To solve this problem, van den Bergh [38] proposed a new algorithm, called Guaranteed Convergence Particle Swarm Optimisation (GCPSO), by inserting a new parameter μ into the SPSO algorithm, which denotes the index of the global best particle in the swarm.

Thus, the velocity and position update equations for the global best particle in the swarm are respectively changed by the following equations:

$$\begin{cases} \vec{V}_{t+1}^{\,\mu} = -\vec{x}_t^{\,\mu} + \vec{g}_t + \omega\vec{V}_t^{\,\mu} + \rho(t)(1 - 2R_2{}_t^{\,i}), \\ \vec{x}_{t+1}^{\,\mu} = \vec{g}_t + \omega\vec{V}_t^{\,\mu} + \rho(t)(1 - 2R_2{}_t^{\,i}). \end{cases} \tag{2.16}$$

The term $-\vec{x}_t^{\,\mu}$ resets the particle's position to the global best position and $\omega\vec{V}_t^{\,\mu}$ sets the search direction; $\rho(t)$ is a function that defines the diameter of the search area surrounding the global best position that will be randomly searched [39].

This significant change was used in several PSO variants (see, e.g., [39–42]).

However, on multimodal functions, the GCPSO algorithm has a higher probability of finding poor solutions when compared to PSO, due to faster convergence of the best particle towards a local extremum. Peer and his collaborators [43] studied this problem for the lbest models.

Nevertheless, as this situation is unlikely to occur, most of the authors do not consider this approach when updating the velocity and the position of the best particle in the swarm.

2.4 Particle Swarm Optimisation Variants

2.4.1 Cooperative Particle Swarm Optimisation

Due to the similarities between GA and PSO algorithms, some researchers started to propose PSO variants that combined the PSO algorithm with the operations used in GA.

An example of this is the Cooperative Particle Swarm Optimisation (CPSO), a PSO variant proposed by van den Berg and Engelbrecht [44] and improved later by the same authors [45]. The CPSO algorithm incorporates the concept of cooperation used in GA, wherein all subpopulations have to cooperate by contributing and exchanging information.

They suggested that this concept can also be applied to PSO by using a number of swarms for each dimension, instead of having only one for all dimensions. Thus, each subpopulation has only to optimise a one-dimensional vector. Although this approach seems simple, some changes on the original algorithm have to be made, especially to the evaluation of the objective function, which still requires a d-dimensional array as input.

Thus, a context vector was used to overcome the problem of the objective function evaluation. This vector is built at every iteration and has a component from each best particle's dimension. Then, for each component, if the new value is better than the previous one, that specific component of the context vector is updated (and so the best individual fitness value).

The first variant splits the search space into exactly d subspaces [44]. On the other hand, and motivated by the fact that components may be correlated, in the $CPSO\text{-}S_k$ algorithm, proposed later by van den Bergh and Engelbrecht [45], the search space is divided into k subspaces, where $k \leq d$, which makes it a generalisation of the CPSO algorithm.

The $CPSO\text{-}S_k$ converges to the local optima of the respective subspaces, which makes it more propitious to be trapped into local optima. However, according to those authors, it has faster convergence when compared to PSO.

PSO, on the other hand, is more unlikely to be trapped into local optimum positions when compared to the $CPSO\text{-}S_k$ algorithm, because the optimisation process considers the dimensions as a whole.

Thus, $CPSO\text{-}H_k$, a hybrid approach using $CPSO\text{-}S_k$ and PSO, was suggested by van den Bergh and Engelbrecht [45] to take advantage of the proprieties of both algorithms, resulting in a fast one with an improved local escape mechanism.

In an overall assessment, the $CPSO\text{-}S_k$ and $CPSO\text{-}H_k$ algorithms performed better than PSO both in terms of quality of the solutions found and performance [45], especially when the dimensionality of the problem increases.

Two Steps Forward, One Step Back

Before getting into the details of the $CPSO\text{-}S_k$ and $CPSO\text{-}H_k$ [45] algorithms, van den Bergh and Engelbrecht [44] stated one problem with PSO, which they named two steps forward, one step back. They found that, at each iteration, PSO changes the elements of the d-dimensional vector, making some components move close to the optimal solution, although

others can move away from it. Thus, PSO can accept a new candidate solution if its fitness value is lower than the previous one (when considering minimisation problems).

In their paper, they showed an example of this weakness of PSO with a vector with three components, wherein one component already had the optimal value, but its value changed in the next iteration to a poor one. Despite that, the other two components improved, and so did the fitness value.

In this case, two components improved, although one did not, taking the algorithm two steps forward and one step back. To overcome this problem, van den Bergh suggested evaluating the fitness function as soon as a component changes, while keeping constant the other $d-1$ components with the values of the previous iteration.

2.4.2 Adaptive Particle Swarm Optimisation

In 2009, one important approach for solving both unimodal and multimodal functions effectively, as well as improving the search efficacy and the converge speed of PSO while preserving premature convergence, was proposed by Zhan et al. [46].

The Adaptive Particle Swarm Optimisation (APSO) presented by those authors defines four evolutionary states for the PSO algorithm: exploration, exploitation, convergence and jumping out, according to the evaluation of the swarm's distribution and each particle's fitness. Thus, for each state, different strategies can be applied, such as parameter adaptation.

The swarm's distribution can be assessed by the mean distance of each particle to all other particles using the following Euclidean metric:

$$D_t^i = \frac{1}{s-1} \sum_{j=1, j\neq i}^{s} \sqrt{\sum_{i=1}^{d} \left(\vec{x}_t^{\,i} - \vec{x}_t^{\,j}\right)^2}, \tag{2.17}$$

where s is the size of the swarm and d is the number of dimensions.

Then, an evolutionary factor, e_f, is computed by:

$$e_f = \frac{D_g - D_{\min}}{D_{\max} - D_{\min}} \in [0,1], \tag{2.18}$$

where $D_{\max}$ and $D_{\min}$ are respectively the maximum and minimum distances among the particles, and D_g is the value of D_t^i of the globally best particle in the swarm.

Based on this factor, the algorithm can be then classified in one of the evolutionary states. For example, a medium to substantial value of e_f indicates the exploration state, while a shrunk value of e_f means exploitation. In turn, the convergence state happens when a minimum value of e_f is reached, and the jumping out state when the mean distance value for the best particle is significantly higher than the mean distance value for the other particles.

An adaptive e_f-dependent inertia weight was also suggested by the same authors and is given by:

$$\omega(e_f) = \frac{1}{1 + 1.5e^{-2.6e_f}} \in [0.4, 0.9],\ \forall\, e_f \in [0,1]. \tag{2.19}$$

Thus, when e_f is large (jumping out or exploration state), $\omega(e_f)$ makes the algorithm give more importance to the particle's self-knowledge, thereby benefiting the global search. On the other hand, when e_f is small (exploitation or convergence state), the swarm's knowledge is more relevant than the self-knowledge of each particle, giving priority to the local search.

The cognitive and social weights are also changed, according to the evolutionary state, and a Gaussian mutation operation is applied to the best particle in the swarm to enable it to jump out of a local optimum or to refine the global best solution.

If the new position found is better than the best particle's solution, the new one replaces the best particle's position. Otherwise, the worst particle's solution is replaced by this new position.

The velocity and the position of each particle are computed, and as usual, the PSO algorithm keeps iterating until the stopping criterion is met.

When tested with some unimodal and multimodal functions, APSO showed itself to be efficient at improving the convergence speed, and most importantly, at enhancing the accuracy of the algorithm when compared to other well known approaches.

2.4.3 Constrained Optimisation Problems

On the other hand, Parsopoulos and Vrahatis [47] proposed a method based on a penalty function and on the constriction factor for constraint handling with PSO. To the author's best knowledge, this was the first paper that proposed a method to use PSO to optimise constrained optimisation problems.

A Constrained Optimisation Problem (COP) can be transformed into an unconstrained problem by using a penalty function that penalises the objective function if the conditions on the variables are not held. Therefore, a single objective function is built and optimised using a standard unconstrained optimisation algorithm.

A penalty function, $F(\vec{x})$, can be defined as:

$$F(\vec{x}) = f(\vec{x}) + h_p(t)H_p(\vec{x}), \;\; \vec{x} \in S \subseteq \mathbb{R}^d, \tag{2.20}$$

where $f(\vec{x})$ is the original objective function to be optimised, $h_p(t)$ is a dynamic modified penalty value, and $H_p(\vec{x})$ is the penalty factor defined as:

$$H_p(\vec{x}) = \sum_{i=1}^{m} \theta\big(q_i(\vec{x})\big)\, q_i(\vec{x})^{\gamma(q_i(\vec{x}))}, \tag{2.21}$$

where $q_i(\vec{x}) = \max\big(\{0, g_i(\vec{x})\}\big)$ for $i = 1, \ldots m$, $\theta\big(q_i(\vec{x})\big)$ is a multi-stage assignment function, and $\gamma\big(q_i(\vec{x})\big)$ is the power of the penalty function. Note that although the equality constraints h_i were not considered, they can be transformed into two inequality constraints, such as $g_i(\vec{x}) \leq 0$ and $-g_i(\vec{x}) \geq 0$.

Although COPs can be transformed into unconstrained problems by using a penalty function, they require more parameters to be fine-tuned (in this case, $h_p(k)$, $\theta\big(q_i(\vec{x})\big)$ and $\gamma\big(q_i(\vec{x})\big)$) in order to prevent premature convergence.

Hu and Eberhart [48,49] proposed a more straightforward, brute-force method to optimise COPs, known as the Preservation of Feasible Solutions Method (FSM).

In their proposal, all feasible solutions found during the search process in the whole search space are preserved. After a stopping criterion is met, the optimal solution that fulfils all the problem's constraints may be found.

When these two methods are compared using the same problems, fine-tuning of the penalty function parameters may result in better average optimal solutions when compared

to FSM, but the choice of which constraint handling method to be used may be very problem dependent [50].

He et al. [51] introduced into PSO a different constraint handling method, called fly-back mechanism. The idea is simple: when a particle fly to a non-feasible region of the search space, its position is reset to the previous (feasible) position.

On the other hand, Sun et al. [52] proposed a more advanced approach, in which once a particle enters a non-feasible region, a new feasible position is computed by:

$$\vec{x}\,'^{\,i}_{t+1} = \vec{x}^{\,i}_{t} + \alpha \vec{V}^{\,i}_{t+1}, \tag{2.22}$$

where the coefficient α is a diagonal matrix whose diagonal values are set within the range of $[0, 1]$. Thus, if $\alpha_{ii} = 1$ for $i = 1, \ldots, d$, then this means that $\vec{x}^{\,i}_{t+1}$ is a feasible position.

If $\vec{x}^{\,i}_{t+1}$ is not in a feasible position, α must be adjusted to bring the particle back to a feasible position.

Sun et al. [52] suggest that α should be found by:

$$\min\Big(\prod_{k=1}^{m+2d} e^{\max(0,\, g_k(\vec{x}^{\,i}_{t}+\alpha\vec{V}^{\,i}_{t+1}))} \prod_{j=1}^{p} e^{\max(0,\, \mathrm{abs}(h_j(\vec{x}^{\,i}_{t}+\alpha\vec{V}^{\,i}_{t+1})))}\Big). \tag{2.23}$$

Note that the superscript $m + 2d$ on the first product symbol includes both the number of inequality constraints, as well as the search space's boundaries, that are transformed into two inequality constraints.

Then, the algorithm proceeds like the PSO algorithm until a stopping criterion is met.

Results show that this algorithm is suitable for solving COPs. However, it did not perform as well when the optimal values were at the boundaries of the search space.

2.4.4 Multi-Objective Optimisation

Initially, research on PSO was made considering only the optimisation of one function. However, in real-world problems, it is rare to have only a single objective to optimise, but multiple objectives that should be optimised simultaneously.

At first glance, the different functions can be optimised running the algorithm independently for each of them, but optimal solutions seldom are found, because the objectives may conflict with each other (e.g., price–quality relationship).

The multi-objective optimisation problems can be modelled as finding $\vec{x} \in S \subseteq \mathbb{R}^d$ that minimises $f(\vec{x}) = \big[f_1(\vec{x}), f_2(\vec{x}), \ldots, f_k(\vec{x})\big]^T$.

In most of the multi-objective optimisation problems, there is no single solution that simultaneously optimises each objective but a set of feasible solutions called Pareto optimal solutions, $\vec{y^*}$. In other words, there is no feasible vector $\vec{x}$ that would optimise some objective values without penalising at least one other objective value.

This set of feasible solutions forms the so-called Pareto front. The user/researcher is then responsible for choosing what he considers to be the best solution to the problem at hands.

This introduces a notion of dominance, namely the Pareto Dominance: a vector $\vec{u} = [u_1, u_2, \ldots, u_k]$ is said to dominate $\vec{v} = [v_1, v_2, \ldots, v_k]$ if $\forall i \in \{1, 2, \ldots, k\}, u_i \leq v_i \wedge \exists i \in \{1, 2, \ldots, k\} : u_i < v_i$.

Hu and Eberhart [53] proposed an approach to solving multi-objective optimisation problems with a PSO algorithm based mainly on the concept of Pareto optimally.

They presented a dynamic neighbourhood version of PSO, such that, at every iteration, each particle has a different neighbourhood than it had in the previous iteration.

Each particle's neighbourhood is chosen based on the distances from the current particle to the other particles in the fitness value space of the first objective function to be optimised.

Within its neighbourhood, each particle chooses the local best (lbest) particle, considering the fitness value of the second objective function.

The new $\vec{p}\,_t^{\,i}$ is only set when a new solution that dominates the current $\vec{p}\,_t^{\,i}$ is found.

Unfortunately, Hu and Eberhart only used two objective functions to describe their proposal and did not provide enough details on how the algorithm was implemented. Besides that, their proposal, in essence, only optimises one objective function, and nothing guarantees that the optimal solution for the second function is also the optimal solution for the first one.

Coello Coello and his collaborators [54,55], on the other hand, introduced the notion of an external (or secondary) repository, proposing a PSO variant called Multi-Objective Particle Swarm Optimisation (MOPSO). The external repository stores non-dominated vectors of particles' positions used to compute the velocity of each particle at each iteration (replacing $\vec{g}_t$ in (2.5)). This repository is dynamically chosen within each iteration. For example, if none of the elements contained in the external population dominates the new solution found, then such a solution is stored in the external repository. They also used a constraint handling mechanism to solve multi-objective constraint optimisation problems with PSO, and a mutation operator to ensure the diversity of the particles, to slow down the convergence speed and to prevent premature convergence to a local optimum.

The constraint handling mechanism can do one of two things if a particle goes beyond the boundaries: either set it to its corresponding boundary, or multiply its velocity by -1 in order to search in the opposite direction.

According to a certain probability, a mutation operator is applied to only one randomly chosen dimension of each particle by changing its value according to the current and total number of iterations, taking into account its boundaries, however. This was the first mutation operation proposed to solve optimisation problems with constraints using PSO.

The algorithm then proceeds as the standard PSO until a stopping criterion is met. The output of the algorithm is a Pareto front, which is built upon each iteration as a grid using the values of the external repository.

The MOPSO approach showed better results than other multi-objective evolutionary algorithms and required low computational time to run the algorithm.

These approaches were the first steps of the research on solving multi-objective parameter optimisation problems using PSO. The MOPSO algorithm was improved by Fieldsend [56] and later by Mostaghim [57].

2.4.5 Multimodal Function Optimisation

Simultaneously, efforts were made to extend the PSO algorithm for multimodal function optimisation; that is, for finding all the global best positions (and eventually other local optimal solutions) of an equation or system of equations.

This type of optimisation is especially useful for the decision makers, so that decisions can be made taking into account, e.g., physical and cost constraints, having, however, multiple optimal solutions at hand.

Due to the existence of multiple local and global optima, all these problems can not be solved by classical non-linear programming techniques. On the other hand, when using Evolutionary Algorithm (EA) and PSO, the optimum positions can be found faster than by traditional optimisation techniques [58].

However, PSO was designed to find only one optimum of a function, and so some changes are required. In fact, PSO can be applied multiple times on the same function to find all the desired minima. Nevertheless, it is not guaranteed that all will be found.

In this type of optimisation, fast convergence can sometimes lead to premature convergence, because PSO (or other EAs) may get trapped into local optima. Thus, it is important to maintain the population diversity before some goal is met.

At first glance, the lbest models can be thought of as potential candidates to find multiple solutions, in which each neighbourhood will represent a candidate solution. However, one particle can be in several neighbourhoods at the same time, causing all the particles in these neighbourhoods to converge to the same point in case that particle has the best fitness among all the points in the neighbourhoods it belongs to. Consequently, if that point is a local optimum, these neighbourhoods will be biased towards that position, making the algorithm converge prematurely.

Thus, many approaches to tackling this kind of problem have been suggested, and the most relevant will be described in the next subsections.

Objective Function Stretching

Multimodal function optimisation with PSO was first introduced by Parsopoulos et al. [58] with the main objective of finding a global minimum of a multimodal function, avoiding the algorithm being trapped into local optima. They named their approach as Stretched Particle Swarm Optimisation (STPSO).

To do so, they defined a two-stage transformation on the objective function that is applied to it as soon as a local optimum (minimum) is found, using a function stretching technique.

A function stretching ($H(\vec{x})$) acts as a filter, transforming the form of the original function in a more flatter surface yet highlighting possible global and local optimum positions.

As already said, this transformation is applied as soon as a local minimum is found, in order to repel the rest of the swarm from moving towards that position. After that, $f(\vec{x})$ is replaced by $H(\vec{x})$ and the PSO algorithm is applied until a specific stopping criterion is met.

Parsopoulos and Vrahatis [59] extended this approach to find all globally optimal solutions and showed that this new approach could be effective and efficient.

They defined a threshold, ϵ, related to the requested accuracy so that when the value of the objective function applied to the particle is lower than ϵ, this particle is pulled away from the swarm and a function stretching is applied at that point to avoid the rest of the swarm from moving towards that position.

After this transformation, a new particle is randomly added to the swarm, to replace the one that was isolated from it. Then, if the function value of the isolated particle is higher than the desired accuracy, a new sub-swarm is created (which is considered a niching technique), and a new instance of the algorithm is executed, although being conditioned to that search area.

The algorithm stops when the number of global minimisers reaches a known one, or when the number of global minimisers is unknown, at the maximum number of iterations.

Unfortunately, this stretching transformation (that can also be considered as a convergence acceleration technique) may create local minima that were not present in the original objective function. This may require some restarts of the PSO algorithm until a global minimum is found [60].

Thus, Parsopoulous and Vrahatis [61] improved their method again by introducing deflection – a technique that incorporates knowledge from previously detected minimisers into the objective function – and a better repulsion technique (which ensures that if a particle moves towards one of the detected local optima, it will be repelled away from it).

Nbest Technique

In 2002, Brits et al. [60] proposed a new PSO-based technique, known as neighbourhood best or nbest PSO, and showed its successful application in solving systems of unconstrained equations.

A system of equations with k equations can be transformed into one fitness function:

$$f(\vec{x}) = \sum_{i=1}^{k} |f_i(\vec{x})|, \tag{2.24}$$

where each equation is algebraically rewritten to be equal to zero. However, the formulation of the problem using this transformation fails when multiple solutions are present in the search space.

To overcome this problem, they redefined the objective function as the minimum of the fitness function with respect to other equations. That is, as in the example given by Brits and his collaborators, when a system of equations has three equations (A, B and C), the objective function is defined as the minimum of the combinations of those equations:

$$f(\vec{x}) = \min\big(\{f_{AB}(\vec{x}), f_{AC}(\vec{x}), f_{BC}(\vec{x})\}\big). \tag{2.25}$$

Thus, particles that are close to one of the solutions are rewarded and do not suffer any penalisation if they are still far from the global best particle.

The nbest technique uses a dynamic neighbourhood approach, based on the Euclidean distance between the particles, to change the biased information towards a single optimal solution.

It is noteworthy that the Euclidean distance is computationally intensive to calculate, and besides that, choosing the neighbourhood based on it led to undesirable convergence properties. Thus, later, Euclidean neighbourhood was abandoned.

After computing the Euclidean distance from each particle to each other one, the neighbourhood of each particle is defined and the centre of mass of the positions is kept as neighbourhood best, and the PSO algorithm proceeds normally until a stopping criterion is met.

The results presented by those authors showed that the nbest technique can find all globally best solutions. However, in real-world applications, the systems of equations to optimise are usually not limited to three equations, and frequently the number of them is much higher. Thus, in such cases, this solution may face performance issues as the number of combinations can increase rapidly.

Subpopulations and Multi-Swarm

Another strand for the neighbourhood structure of communication happens when some subpopulations are watching over the best local optimum. That is, when a local optimum is found, the original swarm is split. One fraction of the swarm remains to explore the local optimum, and the other continues the search on a different portion of the search space [62].

In natural ecosystems, animals live and reproduce in the same groups of their own species, called niches. Based on this idea, niching techniques were proposed and implemented successfully with GA and later with PSO.

This type of technique is most commonly used in multimodal search spaces, because groups of individuals can move simultaneously into different search space regions. Note that individuals can be grouped by similar fitness values, by their distance from others or other similarity criteria.

Brits et al. [39] suggested the first PSO niching technique, named NichePSO, for successfully locating multiple optimal solutions in multimodal optimisation problems simultaneously.

In their proposal, they used a main swarm and a number of sub-swarms, as well as two variants of the PSO algorithm, namely, GCPSO [38] and the cognition-only model proposed by Kennedy [63], where Equation (2.1) is changed to only include the cognitive weight; i.e.,

$$\vec{V}_{t+1}^{\,i} = \vec{V}_{t}^{\,i} + \varphi_1 R_{1\,t}^{\,i}(\vec{p}_{t}^{\,i} - \vec{x}_{t}^{\,i}), \tag{2.26}$$

thereby allowing each particle to perform a local search, preventing the situation in which all particles get pulled towards a single solution due to the influence of the best particle or particles in the neighbourhood.

The cognition-only PSO variant is run for one iteration in the main swarm. Particles are then grouped by a given accuracy threshold (similar to the one used by Parsopoulos and Vrahatis [47] in the constriction factor PSO approach), and then, for each sub-swarm, GCPSO is run.

After that, the sub-swarms that are too close can be merged and can absorb particles from the main swarm when they move into them. Finally, the algorithm checks in the main swarm for the need to split it in other swarms, and iterates until a stopping criterion is found (e.g., when it reaches a certain number of globally optimal solutions, when known).

Later, Engelbrecht [42] improved the NichePSO by changing the merging and absorption strategies that were proposed in the original approach. Schoeman and Engelbrecht [64] proposed a PSO approach (which can be considered as a sequential niching PSO) that uses an additional vector operation, namely, the dot product, to change the direction in which particles should be headed to; viz., towards an already located niche or to explore and search for a new niche. Shortly after that, the same authors [65] proposed a parallel vector-based approach wherein all particles are updated simultaneously.

Li [66] extended the FDR-PSO algorithm to multimodal optimisation problems by introducing two mechanisms in the original FDR-PSO: the memory-swarm and the explorer-swarm.

The memory-swarm saves the personal best positions found so far by the population. During its turn, the explorer-swarm saves the current state of the particles and is used to explore the search space.

The best positions in the memory-swarm are used as anchors, and as the algorithm runs, niches are created around the best positions, according to the fitness-Euclidean distance ratio between a particle's personal best and other personal bests of the particles in the population.

The fitness-Euclidean distance ratio technique is an improved version of FDR that has a scaling factor computed using the worst and best-fitted particles in the swarm.

Li et al. [67] split the population into species, according to the distances between the particles. Based on this idea and the ideas presented in [68, 69], Parrott and Li [70] incorporated the concept of speciation into the constriction factor approach of PSO for solving multimodal optimisation problems.

It is important to note that, although different terminology is used, both niching and speciation techniques group similar particles by a given criteria.

In the resulting species-based algorithm, the particles are dynamically and adaptively grouped into species around dominating particles called species seeds, each species being used to track an optimum point.

Li [71] also presented a niching, parameter-free algorithm with ring topology for multimodal optimisation, which is able to form stable niches across different local neighbourhoods.

Four variants of this lbest PSO niching algorithm with ring topology were also suggested by Li [71], two of them (r2pso and r3pso) with an overlapping ring topology; the other two variants, namely, r2pso-lhc and r3pso-lhc, being lbest PSO algorithms with a non-overlapping ring topology.

Recently, Yue et al. [72] improved the lbest PSO niching algorithm by including a Special Crowding Distance (SCD) for solving multimodal multi-objective problems and reported that the algorithm was able to find a more significant number of Pareto-optimal solutions when compared to other well known algorithms.

2.4.6 The Fully Informed Particle Swarm Optimisation

In 2004, Mendes et al. [73] introduced the Fully Informed Particle Swarm (FIPS) optimisation algorithm, because they were convinced that each particle should not be influenced only by the best particle among its neighbours, but all the neighbours must contribute to the velocity adjustment of each particle; i.e., the particles should be fully informed.

They integrated the constriction factor approach of PSO with a new velocity update equation, wherein the social component is not explicitly considered, given by:

$$\vec{V}_{t+1}^{\,i} = K\left(\vec{V}_{t}^{\,i} + \varphi\left(\vec{p}_{t}^{\,i} - \vec{x}_{t}^{\,i}\right)\right). \tag{2.27}$$

Typically $\varphi = 4.1$ and $K \approx 0.7298$. The particle's individual best position $\vec{p}_{t}^{\,i}$ is given by:

$$\vec{p}_{t}^{\,i} = \frac{\sum_{i=1}^{l} \sigma(i)\,\vec{\varphi}_i \times \vec{p}_{t}^{\,i}}{\sum_{i=1}^{l} \sigma(i)\,\vec{\varphi}_i}, \tag{2.28}$$

with

$$\vec{\varphi}_i = \vec{U}\left[0, \frac{\varphi_{\max}}{l}\right],\ \forall i \in \{1, \ldots, l\}, \tag{2.29}$$

where l is the number of particles in the population, and $\vec{U}$ is a function that returns a position vector generated randomly from a uniform distribution between 0 and $\varphi_{\max}/l$.

The function $\sigma(i)$ can return a constant value over the iterations, or as Mendes et al. [73] also did in their experiments, return the fitness value of the best position found by the particle i or the distance from that particle to the current particle.

Although in this variant all particles contribute equally for the change in the next velocity calculation, those authors also suggested a weighted version of the FIPS algorithm, in which contributions are given according to the fitness value of the previous best position or the distance in the search space to the target particle.

They were, in fact, right, since both FIPS variants performed well on the considered neighbourhood architectures (except on the all-connected-to-all), finding at all times the minimum of the benchmark functions. The weighted versions require an extra computational cost, and such cost may not be justified, since the unweighted version performed quite well in their study [73].

2.4.7 Parallel Implementations of Particle Swarm Optimisation

Besides being trapped into local optima, PSO has another problem: its performance becomes progressively worse as the dimensions of the problem increase [74]. To alleviate this problem, some approaches were suggested, such as the use of multiple processing units of a computer system to distribute processing among them, creating sub-swarms, and thus speeding up the execution of the algorithm.

As each sub-swarm can be thought to be independent, PSO maps well to the parallel computing paradigm. In this section, a survey of the most common approaches to Parallelised Particle Swarm Optimisation (PPSO) will be described.

For PPSO approaches, a multi-core Central Processing Unit (CPU) or a Graphics Processing Unit (GPU) can be used to process the tasks of each parallel sub-swarm, along with some mechanism to exchange information among them. The exchange of information can be made synchronously or asynchronously.

Synchronous exchange is made when particles of each sub-swarm are synchronised with each other, i.e., the particles wait for the others to move to the next iteration, leading to the same result as the sequential approach, although its processing is done in parallel. On the other hand, when the exchange of information is made asynchronously, the sub-swarms are independent of each other, and thus, at the end of an iteration, each particle uses the information available at the moment (especially the global best position information) to move to the next position.

In addition, different architectures can be used to control the exchange of information, such as master-slave (where there is one processing unit that controls the execution of the other processing units), fine-grained (in which the swarm is split into sub-swarms and arranged in a two-dimensional grid, wherein the communication is only made within the neighbours of each sub-swarm) and coarse-grained (where the swarm is also split into sub-swarms independent of each other; however, from time to time, they exchange particles between them) [31, 74, 75].

Gies and Rahmat-Samii [76] proposed the first PPSO. They reported a performance gain of eight-fold (when compared with sequential PSO) with the PPSO algorithm for finding the optimal antenna array design. The results of this first work about PPSO motivated other researchers, such as Baskar and Suganthan [77], who improved the performance of FDR-PSO [37] by introducing a novel concurrent approach, called CONPSO.

Three communication strategies were presented in [78, 79] by using the GA's migration technique to spread the gbest position of each sub-swarm to the others. In the first one, the best particle of each sub-swarm is mutated and migrated to another sub-swarm to replace the poorest candidate solutions. In the second strategy, on the other hand, although similar to the previous one, the exchange of information only happens in neighbour sub-swarms. Finally, the later solution is a hybrid between the first and the second strategy.

Schutte et al. [80, 81] used a synchronous master-slave architecture for a bio-mechanical system identification problem. All particles were evaluated using parallel processes; however, all processes had to finish in order to update the next velocities and positions of all particles. Additionally, they reported that the time required to solve the system identification problem considered was reduced substantially when compared to traditional approaches.

As stated by Schutte et al. [81], synchronous implementations of PPSO are easy to produce. Nevertheless, such implementations usually have a poor parallel efficiency, since some processing units may be idle. Due to this fact, Venter and Sobieszczanski-Sobieski [82] proposed a master-slave asynchronous implementation PPSO algorithm and compared it with a synchronous PPSO.

One can consider the fact that the behaviour of each particle depends on the information available (possibly not from all other sub-swarms) at the start of a new iteration as a drawback of asynchronous approaches. However, in the author's opinion, this can be negligible because, although particles may not have updated information about the best solution before moving to a next position in the search space, communication always exists between particles and sub-swarms. Thus, in further iterations, the information about the best position found so far will inevitably be shared.

Koh et al. [83] introduced a point-to-point communication strategy between the master and each slave processing unit in an asynchronous implementation of PPSO for heterogeneous computing conditions. This condition happens, e.g., when the number of parallel sub-swarms can not be equally distributed among the available processors. In this type of condition, a load balance technique is essential for the robustness of the algorithm.

The results obtained by Koh et al. [83] were compared to the algorithm presented by Schutte et al. [81], and showed that the asynchronous implementation performs better, in terms of parallel efficiency, when a large number of processors are used.

In 2007, McNabb et al. [84] introduced the MapReduce function for the PPSO. This function has two sub-functions: map and reduce.

On the one hand, the map function finds a new position, computes the velocity of the particle, evaluates the objective function on its position, updates the information of the personal best position and shares this information among all dependent particles. On the other hand, the reduce function receives the information and updates the global best position information.

This type of formulation allows the algorithm to be split into small procedures and easily balanced and scaled across multiple processing units, following the divide-and-conquer parallel approach.

Aljarah and Ludwig [85] proposed a PPSO optimisation clustering algorithm (MR-CPSO) based on the MapReduce approach. This parallel PSO-based algorithm showed efficient processing when large data sets were used.

Han et al. [86], in turn, included constraint handling in PPSO, whereas Gülcü and Kodaz [87] proposed a synchronous parallel multi-swarm strategy for PPSO.

In this multi-swarm approach, a population is divided into subpopulations: one master-swarm and several slave-swarms which independently run a PSO variant. However, the slave-swarms cannot communicate with each other, since communication is made through the master-swarm by migrating particles. The parallel multi-swarm algorithm also uses a new cooperation strategy, called Greed Information Swap [87]. This work was extended by Cao et al. [88] to include multi-objective optimisation.

Lorion et al. [89], in turn, proposed an agent-based PPSO that splits PPSO into sub-problems. There are two types of agents: one coordination agent and several swarm agents, which, similarly to the multi-swarm strategy, do not communicate with each other.

Then, a strategical niching technique is used to increase the quality gain. A fault tolerance (e.g., when a processing unit stops responding to requests) was also implemented, by either saving agent's state in other swarm agents or by using the coordination agent's information available at the moment about the failed agent.

Along with all these developments, some researchers suggested approaches that used a GPU instead of using a CPU, especially when the CUDA development kit of NVIDIA was released. GPUs are designed for image processing and graphics applications, although they have more processing capacity (since they have more processing elements) than CPUs.

Developing parallel algorithms on a GPU is far more complicated than the corresponding implementations on a CPU [90]. However, several studies have reported significant improvements in terms of execution time when a GPU implementation of the PPSO is compared to its corresponding implementation on a CPU (see, e.g., [91–94]).

A GPU-based fine-grained PPSO was proposed by Li et al. [67]. In turn, the performance of the Euclidean PSO, proposed by Zhu et al. [95], was improved by Dali and Bouamama [96], where a GPU-based parallel implementation of the original algorithm was presented.

Finally, it is also worth mentioning the distributed and load balancing versions of the PSO algorithm on GPU developed by using a grid of multiple threads [97] or distributed memory clusters [98], along with the OpenMP API.

2.5 Connections to Other Artificial Intelligence Tools

2.5.1 Hybrid Variants of Particle Swarm Optimisation

A PSO variant is called hybrid when the PSO algorithm is combined with other optimisation techniques, such as the operators used in GA (e.g., selection, crossover or recombination and mutation) and other population-based algorithms.

The objective of hybridization is to increase the quality of particles in a swarm and improve the effectiveness and efficiency of the algorithm. The PSO algorithm is known by its tendency to become trapped into local optima, which prevents it from exploring other regions of the search space. Combining PSO with other EAs can overcome this difficulty in escaping from local optimal solutions and suppress the inherent deficiencies of other algorithms with which it is hybridised.

Evolutionary Computation Operators

In 1998, Angeline [99] incorporated a selection mechanism into PSO similar to those used in more traditional evolutionary algorithms, thereby producing what is considered the first

hybrid PSO algorithm.

That mechanism compares the current fitness of each particle with the fitnesses of other particles, and the least fit score a point. Then, the population is sorted using this score.

Current positions and velocities of the worst half of the population are then replaced with the positions and velocities of the best half, leaving the personal best position unchanged. Thus, the selection process resets the low-scored particles to locations within the search space that have yielded better results.

It was shown that this truncation selection mechanism incorporated into PSO improves the performance of the algorithm significantly on most of the tested functions. The roulette wheel selection operator was also used by Yang et al. [100], wherein the best particles in the swarm are the more likely to be selected.

On the other hand, many researchers then suggested and reported good performance by combining PSO with crossover operators (see, e.g., [12,42]) and different mutation strategies, such as Gaussian and Cauchy mutations [40,101–103]. These researches were essentially motivated by the fact that PSO presents difficulty in finding optimal or near-optimal solutions for many complex optimisation problems, including multimodal function optimisation and multi-objective optimisation.

Mutation is a genetic operator, analogous to the biological mutation, which, with a certain probability, changes the value of $\vec{g}_t$ or the next particle's position from its current state, hoping to find a better solution, while maintaining the population diversity. This operation provides strong exploration and exploitation capabilities to the swarm and also prevents premature convergence to a local optimum.

For example, the Cauchy mutation operator can be implemented as follows [101]:

$$(\vec{g}_{t_1})_d = (\vec{g}_t)_d + \left((\vec{x}_{\max})_d - (\vec{x}_{\min})_d\right) \times \text{Cauchy}(0, \sigma), \tag{2.30}$$

where $(\vec{g}_t)_d$ and $(\vec{g}_{t_1})_d$ are, respectively, the current and the new values of the global best position for dimension d, and $(\vec{x}_{\max})_d$ and $(\vec{x}_{\min})_d$ are the upper and lower limits of the dimension d. Finally, σ is the scale parameter of the Cauchy mutation, which is updated as follows:

$$\sigma_{t+1} = \sigma_t - \frac{1}{t_{\max}}. \tag{2.31}$$

As can be seen, σ linearly decreases at each iteration, so that, in the first iterations, the exploration capability is stronger, while in the last ones, the exploitation ability is privileged. Naturally, this mutation operator can be applied to both gbest and lbest models, and often $\sigma_0 = 1$.

On the other hand, reproduction or breeding is the process of combining any two particles (chosen among the particles selected for breeding at a given breeding probability) and performing a crossover operation that generates two new particles based on the characteristics of their parents (which are replaced by those new particles). In their hybrid algorithm, Løvbjerg et al. [12] used an arithmetic crossover operator, so the position of each new child particle is computed as follows:

$$\begin{aligned} \vec{x}_{c_1\,t}^{\;i} &= r \times \vec{x}_{p_1\,t}^{\;i} + (1 - r) \times \vec{x}_{p_2\,t}^{\;i}, \\ \vec{x}_{c_2\,t}^{\;i} &= r \times \vec{x}_{p_2\,t}^{\;i} + (1 - r) \times \vec{x}_{p_1\,t}^{\;i}, \end{aligned} \tag{2.32}$$

where r is a uniformly distributed random value between 0 and 1, and the velocities are given by [12]:

$$\begin{aligned}\vec{V}_{c_1\ t}^{\ i} &= \frac{\vec{V}_{p_1\ t}^{\ i} + \vec{V}_{p_2\ t}^{\ i}}{\left|\vec{V}_{p_1\ t}^{\ i} + \vec{V}_{p_2\ t}^{\ i}\right|}\left|\vec{V}_{p_1\ t}^{\ i}\right|, \\ \vec{V}_{c_2\ t}^{\ i} &= \frac{\vec{V}_{p_1\ t}^{\ i} + \vec{V}_{p_2\ t}^{\ i}}{\left|\vec{V}_{p_1\ t}^{\ i} + \vec{V}_{p_2\ t}^{\ i}\right|}\left|\vec{V}_{p_2\ t}^{\ i}\right|.\end{aligned} \tag{2.33}$$

In the last two equations, the subscript c indicates the position or velocity of a child particle, while the subscript p identifies a parent particle.

These evolutionary computation operators aim to reduce the diversity loss in the swarm and can be combined with others. Despite usually slowing down the efficiency of the algorithm, they can produce better results, especially when faced with multimodal functions.

In 2002, Miranda and Fonseca [104] proposed an approach, denoted Evolutionary Particle Swarm Optimisation (EPSO), which merged the concepts of evolutionary computation with PSO. In their algorithm, the operations of replication (where each particle is replaced r times; usually $r = 1$), mutation (on the cognitive, social, and inertia weights), crossover and selection (before evaluation) were used to generate diversity and to enable the fittest particle to survive and propagate. This is analogous to the mechanism of survival of the fittest of natural selection, from the Darwinian theory of evolution [105].

Wang et al., in 2013, proposed the Diversity Enhanced Particle Swarm Optimisation with Neighbourhood Search (DNSPSO) [106], a PSO variant that includes a new diversity enhanced mechanism using a crossover operation, and a new neighbourhood search strategy.

The crossover operation is applied to each dimension of the current particle's position, by replacing it with the previous correspondent dimension where the particle was in the search space. This operation is, however, applied according to a uniform random number within the range $[0, 1]$ generated for each dimension. If the generated random number is lower than a predefined probability, the particle's position is recombined with the previous dimension. Otherwise, it remains unchanged.

This operation creates what those authors called a trial particle that replaces the current particle only if its fitness value is lower than the current fitness (for minimisation problems).

In turn, the neighbourhood search strategy interestingly combines the gbest and lbest models, creating two more trial particles, based on the gbest and the lbest information. This search strategy is applied according to a predefined probability, and it was developed to improve the exploration of the search space by the particles in the swarm.

Then, the current particle is replaced by the most fitted particle among the current particle, the trial particle derived from the gbest information and the one from the lbest information.

The results presented by those authors showed that the DNSPSO algorithm achieved better results when compared to other PSO variants, both in terms of the quality of the solutions found and performance.

PSO with Genetic Algorithms

On the other hand, PSO was also combined with GA. In GA, similarly to PSO, there is a population of potential candidate solutions. Each element of the population has chromosomes that are mutated, based on a certain probability, to maintain a certain level of population diversity and improve the solution.

Each iteration is called a generation, and the algorithm reflects the process of natural selection, wherein the best fit individuals are chosen for reproduction in order to produce the next generation (which is expected to be better than the previous one).

PSO is known for not being able to effectively avoid being trapped into local optima during the search process. However, the GA algorithm can be used, along with its operators, to reduce this weakness.

On the other hand, GA has a slower convergence speed when compared to PSO [100, 107]. These advantages and disadvantages motivated the researchers to develop optimisation algorithms that combine PSO with GA.

Robinson et al. [107] introduced the first hybrid approach using PSO and GA for optimisation of a profiled corrugated horn antenna.

In their approach, they used the result of the execution of one of the algorithms as a starting point to the other. They either first use PSO and then GA (PSO-GA), or vice-versa (GA-PSO).

When the solutions found by one of the algorithms show no improvement, the algorithm is changed to either PSO or GA.

Some other applications using PSO combined with GA were suggested to, e.g., recurrent network design [108], wherein individuals in a new generation are created by crossover and mutation operations as in GA, but also by running an instance of PSO.

However, unlike the previous approach, GA and PSO both work with the same population. In each generation, after the fitness values are computed, the top 50 % of elements are marked for maturing (and the other half is discarded).

The maturing technique, handled by the PSO algorithm, is used to enhance the best-performing elements, instead of using them directly to reproduce and generate the next generation.

Parents are then chosen based on a tournament selection, and then crossover and mutation are applied to produce the next offspring.

Yang et al. [100] suggested a PSO-based hybrid combining PSO with the genetic operations of selection, reproduction, crossover and mutation.

Like the previous approach, the same population is used as input for the GA and PSO algorithm, but the enhancement of the population is done by applying the motional behaviour of the PSO algorithm, while the population diversity is maintained by the genetic mechanisms (selection, reproduction, crossover and mutation). Additionally, they showed the application of the algorithm to solve three unconstrained optimisation problems and three COP.

Valdez et al. [109] tried to integrate the results given by the PSO algorithm and GA by using fuzzy logic. In their approach, a fuzzy system is responsible for choosing, according to the last results of the execution of either the GA or the PSO algorithm, which one should be executed next. Besides that, other two fuzzy systems are also used, one to change the crossover probability and the mutation rate of the GA, and the other to adjust the cognitive

and social acceleration factors of PSO.

They compared the hybrid variant with the individual GA and PSO approaches, and the hybrid algorithm was shown to be superior to the individual evolutionary methods.

Some hybrid variants of the PSO algorithm with GA were used, e.g., for cancer classification [110], route planning [111], task allocation and scheduling [112, 113] and image classification [114].

PSO With Differential Evolution

Differential Evolution (DE) also belongs to the class of evolutionary computation methods. Like PSO, DE tries to optimise a problem by iteratively improving a candidate solution (called agent, that belongs to a population of candidates) using metaheuristics.

In addition, this method does not require that the functions involved are differentiable, and it was designed to solve optimisation problems with real-valued parameters.

Although it is not guaranteed that an optimal solution is ever found, it has a great ability to maintain an adequate level of diversity within the population, and to perform a local search in specific areas of the search space. However, it has no mechanism to memorise the previous process, so the combination of DE and PSO is promising.

Each agent is represented by a set of real numbers (the parameters of the objective function) and moves around in the hyperplane until a stopping criterion (e.g., accuracy or number of iterations) is satisfied.

DE uses mutation and crossover (using three different agents) for generating a new trial parameter vector. If the new parameter vector is better than the previous one when evaluated in the objective function, then the newly generated vector replaces the current vector [4,115], in accordance with the principle of the survival of the fittest [105].

Hendtlass [116] proposed the first hybrid approach using PSO and DE. In his simple approach, the PSO algorithm runs conventionally, and from time to time, the DE algorithm takes place to move the particles to better positions.

Two years later, Zang and Xie proposed the DEPSO algorithm [117]. In this case, PSO and DE run alternately according to the number of the current iteration. If the current iteration number is odd, then PSO runs; if is even, then DE is executed (or the other way around).

Additionally, the algorithm uses a bell-shaped mutation and crossover to increase the population diversity, but instead of applying both changes at the same time (as DE originally does), different operations are applied at a random probability.

Several applications of this hybrid algorithm based on PSO and DE have emerged, including digital filter design [118], multimodal image registration [119] and data clustering [120].

In 2003, inspired by EPSO, Miranda and Alves [121] proposed the Differential Evolutionary Particle Swarm Optimisation (DEEPSO), an algorithm that is similar to the EPSO sequence, but in which the velocity of each particle is calculated as:

$$\vec{V}_{t+1}^{\,i} = \omega \vec{V}_{t}^{\,i} + \varphi_1 R_{1\,t}^{\,i}(\vec{x}_{t}^{\,r} - \vec{x}_{t}^{\,i}) + \rho\left(\varphi_2 R_{2\,t}^{\,i}(\vec{g}_{t}^{\,*} - \vec{x}_{t}^{\,i})\right), \tag{2.34}$$

where ρ is a diagonal matrix with 0 s and 1 s that controls the flow of information within the swarm (and can be seen as defining the communication topology among particles). $\vec{x}_{t}^{\,r}$ is a distinct particle from $\vec{x}_{t}^{\,i}$ that belongs to the set of particles currently in the search space or

from the previous best particles, and can be chosen at random in the current iteration and be the same for all particles or different for each one.

Finally, $\vec{g}_t^*$ is given by:

$$\vec{g}_t^* = \vec{g}_t\big(1 + w_t N(0,1)\big), \tag{2.35}$$

where w_t is a parameter or weight in the form of a diagonal matrix to add noise to the best position in the swarm, and $N(0,1)$ is the standard normal distribution.

Those authors suggest that $\vec{g}_t^*$ can be chosen from the past bests and sampled once from ρ or can be sampled from ρ to each particle, although, according to the results presented, sampling $\vec{g}_t^*$ from past bests to each particle leads to the best results.

Other hybrid approaches using PSO and DE have been proposed. These include, e.g., the LEPSO algorithm, developed by Abdullah et al. [122] with the objective of improving local best particle searching; and the enhanced DEPSO with adaptive parameters for the position update equation presented by Omran et al. [123]. On the other hand, Pant et al. [124] incorporated the PSO algorithm in DE to create a perturbation in the population that helps maintain diversity within the population and produce a good optimal solution; meanwhile, Epitropakis et al. [125], in addition to the social and cognitive experience of the swarm, included the personal experience of each particle in their hybrid approach.

Zhang et al. [126] used PSO and DE alternately, including the lbest and gbest models of the PSO algorithm. Xiao and Zuo [127] used a multi-population strategy in the DEPSO algorithm to improve diversity and keep each subpopulation on a different peak. In turn, Omran [128] presented a DE-PSO algorithm with a constriction factor, whereas Das et al. [129] used a hybrid approach of these algorithms combined with some concepts of SA, such as the probability of accepting poor solutions.

Either way, these authors did not explore the possibility of executing the PSO in the initial iterations and a change coming to the DE algorithm at the final stages of the algorithm, wherein local search around a potential solution to a problem is more advantageous.

PSO with Simulated Annealing

Simulated Annealing (SA) is also a metaheuristic optimisation algorithm which is based on the thermodynamic process of annealing; that consists of the slow and controlled cooling of a metallic material, in order to alter its microstructure, and with this, change and improve its main properties, such as strength, hardness and ductility. This process ends when the material reaches a state of minimum energy.

As other metaheuristics, SA does not make any assumption on the continuity, differentiability or convexity of the cost and constraint functions of the problem. However, unlike PSO, SA accepts poor solutions by a given probability to maintain the diversity and improve the search process [130].

SA incorporates an important mechanism called cooling schedule, which controls the decreasing of temperature during the optimisation process and the deteriorations in the objective function.

At the very beginning, the annealing process (and SA) requires higher temperatures. Then, the temperature is decreased, and some candidates are generated at that temperature level.

A candidate solution is accepted when its fitness value is lower than the current configuration (for minimisation problems). Otherwise, it may still be accepted with a certain

probability, but as temperature decreases, only small deteriorations are accepted.

This strategy allows uphill moves that help SA to escape from optimal local solutions towards the end of the algorithm, when no deteriorations of the objective function are accepted.

Hybrid variants of PSO and SA were proposed because of the well known inability of the PSO algorithm to jump out of local optima, and because the SA algorithm is known for making upward movements and escaping from those solutions, avoiding premature convergence. Unfortunately, this does not ensure that the algorithm can always converge to the global minimum. Besides that, the computational effectiveness and efficiency of these hybrid algorithms can also be compromised.

The first studies on a hybrid algorithm based on PSO and SA were made by Wang and Li [131], Zhao et al. [132] and Yang et al. [130]. Wang and Li showed that, after evaluating each particle's fitness, running SA independently on each of them and changing the movement according to the PSO algorithm can speed up the rate of convergence and enable the algorithm to escape from local optimal solutions. The algorithm was named SAPSO [131].

Zhao et al. [132] proposed the HPSO algorithm, in which the PSO runs first, providing an initial solution for SA during the hybrid search process. On the other hand, the PSOSA algorithm, proposed by Yang et al. [130], runs the PSO and the SA algorithm simultaneously; that is, after computing the velocity and position of each particle in the swarm using Equations (2.5) and (2.2), a Gaussian mutation operation is applied on each particle's personal best position. If the new value found is lower than the previous one (in the case of a minimisation problem), then it is replaced by this new value; otherwise, the solution can still be accepted according to a certain probability. A similar algorithm was proposed by Sadati et al. [133].

Both hybrid algorithms showed to be successful when compared to the PSO algorithm and the SA algorithm separately, in terms of search behaviour (and thus the quality of the solutions found), performance and computation speed.

Xia and Wu [134] proposed another hybrid approach combining PSO and SA, in this case, for the job-shop scheduling problem. Like in HPSO, in this hybrid algorithm, PSO provides an initial solution for SA. Chu et al. [135], in turn, proposed a parallel PSO algorithm with adaptive SA (ASA-PPSO).

PSO algorithms with SA were also used by Shieh et al. [136] and Deng et al. [137], in which the Metropolis criterion was used to determine the acceptance of a new-found solution that is worse than the previous one. A hybrid discrete PSO-SA algorithm was proposed by Dong et al. [138] for the optimal elimination ordering problem in Bayesian networks. In turn, He and Wang [139] suggested a hybrid approach involving PSO and SA for constrained optimisation problems, which applies SA to the best solution of the swarm to help the algorithm in escaping from local minima.

PSO With Other Evolutionary Algorithms

GA, DE and SA were not the only metaheuristics that were combined with PSO.

In the literature (see, e.g., [25]), it is possible to find PSO-based hybrid algorithms that use, e.g., Ant Colony Optimisation (ACO) [140], a population-based metaheuristic algorithm inspired by the social behaviour of real-life ants searching for food; Cuckoo Search [141], a

metaheuristic approach idealised to reproduce the breeding behaviour of cuckoo birds, who leave their eggs in the nests of other host birds of different species; and also the Artificial Bee Colony (ABC) optimisation [142], a swarm-based metaheuristic algorithm based on the behaviour of real honey bee colonies, which are organised in groups of bees to maximise the nectar amount found in a food source.

It is important to note here that, considering the large number of new developments in this field, especially in the last decade, only the hybrid PSO-based algorithms that are most relevant in practice or future research have been addressed and emphasised in this section.

2.5.2 Artificial Neural Networks with Particle Swarm Optimisation

The first experiment on using PSO to train ANN weights was done by Eberhart and Kennedy in the two papers that introduced PSO [10, 11].

They claimed to have successfully trained a feedforward multilayer perceptron ANN using PSO to solve the exclusive OR (XOR) problem and to classify the Fisher's Iris data set, which lead to the same, and sometimes better, results as the backpropagation algorithm.

It should be noted that the inertia weight is similar to the momentum term in a gradient descent ANN training algorithm [45].

Eberhart and Hu [143] showed the use of sigmoid activation functions in training a feedforward multilayer perceptron ANN using PSO to classify tremor types in Parkinson's disease.

They used an ANN with 60 inputs, 12 hidden nodes and two outputs nodes. Despite the small size of the data set, PSO has been successfully applied to train the ANN with low error and high performance.

In turn, Engelbrecht and Ismail [144] showed that the PSO could also be used to train product unit ANNs (in which the output of each node is computed as a weighted product), and when compared to other training algorithms, such as GA, the PSO showed the lowest errors.

Kennedy [63] used the social-only and the cognition-only models to train an ANN for solving the XOR problem, and showed that the social-only version outperformed the cognition-only model.

The cooperative learning approach presented in Section 2.4.1 was used by van den Bergh and Engelbrecht [44], and different two-layered network architectures were considered for testing; namely, plain (where a single swarm was used to train all the weights and bias), Lsplit (in which two swarms were used to train each layer), Esplit (where one swarm optimised 90 % of the weights and the other swarm optimised the remaining) and Nsplit (similar to Esplit, but in which weights were split according to a function).

These authors performed some tests on various databases, and split architectures (especially the Esplit architecture) outperformed the plain architecture in terms of performance, although correlated variables should be removed of the data set beforehand to improve the effectiveness of these type of architectures.

Zhang and Shao [145] split the data set into three sets, a training set, a validation set and a testing set, and used the PSO to train the architecture of ANN, including the number of nodes, generated at algorithm initialisation.

Chatterjee and his collaborators [146] showed that the PSO algorithm can be used to train the weights of a Takagi-Sugeno neuro-fuzzy network for voice-controlled robot systems.

A detailed comparison of PSO and backpropagation as training algorithms for ANN was made by Gudise and Venayagamoorthy [147]. Results showed that the ANN's weights converge faster with the PSO than with the backpropagation algorithm to achieve the same error goal.

On the other hand, Mendes et al. [148] showed that, for the problems they considered, PSO is not the best algorithm for ANN training, but it is the best one when a high number of local minima is known to exist.

Juang [108] applied PSO to recurrent neural/fuzzy network training, by combining GA, PSO and the concept of elite strategy to produce the best network design.

Ince et al. [149] used a modified version of the PSO algorithm, called MD PSO, to find the optimal architecture and weights of a feedforward multilayer perceptron ANN for the classification of electrocardiogram signals.

The MD PSO algorithm was proposed with the aim of finding the optimal solution in the search space, but also the best number of dimensions for that search space; that is, the particles explore the search space with different dimensions, and at the end of the algorithm, the optimal global solution is chosen according to the lowest optimal solution found from each dimension.

Interestingly, a hash function was used to set higher hash indexes to ANNs with higher complexity, i.e., with higher numbers of hidden layers and neural units per hidden layer, and thus the MD PSO can be used to optimise this unique dimension and find the simplest ANN that is able to classify electrocardiogram signals correctly

It was also shown by Ince et al. [149] that the proposed algorithm strategy performs better than most of the existing algorithms for classification of electrocardiogram patterns.

Pehlivanoglu [150], in turn, used a periodic mutation strategy to determine which particles should be mutated, when the operation should happen, and which ones should be added to the swarm.

Quan et al. [151] also integrated mutation in the PSO algorithm to train a feedforward ANN to short-term load and wind power forecast.

Besides optimising the network architecture and the weights of each connection, Garro and colleagues [152] also computed the best transfer (or activation) function for the problems at hand.

Al-Kazemi and Mohan [153] used a PSO variant named Multi-Phase Particle Swarm Optimisation (MPPSO) algorithm with ANNs. This variant of the PSO algorithm uses niching techniques to increase the diversity and the exploration of the search space. Besides that, according to the phase of PSO execution, the direction of each particle changes, and the particles only move to positions that will increase their fitness [154].

When compared to the backpropagation algorithm, MPPSO showed to be the more stable algorithm for optimising the ANN weights for the problems considered.

Conforth et al. [155], on the other hand, used a hybrid PSO approach, combining PSO and ACO, to adjust the ANN connection weights for the XOR problem.

In the aforementioned approaches, the PSO algorithm and its variants are used for ANN training. The use of the backpropagation algorithm for network training is neglected, since, in addition to requiring gradient and differentiable information, it also suffers from slow convergence and has a high probability of getting trapped into local minima when compared to PSO [156, 157].

Furthermore, in most of these approaches, PSO seems to need fewer epochs to get good results when compared to the backpropagation algorithm.

As can be seen in the previous sections, PSO is one of the most used metaheuristic optimisation algorithms, and is currently being applied for different purposes, as can be seen in Figure 2.3.

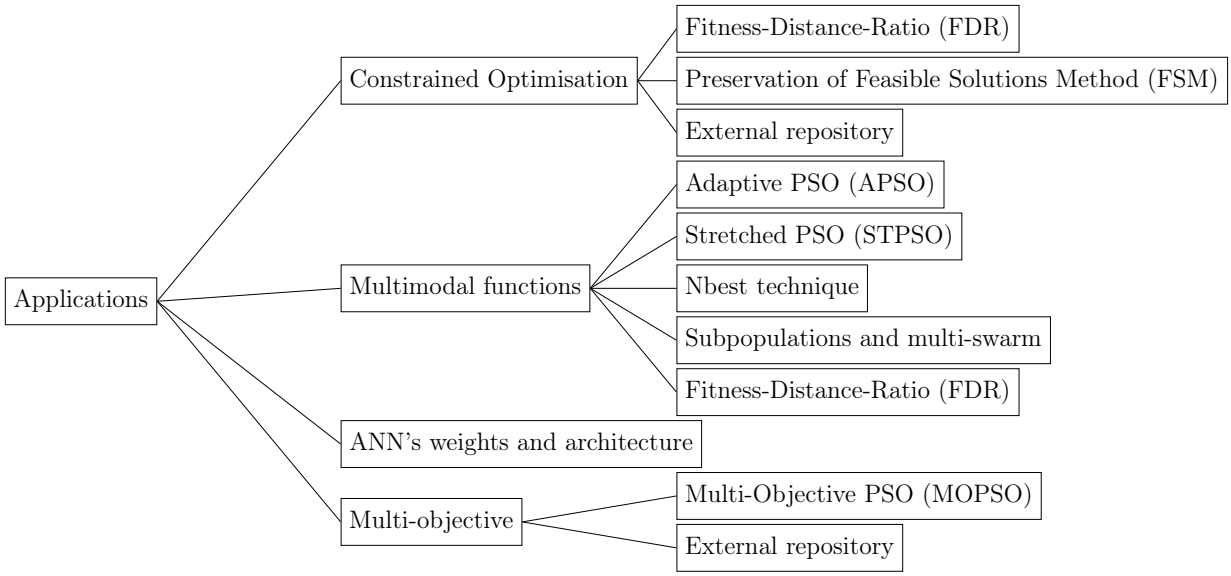


Figure 2.3: Summary of the most important applications of PSO.

2.6 Conclusion

In the previous sections, a literature review focusing on the PSO algorithm and its variants was presented, describing the most important developments in this field since the introduction of the algorithm in the mid-1990s.

The PSO algorithm was inspired by some characteristics of the collective behaviour observed in the natural world, in which elements of a population cooperate with each other seeking to obtain the greatest mutual benefit.

Over the years, the PSO algorithm has gained attention from many researchers due to its simplicity and because it does not make assumptions on the characteristics and properties (such as continuity or differentiability) of the objective function to be optimised.

Inevitably, the algorithm has suffered changes to, e.g., improve its effectiveness and efficiency.

The use of different topologies was one of the first suggestions to improve the algorithm. However, a conclusion was reached: the topologies of communication are problem dependent.

PSO was widely used for different applications, which led to some researchers to report convergence problems with the algorithm. To lessen this problem, changes were made, mostly by the introduction of new parameters, or by combining PSO with other operators or algorithms.

The algorithm has also been extended to solve a panoply of different problems and applications since its original formulation in 1995. Constrained, multi-objective and multimodal

optimisation problems were some of the most relevant applications and problems solved with the PSO approach.

To conclude, PSO is one of the leading swarm intelligence algorithms and is superior when compared to other optimisation algorithms in some fields of application. Although it has some drawbacks, those were lessened by using different types of strategies and modifications to the original version of the algorithm. PSO is also a problem-independent algorithm; i.e., it can be used in a wide range of applications due to its great capacity for abstraction, which further highlights its importance.

Chapter 3

On the Use of Particle Swarm Optimisation for Root-Finding

> — Progress is made by trial and failure; the failures are generally a hundred times more numerous than the successes; yet they are usually left unchronicled.
>
> *William Ramsay* (1852–1916)

Abstract – Particle Swarm Optimisation (PSO) is a bio-inspired algorithm motivated by the social and biological behaviour of bird flocks searching for food. PSO is considered a stochastic algorithm and uses particles, organised in a swarm, and the information gathered during the search process, in order to find the global best position for a given problem. Each particle has a position and a velocity, which enables it to explore the problem's solution space. Throughout the years, many PSO algorithm variants were suggested, leading to the inclusion of new parameters into the original PSO, such as the constriction term in the velocity update equation, and the adoption of new communication structures. Since then, the PSO algorithm is being used in a myriad of applications. In this chapter, the ability of PSO to find the roots of a given non-linear equation or a system of non-linear equations is examined. The PSO algorithm is adapted to enable it for root-finding, and the optimal parameters are found by conducting a set of statistical tests on the different executions. The objective is to give the user/researcher a framework that, according to the purpose, can choose the best parameters set for PSO. In the end, a concurrently PSO algorithm variant– Multiple Root-Finding Particle Swarm Optimisation (MRF-PSO)) – is suggested to find the approximations to the roots of a given non-linear function simultaneously. Results showed that swarms with 24 particles organised in an all-connected-to-all topology, with the constriction term, are the optimal parameter configuration, in terms of the accuracy of the roots approximations found, number of iterations and execution time. MRF-PSO, on the other hand, showed to be effective for root-finding.

3.1 Introduction

The Particle Swarm Optimisation (PSO) algorithm is a population-based stochastic search algorithm that uses particles and the information gathered throughout the search process, in order to find the optimal solution(s) in a (possible complex) search space (S) defined by a given problem.

Particles are placed randomly in the search space and move stochastically by exchanging information among all the other particles in the search space, or among the neighbourhood of particles to which they belong.

Each particle has a memory mechanism that enables it to store the exchanged information, as well as saving information about itself, such as the best position found by it. This last type of information is often referred to as the particle's experience.

Based on this information, particles explore the search space seeking for the best global position(s) regarding minimisation or maximisation problems. Different strategies to update the particles' position were, however, proposed, tested, and discussed, since the original version of the algorithm developed by Eberhart and Kennedy [10, 11] in 1995.

The PSO algorithm stood out among all the other methods available for optimisation (especially exact methods) because, in addition to being simple to implement and having few parameters to adjust, PSO does not make any assumption on the continuity and differentiability of the objective function to be optimised, as most of the exact methods require.

Inevitably, the algorithm is currently being used in different applications and problems, e.g., engineering optimisation problems [158], vehicle crash research [159] and financial risk early-warning [160]. In this chapter, however, the author wants to analyse how the PSO algorithm can be used to find approximations for the roots of non-linear equations or systems of non-linear equations, and what are the parameters that leverage the success and efficiency of PSO.

Therefore, this chapter is intended to: (i) analyse how the PSO algorithm can be used for root-finding; (ii) assess the influence of the different parameters settings, and (iii) introduce and test a concurrently-based approach of PSO for finding approximations for the roots of a given non-linear equation or system of non-linear equations.

The rest of this chapter is organised as follows: after the introduction, Section 3.2 presents the PSO algorithm and the most important convergence improvements suggested for the algorithm. On the other hand, in Section 3.3, a survey of the related approaches is done. Section 3.4 presents the changes in PSO for root-finding and, based on an Analysis of Variance (ANOVA), on Kruskal–Wallis tests, and followed by post hoc analysis, the optimal parameters for the PSO algorithm are found. In Section 3.5, a PSO algorithm variant for finding approximations for the roots of a given non-linear equation or system of non-linear equations simultaneously is suggested and investigated. Finally, the last section presents some concluding remarks.

3.2 Particle Swarm Optimisation

As already mentioned, the PSO algorithm was introduced in 1995 by an electrical engineer, Dr Russell C. Eberhart, and the social psychologist Dr James Kennedy [10, 11], inspired by a computational simulation of a social model of a bird flock seeking for food.

In order to explore the search space of a given problem, the PSO algorithm uses agents, called in this context particles. Each particle, in turn, has a velocity and a position in the search space and belongs to a set of particles, or, as Eberhart and Kennedy named, to a swarm of particles.

Besides that, the PSO algorithm also takes advantage of the cognitive and social information of each particle, in order to move it around the search space. The cognitive information (also known as particle's experience) is made by the best position found by the particle during the search process. On the other hand, the social information (also known as swarm's experience) is the best position found by the swarm of particles. This last bit of information is shared among particles at every iteration using imaginary connections between them, which enables the particles to be connected to all particles in the swarm. This model of communication is designated as gbest model.

In this way, the equations of motion of each particle i in the swarm, at every iteration t, are defined as follows:

$$\begin{cases} \vec{V}\,{}^{i}_{t+1} = \vec{V}\,{}^{i}_{t} + \varphi_1 R_1{}^{i}_{t}(\vec{p}\,{}^{i}_{t} - \vec{x}\,{}^{i}_{t}) + \varphi_2 R_2{}^{i}_{t}(\vec{g}_{\,t} - \vec{x}\,{}^{i}_{t}), \\ \vec{x}\,{}^{i}_{t+1} = \vec{x}\,{}^{i}_{t} + \vec{V}\,{}^{i}_{t+1}, \end{cases} \tag{3.1}$$

where φ_1 and φ_2 are respectively the cognitive and social weights, controlling how much the particle's own experience and the swarm's experience should influence the particle's movement, whereas R_1 and R_2 are uniformly distributed random vectors between the search space's boundaries, being responsible for adding diversity to the swarm, and thus try to lessen the premature converge of the algorithm to a local minimum or maximum. Lastly, $\vec{p}\,{}^{i}_{t}$ and $\vec{g}_{\,t}$ denote the personal best position of particle i and the current global best position of the swarm at iteration t, respectively. It is important to note that $\vec{V}\,{}^{i}_{t+1}$, R_1, R_2 and $\vec{x}\,{}^{i}_{t+1}$ are d-dimensional vectors, where d is the number of dimensions of the problem's solution space.

It is noteworthy that both R_1 and R_2 are generated for every dimension, in order to increase the variability in each component, meaning that it is not the same random number for all dimensions. This is a common error that one incurs because the equations of motion are usually written in vector notation, affecting the efficacy and effectiveness of the search process, since there is a loss of diversity.

In short, the PSO algorithm can be described in three main steps: (i) initialise the swarm; until a stopping criterion is met: (ii) compute, for each particle, the new velocity and the new position using Equation (3.1), and (iii) update the personal and global best position (if necessary) for each particle.

Equation (3.1) and the social information sharing process presented above correspond to the original PSO approach [10, 11]. However, different velocity update equations and information sharing processes (i.e., different swarm communication structures) were proposed, in order to improve the convergence characteristics of PSO.

3.2.1 Convergence Improvements

Since early, when it was first introduced, the PSO algorithm became one of the most well known and widely used swarm intelligence algorithm and metaheuristic technique, because of its simplicity and since it is considered a problem-independent algorithm, meaning that it can be used in a wide range of applications.

However, it was reported by some authors that PSO suffers from convergence problems, making the algorithm converge to a position that is not granted to be the global best position for the problem but only a local optimum.

This loss of exploration capabilities of the search space is caused when one particle founds a local optimum and reports this position to the other particles. The remaining particles will then fly headed to that position without exploring new areas of the search space, causing the algorithm to get stuck into a local optimum.

This problem motivated researchers to develop in-depth studies and propose different PSO approaches with the objective of lessening this problem, leading to the introduction of new parameters (such as the inertia weight parameter and the constriction factor) or by changing how the particles communicate between each other (giving rise to the lbest models, as an alternative to the gbest model).

The Inertia Weight Parameter

One of the most known and used PSO variant is known as Standard Particle Swarm Optimisation (SPSO), and was presented by Shi and Eberhart [23] in 1998.

They introduced the inertia weight parameter, ω, as a mean of controlling the influence of the previous velocity in the next particle's velocity, and thus controlling the local and global search strategy.

With the introduction of the inertia parameter [23], the equations of motion (3.1) are now given by:

$$\begin{cases} \vec{V}\,{}^{i}_{t+1} = \omega \vec{V}\,{}^{i}_{t} + \varphi_1 R_1\,{}^{i}_{t}(\vec{p}\,{}^{i}_{t} - \vec{x}\,{}^{i}_{t}) + \varphi_2 R_2\,{}^{i}_{t}(\vec{g}_{\,t} - \vec{x}\,{}^{i}_{t}), \\ \vec{x}\,{}^{i}_{t+1} = \vec{x}\,{}^{i}_{t} + \vec{V}\,{}^{i}_{t+1}. \end{cases} \tag{3.2}$$

In turn, ω can take different forms, such as being a simple constant value, but can also vary according to the iteration number, or even be a random number [26]. When the inertia parameter was introduced by Shi and Eberhart [23], they used different constant values for the ω parameter and found that when $\omega \in [0.9, 1.2]$, the SPSO has a higher chance of finding the global optimum within an acceptable number of iterations, when compared with the original PSO.

On the other hand, some authors [28, 29] proposed a negative linear time-varying inertia weight, that can be given by:

$$\omega(t) = \omega_{\max} - \frac{\omega_{\max} - \omega_{\min}}{t_{\max}} \times t, \tag{3.3}$$

where t denotes the current iteration number, $t_{\max}$ is the maximum number of iterations defined, and $\omega_{\max}$ and $\omega_{\min}$ are respectively the maximum and the minimum value that the inertia parameter can assume. The user/researcher is responsible for choosing these values, but usually, $\omega_{\max}$ is set to be equal to 0.9 and $\omega_{\min}$ to 0.4.

Another way of defining the inertia weight parameter is, as suggested by Chatterjee and Siarry [30], to use a non-linear time-dependent function, such as:

$$\omega(t) = \left(\frac{t_{\max} - t}{t_{\max}}\right)^{v} (\omega_{\max} - \omega_{\min}) + \omega_{\min}, \tag{3.4}$$

where υ is the non-linear modulation index. Usually, according to those authors, $\upsilon \in [0.9, 1.3]$ is satisfactory.

The time-varying inertia weight approaches make a trade-off between exploration and exploitation of the search space, avoiding that PSO gets stuck into a local optimum. In this way, at the initial stages of the SPSO algorithm, the particles should explore all areas of the search space, and thus have a higher inertia value. Differently, and as soon as a potential area of the search space is found, particles should exploit that area, reducing the inertia value, and consequently the velocity of each particle.

Lastly, the inertia weight parameter can be a random number [26], such as:

$$\omega = 0.5 + \frac{\xi}{2}, \tag{3.5}$$

with ξ being a random variable following a uniform distribution over the closed interval of 0 to 1. This expression turns ω into a uniform random number between 0.5 and 1, with a mean value of 0.75.

The Constriction Factor

In 1999, Maurice Clerc [14] suggested that the use of a constriction factor could improve the effectiveness, and efficiency of the PSO algorithm, and thus ensuring the convergence of the algorithm by making a trade-off between exploration and exploitation, similarly to the inertia weight parameter.

In 2000, Eberhart and Shi [28] proposed one of the first and most known constriction factors. In their approach, this factor is given by:

$$K = \frac{2}{\left|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}\right|}, \tag{3.6}$$

where $\varphi = \varphi_1 + \varphi_2$ and $\varphi > 4$. Typically, $\varphi = 4.1$, and thus $K \approx 0.7298$.

In turn, Equation (3.1) should be changed in order to accommodate this new parameter, that is:

$$\begin{cases} \vec{V}\,{}^{i}_{t+1} = K\left[\vec{V}\,{}^{i}_{t} + \varphi_1 R_1\,{}^{i}_{t}(\vec{p}\,{}^{i}_{t} - \vec{x}\,{}^{i}_{t}) + \varphi_2 R_2\,{}^{i}_{t}(\vec{g}_{\,t} - \vec{x}\,{}^{i}_{t})\right], \\ \vec{x}\,{}^{i}_{t+1} = \vec{x}\,{}^{i}_{t} + \vec{V}\,{}^{i}_{t+1}. \end{cases} \tag{3.7}$$

Other authors [14, 161] suggested different constriction factors and deeply studied their effect in the particles' search process, while some combined the constriction factor and the inertia factor in the velocity update equation [32].

Nevertheless, Equation (3.7) is still the most used approach when combining the PSO algorithm with a constriction factor and will be the only one considered in this work.

The Communication Structure

The communication structure, or the swarm topology, defines which particles share information with other particles, directly influencing how particles explore the search space, since the global best information is shared among a restricted set of neighbours (lbest models) or among the entire swarm (gbest model).

In the original approach of the PSO algorithm [10, 11], particles were all connected to each other. This type of communication structure is known as the gbest model and can be formally expressed by:

$$\vec{g}_t \in \{\vec{p}^{\,1}_t, \vec{p}^{\,2}_t, \ldots, \vec{p}^{\,s}_t\} \mid f(\vec{g}_t) = \min\Big(\{f(\vec{p}^{\,1}_t), f(\vec{p}^{\,2}_t), \ldots, f(\vec{p}^{\,s}_t)\}\Big), \tag{3.8}$$

where f is the objective function and s is the total number of particles placed in the search space.

However, some researchers soon detected that the gbest model has a high probability of getting PSO trapped into a local minimum since particles are affected by only one particle: the best particle in the swarm. This problem motivated the development of other communication structures, giving rise to the lbest models, and the concept of neighbourhoods [13,33].

The lbest models define neighbourhoods of particles, in which the knowledge is only shared among particles that belong to the same neighbourhood. It is important to note that the same particle can belong to more than one neighbourhood, and although it will not be considered in this work, the neighbourhood of each particle can also be time-varying, i.e., it can change according to the current iteration number [24,34,35]. (The neighbourhood of the particle i with l neighbouring particles is denoted here as N_i.)

Differently from the gbest model, the lbest model can be formally expressed as follows:

$$\vec{l}^{\,i}_t \in N_i \mid f(\vec{l}^{\,i}_t) = \min\Big(\{f(\vec{a})\}\Big),\ \forall \vec{a} \in N_i, \tag{3.9}$$

where $\vec{l}^{\,i}_t$ denotes the local best position, at iteration t, between all the particles belonging to the neighbourhood of the particle i.

Examples of lbest model's swarm topologies include the mesh, pyramid, random, ring, star, and toroid structures [13, 33]. A more detailed description of these topologies will be presented in the subsequent sections.

The lbest variant of the algorithm will be the communication model used from now on, since it can be seen as a generalisation of the gbest model, i.e., when the neighbourhood of each particle is the set of all particles in the swarm, then the two models are analogous.

3.2.2 Swarm Initialisation

The first step of the PSO algorithm is to randomly initialise each particle of the swarm, within the search space bounds ($\vec{x}_{\min}$, $\vec{x}_{\max}$), using a uniformly distributed random vector with the same dimensions of the search space (d).

In turn, and because there is no previous position visited by the particles, the personal best position of each particle is initialised with the position occupied at that moment by the particle. On the other hand, the neighbourhood's best position of each particle corresponds to the position of the particle that has the lowest cost (when considering minimisation problems) in the set of neighbouring particles. However, the neighbourhood's best position is only updated with the position of the best particle in the neighbourhood if it has a lower cost when compared to the current cost of the particle. Otherwise, the neighbourhood's best position is initialised with the current personal best position. Note that the cost of each particle, in all phases of the algorithm, is given by the evaluation of the objective function f in the position that the particle is at that iteration.

Like the position, the initial velocity is also a uniformly distributed random vector, generated within the boundaries of the search space.

This initialisation scheme is presented in Algorithm 1 (refer to Appendix A), and it was chosen by the author in order to increase the initial diversity of the swarm between the boundaries of the search space since the PSO algorithm was shown to be highly influenced by its initial diversity [162]. However, other schemes exist [163], and other strategies may be used, such as placing the particles in specific positions, especially when there is a priori knowledge of which areas of the search space are more favourable than others.

3.2.3 Optimisation Cycle

At each iteration, until a stopping criterion is met, the position and the velocity of each particle are updated according to Equation (3.1), Equation (3.2) or Equation (3.7). At the end of each iteration, if necessary, the best personal position and the neighbourhood best position are synchronously updated, as can be seen by the flowchart of PSO presented in Figure B.1.

In this work, particles are identified by an index, i, because it is computationally inexpensive to share information among the particles using this strategy. Also, it simplifies the algorithm and makes it less prone to human errors.

When a stopping criterion is met, the output of the algorithm is the position, within the search space, where particles found the lowest cost (when considering minimisation problems), regardless of the swarms' topology, i.e., the output of the algorithm is the position of the particle in the swarm with the lowest cost value.

3.2.4 Stopping Criterion

The stopping criterion defines the objective of the execution of the algorithm. For example, the algorithm can stop when no improvements are detected, when the number of iterations reached the maximum allowed, when the standard deviation of the particles' positions is below a predefined threshold [164], or when the cost of the best particle in the swarm is lower than a predefined threshold ϵ.

A combination of two stopping criteria can also be used, especially when one of the stopping criteria is the maximum number of iterations, in order to force the termination of the algorithm when it fails to converge.

Although the stopping criterion defined by the maximum number of iterations does not require a priori knowledge about the problem at hands, care should be taken when choosing the maximum number of iterations in order to prevent the algorithm from terminating prematurely. The use of a stopping criterion that requires a frequent calculation of the fitness function should also be avoided.

The stopping criterion is, thus, one of the algorithm's characteristics that is crucial for its performance and solution quality.

Concluding, as can be seen, PSO is relatively easily implemented, does not require a high computational capacity to be run and can be used in a wide range of applications, since its application is problem independent.

For its natural division of tasks and processes, the PSO algorithm can be easily split into small computer processes for each particle or for each swarm (when multiple swarms

are placed in the search space), that can be executed concurrently in a multi-processor and multi-core system, improving the efficiency of PSO significantly. This will be explored later in this chapter.

3.3 Related Work

The first report related to solving systems of equations with PSO was done by van den Bergh and colleagues [24, 60] in 2002. They introduced the nbest PSO, in order to locate multiple global solutions in a problem's solution space.

For that purpose, they redefined the objective function as the minimum of the fitness function with respect to other equations. For example, when the given system of equations has three equations (say A, B and C), the fitness function is given by:

$$f(\vec{x}) = \min\big(\{f_{AB}(\vec{x}), f_{AC}(\vec{x}), f_{BC}(\vec{x})\}\big), \tag{3.10}$$

where f_{AB} denotes the fitness function with respect to equation A and B, f_{AC} to A and C, and f_{BC} to B and C. This strategy rewards particles that are close to one of the solutions, and does not apply any penalisation if they are still far from the global best particle. This way, particles will find the position where all lines of the given system of equations intersect.

Besides changing the fitness function, the nbest technique uses a dynamic neighbourhood approach, based on the Euclidean distance, in order to define the neighbourhood of each particle, and thus which particles communicate with other particles. The effect of the neighbourhood size was also investigated [24].

Their strategy has revealed some promising results when compared to the gbest and some lbest architectures. Nevertheless, in real-world applications, the systems of equations are usually not limited to three equations. Thus, as the number of equations increases, the number of combinations also increases, causing the algorithm to face performance issues.

Wang and collaborators [131], also with the objective of solving systems of equations, opted to control the adjustment of the inertia term (ω) during the execution of the PSO algorithm.

Interestingly, they proposed two approaches for adjusting the inertia term. The first approach uses a Proportional Integral (PI) controller to perform the adjustments on the inertia term based on the current error ($e(t)$), given by the difference between the desired output ($r(t)$) and the output given by PSO ($P(\vec{x})$), and on its past values.

On the other hand, an Artificial Neural Network (ANN) based approach was used along with a Proportional Integral Derivative (PID) controller, in order to tune the PID's constants and add the information about the current rate of change of $e(t)$, besides the information related to the current and past information about $e(t)$ (that were previously integrated into the model by the PI controller).

The ANN is used, in turn, to tune the coefficients for the proportional, integral, and derivative terms of the PID controller, denoted here by K_p, K_i and K_d, respectively. These coefficients represent the weights of a single neuron ANN with three inputs.

Figure 3.1 summarises this approach. In this figure, I_1, I_2 and I_3 are the inputs of the ANN, which are defined by the difference between the desired output and the actual output, i.e., $e(t)$, and the first and second-order differences of the error, respectively. These differences are computed by the SG module. In this way, I_1 constitutes the proportional

part, I_2 the derivative part and I_3 the integral part in the PID controller, being w_{11},w_{21} and w_{31} the same as K_p, K_i and K_d.

Finally, the inertia term, in this case, the control variable in the PID controller, is passed to the PSO algorithm, which is then run. Therefore, the PID controller is responsible for balancing the global and local search strategies.

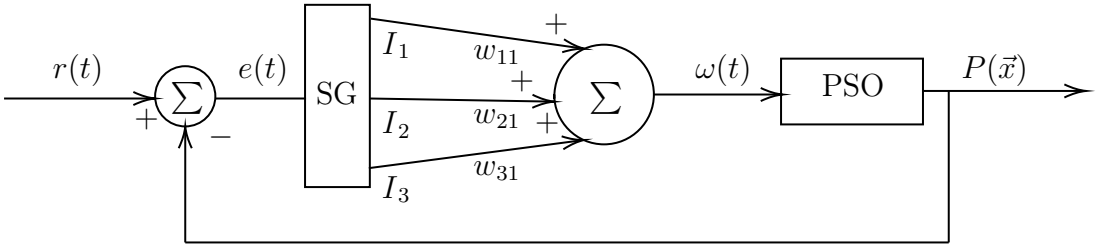


Figure 3.1: PID controller, proposed by Wang et al., to control the inertia term.

Abraham et al. [165] then used the PSO algorithm to find the integer numerical approximations for the solutions of the Diophantine equation, which represents an elliptic curved function. Later, Pérez and collaborators [166] proposed a discrete version of the PSO algorithm for solving identical problems.

Amaya and colleagues [167] tested the PSO algorithm for solving systems of non-linear equations and studied the effect of considering different swarms sizes. They found that, as the swarms get bigger, the PSO algorithm required fewer iterations to achieve the same accuracy level, and thus have a faster convergence speed; however, the algorithm's execution time increased.

Grailoo and collaborators [168] suggested to change the position update equation as follows:

$$\vec{x}\,^{i}_{t+1} = \vec{x}\,^{i}_{t} + C(t+1)\vec{V}\,^{i}_{t+1}, \tag{3.11}$$

with $C(t)$ given by:

$$C(t) = \frac{1}{e^{\alpha t}}, \tag{3.12}$$

where $\alpha = \dfrac{\mu}{t_{\max}}$, and $t_{\max}$ and μ are the maximum number of iterations and a parameter, both defined by the user/researcher.

This strategy changes the weighting given to the velocity of the particles, by using an exponential decreasing relationship between the iteration number and the parameter μ. Thus, particles tend to give privilege to the exploration of the search space initially and, on the last stages of the algorithm, the exploitation of it.

The experimental results were compared to some classical methods for root-finding (e.g., Newton's method and Broyden's method) and the proposed algorithm revealed to be more accurate.

Jaberipour et al. [169], on the other hand, proposed to change both the velocity and position update equations as follows:

$$\begin{cases} \vec{V}\,^{i}_{t+1} = (2\xi_1 - 0.5)\vec{V}\,^{i}_{t} + (2\xi_2 - 0.5)(\vec{p}\,^{i}_{t} - \vec{x}\,^{i}_{t}) + (2\xi_3 - 0.5)(\vec{g}_{t} - \vec{x}\,^{i}_{t}), \\ \vec{\omega}\,^{i}_{t+1} = (2\xi_4 - 0.5)(\vec{g}_{t} - \vec{p}\,^{i}_{t}) + (2\xi_5 - 0.5)(\vec{g}_{t} - \vec{x}\,^{i}_{t}), \\ \vec{x}\,^{i}_{t+1} = \vec{p}\,^{i}_{t} + (2\xi_6 - 0.5)\vec{V}\,^{i}_{t+1} + (2\xi_7 - 0.5)\vec{\omega}\,^{i}_{t+1}, \end{cases} \tag{3.13}$$

where ξ_1, ξ_2, ξ_3, ξ_4, ξ_5, ξ_6 and ξ_7 are randomly generated numbers between 0 and 1.

After updating the velocity and the position of each particle, the personal best position is, if necessary, updated, and stored in a matrix. Then, one of the dimensions of the worst personal best position in terms of fitness value is mutated and is updated with the mutated position if its fitness is lower (considering minimisation problems) than the previous fitness value. Finally, the global best position of the swarm is found and the algorithm proceeds as the original PSO's strategy. A similar approach was also followed by Salomon et al. [170], but for load-flow studies.

Besides changing the equations of motion, Zhao and collaborators [171] applied a speciation strategy, where particles are grouped in neighbours according to the Euclidean distance between them. Thus, the information sharing is reserved for particles belonging to the same species, making the algorithm able to find multiple solutions, more specifically, a solution per species.

Reyes-Sierra and collaborators [172] showed that the PSO algorithm can be used to find both real and complex roots of a given non-linear system of equations, by changing the velocity update equation as:

$$\begin{aligned}\vec{V}\,{}^{i}_{t+1} = \omega\vec{V}\,{}^{i}_{t} + \varphi_1 R_1\,{}^{i}_{t}\Big(\mathrm{Re}(\vec{p}\,{}^{i}_{t} - \vec{x}\,{}^{i}_{t})\Big) + \varphi_2 R_2\,{}^{i}_{t}\Big(\mathrm{Re}(\vec{g}_t - \vec{x}\,{}^{i}_{t})\Big) + \\ \varphi_1 R_1\,{}^{i}_{t}\Big(\mathrm{Im}(\vec{p}\,{}^{i}_{t} - \vec{x}\,{}^{i}_{t})\Big) + \varphi_2 R_2\,{}^{i}_{t}\Big(\mathrm{Im}(\vec{g}_t - \vec{x}\,{}^{i}_{t})\Big),\end{aligned} \tag{3.14}$$

where Re and Im are respectively the real part and the imaginary part of a complex number.

Mai and Li [173] hybridised the Bacterial Foraging Optimisation (BFO) with the PSO algorithm with linear time-varying inertia [28,29] for solving systems of non-linear equations. On the other hand, Ibrahim and Tawhid [174] hybridised the Cuckoo Search Optimisation algorithm with PSO for the same task.

3.4 Root-Finding Particle Swarm Optimisation

Although there are many iterative exact methods for finding one root or a pair of roots of a given equation, these methods require repeated deflations in order to find the set of arguments where the function assumes the value of zero. That is, these algorithms use the last computed root approximation for computing a new approximation.

However, this repeated approximation technique can lead to very inaccurate results due to the problem of accumulating rounding errors, even if all machine (finite) precision for floating-point arithmetic is used. Examples of iterative methods for root-finding include the well known Newton's method (also known as Newton–Raphson method), based on the 2^{nd} order Taylor series expansion.

The Newton's method has a fast convergence speed, being quadratically when the method converges. However, it needs very good initial approximations for all roots in order to converge, which makes it complex to be used when no a priori information about the function is available. It also requires the computation of the derivative information (like the Jacobian matrix) of the function whose roots are being computed, which is not always possible. Finally, and although it is simple to implement, the Newton's method is very computationally expensive to execute [175].

The secant method and other quasi-Newton variations (such as the Broyden's method and the Broyden–Fletcher–Goldfarb–Shanno [BFGS] algorithm) [176] facilitate the computation of the exact derivative of the equation by using an approximation; however, it is still a repeated approximation technique with an associated medium-high computational effort and, in some cases, a slow convergence rate.

On the other hand, due to the search strategy implemented in PSO, the algorithm is able to explore all the search space and progressively converge to the most propitious areas of it. Thus, there is no need to have good initial approximations for the roots of a given function for the algorithm to converge.

Besides that, the PSO does not need the derivative information to be computed, since the cost of each particle is given by evaluating the function on the particles' position, nor is a deflation technique; therefore, high-quality solutions may be obtained.

In this way, the PSO algorithm is tested here for finding approximations for the real roots of non-linear functions, using different parameters configuration and communication structures, in order to find the best set of parameters for root-finding with PSO.

Later, a PSO variant for finding approximations for all roots of a given non-linear equation or system of non-linear equations simultaneously is suggested and tested. Results, however, cannot be compared to the previously mentioned exact approaches, since they were not designed to find all roots simultaneously. As an alternative, it is possible to find each root sequentially by using different initial guesses, but in the author's opinion, this is not a fair comparison for both approaches.

3.4.1 Adaptation of Particle Swarm Optimisation

Using the PSO algorithm for root-finding is, in essence, very similar to the use of PSO for minimisation or maximisation problems. Changes were made specifically in the way of computing the cost of each particle, as well as ensuring that the particles stay within the search space's bounds in the course of the algorithm.

On the other hand, the search strategy was not directly changed, since particles should always move into the best-fitness particle, in this case, to the particle with the lowest root value.

The algorithm then iterates until the maximum number of iterations is reached or until the root value is lower than a pre-established tolerance ϵ (usually, $\epsilon = 10^{-12}$).

Cost of Each Particle

In the PSO algorithm for root-finding, the cost of each particle is given by the absolute value of the evaluation of the function at the current position of the particle, i.e., the root value.

It is important to note that since the PSO algorithm was designed for optimisation problems, where particles with lower cost (considering minimisation problems) have a higher probability of being at the best global position, the same may not happen with the roots of a given function.

If the absolute value of the evaluation of the function in some position was not taken as the cost of each particle, particles far from the function's root with negative costs will have more influence than particles that may even be closer to the root but have a higher fitness value.

Since there are no negative costs, the root value can be seen as a measure of the quality of each particle; thus the pbest and lbest information will still be used to influence particles to move towards the best-fit particle, in this case, to the particle that is closer to the root of the given non-linear function.

Search Space's Bounds

Even though in the first iteration of PSO particles are inevitably positioned within the search space's bounds, during the later iterations, some particles can leave the search domain, and thus the feasible region. Consequently, particles may take some iterations to return to the feasible region again, or they may not even come back, and influence other particles to move towards invalid positions, affecting the quality of the solutions found by the algorithm.

To overcome this problem, particle's dimensions that exceed its bounds are reset to the nearest available boundary, such that:

$$(\vec{x}\,{}^{i}_{t+1})_d = \begin{cases} (\vec{x}_{\min})_d & \text{if } (\vec{x}\,{}^{i}_{t+1})_d < (\vec{x}_{\min})_d, \\ (\vec{x}_{\max})_d & \text{if } (\vec{x}\,{}^{i}_{t+1})_d > (\vec{x}_{\max})_d, \\ (\vec{x}\,{}^{i}_{t+1})_d & \text{otherwise.} \end{cases} \tag{3.15}$$

Nevertheless, others strategies exist when particles exceed the feasible region, such as position randomly the particle inside the dimension's bounds, position the particle in the midpoint between its current position and the dimension's bound, or re-adjust the particle's velocity using a shrinking method [177, 178].

The pseudo-code for the optimisation cycle is listed in Algorithm 2 (refer to Appendix C), where $\hat{y}$ denotes the root approximation found by the PSO algorithm. It is also important to note here that Algorithm 2 adopted the velocity update equation given in Equation (3.2), but other velocity update equations will also be considered in this work.

Particles, in turn, are initialised, at the first iteration, as the initialisation scheme presented in Algorithm 1.

3.4.2 Parameter Tuning

Although PSO has few parameters to be regulated when compared to other optimisation techniques, these need to be chosen and have a significant impact on the successful execution of the algorithm. Unfortunately, most of the times, the right-tune of these parameters can only be made experimentally, using a trial and error methodology.

In this section, a set of test functions commonly used for testing optimisation algorithms will be used to test how the parameters affect the convergence, performance and the effectiveness of the PSO algorithm adapted for root-finding. The author also intended to provide a framework to other users/researchers so that, according to their purposes (e.g., accuracy or efficacy), one can choose the best set of parameters for some application taking into consideration the results presented in the following subsections.

Nevertheless, the optimal combination of parameters (considering accuracy and efficacy) will be drawn from the tests and will be used with the proposed PSO algorithm variant for simultaneously finding approximations for the roots of a given non-linear equation or system of non-linear equations.

Experimental Setting

To assess the impact of the different parameters in the execution of the PSO for root-finding, a set of tests were performed using different test functions [179, 180].

Test functions are functions that were intentionally developed to test, evaluate and compare different optimisation algorithms and their parameters. For this work, functions were chosen based on the number of local minima and function's shape and taking into account that its optimal solution coincides with the root of the function. In this view, it is possible to test the convergence rate, accuracy, effectiveness and efficiency of the PSO algorithm for root-finding.

The parameters considered in each test include: (i) the number particles in the swarm (24, 36, 48 and 60); (ii) the communication structure (all-connected-to-all [gbest model], mesh, pyramid, random, ring, star and toroid [lbest models]), and (iii) the velocity update equation (constant, linear and non-linear time-dependent inertia, stochastic inertia and constriction factor).

The value found by each swarm, the number of iterations, the total execution time and the number of roots that were not found are the metrics that were collected from each test. Note that, when the algorithm fails to converge to an acceptable root approximation according to the predefined accuracy, it is said that the algorithm has not found a root.

Due to the high number of dimensions and the complexity of the search space, ϵ was set to 10^{-5}. In addition to setting the stopping criterion by the value of the objective function in a given point, in order ensure that the algorithm will end, it was also defined that the algorithm can have a maximum of 5 000 iterations.

In total, 7 000 tests were executed. That is, ten tests were executed for each combination of the four number particles in the swarm, the seven communication structures, the five velocity update equations and the five test functions. Besides that, all the tests were executed in the same machine with an Intel Core i7-7560U CPU, running at 2.40 GHz, with 16 GB of RAM.

Test Functions

The Ackley, Rastrigin, Rosenbrock and sphere functions, and the Schaffer function no. 2 were used in this work to find the best parameters for root-finding. Except for the Schaffer function no. 2, which is defined only for two dimensions, 30 dimensions were used in all of the test functions. In Appendix D, all the functions' graphical illustrations, expressions and search-domains are presented.

Functions with flat surfaces threat the convergence of the algorithm since there is no information about which direction are the promising areas of the search space. An example of a function with a flat surface is the Ackley function [181, 182].

This symmetric function has many local minima in the outer region, whereas at the centre of the search space there is a large hole, where the root (and the global minimum) of function is located. Besides that, the function is continuous and differentiable.

Another well know test function is the multimodal Rastrigin function [183], in which the search space besides being large and complex, has many local minima with the particularity that they are regularly distributed, due to the cosine function. Like the previous test function, this function is non-convex with its root sitting at the origin.

The Rosenbrock function [184] is a unimodal function with the global minimum, in this case, also the root of the function, located in a long and narrow parabolic valley in the search space, more specifically at $(1, \ldots, 1)$.

In addition to varying rapidly [185], tests made by Picheny et al. [186] report that it is easy to find the valley. However, the global minimum is more complex to be found, and this is one of the reasons why this function is considered by many researchers to be a proper test function.

On the other hand, the Schaffer function no. 2 [187] is only defined in a 2-dimensional space. Nevertheless, the search process is still laborious because it is extensive and has many local minima in the sided wells. The function is non-convex, unimodal and differentiable, and has its root at $(0, 0)$.

The sphere function [188], in turn, is a bowl-shaped function where all the points are equidistant between them and the centre of the sphere, where the root is sitting. This function is symmetric, unimodal and convex.

In terms of complexity of the search process, the Ackley function and the Schaffer function no. 2 are considered hard; the Rastrigin and Rosenbrock functions are considered medium-level, whereas the sphere function has the most simple search space, being categorised into the easy level.

Analysis of Variance and the Kruskal–Wallis Test

The Analysis of Variance (ANOVA), introduced by Sir Ronald Fisher [189], is a statistical technique that is used to compare the average values of random variables when subject to different experimental conditions, identified by one or more independent variables, called factors [190].

To conduct an ANOVA, the experimental procedures are applied to $k \geq 2$ populations, to test the differences between the populations' averages. Besides that, the ANOVA is also used to determine which of the factors are more relevant to explain the variability of the dependent variable(s).

For example, an ANOVA should be conducted when one wants to compare the grades of students, from different colleges, that took the same exam, in order to verify either if the exam performance is the same or if one college is better than the other.

For this work, a one-way multiple ANOVAs were conducted. That is, only one dependent variable was considered at a time and analysed in each of the three independent variables, indicated in Table 3.1.

Table 3.1: Independent variables and their respective levels.

Independent variable	**Levels of the independent variable (factors)**
Population size	24, 36, 48 and 60.
Velocity update equation	Constant, linear, non-linear, and stochastic inertia and constriction factor.
Communication structure	All-connected-to-all, mesh, pyramid, random, ring, star and toroid.

The dependent variables for this study are: (i) the root approximation value found by the swarms; (ii) the number of iterations, (iii) the total time of execution (in seconds), and (iv) the number of roots not found.

The ANOVA procedure tests if the mean values of a random variable in the k populations are equal, that is, if the effect of the factor is null, determining whether the populations are all part of a larger population or are populations with different characteristics. In this view, the hypotheses of interest can be given as:

$$H_0 : \tau_1 = \tau_2 = \cdots = \tau_k \quad \text{vs.}$$
$$H_1 : \exists i, j : \tau_i \neq \tau_j,\ i, j = 2, \cdots, k,$$

where τ_k denotes the average of the population k.

If the null hypothesis is accepted, under a certain significance level, the average value between the populations is assumed to be equal; on the other hand, when the null hypothesis is rejected, at least one population's average is different from at least one other population's average. In this last case, post hoc tests may be performed in order to detect which groups have a statistically significantly different average values from the other groups. (This procedure is known as Multiple Comparison Analysis.)

However, the ANOVA procedure can only be applied when all the following assumptions are met [191]:

- Independent and randomly selected samples.
- The distribution of each group follows a normal distribution (which can be checked using Shapiro–Wilk Test or the Kolmogorov–Smirnov test).
- Equal variances between the groups, i.e., groups must be homogeneous (which can be examined using Levene's test). It is important to note that if the hypothesis of homogeneity of variances is not satisfied and groups have the same number of samples, the ANOVA procedure is little affected.

If the ANOVA's assumptions are met and the null hypothesis is rejected, then differences between specific groups can be detected using the Tukey post hoc test (or the Tuckey–Kramer's when there are unequal sample sizes).

There is also the case when the ANOVA's assumptions are not met. In these situations, the Kruskal–Wallis test, a non-parametric alternative to the usual ANOVA, may be used along with the null hypothesis that the samples come from similar population distributions.

It is important to note here that the ANOVA's hypotheses of interest are written in terms of the population's average value whereas the Kruskal–Wallis are written in terms of the median value. Besides that, when the null hypothesis is rejected on the Kruskal–Wallis test, the Dunn's test can be used in order to make the comparisons between any two groups levels and detect which specific groups are statistically significantly different from the others. (That is, the Dunn's test will be used here as a post hoc test to test for median statistically significantly difference.)

The ANOVA results will be presented here as following $[\mathrm{F}(\mathrm{df}_1, \mathrm{df}_2) = \cdot,\ p = \cdot]$, where F represents the statistic value using df_1 degrees of freedom between groups and df_2 within groups, and p the p-value. On the other hand, the Kruskal–Wallis test will be presented as

follows: $[\chi^2(\text{df}) = \cdot,\ p = \cdot]$, where χ^2 denotes the chi-squared statistic value, and df denotes the degrees of freedom.

Finally, the Pearson's chi-squared test for independence, which is suitable for categorical data, was used to test if the different parameters influence the number of roots not found by the PSO for root-finding (in this case, true, if the algorithm found a root; false, otherwise). In a similar way to the Kruskal–Wallis test's notation, the Pearson's chi-squared tests' results will be presented as follows: $[\chi^2(\text{df}) = \cdot,\ p = \cdot]$.

All the tests were made considering the significance level of .05 and using the `R` software version 3.6.3 [192], and taking into account only the successfully executions of PSO, except for the test of the number of roots not found. It is also important to note that the Dunn's test was applied using the Bonferroni adjustment for the p-value.

In what follows, the different PSO algorithm parameters will be introduced and its effect on the execution of the PSO algorithm for root-finding will be assessed. Throughout the analysis, auxiliary tables will be presented with the average (when the ANOVA is used) or with the median values (when the Kruskal–Wallis test is utilised) for each level.

3.4.3 Population Size

The population size, i.e., the number of particles in the swarm, is one of the first parameters to be considered. Usually, the number of particles varies from 20 to 60.

However, if a small number of particles is chosen, particles tend to take longer (i.e., require more iterations) to explore the entire search space, slowing down the convergence speed, which may help the algorithm to escape from local optimum positions. Still, there are few function evaluations, when compared to a larger number of particles.

On the other hand, when a higher number of particles is used, there are more interactions and exchanges of information among the swarm, thus PSO has a higher probability of increasing the quality of the solution(s) found when compared to a smaller number of particles.

Nevertheless, some studies [28, 193] reported that the effect of the population size does not have a significant impact on the overall performance of the PSO algorithm.

El-Gallad et al. [194] tested some different swarm sizes with PSO for solving Constrained Optimisation Problems (COPs), and they concluded that increasing the number of particles in the swarm also increased the quality of the solutions found. However, increasing the swarm size from 30 to 50 did not result in a significantly better solution. Similar results were also reported by Dai et al. [195].

Notwithstanding, a compromise must be reached between quality and performance.

Tests and Results

A different number of particles was tested with different combinations of parameters, to choose the number of particles that, on average, gives the best results in terms of the value of the root found, the number of iterations required, execution time and the number of roots not found.

Table 3.2 serves as an auxiliary table for selecting the best number of particles for each test, considering the results of the multiple comparison analysis.

Table 3.2: Auxiliary table used to compare the effect of the different number of particles in the swarm on the PSO algorithm.

No. of particles	Median root value	Average no. iterations	Median execution time (s)	No. roots not found
24	4.990e−06	276.966	0.019	80.057 %
36	6.580e−06	296.063	0.022	78.343 %
48	6.545e−06	355.203	0.028	76.343 %
60	6.880e−06	295.169	0.030	75.371 %

For the first experiment, a Kruskal–Wallis test was conducted to compare the effect of the number of particles on the accuracy of the roots found by the PSO algorithm with swarms with 24, 36, 48 and 60 particles. The choice of this test is based on the fact that the groups' sizes are different and, according to the Levene's test, the populations' variances are considered to be not equal ($p \approx .010$).

In this view, the analysis suggests, at the level of significance of .05, that there are statistically significant differences between the effect of the number of particles in the swarm on the value of the root found by the algorithm for the four population's size [$\chi^2(3) \approx 12.337,\ p \approx .006$]. Post hoc comparisons using the Dunn's test then indicated that swarms with 24 particles differ statistically significantly from swarms with 60 particles ($p \approx .003$); nevertheless, the comparison between the pairs with 36, 48 and 60 did not show statistical evidence that they differ.

Taking into consideration the results presented in Table 3.2, it is possible to verify that swarms with 24 particles revealed to have the lowest median root value. Thus, in what concerns to the root values' accuracy, swarms with 24 particles found more accurate root approximations.

Regarding the number of iterations and since all ANOVA's assumptions were fulfilled, a one-way ANOVA was conducted to compare the effect of the number of particles on the number of iterations required for the algorithm to be successful. Results showed, at the level of significance of .05, that there was not a statistically significant evidence that the different number of particles affects the number of iterations [$\mathrm{F}(3, 1569) = 1.364,\ p \approx .252$].

In order to examine the effect on the execution time when a different number of particles in the swarm is considered, a Kruskal–Wallis test was conducted and reported that the number of particles has a statistically significant effect in the execution time of the algorithm [$\chi^2(3) \approx 22.244,\ p < .001$]. (This test was used due to the populations' variances are considered to be not equal [$p < .001$] and since the groups have a different number of samples.) Thus, for any level of significance, at least one number of particles in the swarm is statistically different from the others in terms of execution time.

In fact, the Dunn's test revealed that no statistically significant differences were found in the pair 24-36 ($p \approx 1.000$), which, according to Table 3.2, has the shortest execution times. On the other hand, pairs 24-48 ($p \approx .004$), 24-60 ($p < .001$) and 36-60 ($p \approx .020$) were found to have statistically significant differences. Thus, swarms with 24 or 36 particles are the optimal numbers of particles in the swarm in terms of execution time. It is also important to note that, as one would expect, as the number of particles increases, the execution time required for the execution of the PSO algorithm also increases.

Finally, a Pearson's chi-squared test was performed comparing the frequency of the number of roots found in the different population sizes. The result of this test showed that there

is a statistically significant relationship between these variables [$\chi^2(3) \approx 13.175$, $p \approx .004$]. That is, there is a statistically significant evidence that increasing or decreasing the number of particles in the swarm increases or decreases the probability of PSO to find the root of a given function.

In order to detect the pairs that are statistically significantly different, the Pearson's chi-squared was used again, but with the Yates' correction for continuity. As a result, swarms with 60 particles showed to be statistically significantly different from particles with 24 ($p \approx .001$) and 36 ($p \approx .041$); however, no statistically significant differences were found in between swarms with 60 and 48 particles ($p \approx .527$). As can be seen by the results of Table 3.2, swarms with 60 and 48 particles have a higher success rate when compared to other swarms with a lower number of particles. (In fact, increasing the number of particles reflects an increase in the success rate of the algorithm.)

The analysis of all these tests allowed to conclude that, for the test functions considered, swarms with 24 particles perform better than any other number of particles in the swarm, proving to be a good approach in terms of root accuracy and efficiency, at the expense of a decrease in the effectiveness of PSO.

A summary of the statistical results on different swarms' sizes can be found in Table 3.3.

Table 3.3: Comparison of the different number of particles in the swarm (X means performed better.)

No. of particles	**Root value**	**No. iterations**	**Execution time**	**No. roots not found**	**Total**
24	X	X	X		**3**
36		X	X		**2**
48		X		X	**2**
60		X		X	**2**

3.4.4 Velocity Update Equation

The velocity update equation is the mechanism that makes particles move around in the search space, being the most critical tuning strategy for the successful execution of the algorithm.

In most of the PSO algorithm variants, the velocity update equation is made of an inertia or a constriction term, cognitive and social acceleration coefficients, and stochastic components to create diversity in the swarm's movement.

Inertia Weight and Constriction Term

The inertia weight parameter was introduced by Shi and Eberhart [23] to balance the global and local search in the defined search space, controlling how much the previous velocity should affect the next velocity and position update.

Therefore, if a high value is assigned to the inertia component, particles will give more importance to the self-knowledge, whereas a small inertia gives a greater weighting to the social's knowledge, helping the algorithm to escape from local minima.

To control the exploration and exploitation capabilities, Shi and Eberhart suggested a constant inertia weight parameter. However, many researchers reported that using a dynamic inertia weight (e.g., based on the iteration number) increased the capabilities of the PSO algorithm [196].

On the other hand, the constriction factor (K) was suggested by Maurice Clerc [14] and later introduced by Eberhart and Shi [28] in the PSO velocity update equation, with the objective of ensuring, based on mathematical theory, the stable convergence of the algorithm [161]. Besides that, this constriction term can be used to prevent the rapid increase in the velocity of the particles towards infinity.

The inertia weight and the constriction factor can be seen as similar approaches since the previous velocity is multiplied by a constant. Typically, when the constriction term is used, the sum of the cognitive and social accelerations coefficients is equal to 4.1 ($\varphi_1 = \varphi_2 = 2.05$), and thus, $K \approx 0.729$. However, in the velocity update equation of (3.7), φ_1 and φ_2 are not equal to 2.05, but approximately equal to 1.494 ($0.729 \times 2.05 \approx 1.494$). Thus, when $\omega = K$ and $\varphi_1 = \varphi_2 \approx 1.494$, Equation (3.2) can be obtained.

Table 3.4 lists the inertia weights variations and the constriction term used in the tests, as well as the value of the parameters used.

Table 3.4: Inertia weights variations and the constriction term used in the tests, and the values of the parameters used.

Name	**Function**	**Experiments**
Constant [23]	$\omega = c$	$c = 0.9$
Linear decreasing [28, 29]	$\omega(t) = \omega_{\max} - \frac{\omega_{\max} - \omega_{\min}}{t_{\max}} \times t$	$\omega_{\max} = 0.9$ and $\omega_{\min} = 0.4$
Non-linear decreasing [146]	$\omega(t) = \left(\frac{t_{\max} - t}{t_{\max}}\right)^{\upsilon} (\omega_{\max} - \omega_{\min}) + \omega_{\min}$	$\upsilon = 1.2$, $\omega_{\max} = 0.9$ and $\omega_{\min} = 0.4$
Stochastic [26]	$\omega = 0.5 + \frac{\xi}{2}$	n/a
Constriction factor [23]	$K = \frac{2}{\left\|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}\right\|}$	$\varphi = \varphi_1 + \varphi_1$ and $\varphi_1 = \varphi_2 = 2.05$

Acceleration Coefficients

It is already known that the acceleration coefficients (φ_1 and φ_2) can affect stochastically the magnitude of the search, how much importance is given to self and swarm's experience and the convergence behaviour.

When $\varphi_1 \gg \varphi_2$ or $\varphi_2 \gg \varphi_1$, particles will be more attracted to $\vec{p}\,_t^{\,i}$ or $\vec{l}\,_t^{\,i}$, respectively. However, keeping one of these conditions throughout the execution of the algorithm can cause the particles to wander in the search space or the algorithm to converge prematurely.

To enhance the exploration capabilities of the swarm, high values should be assigned to both φ_1 and φ_2. In this case, particles will be more sensitive to the difference between the

current position and the best position in the neighbourhood and thus will move faster and to distant regions of the search space.

When a refined local search is better for the search strategy, these parameters should be set to lower values, since particles will move slower and will continuously improve an objective and the exploitation capabilities of the swarm.

Usually, their values are static, but may also vary according to the iteration number [34, 193] to give the swarm better exploration capabilities at the beginning and, when a promising area of the search space is found, enhance the exploitation capabilities of the swarm.

Besides that, both should be greater than zero, although, when $\varphi_1 = 0$, particles have no cognitive abilities and are influenced only by the best particle in the neighbourhood. Thus, the algorithm is more susceptible to get stuck into a local optimum. On the other hand, when $\varphi_2 = 0$, the particles do not have a social component, i.e., there is no cooperation and communication among particles, making the algorithm more time consuming and less likely to find the optimum position [197]. In most applications, $\varphi_1 = \varphi_2$, since particles are attracted to the mean position between its personal and neighbourhood's best position [163].

The best settings for φ_1 and φ_2 is, derived from theoretical studies and benchmark tests, $\varphi_1 = \varphi_2 = 2$ or $\varphi_1 = 2.5$ and $\varphi_2 = 1.5$. When the constriction factor is used with PSO, typically $\varphi_1 + \varphi_2 = 4.1$.

φ_1 and φ_2 can also be related with ω, such that $\varphi_1 = \varphi_2 = 1.4961$, when $\omega = 0.72984$; $\varphi_1 = \varphi_2 = 1.193$ when $\omega = 0.721$ [24, 198].

For the tests of this work, except for the constriction factor approach, $\varphi_1 = 0.5$ and $\varphi_2 = 0.3$, since a priori tests conducted by the author revealed that, for the suit of test functions considered, the algorithm is most likely to succeed.

Tests and Results

In this subsection, different velocity update equations will be tested and compared in order to find which is (or are) the best equation(s) among all the velocity update equations considered in this study.

Table 3.5 complements the information that will be presented bellow and shows a comparison between the effects of the different particle's velocity update equations. Beforehand, it is important to note that the non-linear inertia term is not, by any means, a good approach for finding the roots of a given function, since it was not successful in all the executions.

Table 3.5: Auxiliary table used to compare the effect of the different particle's velocity update equations on the PSO algorithm.

Equation	Median root value	Median no. iterations	Median execution time (s)	No. roots not found
Constant	8.805e−06	83	0.046	63.429 %
Linear	6.926e−06	75	0.021	68.357 %
Non-linear	n/a	n/a	n/a	100.000 %
Stochastic	5.881e−06	44	0.012	75.857 %
Constriction factor	3.805e−06	128	0.031	80.000 %

Since the populations' variances are considered to be not equal ($p \approx .002$), a Kruskal–Wallis test was performed to explore the different equations effects on the value of the roots found by the algorithm, i.e., which velocity update equation(s) found more accurate root values. In this regard, the relation between the different roots' values and the velocity

update equations showed to be statistically significant at the level of significance of .05 [$\chi^2(3) \approx 98.144,\ p < .001$].

Using the Bonferroni adjustment, Dunn's post hoc tests were carried out on each pair of equations, finding that the velocity equation that accommodates the constriction factor was statistically significantly different from all the other pairs (for all cases, [$p < .001$]). Nevertheless, no statistically significant differences were detected between the linear and stochastic equations ($p \approx .487$) and in between the linear and the constant inertia term ($p \approx .508$). Referring to the Table 3.5, it is possible to conclude that the constriction factor term is the best strategy for updating the particles' velocity, since it found, in terms of median value, the lowest (i.e., more accurate) root approximation values.

In turn, a Kruskal–Wallis test showed that choosing a specific velocity update equation statistically significantly affects, at the level of significance of .05, the number of iterations of the PSO algorithm [$\chi^2(3) \approx 93.787,\ p < .001$]. (The populations' variances are considered to be not equal [$p < .001$].) The pairwise comparisons using the Dunn's test then revealed that the stochastic update equation performed statistically significantly different from the other velocity update equations (in all comparisons, $p < .001$). In addition to this, no statistically significant differences were detected between the pairs constant-linear ($p \approx .285$) and constant-constriction factor ($p \approx .673$). Thus, using the information available in Table 3.5, it is possible to conclude that, in terms of the number of iterations, the best velocity update equation is the one with a stochastic inertia term.

Following a similar approach, after verifying that the variance between the groups was not equal ($p < .001$), a Kruskal–Wallis was conducted to compare the effect of the different velocity update equations on the execution time of the PSO algorithm. Tests showed that there are statistically significant differences in the execution time between all the update equations [$\chi^2(3) \approx 40.009,\ p < .001$].

Dunn's test revealed again that, at the level of significance of .05, the stochastic velocity update equation is statistically significantly different from the other equations (with $p < .001$, for all pairs). On the other hand, the constant, linear and non-linear inertia statistics did not provide sufficient evidence that they are statistically significantly different. Taking into account the values presented in Table 3.5, the shortest execution times are associated with velocity update equation with a stochastic inertia term, being this the best approach for updating the particles' velocity when considering the algorithm's execution time.

To assess the influence of the different velocity update equations on the number of roots found by the PSO algorithm, a Pearson's chi-squared test was carried out and found that the velocity update equations are associated with the number of roots found by the PSO algorithm [$\chi^2(4) \approx 640.295,\ p < .001$]. Using the same test, but with the Yates' correction for continuity, the statistical evidence provided showed that the constant inertia found more roots when compared to the other velocity update equations. (For all comparisons with the constant inertia, $p < .001$, except for the linear inertia, where $p \approx .007$).

As can be seen in Table 3.6, if one is interested in obtaining accurate root approximations, then one should choose the constriction factor approach. However, the constant inertia should be chosen when the objective is to increase the number of successful executions. Nevertheless, to leverage the efficacy of the PSO algorithm, the stochastic inertia term is the best option. In the context of this work, the constriction factor approach will be, from now on, the strategy used for updating the velocity of the particles.

Table 3.6: Comparison of the different particle's velocity update equations (X means performed better.)

Equation	Root value	No. iterations	Execution time	No. roots not found	Total
Constant				X	**1**
Linear					**0**
Non-linear					**0**
Stochastic		X	X		**2**
Constr. factor	X				**1**

3.4.5 Communication Structure

Selecting the best swarm topology for a given problem is crucial for the successful execution of the algorithm [73]. However, the task of choosing the best swarm topology has shown to be complex, since no swarm topology is superior to all others [199] when different optimisation problems are considered.

Studies were done to understand better the different impacts of the swarm communication structure in the algorithm [13, 33, 199], all concluding that the swarm topology is problem-dependent and cannot be easily generalised to all optimisations problems.

In general, a faster convergence speed can be obtained when gbest social communication structure is used, and thus a better performance when compared to the lbest model. On the other hand, lbest models are better to escape from a local minimum, particularly on multimodal problems. However, they require more iterations.

Engelbrecht [200] even reported that the gbest and lbest are equivalent, in terms of solution accuracy, based on 60 benchmark tests.

In the subsequent sections, the different static communication structures will be introduced and its effects on the PSO for root-finding, as before, will be assessed and compared in order to find the optimal communication structure(s), based on the effectiveness and efficiency.

All-Connected-To-All

The all-connected-to-all communication structure was the first swarm communication structure introduced with the PSO algorithm [10, 11], and it is characterised by a fully connected network, as illustrated in Figure E.1 (refer to Appendix E).

In this communication structure, particles are all connected to other particles and spread information among all particles in the swarm, making the best performing particle in the population the only social source of influence [201]. The other particles, in turn, are biased towards the position of that particle.

Because this communication structure has a greater connectivity when compared to other communication structures, it can speed up the converge of the algorithm. However, there is less exploration of the search space, and thus the algorithm is susceptible to being trapped into a local minimum. Therefore, this communication structure should be used in unimodal optimisation problems with a simple search space.

Ring and Pyramid

In a ring communication structure, each particle has two immediately adjacent neighbourhoods, based, e.g., on the index of each particle, which they share information with, as can be seen in Figure E.2a.

Since particles have few connections between them, the converging speed of the algorithm is slower when compared to the gbest communication structure, being more useful for multimodal optimisation problems.

Particles are, thus, influenced by the best of the two neighbours, making indirect, remote communications and dividing the population into parts, so that different areas of the search space can be explored simultaneously until the best position of the areas have been found and influence all particles towards that position.

On the other hand, the pyramid communication structure is similar to the ring communication structure; however, all particles have tree connections, being one of them with a common neighbour to all particles, as illustrated in Figure E.2b. This particle can be chosen randomly, or other different strategies can be employed to choose it, such as the best or worst particle in the swarm, or even the most distant particle in the swarm.

For this study, the common particle was chosen randomly at the beginning of the execution of the PSO algorithm and did not change throughout the iterations.

Random

A random communication structure, as presented in Figure E.3, is also an alternative for the swarm communication structure. For this communication structure strategy, each particle has a random number of connections, as well as a random set of neighbours.

In this communication structure, each particle can be affected by any particle in the swarm. Connections between the particles, in its turn, can be bidirectional as well as unidirectional, and can be changed randomly throughout the algorithm, e.g., when no improvements are detected in the global or local best particle in the swarm [202].

Howsoever, the successful execution of the PSO algorithm is always dependent on the luck factor associated with the random number of neighbours, as well as which neighbours each particle will neighbour with.

In this case, and for the sake of simplification, the number of unidirectional links between each particle was generated randomly at the beginning of the algorithm. Each particle has at least one connection to another particle, and at most, it can be linked to all particles in the swarm.

Moreover, the set of neighbours of each particle was also generated randomly at the beginning of the algorithm and did not change during the execution of PSO.

Mesh and Toroid

In a mesh communication structure, particles are connected in a two-dimensional grid network layout, where each inner particle is connected to the left, right, upper, and lower neighbourhoods, and the outer particles are connected to the left and right neighbourhoods when possible, as can be seen in Figure E.4a. As other lbest social topologies, the information is still shared among all particles in the swarm; however, delayed.

Another variant is the toroid communication structure, also known as von Neumann communication structure, in which every particle has four connections in a tree-dimensional grid network layout. Inner particles are still connected to the left, right, upper and lower neighbourhoods; however, outer particles are connected to the opposite particles, enclosing the edges, as illustrated in Figure E.4b.

Studies reported that the toroid communication structure [33, 201, 203] performed, in general, better than any other swarm communication structure. Howsoever, all the communication structures will be tested in this work.

Star

In a star swarm communication structure, particles are independent or isolated from others, except one central particle (also known in the literature as the focal point), that intermediates the communication between the particles.

As illustrated in Figure E.5, the central particle knows the performance of every particle in the swarm and changes its position according to the best particle's position. If its new position is better than the previous best global position, then the central particle is responsible for spreading this discovery [162, 204]. Thus, this communication structure is centralised, since the central particle is influenced and is the only particle that influences the remaining particles.

From all the communication structures considered, this is the one that has the lowest converge speed. However, it must have a higher probability of achieving good results, when compared to other topologies, especially in multimodal search spaces.

The central particle can be randomly chosen at the very beginning of the algorithm, and be the same until the end (this was the strategy followed in this work); however, it can vary throughout the algorithm's execution and may not be necessarily chosen at random but take into account the state of the search in the search space.

Tests and Results

Finally, auxiliary Table 3.7 shows the comparisons of the different swarm's communication structures effects in the behaviour of the algorithm.

Table 3.7: Auxiliary table used to compare the effect of the different swarm's communication structures on the PSO algorithm.

Architecture	Mean root value	Median no. iterations	Median execution time (s)	No. roots not found
All-connected-to-all	4.884e−06	42	0.005	81.100 %
Mesh	6.306e−06	113	0.028	74.200 %
Pyramid	5.424e−06	67	0.014	80.300 %
Random	6.106e−06	66	4.788	76.800 %
Ring	6.809e−06	165	0.041	72.400 %
Star	3.988e−06	79	0.016	85.100 %
Toroid	6.508e−06	116	0.025	72.800 %

Starting with the root value, a one-way ANOVA was performed to find if the communication structure of the swarm influences the result of the execution of the algorithm. Results found, at the level of significance of .05, that there was a statistically significant

effect of the swarm's communication structure on the roots found by the PSO algorithm [$\mathrm{F}(6, 1566) = 16.207,\ p < .001$].

Besides that, the Tukey post hoc tests revealed that the star communication structure is statistically significantly different from the other communication structures, except for the all-connected-to-all, that did not show to have statistically significant difference ($p \approx .215$). In fact, according to Table 3.7, the star communication and the all-connected-to-all structures showed to produce, on average, more accurate root approximations.

The number of iterations required for the algorithm to stop has also revealed, according to the Kruskal–Wallis test, at the level of significance of .05, that is statistically significantly different among the communication structures [$\chi^2(6) \approx 244.041,\ p < .001$]. (The Kruskal–Wallis was chosen because the populations' variance are considered to be not equal [$p < .001$]).

In this view, the all-connected-to-all communication structure revealed to be statistically significantly different from all other communication structures, with all $p < .001$. Thus, the all-connected-to-all architecture required fewer iterations, in terms of median value, to meet the stopping criteria when compared to the remaining communication structures. These results were found by performing a Dunn's post hoc test with the Bonferroni adjustment, at the level of significance of .05, and by the results presented in Table 3.7.

After checking that, according to the Levene's test, the population's variance is considered to be not equal ($p < .001$), a Kruskal–Wallis test was conducted to compare the effect of the different communication structures on the execution time of the PSO algorithm. The test showed a statistically significant effect on the execution time of the PSO, at the level of significance of .05, for the seven communication structures [$\chi^2(6) \approx 244.041,\ p < .001$].

Post hoc comparisons using the Dunn's test then indicated that, similar to the previous tests, the all-connected-to-all was the communication structure that has statistically significantly differences from all the other communication structures considered for this study (all $p < .001$). When analysing the data from Table 3.7, it can be seen that the all-connected-to-all is the best approach in terms of the median value of the execution time and the random the worst approach, that also revealed to be statistically significantly different from all other pairs, requiring a higher execution time.

To end this test set, the number of roots found by each swarm's communication structure was also compared. Using the Pearson's chi-squared test, it was found a significant relationship between these communication structures and the number of roots found [$\chi^2(6) \approx 79.231,\ p < .001$]. Using the same test, but with the Yates' correction for continuity, the pairs were compared, and the ring architecture revealed to not have a statistical evidence to be different from the mesh ($p \approx .390$) and toroid ($p \approx .880$). These three architectures had, according to Table 3.7, the highest success rate.

As can be seen in Table 3.8, the all-connected-to-all communication structure revealed to be the best communication structure approach, since higher accurate root values and a higher efficiency was observed, when compared to the other topologies; however, it comes with the cost of a medium success algorithm's rate.

It is also interesting to observe that the all-connected-to-all architecture was the communication structure that revealed to have the faster convergence speed, as well as the one with the lowest number of iterations. On the other hand, the lbest models needed more iterations; however, those architectures showed to have better convergence characteristics.

Table 3.8: Comparison of the different swarm's communication structures (X means performed better.)

Architecture	Root value	No. iterations	Execution time	No. roots not found	Total
All-connected-to-all	X	X	X		**3**
Mesh				X	**1**
Pyramid					**0**
Random					**0**
Ring				X	**1**
Star	X				**1**
Toroid				X	**1**

3.4.6 Summary

The tests performed in the last sections allowed to conclude that the best combination of parameters found is given by 24 particles, organised in an all-connected-to-all topology, using the constriction term in the particles' velocity update equation. These parameters showed to find more accurate root approximations with a lower execution time and a reduced number of iterations when compared to other sets of parameters.

On the one hand, increasing or decreasing the number of particles revealed to not affect the number of iterations. On the other hand, as one would expect, increasing the number of particles impose an increase in the execution time. Although swarms with 24 particles found more accurate roots, swarms with 48 and 60 were able to escape from local minima and found more roots. (This is because there are more sources of information about the most propitious areas in the search space and, consequently, more information sharing.)

The non-linear decreasing inertia was the worst approach for updating the particles' velocity, while the stochastic velocity equation revealed to be the best strategy in terms of efficiency. Besides that, there was not a unanimous decision between the use of the constant inertia value (that revealed to have the highest success rate) or the constriction factor approach (that found more accurate roots), being the user/researcher responsible for choosing the best communication structure according to her/his needs.

Furthermore, the all-connected-to-all topology was the architecture with the fastest convergence speed, as well as the one that found more accurate roots; however, lbest architectures were better in escaping from local minima, and thus increasing the number of times that PSO ends successfully.

3.5 Multi Root-Finding Particle Swarm Optimisation

In the previous sections, the PSO was introduced and adapted for root-finding. This algorithm finds, at every execution, only one root approximation. However, in many Engineering, Physics, Chemistry and Economics problems, the common situation is to have and find multiple roots.

In fact, the PSO algorithm could be executed several times in order to find the multiple roots for a given non-linear function. Notwithstanding, the PSO algorithm will tend to

converge to the same roots, and the other (possible) roots seldom are found.

In this view, the Multiple Root-Finding Particle Swarm Optimisation (MRF-PSO), introduced in this section, is proposed to find, for a given non-linear equation or system of non-linear equations, its distinct roots by using multiple swarms that explore different areas of the same search space simultaneously.

In the previous section, swarms with 24 particles organised in an all-connected-to-all communication structure, with the constriction term on the particles' velocity update equation showed to be the best set of parameters in terms of accurate root values, the number of iterations, and also in terms of execution time. Thus, this set of parameters will also be used in MRF-PSO; despite this, this algorithm can be used with all other parameters introduced in Section 3.4.2.

In the MRF-PSO algorithm, multiple swarms are placed in the search space and the algorithm stops when no swarms are exploring it. Swarms do not exchange information among them but between them and a master. The master, as can be seen in Figure 3.2, stores the information about the roots found by each swarm and is responsible for synchronising the execution of each swarm and for spreading the knowledge gained during the execution of the algorithm. In this figure, r denotes the list of roots found, and r_n the root found by Swarm n.

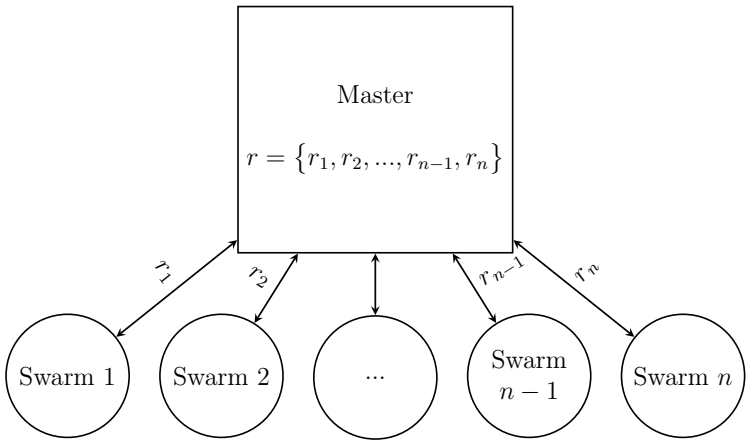


Figure 3.2: The MRF-PSO architecture for sharing information about the roots found during the search process of each swarm.

The MRF-PSO algorithm can be seen as multiple instances of the PSO algorithm running in parallel, where each swarm explores the search space and waits for the other swarms to finish, in order to exchange information.

Some strategies, however, must be applied for the purpose of detecting equal roots and to prevent particles to search in the same search areas where a root was found before.

This algorithm was developed and tested using the Python general-purpose programming language at version 3.7. To leverage the execution time, the Ray framework [205] was used to create a system's process for each swarm. Thus, in multi-core processors systems, several swarms can explore the search space simultaneously. Generally speaking, this means that the time required to find a root is approximately equal to finding multiple roots.

It is important to note that a parallelism architecture was followed to develop MRF-PSO, not a multithreading scheme, by the fact that, in a parallelism paradigm, processes can run

at the same time on different cores. On the other hand, in a multithreading scheme, processes are run in the same core, not substantially improving the execution time of PSO (or any other algorithm).

3.5.1 Detecting Equal Roots

For the master to detect equal roots, a new parameter was added to the PSO algorithm: the absolute tolerance parameter (e), such that

$$nr(\vec{x}, r, e) = \begin{cases} 1, & \left|(\vec{x})_i - (r_j)_i\right| > e \ , \forall i \in 1, ..., d, \\ & \qquad\qquad j \in 1, ..., z, \\ 0, & \text{otherwise}, \end{cases} \tag{3.16}$$

where d is the number of dimensions of the search space, r is the set of roots found by the MRF-PSO and z is the number of roots in r, i.e., $z = |r|$. Thus, if $nr(\vec{x}, r, e) = 1$, then $\vec{x}$ is appended to r, meaning that a new distinct root was found.

Since the execution of the nr function on all swarms is dependent on the order of evaluation, swarms are sorted by ascending order of its cost value (i.e., the root value), such that swarms that found a lower cost value have a higher probability of being appended to r.

To clarify this, consider that $z = 0$, e.g., after the first iteration. (In the MRF-PSO algorithm, the iteration number is incremented after the execution of the PSO algorithm in all swarms.) Now consider that four swarms were placed in a two-dimensional search space and $e = 0.1$.

Swarm 1 found a root at $(1, 2)$ with a root value of 0.01. On the other hand, Swarm 2 found a root at $(1.01, 2.01)$, but with a root value of 0.001. Swarm 3 and 4 found distinct root positions, then the root positions found by those two swarms will be appended to r. Nevertheless, Swarm 1 and Swarm 2, according to the absolute tolerance parameter, found the same root.

In this example, the position found by Swarm 2 represents a more accurate root position than the position found by Swarm 1. However, if Swarm 1 was the first swarm to be evaluated in nr, then its root position will be appended to r, and the root found by the Swarm 2 will be discarded, since a similar root was already found (in this case, by Swarm 1).

Thus, not considering Swarms 3 and 4, if the swarms were sorted by its cost value, then root found by Swarm 2 would be appended to r, and Swarm 1 would be discarded since a similar root (found by Swarm 2) was already in r.

The example presented before is intended to show the importance of sorting the swarms by its cost value after running the PSO algorithm in each swarm. It is important to note that the root values presented before were chosen to illustrate why ordering swarms is important, and do not represent, by any means, the required accuracy imposed to MRF-PSO.

3.5.2 Particle Positioning

Besides the absolute tolerance parameter, another parameter was added to the MRF-PSO algorithm, namely the maximum time to live parameter. This parameter defines how many times should the particles of a given swarm be repositioned in the search space.

Swarms (and its particles) are repositioned in the search space when they found a root that was already found before; however, unlike the algorithm initialisation, particles are not distributed uniformly in all the search space using the Algorithm 1.

As mentioned before, the master is the mean for sharing information about the roots found by other swarms. This information is used not to position particles in places in the search space that have already been detected as being roots. Thus, the algorithm has a higher probability of converging to a root that was not found before.

It is noteworthy that swarms that found distinct roots are removed from the search space.

As an example, consider Figure 3.3. Three roots were detected by Swarm 1, Swarm 3 and Swarm 5. Since those swarms successfully found roots that were not found before, they were removed from the search space. Particles of the remaining swarms, in turn, are repositioned uniformly in the search space in the areas where roots were not detected, i.e., in the remaining search space.

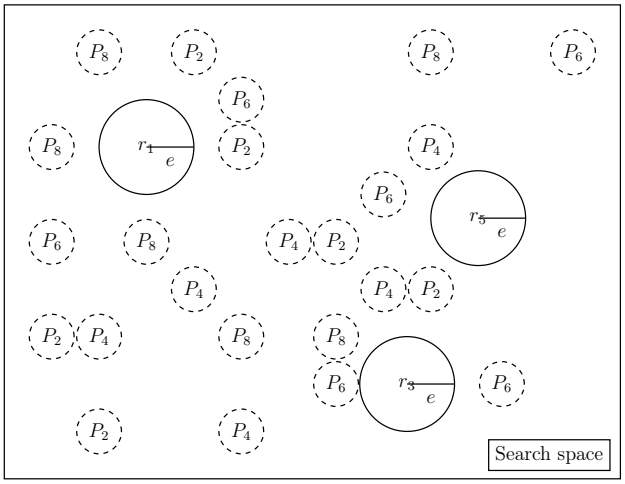


Figure 3.3: Example of particle repositioning when three roots were discovered by Swarms 1, 3 and 5.

Besides that, the number of iterations of the repositioned swarms is reset, since its particles will explore a new search area. However, when the number of times that each swarm is repositioned in the search space reaches the maximum time to live, swarms are removed from the search space and are not repositioned again.

It is important to note here that unsuccessfully swarms, i.e., swarms that were not able to found any root until the maximum number of iterations was reached are directly removed from the search space, without being repositioned, even though they have never been repositioned yet.

The flowchart of Figure F.1 summarises the workflow of the MRF-PSO algorithm.

3.5.3 Examples of Execution

Similarly to what was done for the PSO algorithm, a set of well known test functions [179,206] were chosen to test if the MRF-PSO algorithm was able to find, within a reasonable execution time, all roots for a given non-linear equation or a system of non-linear equations. The convergence of the algorithm was also tested by the average number of roots found.

The test functions used to test the PSO algorithm for the root-finding problems have only one root, thus they were not used here to test the MRF-PSO, since it is an algorithm developed to find approximations for the multiple roots of a given function simultaneously.

It is important to note that the chosen test functions are non-linear equations or system of non-linear equations, and that finding approximations for roots of non-linear functions are one of the most complicated mathematical problem currently in discussion in the research community.

It is also noteworthy that, for root-finding, systems of non-linear equations were considered as a single non-linear equation, such that:

$$f(\cdot) = |f_1(\cdot)| + |f_2(\cdot)| + \ldots + |f_n(\cdot)|, \tag{3.17}$$

where $f_n(\cdot)$ denotes the n-th non-linear equation of a given system of non-linear equations.

Each test function was tested 500 times, and the results are presented below in terms of the average values of all the trials. Except for Price 4 Function (Example 3.5.6), where $\epsilon = 10^{-5}$, for all other test functions $\epsilon = 10^{-12}$. That is, the computed roots approximations had a function value less or equal to 10^{-12}. Finally, $e = 0.1$ and the time to live parameter was set to be equal to 5. The rest of the parameters were kept from the previously executed tests with the PSO algorithm.

The tests were executed in a machine with Dual-Core Intel Core i7 CPU running at 1.70 GHz, and with 8 GB of RAM. It is worth mentioning that, until the date of writing of this chapter, the Ray framework [205] was only available for Linux-based operations systems. So, a different machine than the one used in previous tests was used.

It is important to note that the effectiveness is computed by dividing the average number of roots found by the total number of known roots. On the other hand, the execution time, given in seconds, is the time required by the algorithm to finish its execution, that is, until there are no more swarms exploring the search space, even though the algorithm may have discovered all the roots at the end of the first few seconds. This metric is usefully to study, since in most situations the number of roots is not known. On the other hand, it is also reported the execution time that the algorithm required to find all the known roots, even if the algorithm continues to run.

Example 3.5.1 (Himmelblau's objective function)**.**

$$f(x, y) = \begin{cases} 4x^3 + 4xy - 42x + 2y^2 - 14 = 0 \\ 2x^2 + 4xy + 4y^3 - 26y - 22 = 0 \end{cases} \tag{3.18}$$

- Number of swarms placed in the search space: 18 swarms.
- Average number of roots found: 8.81 (effectiveness: 98 %).
- Number of know roots: 9.
- Average execution time: 31.62 seconds (22.74 seconds to find all the roots).

Example 3.5.2 (Merlet problem)**.**

$$f(x, y) = \begin{cases} -\sin(x)\cos(y) - 2\cos(x)\sin(y) = 0 \\ -\cos(x)\sin(y) - 2\sin(x)\cos(y) = 0 \end{cases} \tag{3.19}$$

- Number of swarms placed in the search space: 26 swarms.
- Average number of roots found: 13 (effectiveness: 100 %).
- Number of know roots: 13.
- Average execution time: 39.03 seconds (20.05 seconds to find all the roots).

Example 3.5.3 (Floudas problem)**.**

$$f(x,y) = \begin{cases} 0.5\sin(xy) - 0.25\dfrac{y}{\pi} - 0.5x = 0 \\ \left(1 - \dfrac{0.25}{\pi}\right)\Big(\exp(2x) - e\Big) + e\dfrac{y}{\pi} - 2ex = 0 \end{cases} \tag{3.20}$$

- Number of swarms placed in the search space: 4 swarms.
- Average number of roots found: 2 (effectiveness: 100 %).
- Number of know roots: 2.
- Average execution time: 6.40 seconds (4.03 seconds to find all the roots).

Example 3.5.4 (Robotic – Planar Parallel Manipulators)**.**

$$\begin{aligned} f(x) = k_0 + k_2x^2 + k_4x^4 + k_6x^6 + \big(k_1x + k_3x^3 + \\ k_5x^5\big)\sqrt{1-x^2} = 0, \end{aligned} \tag{3.21}$$

where $k_0 = 3.9852$, $k_1 = -8.8575$, $k_2 = -10.039$, $k_3 = 20.091$, $k_4 = 7.2338$, $k_5 = -11.177$ and $k_6 = -1.17775$.

- Number of swarms placed in the search space: 12 swarms.
- Average number of roots found: 4.68 (effectiveness: 78 %).
- Number of know roots: 6.
- Average execution time: 23.77 seconds (22.52 seconds to find all the roots).

Example 3.5.5 (Wayburn Seader 1 Function)**.**

$$f(x,y) = \big(x^6 + y^4 - 17\big)^2 + (2x + y - 4)^2 \tag{3.22}$$

- Number of swarms placed in the search space: 2 swarms.
- Average number of roots found: 2 (effectiveness: 100 %).
- Number of know roots: 2.
- Average execution time: 5.62 seconds (2.70 seconds to find all the roots).

Example 3.5.6 (Price 4 Function)**.**

$$f(x,y) = \big(2x^3y - y^3\big)^2 + \big(6x - y^2 + y\big)^2 \tag{3.23}$$

Table 3.9: Comparison of the MRF-PSO with some of the results available on the literature for Example 3.5.1.

Metric	**Value**
Effectiveness MRF-PSO	98 %
Effectiveness [206]	82 %
Effectiveness [207]	100 %
Execution time MRF-PSO	31.62 (22.74)
Execution time [206]	19.917
Execution time [207]	5

- Number of swarms placed in the search space: 6 swarms.
- Average number of roots found: 3 (effectiveness: 100 %).
- Number of know roots: 3.
- Average execution time: 7.91 seconds (4.59 seconds to find all the roots).

The MRF-PSO algorithm showed to be a viable approach for finding the roots of a given non-linear equation or a system of non-linear equations. Even though it only presented difficulties in Example 3.5.4, for the other examples, the effectiveness was always 98 % or more.

Results of Examples 3.5.1–3.5.4 were compared, in terms of the number of the roots found and execution time, with some results available on the literature about the use of stochastic optimisation algorithms, different from the PSO approach, for solving systems of non-linear equations [206–210]. These results were summarised in a paper authored by Ramadas et al. [206], that also proposed a metaheuristic algorithm for finding multiple roots of systems of non-linear equations. The results of the comparison between MRF-PSO and other approaches are presented in Tables 3.9–3.12.

As can be seen, despite not being able to be superior in all considered approaches, the MRF-PSO revealed to be a promising approach in terms of effectiveness, especially when compared to the results presented by Ramadas and collaborators [206].

However, it is possible to note that, in most cases, the MRF-PSO required a higher execution time when compared with the other algorithms. Possible justifications for this result may arise from the different algorithm accuracy considered, and from the programming language used to develop both the algorithms. Nevertheless, it is still a weakness of the MRF-PSO algorithm to be considered in future work.

3.6 Conclusion

In this chapter, the PSO algorithm was described in both theoretical and practical approaches. The algorithm was divided into processes and, for each process, a description was given.

Table 3.10: Comparison of the MRF-PSO with some of the results available on the literature for Example 3.5.2.

Metric	**Value**
Effectiveness MRF-PSO	100 %
Effectiveness [206]	61 %
Effectiveness [207]	100 %
Effectiveness [208]	100 %
Effectiveness [209]	100 %
Execution time MRF-PSO	39.03 (20.05)
Execution time [206]	0.881
Execution time [207]	46
Execution time [208]	20
Execution time [209]	0.03

Table 3.11: Comparison of the MRF-PSO with some of the results available on the literature for Example 3.5.3.

Metric	**Value**
Effectiveness MRF-PSO	100 %
Effectiveness [206]	100 %
Effectiveness [207]	100 %
Effectiveness [208]	100 %
Effectiveness [209]	100 %
Execution time MRF-PSO	6.40 (4.03)
Execution time [206]	0.505
Execution time [207]	0.607
Execution time [208]	0.461
Execution time [209]	0.03

Table 3.12: Comparison of the MRF-PSO with some of the results available on the literature for Example 3.5.4

Metric	**Value**
Effectiveness MRF-PSO	78 %
Effectiveness [206]	88 %
Effectiveness [210]	100 %
Execution time MRF-PSO	23.77 (22.52)
Execution time [206]	5.345
Execution time [210]	n/a

During this work, PSO was adapted for root-finding, by changing the way of computing the cost of each particle, and by adding a strategy to ensure that particles stay within the search space's bounds.

This algorithm was then tested with different parameter settings, and conclusions were drawn regarding the parameters that showed the best results in terms of accuracy, the number of iterations, execution time and the number of roots not found.

Thus, the best combination of parameters was found to be 24 particles, in an all-connected-to-all communication structure, in which the particles' velocity should be updated using the velocity update equation that integrates the constriction term (Equation (3.7)).

This optimal parameter setting was then used in a proposed PSO algorithm variant that concurrently finds approximations to the roots of a given non-linear equation or the solutions of a system of non-linear equations.

Finally, some well known functions were used to test the MRF-PSO and results were compared to other results available on the literature. The MRF-PSO algorithm has shown to be effective in finding multiple roots. However, it demonstrated a weak point, which is the need of a considerable execution time in order to find all of the roots of a given function or to end its execution. This part will be considered in future works.

Chapter 4

A Neural Network-Based Approach for Approximating the Roots of Polynomials

> — The most beautiful thing we can experience is the mysterious. It is the source of all true art and science.
>
> *Albert Einstein* (1879–1955)

Abstract – Finding arbitrary roots of polynomials is a fundamental task in various areas of science and engineering. A myriad of methods was suggested to address these tasks, such as the sequential Newton's method and the Durand–Kerner (D–K) simultaneous iterative method. The sequential iterative methods, however, need to use a deflation procedure in order to compute approximations to all the roots of a given polynomial, which can produce inaccurate results due to the accumulation of rounding errors. In turn, the simultaneous iterative methods require good initial guesses to converge. On the other hand, Artificial Neural Networks (ANNs) are widely known by their capacity to find complex mappings between the dependent and independent variables. This chapter aims to determine, based on comparative results, whether ANNs can be used to compute approximations to the real and complex roots of a given polynomial, as an alternative to simultaneous iterative algorithms like the D–K method. Although the results are very encouraging and demonstrate the viability and potentiality of the suggested approach, the ANNs were not able to surpass the accuracy of the D–K method. The results indicated, however, that the use of the approximations computed by the ANNs as an initialisation scheme for the D–K method can be beneficial to the accuracy of this technique.

4.1 Introduction

An Artificial Neural Network (ANN) is a biologically inspired model that enables a system to learn, as living animals do, from observational data. Today, ANNs are one of the main tools used in machine learning for a myriad of applications.

Each ANN is composed of a set of artificial neurons that are interconnected by synaptic connections, as an analogy to the biological neural network. Mathematically, an ANN is basically a directed graph with vertices and edges.

Every synaptic connection, in turn, has a weight, also known as strength, that will be adjusted according to the examples provided and the feedback from the error (i.e., the difference between the estimated and the true target values).

ANNs are, thus, an adaptive system that maps a given input to a desirable output, having been proven, by the Universal Approximation Theorem [211–213], to be a universal approximator to any continuous function, in a d-dimensional space.

Today, ANNs have shown to be successful in various areas, such as disease risk prediction [214], natural language processing and speech recognition [215, 216], image recognition to identify people or objects [217, 218], among other applications [219].

Notwithstanding, there are some applications (e.g., image recognition) that the ANNs are not even close to human performance (in terms of processing speed and accuracy) since only a small fraction of the functioning of our neural circuit is known [220] and it is not possible yet to implement that behaviour on an ANN. (In fact, the Drosophila and Nematode's brain are the only two brains that are known in detail, and it took over 100 years of research [221,222].)

Polynomial Root-Finding

Finding the arbitrary (real or complex) roots of a given polynomial is a fundamental task in various areas of science and engineering. Applications of root-finding emerge from, e.g., control and communication systems, filter design, signal and image processing, codification and decodification of information.

Most of the methods available in the literature are based on Newton's method or derived from it, such as the Durand–Kerner (D–K) method [223, 224]. The D–K method is an iterative method that finds all the roots of a given polynomial simultaneously, although requires very good starting approximations for all the roots in order to converge.

Due to the drawbacks mentioned above, and since traditional ANNs, or shallow neural networks, are well known for their capability to model data and to find good approximations for complex problems, in this chapter, a different approach for finding the real and complex roots of polynomials based on neural networks is tested, in order to assess its potentiality and limitations in terms of efficiency and accuracy. (It is important to note that this approach uses the inductive inference in order to find the roots of a given polynomial simultaneously.)

The ANN-based approach is compared with D–K method, one of the most traditional simultaneous iterative methods for polynomial root-finding. Finally, the approximations computed by the ANNs are used as an initialisation scheme for the D–K method.

Thus, this chapter is organised as follows: after the introduction, Section 4.2 introduces the key concepts of ANNs. Section 4.3 presents a summarised state-of-the-art from the initial proposals of ANNs up to the current developments. Section 4.4 offers a description of related work about the use of ANNs for root-finding and Section 4.5 and 4.6 describe respectively the methodology followed to address the same problem and the comparative results between the ANN-based approach and the D–K method. In turn, Section 4.7 suggests an enhanced version of the previously proposed approach by using Particle Swarm Optimisation (PSO) as a training algorithm. Finally, the ANN-based approach is tested, in Section 4.8, as an initialisation scheme for the D–K method. This chapter ends with a section dedicated to conclusions.

4.2 Feedforward Artificial Neural Networks

ANNs are composed of artificial neurons organised in layers, typically an input layer, some hidden layers, and an output layer, as illustrated in Figure 4.1. The number of layers gives the depth of the ANN [225]; in this example, three. However, there is still no consensus regarding the depth of an ANN, as some researchers argue that because the input layer and output layer are not responsible for the learning process, they should not be counted for the depth of an ANN. In turn, other researchers claim that the depth corresponds to the total number of layers.

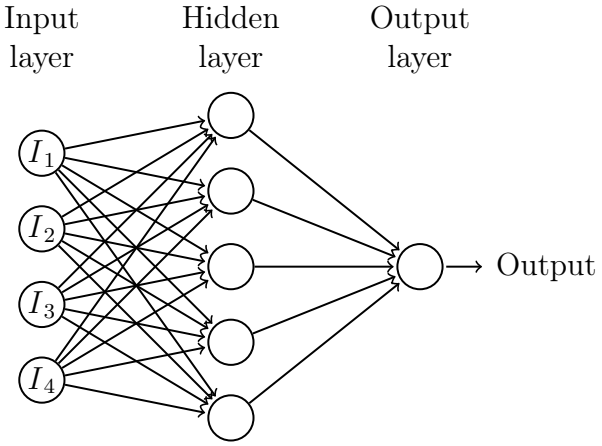


Figure 4.1: An example of a shallow ANN with four inputs nodes, five hidden nodes and an output node.

ANNs with only one hidden layer are called shallow networks. However, when the number of hidden layers is higher than one, the ANN is called a deep network.

Each layer has a set of neuron units that are only connected to the succeeding layer. This type of network architecture is known as Feedforward Artificial Neural Network (FNN) and will be the only one considered for this work, although other architectures exist (e.g., the recurrent neural networks).

Despite the number of input and output neurons are known, the number of units per each hidden layer is unknown, being the user/researcher responsible for deciding the rest of the ANN architecture, including the number of hidden layers.

These two hyperparameters control the effectiveness and efficacy of an ANN, being extremely important to find the best configuration for the problem at hands. The optimal parameters configuration is still an open question in research; thus, a trial and error strategy is often followed in order to find the optimal parameterisation.

4.2.1 Neuron Functions

Each neuron in an ANN performs, as can be seen in Figure 4.2, two functions: a transfer function and an activation function.

The transfer function is given by a weighted sum of the neuron inputs, i.e.:

$$T_j = \sum_{i=1}^{n} I_{ij} \cdot w_{ij}, \tag{4.1}$$

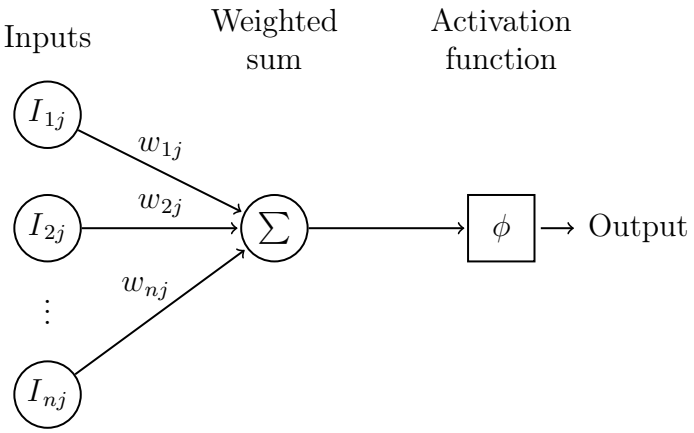


Figure 4.2: Behaviour of a single artificial neuron for shallow networks.

where n is the total number of inputs of the neuron j, $I_{.j}$ and $w_{.j}$ the inputs and the weights of those inputs, respectively.

After the transfer function has been computed, an activation function is applied to the transfer function, and the neuron's output is obtained, such that:

$$O_j = \phi\Big(T_j\Big). \tag{4.2}$$

This output will be then transmitted to the following neuron of the next layer, until the output layer. This transmission process is known as forward propagation [226].

Activation function

Like the biological neurons, the neuron units in an ANN are only fired when the amount of excitation is high; otherwise, they must remain silent [221].

An activation function, as the name implies, is the function responsible for activating or deactivating neuron units, and thus add variability to the dot products between the inputs and the weights of the synaptic connections.

There are two types of activation functions: linear (also known as identity function) and non-linear. Although linear activation functions can be used, non-linear functions are the most used type of activation function, since it enables the network to create non-linear mappings between the inputs and outputs. Heaviside, Logistic, Hyperbolic tangent and later the Rectified Linear Unit (ReLU) functions are the most known and used non-linear activation functions.

In most of the ANN applications (especially regression), hidden layers apply non-linear activation functions, whereas the output layer applies a linear activation function so that the final output is not restricted to the range of values of the previous activation function [227].

4.2.2 Number of Neurons

The input and output layers' number of units is, for most of the applications, reasonably easy to know.

The number of inputs units corresponds to the number of independent variables in the training set, i.e., the number of features or columns in the data set. On the other hand, the number of output neurons is related to the number of dependent variables (for regression problems) or the number of class labels (for classifications problems).

In turn, the number of units in the hidden layer is not straightforward. Like other bio-inspired methods, a trial and error methodology is often required to select the architecture that best fits the problem at hands, and that avoids overfitting or underfitting problems. Some strategies, however, exist.

Kolmogorov's Mapping Neural Network Existence Theorem [228], for example, states than any given continuous function in a d-dimensional unit cube can be implemented by using d neuron units in the input layer and $2d + 1$ in the hidden layers. The formal proof of this theorem can be found in [228].

Simple rule-of-thumb methods also exist, such as the number of hidden neuron units should be 2/3 of the number of neuron units in the input layer, in between the number of input and output layers or should be less than twice the size of neurons in the input layer [229]. On the other hand, Principe and colleagues [230] also suggested that the number of neuron units should be 1/10 of the number of observational data used for training the ANN.

Ultimately, pruning techniques or Evolutionary Algorithms (EAs) (such as the PSO algorithm) can be applied to ANNs [231], in order to produce the optimal ANN structure.

4.2.3 Number of Hidden Layers

ANNs with no hidden layers can only implement linear-separable functions [232]; however, most of the real-world problems are non-linear-separable, motivating the introduction of multiple hidden layers in an ANN.

The Universal Approximation Theorem [211–213] states that any ANN with one hidden layer with a sigmoid activation function and a linear activation function in the output function can approximate any function.

Although according to the Universal Approximation Theorem one hidden layer is enough, deep ANNs have a better generalisation capability; however, it comes with a much higher computational cost and with significantly more complex learning algorithms. Thus, apparently, it is better to start with a simpler ANN (e.g., with only one hidden layer), and, if necessary, increase the number of hidden layers based on the generalisation results.

Similar to the number of neuron units in the hidden layer(s), the optimal number of hidden layers can also be achieved using an EA algorithm with a performance measure, such as the Mean Squared Error (MSE).

4.2.4 Training Algorithms

In order to adjust the weights of the connections between neurons, a training algorithm is run. A training algorithm is an iterative process that, based on an optimisation method, is responsible for finding the optimal set of weights that leverage the ANN's performance, i.e., the weights that reduce as much as possible the error between the target and the value outputted by the network. The process of training a basic ANN can be described, in little detail, as follows:

1. The input units receive the feature values and send them to the hidden layer(s).
2. Based on a mathematical function, each node in the hidden layer outputs a value and passes this value to the next hidden layer (if it exists) or the output layer.
3. The output layer receives the outputs from the last hidden layer and computes the predicted value.
4. This predicted value is then compared to the expected value, and the error is used to update the weights of the synaptic connections, based on a training algorithm.

This training process is applied to each example in the training set until a stopping criterion is met (e.g., the minimum MSE, maximum number of iterations, among others) and is known as supervised learning, since the training set contains some examples of the mapping between the input and the output variables. (It is important to note that although unsupervised learning techniques have already been developed, they will not be studied in this chapter.)

There are two categories of training algorithms [227]: the ones that are based on the derivatives of the error function, and the ones that are based on stochastic methods, e.g., PSO and Genetic Algorithm (GA). The Newton's and Gauss–Newton method and the Levenberg–Marquardt Algorithm (LMA) are, in turn, examples of training algorithms based on derivatives.

Although derivative-based methods are the most used to train ANN, they cannot be used to, e.g., also optimise the architecture of the ANN, which can be done using a stochastic method.

In this work, both the categories of training algorithms will be studied. On the one hand, the LMA will be used as a training algorithm based on derivatives. On the other hand, the PSO algorithm will be used as a stochastic method.

4.3 State-of-the-Art

ANNs are currently an active research area that tries to mimic the human neural circuit, either in the way neurons interconnect and in the form of learning by examples (known as inductive inference), intending to enable machines to behave identically to intelligent agents [233].

McCulloch and Pitts [234], in 1943, developed the first computational model for the ANNs utilizing a linear model. According to these authors, neurons in the brain perform a weighted sum of their inputs, in order to implement any logical operation (such as the operations performed by a Turing machine).

In the late 50s and early 60s, Rosenblatt [235–237] extended the approach suggested by McCulloch and Pitts [234] and introduced the perceptron. Perceptrons are units that perform a weighted sum of their inputs and outputs a binary value when compared to a threshold. If the weighted sum of their inputs is greater than the pre-defined threshold, then the output of the perceptron would be 1. Otherwise, the perceptron unit would be disabled.

This work is considered to be the first model that can learn how to adjust the weights of the connections between neurons, in order to classify the inputs in two categories, based on previously given examples (i.e., supervised learning).

In 1965, Ivakhnenko and Lapa [238] introduced the first deep ANN and used the method of least squares with polynomial activation functions to train the connections' weights.

However, in 1969 Minsky and Papert [239], showed that the techniques developed so far were not able to solve some problems or applications that do not have a linear relationship between the independent variables and the dependent, or handle non-linear separable problems (such as the XOR function).

Nevertheless, the works developed by McCulloch, Pitts, Rosenblatt, Ivakhnenko and Lapa were the most relevant approaches that leveraged the development of the ANN of the current days [221].

Linnainmaa [240] introduced, in his Master's thesis in 1970, the backpropagation and automatic differentiation; however, without mentioning that it could be used to train an ANN, something that would be explored later by Werbos [241,242] and Rumelhart et al. [243], between 1974 and 1985. Until today, the backpropagation is still the most used training algorithm for ANN, and it was responsible, at the time, for the resurgence of research in machine learning areas.

In 1979, Fukushima [244] introduced the convolutional ANN using a gradient-based learning algorithm and in 1989, LeCun and colleagues [245] tested the Fukushima's approach, but using the backpropagation algorithm. They showed that the ANNs were able to classify, with high accuracy, handwritten digits.

In 1982, Hopfield [246] popularised the recurrent ANNs (also known as Hopfield networks) introduced by Little [247] in 1974, where neurons in the same layer can be connected, or a neuron unit from a frontmost layer be connected to a unit from a previous layer, enhancing the ANNs with memory mechanisms.

Ballard [248] in 1987 suggested the Autoencoder ANN, one of the most known unsupervised learning technique that tries to copy the inputs to the outputs using an encoder and a decoder module.

Despite all this studies and works about ANNs, in 1991, Hochreiter [249] reported that training a deep ANN with the backpropagation algorithm is hard due to the vanishing gradient problem, where the gradient gets smaller from layer to layer. The vanishing gradient problem causes the deeper layers not to be able to learn the data as well as the initial layers; thus, affecting all the learning process of the ANN and its results.

In fact, the idea presented by Hopfield [246] of the ANN model to mimic the human memory can be used to solve the vanishing gradient, since it flows back and forth the gradient in the ANN. This originated the Long Short-Term Memory (LSTM) ANN proposed by Hochreiter and Schmidhuber [250] in 1997. In the LSTM ANN, an LSTM neuron unit stores information from past iterations and from time to time that information is erased. Besides solving the vanishing gradient, like the Hopfield networks, the LSTM ANNs are especially useful for natural language processing, since the ANN can memorise the previous words of a given sentence, and thus have a better understanding about its meaning.

In 1995, Cortes and Vapnik [251] introduced the Support-Vector Networks for classifying data in two groups. The idea behind the Support-Vector Networks is to increase the dimensionality of the feature space to the point that categories can be separated linearly, and to maximise the distance between the support vectors, in order to improve the ANN's generalisation capabilities.

Two years later, in 1997, Bidirectional Recurrent ANNs were introduced by Schuster and Paliwal [252] with the objective of reducing the number of samples required to train

an ANN. Bidirectional Recurrent ANNs can be seen as two ANN connected to the same output layer. The first ANN is trained using a feedforward approach and the other by using backpropagation. Thus, the output layer can access information about past and future states.

These advances made ANNs gaining, again, traction among the scientific community, but the computational resources that existed at the time did not allow significant improvements.

In 2006, the Deep Belief ANNs were introduced by Hinton and Salakhutdinov [253]. The technique combined an unsupervised leaning before supervised training the ANN, in order to be used in classification tasks. Bengio et al. [254], in the same year, showed that the Deep Belief ANNs were able to generalise the test data set better than other unsupervised learning techniques.

Deep Belief ANNs were inspired by the autoencoder technique suggested by Ballard [248], since the first ANN's training (unsupervised) is responsible for reconstructing the ANNs' inputs, in order to extract the most important features.

Since then, many researchers started to develop parallel GPU-based versions of the back-propagation algorithm [255–257], and ANNs were wining contests, achieving, most of the times, the human performance [258].

Glorot et al. [259] popularised, in 2011, the ReLU activation function by testing and showing that very deep ANN for image classification tasks can be trained faster with ReLU than other activation functions (such as the hyperbolic tangent).

In 2012, Krizhevsky [260] introduced the AlexNet, a deep convolutional ANN for classifying 1.2 million images from the ImageNet data set.

At the same time, efforts were conducted to enhance the capabilities of the ANN for speech recognition [216, 261].

Srivastava et al. [262], on the other hand, in 2014, introduced the Dropout Regularisation technique, with the objective of reducing the overfitting problem (especially in deep fully-connected ANNs trained with few samples) and, thus, improving the generalisation capabilities.

As the name suggests, the dropout technique drops some neuron units from an ANN during the training phase by a given probability. Neuron units to be dropped are chosen from both the input layer and hidden layer(s) and have the opportunity to re-enter in the network.

In 2014, Kingma and Jimmy [263] introduced the Adam stochastic method for improving the learning ability of deep ANN and increasing the speed of training, by using an adaptive learning rate (also known as the step in classical stochastic gradient descent methods) for each neuron units throughout the learning process.

Goodfellow and collaborators [264] introduced, also in 2014, the Generative Adversarial ANN. The approach uses two ANNs competing with each other: a generative ANN and a discriminative ANN.

The generative ANN generates samples from a given distribution and the discriminative ANN is responsible for detecting if the data is true or generated by the generative ANN.

On the one hand, the objective of the generative ANN is to maximise the error in the discriminative ANN. On the other hand, the discriminative ANN strive to minimise the classification error (i.e., strive to detect fake samples better).

The two networks compete until an equilibrium state where the generative ANN is able to produce samples from the same distribution of the training data set, and the discriminative

ANN is able to identify successfully either if the data is from the training data set or was generated.

More recently, Vaswani et al. [265] introduced the concept of transformers in the ANN for natural language processing, as an alternative to the LSTM ANN. The transformer uses a set of encoders and a set of decoders modules (like those from Ballard [248] in 1987) and is identical to the recurrent ANNs; however, without connections of neuron units from the upper layer to lower layers.

The concept of transformers uses a self-attention and a mechanism of attention. The self-attention mechanism, located in both encoders and decoders, extracts what it considers to be the most important words in a sentence (i.e., keywords). Similarly, the mechanism of attention, located only in the decoder units, extracts the information that it considers to be the most important, but from the encoded words. Keywords then flow by all chained encoders which, in turn, take all the encoded keywords, process them, and output a sentence according to its deducted meaning.

Later, Devlin et al. [266] introduced the Bidirectional Transformers (also known as BERT).

The state-of-the-art presented here does not describe, by any means, all the approaches developed for ANN. The artificial intelligence scientific community is growing very fast, and many works are being published, mainly due to the fact of the development of open-source software libraries, such as TensorFlow [267] and PyTorch [268].

4.4 Related Work

Although there are many iterative methods to calculate one root or a pair of complex conjugate roots of a polynomial, such as the well known Laguerre's and Jenkins–Traub's methods [269], the determination of all roots of a given polynomial by one of such methods involves repeated deflations, which can lead to very inaccurate results due to the problem of accumulating rounding errors when using finite accuracy floating-point arithmetic.

Iterative methods for finding all roots of a polynomial simultaneously, such as the methods of D–K [223, 224] and Ehrlich–Aberth [270, 271], appeared in literature only in the 1960s. The simultaneous root-finding algorithms have the advantage of being inherently parallel; however, they need very good initial approximations for all the roots in order to converge.

On the other hand, the first works about finding the roots of a given polynomial with ANN-based approaches started in 1995, when Hormis and colleagues [272] presented the $\sum - \prod$ ANN with the objective of separate a two-dimensional polynomial into two one-dimensional polynomials (with the same degree).

The $\sum - \prod$ ANN includes two other sub-ANNs with only two layers: an input layer (that receives the values of each independent variable of each training pattern) and an output layer. These two layers are interconnected by synaptic connections associated with a weight, that represents the values of the independent variables in the two one-dimensional polynomials.

On the other hand, the output layer of each sub-ANN has only a single output, that corresponds to a weighted sum according to the weights of the synaptic connection and the coefficients of each independent variable.

The final output of the $\sum - \prod$ ANN consists of multiplying the weight of the output synaptic connection by the two sub-ANNs outputs. The output is then compared to the

two-dimensional polynomial's output using the MSE, and the error is backpropagated and the weights are adjusted.

Perantonis et al. [273] extended the approach proposed by Hormis and colleagues [272] in order to factorise a two-dimensional polynomial as a product of polynomials of lower order.

The ANN structure used contains two hidden neurons that correspond to each one of the factors of the two-dimensional polynomial. The hidden neurons perform a weighted sum of the inputs and then apply a logarithmic activation function. The output layer contains a single neuron unit that simply sums its two inputs.

Besides that, a Constrained Learning Algorithm (CLA) was applied, in order to incorporate additional prior knowledge about the relationship between the two-dimensional polynomial and the coefficients of the desired factored polynomials.

Interestingly, they used the Optimal Brain Surgeon (OBS) weight elimination technique, in order to remove small coefficients in the factored polynomials that should be zero, and with this increase the accuracy of the approach.

In 2000, Huang and Zhao [274] suggested an ANN structure for the same objective of polynomial factorisation. Their approach, however, was designed for factorisation of polynomials with more than two dimensions.

The proposed ANN structure is identical to the $\sum - \prod$ structure and consists of using two hidden layers and one output layer with a single product unit. The first hidden layer performs a weighted sum of its inputs (the values of each independent variables of the training pattern); in turn, the second hidden layer has a number of difference hidden neurons (one for each factored polynomial), that computes the difference between the weighted sum of its inputs and the training pattern. In turn, its outputs are passed to the final layer.

Although their proposal was able to factorise polynomials in several other polynomials with a lower degree, it required a significant number of iterations in order to obtain accurate solutions, mainly due to the fact that the gradient descent backpropagation algorithm was used.

In 2001, Huang and Chi [275] presented an approach for finding the real roots of a given polynomial. This is considered to be, by the author, the first work that addressed the root-finding with an ANN directly.

An n-th degree polynomial (with $n \geq 2$), $P(x)$, can be given by:

$$P(x) = a_0 + a_1 x + \ldots + a_n x^n, \tag{4.3}$$

where $a_n \neq 0$ and usually set to be equal to 1, without loss of generality.

In turn, the polynomial $P(x)$ can be factorised as follows [275]:

$$P(x) \approx \prod_{i=1}^{n} (x - w_i), \tag{4.4}$$

being w_i the calculated value of i-th real root of $P(x)$.

In order to find the real roots of a given polynomial, Huang and Chi [275] used a similar ANN to the one presented by Hormis and colleagues [272] (the $\sum - \prod$ ANN). In essence, the ANN has two inputs corresponding to the terms 1 and the training pattern x. The number of hidden neurons is set to be equal to the degree of $P(x)$, i.e., n.

Each neuron unit in the hidden layer is responsible for computing the difference between the training pattern x and the weight of the synaptic connection that connects the input

with term 1 and the respective hidden neuron. That is, the output of i-th hidden neuron is given by $x - 1 \cdot w_i$.

The single unit in the output layer, in its turn, computes the multiplication of its inputs. The final result of the network is then compared to the output of P for each training pattern x, and the weights of the synaptic connections between the input node with term 1 and each hidden neuron are updated according to the CLA suggested by Perantonis et al. [273].

In their approach, the additional a priori knowledge was incorporated in the form of relationships between the roots and the coefficients of $P(x)$, such that:

$$\begin{cases} \sum_{i=0}^{n} w_i + a_{n-1} = 0, \\ \sum_{i<j}^{n} w_i w_j + a_{n-2} = 0, \\ \vdots \\ w_1 w_2 \cdots w_n - (-1)^n a_0 = 0. \end{cases} \tag{4.5}$$

It is important to note that the weights of each synaptic connection correspond to one real root of $P(x)$.

In the same paper [275], these authors presented an approach for reducing the complexity of the computational calculation in each epoch. Thus, they suggested that the polynomial $P(x)$ can be refactored in order to find i roots at the time ($i < n$) as follows:

$$P(x) \approx \prod_{i=1}^{n} (x - w_1)(x - w_2) \cdots (x - w_i) P_b(x), \tag{4.6}$$

where $P_b(x)$ is the remaining $n - i$ order polynomial given by:

$$P_b(x) = b_0 + b_1 x + \ldots + b_{n-i} x^{n-i}. \tag{4.7}$$

The ANN's structure is changed in order to receive as input the training pattern values of $x, x^2, \cdots, x^{n-i-1}, x^{n-i}$. The number of hidden neurons is $i + 1$, that corresponds to the number of roots to be found at the time and an additional hidden neuron that computes the output of $P_b(x)$. The output of the output neuron is still a product of the $i + 1$ hidden neurons. The weights are now $\{w_1, w_2, \cdots, w_i, b_0, b_1, \cdots, b_{n-i}\}$ and are adjusted using the CLA [273]. This is an iterative process that terminates when all roots were found.

In the same year, these authors extended this approach to also compute the complex roots of a given polynomial [276], with coefficients as being complex numbers. The ANN was trained using the CLA and the relationships of Equation (4.5), that still hold in the complex case, were included as a priori knowledge.

This partitioning approach was then compared with non-neural traditional approaches for root-finding and revealed to be more efficient and effective [277].

In 2003, a dilatation method for finding close arbitrary roots of polynomials was suggested [278]. In essence, the idea is to rewrite the original polynomial as follows:

$$P(x)' = P(\beta x), \tag{4.8}$$

where β is the dilation factor ($0 < \beta < 1$), i.e., $P(x)'$ is given by:

$$P(x)' = P(\beta x) = \beta^n \Big(x^n + \frac{a_0}{\beta^n} + \frac{a_1}{\beta^{n-1}} x + \ldots + \frac{a_{n-1}}{\beta} x^{n-1} \Big). \tag{4.9}$$

This rewrite of $P(x)$ magnifies the distance between close roots, allowing them to be found more easily.

In 2004, Huang et al. [7, 279] integrated into the CLA the root moment method and the Newton identities.

The δ order root moment of a given polynomial $P(x)$ is given by:

$$S_\delta = w_1^\delta + w_2^\delta + \cdots + w_n^\delta = \sum_{i=1}^{n} w_n^\delta. \tag{4.10}$$

The relationship between the δ-order root moment and the coefficients of the polynomial $P(x)$ is known as the Newton identities and is given by:

$$\begin{cases} S_1 + a_{n-1} = 0, \\ S_2 + a_{n-1}S_1 + 2a_{n-2} = 0, \\ \vdots \\ S_\delta + a_{n-1}S_{\delta-1} + \cdots + \delta a_{n-\delta} = 0 \quad (\delta \leq n), \\ S_\delta + a_{n-1}S_{\delta-1} + \cdots + a_0 S_{\delta-n} = 0 \quad (\delta > n), \end{cases} \tag{4.11}$$

when $\delta \geq 0$; otherwise (when $\delta < 0$):

$$\begin{cases} a_0 S_{-1} + a_1 = 0, \\ a_0 S_{-2} + a_1 S_{-1} + 2a_2 = 0, \\ \vdots \\ a_0 S_\delta + a_1 S_{\delta+1} + \cdots + |\delta| a_\delta = 0 \quad (\delta \geq -n, a_n = 0), \\ a_0 S_\delta + a_1 S_{\delta+1} + \cdots + S_{\delta+n} = 0 \qquad\qquad (\delta < n). \end{cases} \tag{4.12}$$

The same authors [279] also showed that the computational complexity for computing the a priori knowledge for the CLA using the Equations (4.11) and (4.12) is significantly lower than Equation (4.5), resulting in a faster training speed. They also studied the effect of the different parameters in the behaviour of the CLA.

Besides that, the $\sum - \prod$ structure and the logarithmic structure proposed by Perantonis et al. [273] were tested and compared, and although the $\sum - \prod$ revealed to compute the roots with a higher accuracy, it had a slower training speed and a higher probability of not converging, when compared to the logarithmic structure. Nevertheless, these two types of a priori knowledge can be used simultaneously in the CLA [7, 280].

Mourrain and colleagues [281] then investigate the determination of the number of real roots of polynomials using an FNN and concluded that the ANNs were capable of performing such task with high accuracy.

Zhang et al. [282] suggested using a discrimination system in order to compute the number of distinct real or complex roots or, in other words, the roots' multiplicities. They used the ANN structure proposed by Huang and Chi [275].

Das and Seal [283] proposed a completely different approach using a set of divisors of $a_0, a_1, \ldots, a_n$, denoted here as D. In this view, if r is a divisor of D and a divisor of a_0, then r is a potential root of the specific polynomial.

The coefficients of a given polynomial are fed into an ANN by the input neurons and then the first hidden neuron in the first hidden layer separates the coefficients list into two other lists. Each list serves as inputs for the two hidden neurons in the second hidden layer. These two nodes compute the divisor of its coefficients in parallel, and those that are also divisors of a_n are passed to the next two hidden neurons of the next hidden layer, who are responsible for improving the candidate roots using a learning algorithm.

The approaches presented above do not make use of the inductive inference capacity of the ANNs to compute an approximation for each root of a given polynomial. This is the focus of this study.

4.5 Methodology

In this section, the steps taken to build a training and a test data set, and to train the ANNs to produce approximations for the arbitrary roots (both real and complex) of polynomials are described.

The main focus of this study is to compute approximations of the roots α_i $(i = 1, 2, \ldots, n)$ of an n-th degree real univariate polynomial, $P(x) = a_0 + a_1 x + \ldots + a_n x^n$, with both real and complex roots, given its real coefficients.

The block diagram of this approach is presented in Figure 4.3, and it shows that the coefficients of a given polynomial are used as the inputs for the ANNs, that are then processed by it, and used to output an approximation for each root.

It is important to note that, according to the Fundamental Theorem of Algebra [284], an n-th degree polynomial has n real or complex roots. Thus, a priori, the number of output nodes is known.

This study was conducted with two different ANN architectures. The first architecture was used for computing the approximations of the real roots from polynomials with only real roots. On the other hand, the second architecture was used for the case when polynomials can have both real and complex roots. Thus, the limitations in terms of efficiency and accuracy can be studied separately.

The results were then compared in terms of accuracy and in terms of execution time with the D–K method described below.

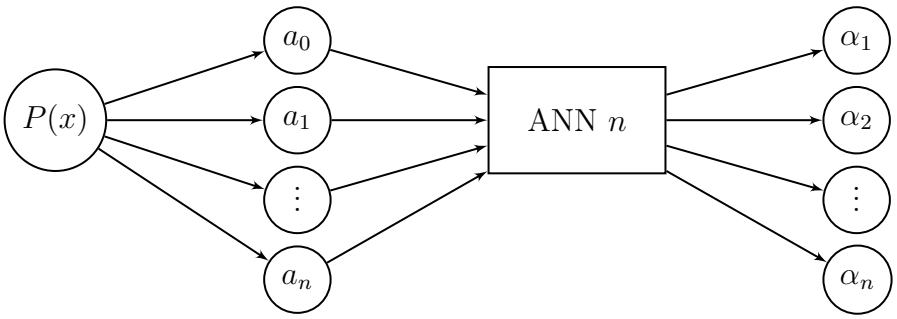


Figure 4.3: Flowchart showing the inputs, processing flow and the outputs of the proposed neural approach for polynomial root-finding.

4.5.1 Durand–Kerner Algorithm

The D–K [223, 224], also known as Weierstrass' or Weierstrass–Dochev's method [271], is a well known iterative method for the simultaneous determination of all roots of a polynomial that does not require the computation of derivatives, but has the drawback of requiring a good initial approximation to each of the roots (which must be obtained using another numerical method) in order to converge and produce approximations to these roots with the required accuracy.

Let $P(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ ($a_n \neq 0$) be an n-th degree univariate polynomial with real (or complex) coefficients. The D–K is given by [285]:

$$\alpha_i^{(t+1)} = \alpha_i^{(t)} - \frac{P(\alpha_i^{(t)})}{a_n \prod\limits_{\substack{j=1 \\ j\neq i}}^{n} (\alpha_i^{(t)} - \alpha_j^{(t)})}, \tag{4.13}$$

where $i = 1, \ldots, n$ and t the current iteration number.

The convergence order of the D–K is quadratic for simple roots but only linear in case of multiple roots [286].

Table 4.1: Example of execution of the D–K algorithm.

Iteration no.	$\boldsymbol{\alpha_1}$	$\boldsymbol{\alpha_2}$	$\boldsymbol{\alpha_3}$
0	1	2	7
1	7.6667	-0.4	6.7333
2	4.9118	1.6420	7.4461
3	4.9340	2.6891	6.3769
4	4.9760	2.9764	6.0476
5	4.9987	2.9999	6.0014
6	5	3	6

Example 4.5.1 (Execution of the D–K method)**.** This example shows the execution of the D–K method for finding the real roots of a polynomial.

The chosen polynomial is given by $P(x) = x^3 - 14x^2 + 63x - 90$, and it has three real roots. $\alpha_1^{(0)}$, $\alpha_2^{(0)}$, $\alpha_3^{(0)}$ were randomly chosen; however, studies were done on how to initialise the roots of a given polynomial to leverage the effectiveness of the D–K method [285].

The first iteration of α_2 is computed as:

$$\begin{aligned}
\alpha_2^{(1)} &= \alpha_2^{(0)} - \frac{P(\alpha_2^{(0)})}{1\prod_{\substack{j=1\\ j\neq i}}^{3}(\alpha_2^{(0)} - \alpha_j^{(0)})} \\
&= 2 - \frac{P(2)}{1\prod_{\substack{j=1\\ j\neq 3}}^{3}(2 - \alpha_j^{(0)})} \\
&= 2 - \frac{-12}{\Big((2-1)(2-7)\Big)} \\
&= -0.4.
\end{aligned}$$

This process is then repeated for the number of roots until a stopping criterion is met. Table 4.1 shows, for this example, the following iterations until the computed roots are fixed. In this case, because the absolute differences between iterations 6 and 7 are all lower than $\epsilon = 10^{-5}$, the algorithm stopped improving; thus, it came to an end.

When 0.1 is added to each initial root value, i.e, when $\alpha_1^{(0)} = 1.1$, $\alpha_2^{(0)} = 2.1$, $\alpha_3^{(0)} = 7.1$, for the same stopping criteria as the example in Table 4.1, the algorithm took 16 iterations to stop. Table 4.2 presents a more detailed comparison between the different initialisation values assigned to $\alpha_1^{(0)}$, $\alpha_2^{(0)}$ and $\alpha_3^{(0)}$, number of iterations and average execution time of 100 trials. In this table, the execution time is given in milliseconds and the variation is computed with respect to the first initialisation scheme.

It can be observed that as the initialisation gets further away from the roots' positions, the number of iterations and the execution time increase approximately exponentially. This demonstrates that the D–K is very sensitive to the initial roots' approximations, and it also provides a motivation for the use of ANNs for root-finding since they do not require initial approximations in order to compute the output.

Table 4.2: Comparison between the different initialisation values assigned to $\alpha_1^{(0)}$, $\alpha_2^{(0)}$ and $\alpha_3^{(0)}$.

Initialisation	**No. of iterations**	**Execution time**	**Variation no. of iterations**	**Variation no. execution time**
$\alpha_1^{(0)} = 1$, $\alpha_2^{(0)} = 2$, $\alpha_3^{(0)} = 7$	7	0.02 ± 0.14		
$\alpha_1^{(0)} = 1.1$, $\alpha_2^{(0)} = 2.1$, $\alpha_3^{(0)} = 7.1$	16	0.05 ± 0.22	128.57%	150.00%
$\alpha_1^{(0)} = 0.001$, $\alpha_2^{(0)} = 0.002$, $\alpha_3^{(0)} = 0.003$	61	0.39 ± 0.55	771,43%	1850.09%

This example also shows the involved repeated deflations required to compute a root, i.e., the computation of the next approximation uses the approximation computed before. This strategy can lead to very inaccurate results due to the problem of accumulating rounding errors.

It is important to note that, although only real roots are being used here, the formulation of the iterative process of the D–K method, utilising Equation (4.13), allows its use to

compute both complex and real roots of a given polynomial. Therefore, if the imaginary part of a complex root is lower than a predefined threshold, the root is considered to be a real root.

4.5.2 Artificial Neural Networks

As described previously, two cases were considered for this study: (i) when polynomials only have real roots, and (ii) when polynomials can have both real and complex roots. The latter case is the most general for real polynomials.

In this study, five neural networks for each case, with three layers (input, hidden, and output layer), were trained using the LMA and PSO, being the inputs the real coefficients of a set of polynomials of degrees 5, 10, 15, 20 and 25. In Figure 4.3, ANN n denotes the neural network that can output the roots of a real n-th degree polynomial.

Data sets

Tables G.1 and G.2 (refer to Appendix G) show the head of the data sets (with 100 000 records) that were used with ANN 5 to compute approximations for the real roots.

To generate these data sets, two algorithms were used to: (i) generate real roots for any polynomial degree in the closed interval of -1 to 1, and (ii) given a set of real roots, compute the respective coefficients.

On the other hand, Tables G.3 and G.4 in Appendix G show the head of the data sets (with 100 000 records) that were used with ANN 5 to compute approximations for both real and complex roots.

Differently from the data set of real roots, where the number of columns is equal to the degree of the polynomials (i.e., n), the output data sets for the ANNs that compute the approximations for both real and complex roots have $n \times 2$ columns. In these data sets, the odd columns represent the real part of the complex number, and the even columns the imaginary part, such as:

$$\alpha_i = \{\mathrm{Re}(\alpha_i), \mathrm{Im}(\alpha_i)\}\,,\ i = 1, 2, \ldots, n. \tag{4.14}$$

Contrary to the strategy employed to generate the databases for the real roots, the coefficients of the polynomials were generated first (in the closed interval of 0 to 1) and from these, the exact roots were calculated (which can be real or complex). Thus, the ANN does not know a priori which roots are complex or real.

For the two cases, it is important to note that, although coefficients and roots are shown with only four decimal places, double-precision values were used to generate the data sets.

From these data sets, 70 % of the samples were used to train the ANNs. The remaining 30 % was used to test the generalisation capabilities of the ANNs, by computing a performance measure on samples that were not used to train the ANNs.

Besides that, this test-train data set division is also important to avoid overtraining, i.e., to avoid the ANN to learn specific information about the data (such as noise) and, thus, losing its capacity to generalise the results.

Architecture

For this study, shallow FNNs were used and, after several tests according to the rule-of-thumb methods presented in Section 4.2.2, it was found that there is little variance resulting from a change in the number of hidden neurons of the neural network. Given this, in this experimental study, ten neurons were used in the hidden layer for all the final tests.

Besides that, the hyperbolic tangent sigmoid (tansig) activation function [287] was applied only in the hidden layer.

The tansig activation function is given by:

$$\phi(x) = \frac{2}{1 + e^{-2x}} - 1, \tag{4.15}$$

and it was chosen to ensure that values stay within a relatively small range and to allow the network to learn non-linear relationships between coefficients and roots.

The use of this anti-symmetric (S-shaped) function for the input to output transformation allows the output of each neuron to assume both positive and negative values in the interval $[-1, 1]$.

It is important to note that a min-max normalisation method [288] was used to scale data within the range of $[-1, 1]$, in order to improve the convergence properties of the training algorithm [227].

In the output layer, no activation function was embedded, since the final output (rescaled) is still a linear function of the original data, allowing the ANNs to output values that are not circumscribed to the range of values of the activation function.

The architecture of the ANN is represented in Figure 4.4. However, it is important to emphasise that for the case of the ANNs that compute the approximations for both complex and real roots, two output units were used for each root: one for the real part and the other for the imaginary part.

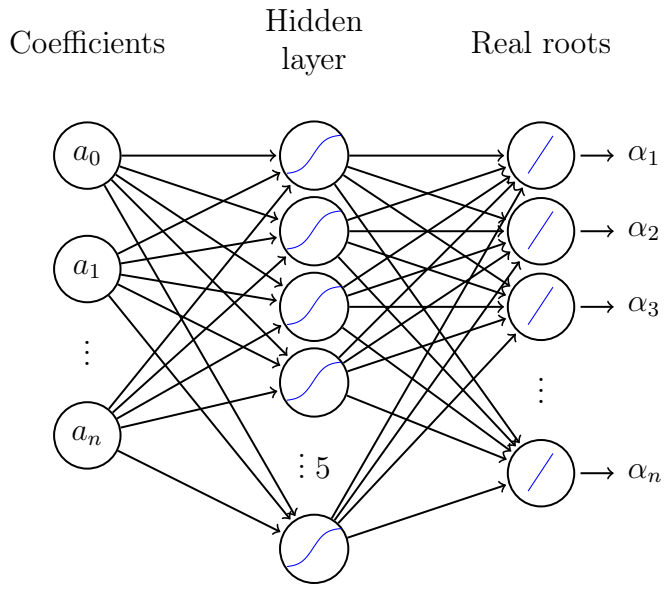


Figure 4.4: Architecture of the ANN n for root-finding.

Training Algorithm

The well known LMA [289, 290] was used for ANN training, due to its efficacy and convergence speed, being one of the fastest methods for training FNNs, especially medium-sized ones. The application of the LMA to neural network training is described, e.g., in [291] and [292], since its details go beyond the scope of this work.

Furthermore, the LMA is a hybrid algorithm that combines the efficacy of the Gauss–Newton method with the robustness of the gradient descent method, making one of these methods more or less dominant at each minimisation step through a non-negative algorithmic parameter (λ) that is adjusted at each iteration [293].

The weights of the synaptic connections are then updated according to the difference between the predicted and the observed value, backpropagating that error, such as:

$$w^{t+1} = w^t - \left(2(J^t)^T \cdot J^t + \lambda^t I\right)^{-1} \cdot \left(2(J^t)^T \cdot e^t\right), \tag{4.16}$$

where I is the identity matrix and e is the squared error between the target and the estimated values. Finally, J is a Jacobian matrix given by $J_{i,j} = \dfrac{\partial e_i}{\partial w_{j.}}$.

The LMA was used following a batch learning strategy, meaning that the network's weights and biases are updated after all the samples in the training set are presented to the network.

4.6 Results and Discussion

In this section, the results obtained with this approach are presented, along with comparisons with the numerical approximations provided by the D–K, in terms of accuracy and execution time, when the polynomials have only real roots and for the case when the polynomials have both real and complex roots.

4.6.1 Polynomials with Only Real Roots

It is important to note that the D–K method and the ANNs' training used the same stopping criteria, i.e., the maximum number of iterations is 5 000 and $\epsilon = 10^{-12}$ (this means that the value of the polynomial, when evaluated on the position of the root found, is less or equal to 10^{-12}). Besides that, for the D–K method, the Cauchy's upper bound [294] was used in order to compute an initial approximation for the roots of a given polynomial, such as:

$$ub = 1 + \max\left\{\left|\frac{a_0}{a_n}\right|, \cdots, \left|\frac{a_{n-2}}{a_n}\right|, \left|\frac{a_{n-1}}{a_n}\right|\right\}. \tag{4.17}$$

Thus, the initial approximation for each root is chosen randomly from a uniform distribution between $-ub$ and ub, inclusive.

Table 4.3 shows the MSE for each of the five polynomial degrees with only real roots considered. The MSE is computed as follows, where D_i denotes the actual i-th root value ($i = 1, \ldots, n$) in the test data set, and N_i the corresponding approximation obtained with the D–K method or the proposed ANN approach:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(D_i - N_i)^2. \tag{4.18}$$

As already mentioned, in order to compute the MSE, 30 % of the original data set was reserved, and it contains samples randomly chosen that were not employed to train the networks.

It should be pointed out that all the ANN were trained 20 times and based on the lowest MSE on the test data set, only one was chosen to be compared to the numerical approximations provided by the D–K method.

Table 4.3: Comparison between ANN and D–K methods in terms of the MSE for polynomials with only real roots.

Degree	**MSE of ANN**	**MSE of D–K**
5	0.4428	9.4418e−16
10	0.6032	2.8480e−15
15	0.6474	2.1328e−11
20	0.6213	1.8378e−09
25	0.4731	7.2990e−09

Table 4.4: Comparison between ANN and D–K methods in terms of average execution time (in seconds) for polynomials with only real roots.

Degree	**ANN**	**D–K**
5	0.007	0.0402
10	0.009	0.4386
15	0.009	0.5526
20	0.013	2.4546
25	0.016	3.1650

Table 4.3 shows that this approach does not yet surpass the accuracy of the D–K method, that always finds the roots with an accuracy greater than 10e−9. It is important to note that, as expected, as the degree of the polynomials increases, the D–K method shows more difficulty in finding the roots, thus justifying the increase in the MSE values. This situation is not, however, clearly observed in the ANN approach, since polynomials of 5^{th} and 25^{th} had similar MSE values.

Since all the MSE values presented in Table 4.3 are significantly less than one, it can be inferred that the networks have a good capacity to generalise the space of results. Thus, with some confidence, it is possible to conclude that the networks can solve any real univariate polynomial of the respective degree with only real roots.

The results on the execution time for both methods, showed in Table 4.4 and in the following tables, were obtained using a personal computer equipped with a 7^{th} generation Intel Core i7 processor and 16 GB of RAM.

Table 4.4 shows that, when the degree of the polynomial increases, the execution time with ANN remains almost constant. The opposite happens with the D–K, which with an increase in the degree of the polynomial implies an increase in the execution time.

Comparing the execution times of both methods, it can be observed that the execution time required to compute the approximations to the roots using ANN is significantly lower than that of the D–K. This result was already expected because computing polynomial roots using the D–K method, unlike ANN, is a pure iterative procedure.

Thus, the results obtained with the ANN-based approach, although limited, are very encouraging and demonstrate the viability and potentiality of the approach for approximating real roots of polynomials.

4.6.2 Polynomials with Both Real and Complex Roots

In what follows, the results of the use of the proposed approach for computing the real and complex roots of a given polynomial will be presented.

It is noteworthy that, since the proposed approach computes the real and imaginary part separately, in order to perform a comparison with the D–K, the results of the D–K method were also split into real and imaginary parts.

Besides that, the Cauchy's upper bound [294] is still valid for the case of complex roots. Nevertheless, it represents the upper bound (ub) of the modulus of the complex roots, forming a circle in the complex plane centered at the origin with a radius equal to the upper bound where all the real and complex roots are located.

In this view, the initial root values $\alpha_i^0 \, (i = 1, 2, \ldots, n)$ of an n-degree real univariate polynomial will be set randomly inside the circle with a radius equal to ub, such as:

$$\alpha_i^0 = \cos(\theta) \times ub + \sin(\theta) j \times ub, \tag{4.19}$$

where $j = \sqrt{-1}$ and θ given by:

$$\theta = r \times \pi \times 2, \tag{4.20}$$

being r a random number between 0 and 1 (exclusive).

Table 4.5 shows the MSE of the ANN-based approach for polynomials with both real and complex roots. In this table, it is possible to observe that the ANN-based approach failed to converge, in all the trials, to an acceptable accuracy, and that the MSE is not related with the degree of the polynomial, since polynomials of 10^{th} degree had a worse MSE when compared to the results from polynomials of 15^{th} degree.

On the other hand, the D–K method is still an available approach to compute accurately both the real and imaginary parts of the roots of a given polynomial. However, interestingly, as the degree of the polynomials increases, the D–K method becomes more accurate. Also, when compared with the results of Table 4.3, it can be perceived that, for lower degree polynomials, the D–K method does not calculate complex roots as accurately as real roots. In any case, the D–K is still an accurate alternative to compute the approximations for both real and complex roots of a given polynomial.

In terms of execution time, as one would anticipate from the previous result where only real roots where considered, the ANN-based approach performed better to compute the real and complex roots when compared to the D–K method.

It is important to note that the ANN-based approach for computing the real and the complex roots is more time consuming than the ANN-based for computing only the real roots, since the number of weights doubled.

Table 4.5: Comparison between ANN and D–K methods in terms of the MSE for polynomials with both real and complex roots.

Degree	MSE of ANN	MSE of D–K
5	36.5387	4.9290e−09
10	386.7120	3.4800e−09
15	2.7157	2.6766e−09
20	146.2455	1.8608e−09
25	4.6057	9.7454e−10

Table 4.6: Comparison between ANN and D–K methods in terms of the average execution time (in seconds) for polynomials with both real and complex roots.

Degree	ANN	D–K
5	0.018	0.0842
10	0.015	0.4098
15	0.027	1.2728
20	0.025	3.0516
25	0.028	5.4552

Considering that accurate results were not achieved when using the ANN-based approach for computing the real and the complex roots using the real and the imaginary parts of the roots in the test data sets, the roots were transformed into polar coordinates, in order to assess its impact on the accuracy of the approach, such as:

$$\alpha_i' = \left\{ \sqrt{x^2 + y^2}, \tan^{-1}\left(\frac{y}{x}\right) \right\}, \tag{4.21}$$

where x and y correspond to the real and the imaginary of α_i, respectively.

However, and as can be seen from Table 4.7, the MSE obtained is even higher in most of the polynomials degrees when compared to the previous approach.

Interestingly, in this approach, polynomials with a higher-order degree had a lower MSE. Nevertheless, for all the polynomial degrees, the LMA was not successful in updating the ANNs' weights in order to reduce the MSE.

Table 4.7: Comparison of the MSE between complex roots in polar coordinates, and in real and imaginary parts for ANN-based approach.

Degree	MSE (polar coordinates)	MSE (real and imaginary parts)
5	678.9594	36.5387
10	249.7066	386.7120
15	210.9323	2.7157
20	146.6783	146.2455
25	114.6380	4.6057

Concluding, the proposed approach showed to have limitations in terms of accuracy when compared to the D–K method, especially when computing the roots of polynomials with both real and complex roots. Nevertheless, this approach surpassed the efficiency of the D–K method, since the ANN-based approach required a lower execution time in all the executed tests when compared with the D–K method.

Motivated by these results, in the next section, the ANN-based approach is enhanced using the PSO algorithm, introduced the previous chapter.

4.7 Enhancing the Artificial Neural Network for Root-Finding

In this section, the ANN approach for root-finding is enhanced by using the PSO algorithm, as an alternative to the LMA.

The PSO algorithm runs in every iteration to minimise the cost function defined by the MSE between the output of the ANN and the expected value.

The structure, i.e., the number of neuron units in the hidden layer and the activation functions of the ANN, as well as the data sets, were kept the same as in the previous sections in order to enable a fair comparison between the two approaches: ANN trained with the LMA and ANN trained with the PSO algorithm. (Although it has not been investigated in this work, it is important to note that the PSO algorithm could also be used to obtain the optimal ANN architecture that most minimises the difference between the observed and the predicted values.

For the purpose of this work, the PSO will minimise the cost function, i.e., the MSE, in a search space with $10+21n$ (for polynomials with only real roots) or $10+32n$ (for polynomials with both real and complex roots) dimensions, each one corresponding to each synaptic connection between the neuron units in an ANN with one hidden layer. (For example, for the ANN 25 to be used on polynomials with real roots, 260 weights are needed to connect the input layer to the hidden layer [26 coefficients neurons $\times$ 10 hidden neurons]. On the other hand, 275 synaptic connections are required in order to connect the hidden neurons and the one bias neuron to the nodes in the output layer [(10+1) hidden neurons with bias $\times$ 25 neurons for each root]. This defines a search space with 535 dimensions to optimise using PSO.)

The LMA is known for being a successful algorithm even if it starts in a zone far from the optimal one. However, the calculation of the Jacobin matrix may cause some performance issues in the algorithm, especially in deep ANNs, in ANNs with many hidden neuron units, or in big data sets [295]. PSO, on the other hand, also has a good ability to explore the search space; however, the calculations needed for the algorithm's execution are far more simple and expected to be less computationally demanding when compared to LMA.

24 particles, organised in star communication structure, were used in the PSO algorithm's. Besides that, the constriction term, proposed by Eberhart and Shi [28], was used in the particles' velocity update equation (Equation (3.7)), with $\varphi = \varphi_1 + \varphi_1$ and $\varphi_1 = \varphi_2 = 2.05$. (A priori tests using the same optimal parameters as the ones found in Chapter 3 did not reveal such accurate results [refer to Appendix H for the generalisations results] when compared to this configuration.)

Each particle corresponds to an ANN with distinct weights configuration from other particles in the swarm. The PSO then proceeds normally until a stopping criterion is met. In this case, the algorithm stops when it reaches the maximum number of iterations (5 000 iterations) or when the MSE derived from the training data set is lower than a predefined ϵ (in this case, $\epsilon = 10^{-12}$).

For each ANN, ten tests were executed and, for each experiment, the MSE was computed as Equation (4.18). Nevertheless, only the ANN with the lowest MSE was kept for comparison. The MSE results are presented in Table 4.8 for the case of real roots.

Table 4.8: Comparison of the capacity of the networks to generalise the outputs between the D–K method, ANN trained with LMA and with the PSO algorithm for the case with real roots.

Degree	**MSE of D–K**	**MSE of LMA**	**MSE of PSO**
5	9.4418e−16	0.4428	0.0665
10	2.8480e−15	0.6032	0.0763
15	2.1328e−11	0.6474	0.0799
20	1.8378e−09	0.6213	0.0797
25	7.2990e−09	0.4731	0.0816

As can be seen, the MSE of the ANN training using the PSO algorithm is almost constant among the different degrees of the polynomials considered and is always lower when compared to the MSE of the ANN trained with the LMA.

An improvement in terms of accuracy was also detected for the case of the complex roots, as can be seen in Table 4.9. Besides that, is possible to observe that as the degree of the polynomial increase, the accuracy of the ANN trained using PSO is little affected, except for the case of polynomials of degree 20 and 25. Nevertheless, polynomials of degree 20 and 25 showed a similar MSE value.

Table 4.9: Comparison of the capacity of the networks to generalise the outputs between the D–K method, ANN trained with LMA and with the PSO algorithm for the case with complex roots.

Degree	**MSE of D–K**	**MSE of LMA**	**MSE of PSO**
5	4.9290e−09	36.5387	0.0036
10	3.4800e−09	386.7120	0.0069
15	2.6766e−09	2.7157	0.0083
20	1.8608e−09	146.2455	0.0121
25	9.7454e−10	4.6057	0.0111

Similarly, the accuracy of the results obtained with the ANN-based approach for computing both real and complex roots in polar coordinates was also improved by training the ANN with the PSO algorithm, when compared to the LMA. The results of this approach are presented in Table 4.10, and show that the use of the PSO algorithm produced a stable MSE, especially in higher-order degree polynomials.

It is noteworthy that converting the complex roots to polar coordinates resulted, in all the polynomials degrees considered, in a loss of generalisation capabilities. Thus, the real and imaginary parts should be used, along with the ANN trained using the PSO algorithm, in order to compute both real and complex roots of a given polynomial.

Table 4.10: Comparison of the capacity of the networks to generalise the outputs between ANN trained with LMA and with the PSO algorithm for the case with complex roots in polar coordinates.

Degree	MSE of LMA	MSE of PSO
5	678.9594	0.0159
10	249.7066	0.0253
15	210.9323	0.0343
20	146.6783	0.0344
25	114.6380	0.0397

This comparison allowed to conclude that the ANN trained with the PSO algorithm seems to be a more effective alternative when compared to the training using the LMA since it was able to generalise better the results (in this case, the roots of a given polynomial). As noted by Mendes et al. [148], one possible justification for the fact of the PSO had surpassed the LMA generalisation capabilities may be related to the number of local minima present in the search space, since PSO revealed to be the best training algorithm when a high number of local minima exist.

Besides that, even for higher-order degree polynomials, the ANN trained with the PSO algorithm revealed a modest MSE. However, these results are, again, limited, especially when compared to the D–K method, but it was already a step towards the improvement of the initially proposed approach.

4.8 Initialisation Scheme for the Durand–Kerner Method

This section is intended to focus on the use of the ANN-based approach, trained with the PSO algorithm, in order to compute the initial guesses for the roots of a given polynomial, i.e., the initial approximations to the roots to be used with the D–K method. This section appears since the presented results of the ANN-based approach are not superior to the ones computed by the D–K method. So, instead of competing, the ANN-based approach will hopefully help to leverage the effectiveness and efficiency of the D–K method.

As already mentioned, the D–K method requires good initial approximations for all the roots in order to converge. Thus, the objective of this section is to compare the use of the D–K method when it is initialised using the Cauchy's upper bound or by using the ANNs introduced in the previous section.

Results will be compared in terms of the MSE, execution time (in seconds), the number of iterations and effectiveness. In turn, the effectiveness is computed by dividing the number of times that the D–K method found the roots of a given polynomial by the total number of tests. It is also important to note that all the presented results correspond to the average of the different tests that were run.

Like in the previous sections, and for the sake of comparison, only the test data set containing the coefficients and the polynomial roots will be used. Following a similar approach, it will be considered the case when only real roots exist, and the case when both real and complex roots exist.

Table 4.11 presents the comparison between the Cauchy's upper bound and the ANN-based approach for the case when the polynomials have only real roots.

In that table, it can be analysed that using the ANN, in order to compute the initial approximation for each root for the D–K method, represent a significant advantage to the method, when compared to the Cauchy's upper bound in all parameters of the tests.

Comparing the MSE of the Cauchy's upper bound and the ANN, for all polynomial degrees considered, the ANN allowed the D–K method to find more accurately, on average, the roots of the polynomials. On the one hand, using the ANN-based approach, the algorithm needed more iterations or a longer execution time in polynomials of degree 5 and 15; however, when considering the other polynomial degrees, this situation no longer exists, since the use of the ANN in the D–K method required a lower number of iterations and a lower execution time, when compared to the use of the Cauchy's upper bound in the D–K method.

Finally, in terms of effectiveness, the ANN-based approach was able to leverage the number of times that the D–K method was successful when compared to the Cauchy's upper bound initialisation, by the fact that the use of the Cauchy's upper bound in the D–K was not able to find all roots of polynomials with degree exceeding 5 in the test data set, and the ANN initialisation only presented difficulties from polynomials of 20^{th} and 25^{th} degree.

Table 4.11: Comparison between the Cauchy's upper bound and ANN-based approach to provide the initial approximation for each root to the D–K method for the case with real roots.

	Cauchy's upper bound				ANN			
Degree	**MSE**	**Exec. time**	**No. iterations**	**Effectiveness**	**MSE**	**Exec. time**	**No. iterations**	**Effectiveness**
5	9.4418e−16	0.0402	11.7425	100.00 %	2.4284e−18	0.0638	12.0500	100.00 %
10	2.8480e−15	0.4386	26.8239	99.99 %	2.5737e−17	0.3050	19.6400	100.00 %
15	2.1328e−11	0.5526	50.0772	99.89 %	6.3421e−14	1.0206	29.4500	100.00 %
20	1.8378e−09	2.4546	84.9369	98.65 %	6.3063e−11	1.3748	41.3939	99.99 %
25	7.2990e−09	3.1650	138.1522	92.00 %	1.6325e−09	2.0469	41.8125	96.00 %

Following a similar approach, the case when polynomials have both real and complex roots was also tested, and results are presented in Table 4.12.

The comparison between the Cauchy's upper bound and ANN-based approach, in this case, leads to the conclusion that the ANN-based approach can be used only to improve the accuracy of the D–K method, surpassing the Cauchy's upper bound in all tests; however, if one is interested in enhancing the performance of the algorithm, then the use of the D–K using the Cauchy's upper bound initialisation technique should be considered, taking into account that in all degrees of the polynomials, this strategy required a shorter execution time and a smaller number of iterations.

Besides, in terms of effectiveness, the Cauchy's upper bound strategy was superior to the ANN-based approach, especially in higher-order degree polynomials. Thus, concluding that the ANN-based approach can be used only to improve the accuracy of the D–K method.

Finally, this section shows that the ANN-based approach trained with PSO can be used as an initialisation scheme for the D–K method, since it found, on average, more accurate roots,

Table 4.12: Comparison between the Cauchy's upper bound and ANN-based approach to provide the initial approximation for each root to the D–K method for the case with complex roots.

	Cauchy's upper bound				ANN			
Degree	**MSE**	**Exec. time**	**No. iterations**	**Effectiveness**	**MSE**	**Exec. time**	**No. iterations**	**Effectiveness**
5	4.9290e−09	0.0842	67.5155	99.82 %	4.5822e−09	0.1008	77.7170	99.84 %
10	3.4800e−09	0.4098	83.1480	99.77 %	2.7444e−09	0.6589	105.5526	99.67 %
15	2.6766e−09	1.2728	103.4006	99.72 %	1.4705e−09	2.3469	151.4768	99.43 %
20	1.8608e−09	3.0516	127.3676	99.68 %	9.2850e−10	4.3203	184.3324	99.33 %
25	9.7454e−10	5.4552	143.9964	99.58 %	4.8588e−10	8.6969	226.1480	99.01 %

when compared to the Cauchy's upper bound initialisation technique. It is also important to note that, for the case when polynomials have only real roots, the ANN-based approach should always be chosen by the user/researcher; however, when polynomials have both real and complex roots, one should choose between accuracy, giving preference to the ANN-based approach, or efficiency, giving preference to the Cauchy's upper bound.

4.9 Conclusion

This chapter introduces an approach for computing approximations for both real and complex roots of a given polynomial, based on the inductive inference capabilities of the ANNs. Results were then compared in terms of effectiveness and efficacy to the D–K method.

The author started by training the ANNs using the LMA algorithm and as inputs the coefficients of the polynomials and the LMA to adjust the weights based on the error between the true values and the values produced by the ANN, i.e., the roots of the polynomials.

Although for the cases when polynomials have only real roots the proposed approach revealed a modest MSE, for the case when both real and complex roots exit, the LMA was unable to converge to an acceptable solution. The approach shows, however, advantages in terms of performance.

These results were then significantly improved by using PSO acting like a training algorithm, as an alternative to the LMA. Nevertheless, results still showed that ANNs were not able to surpass the accuracy of the D–K method.

As an example of the utility of this approach, the author tested the use of the ANNs as an initialisation scheme for the D–K method and reached to the conclusion that the ANN-based approach is a viable alternative when compared to the initialisation scheme provided by the Cauchy's upper bound, especially in terms of accuracy and mostly for real roots.

Chapter 5

Conclusions and Future Work

— The science of today is the technology of tomorrow.

Edward Teller (1908–2003)

5.1 Conclusion

This dissertation started with a systematic review of the PSO algorithm, taking into consideration the initial developments up to the recent stages of the algorithm. This gives to the reader a broad view of PSO and the most relevant approaches and applications. The interested reader has then the opportunity to explore the different PSO modifications and variants and apply them to real-world problems.

After the state-of-the-art chapter, the document focused on the use of the PSO algorithm for solving complicated non-linear equations and systems of non-linear equations, which is considered one of the most difficult numerical problems to solve. This served as motivation for the study on the effect of the parameters on the execution of the PSO for root-finding, and later the proposal of a PSO variant able to find the roots of a given non-linear function simultaneously.

The last chapter presented an ANN-based approach for approximating both real and complex roots of a given polynomial. Polynomials of degree 5, 10, 15, 20 and 25 were tested, and the algorithm revealed a modest MSE, but still suggesting its viability and potentiality.

Thus, the use of PSO for root-finding was investigated and two distinct nature-inspired algorithms for the same task were proposed and tested using some difficult problems with applications in Science and Engineering. These were the main contributions of this dissertation.

5.1.1 Parameter Selection for Root-Finding with Particle Swarm Optimisation

The original PSO was adapted for root-finding, and different combinations of parameters were used to test the effectiveness and efficiency of the PSO algorithm. By conducting an ANOVA, swarms with 24 particles organised in a all-connected-to-all communication

structure, with the constriction term in the velocity update equation revealed, on average, to be the best parameter configuration in terms of the accuracy of the root approximations found, number of iterations and execution time.

5.1.2 Multi Root-Finding Particle Swarm Optimisation

The MRF-PSO algorithm is a bio-inspired stochastic algorithm that uses the search strategy implemented in the PSO algorithm in order to compute approximations for the roots of a given non-linear equation or system of non-linear equations.

Thus, multiple swarms are placed in the search space and, following a parallel architecture, they are able to explore the search space at the same time, using different cores of a multi-core processor, and report the results to the master.

The master, in turn, stores all the roots that were found during the search process, and is responsible for randomly positioning the particles of the swarms uniformly in the search areas where a root was not found yet.

For this algorithm to find the roots of a function simultaneously, two parameters were introduced into the MRF-PSO: the absolute tolerance (e) and the time to live parameter. On the one hand, the absolute tolerance is responsible for detecting equal roots, and it is also used to prevent particles from searching in the same search space areas where a root was found before. On the other hand, the time to live parameter defines how many times should the particles of a swarm be repositioned in the search space after not founding a root or after finding a root that has already been found.

The MRF-PSO algorithm was tested with different, commonly used non-linear functions. The results of the executions were compared with other results available on the literature and, although the MRF-PSO can be considered an effective algorithm on finding the multiple roots of a given function, it showed some issues related with the time required to compute an approximation for all roots.

5.1.3 Neural Network-Based Approach for Approximating Roots of Polynomials

The implementation of an ANN for finding the arbitrary roots (real and complex) of a given polynomial simultaneously is also considered one of the main contributions of this dissertation.

This approach uses the inductive inference capabilities of the ANN in order to approximate the roots of polynomials of degree 5, 10, 15, 20 or 25, giving its coefficients. The ANNs were trained using the LMA algorithm and PSO for finding the optimal networks weights that minimise the MSE. Hence, it resulted that training the ANNs with PSO improved the generalisation capabilities of the networks, since the LMA was not capable of converging to an acceptable solution.

Although this approach was not able to surpass the accuracy of the D–K method, it presented advantages in terms of performance.

5.1.4 Initialisation Scheme for the Durand–Kerner Method

Since the D–K method requires good initial approximations for all the roots in order to converge, the results provided by the ANN approach, suggested for approximating the roots of polynomials, were used as initial guesses for the roots. When compared with the initial approximations obtained from the Cauchy's upper bound, the ANN approach revealed to better leverage the accuracy of the approximations calculated by the D–K method.

5.2 Future Work

Concerning Chapter 3, many different PSO modifications and variants have been left for future research due to time restrictions. The use of dynamic neighbours with the concept of stereotyping [24, 34, 35], Fully Informed Particle Swarm (FIPS) [73], and the Adaptive Particle Swarm Optimisation (APSO) [46] are some ideas that the author would have liked to test.

These ideas would, for instance, improve the convergence behaviour of the MRF-PSO algorithm, enhancing its effectiveness and efficiency. The author is also considering rewriting the MRF-PSO algorithm in another computer programming language, such as the Julia Language [296], in order to reduce the total execution time.

The way the algorithm is designed does not enable it to solve non-linear equations or systems of non-linear equations subject to a set of constraints. A penalty function strategy, like the one introduced by Parsopoulos and Vrahatis [47], can be a good starting point.

In terms of the ANN-based approach, presented in Chapter 4 for approximating the roots of polynomials, another training algorithms could be investigated, such as the Adam stochastic method [263], stochastic gradient descent [236] or other quasi-Newton methods.

It is of interest to test the mini-batch approach, meaning that the training data set is split into small data sets, denominated batches. The mini-batch strategy is often suggested [297] in order to improve the robustness convergence, avoiding the training algorithm to be trapped into a local optima.

Finally, the early stopping regularisation should also be considered in future works as a mean to improve the generalisation capabilities of the ANN, i.e., as a mean to avoid model overfitting.

Bibliography

[1] J. Pintér, "Operations research. From MathWorld—A Wolfram Web Resource." [Online] `http://mathworld.wolfram.com/OperationsResearch.html`, accessed on 14 February 2019.

[2] K. Parsopoulos and M. Vrahatis, *Particle Swarm Optimization and Intelligence: Advances and Applications*. Advances in Computational Intelligence and Robotics, Hershey, PA, USA: Information Science Reference, 2010.

[3] M. R. Bonyadi and Z. Michalewicz, "Particle swarm optimization for single objective continuous space problems: A review," *Evol. Comput.*, vol. 25, pp. 1–54, Mar.–June 2017.

[4] S. Das and P. N. Suganthan, "Differential evolution: A survey of the state-of-the-art," *IEEE T. Evolut. Comput.*, vol. 15, pp. 4–31, Feb. 2011.

[5] M. Juneja and S. K. Nagar, "Particle swarm optimization algorithm and its parameters: A review," in *Proc. of the International Conference on Control, Computing, Communication and Materials (ICCCCM)*, (Allahbad, India), pp. 1–5, Oct. 2016.

[6] Y. Zhang and Z. Zeng, "A new method for simultaneous extraction of all roots of algebraic polynomial," in *Proc. of the International Conference on Computational Intelligence and Security (CIS)*, vol. 1, (Beijing, China), pp. 197–200, Dec. 2009.

[7] D.-S. Huang, "A constructive approach for finding arbitrary roots of polynomials by neural networks," *IEEE T. Neural Networ.*, vol. 15, pp. 477–491, Mar. 2004.

[8] D. Freitas, L. G. Lopes, and F. Morgado-Dias, "A neural network based approach for approximating real roots of polynomials," in *Proc. of the International Conference on Mathematical Applications (ICMA)*, (Funchal, Portugal), pp. 44–47, July 2018.

[9] D. Freitas, L. G. Lopes, and F. Morgado-Dias, "Particle swarm optimisation: A historical review up to the current developments," *Entropy*, vol. 22, p. 362, Mar. 2020.

[10] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *Proc. of the 6th International Symposium on Micro Machine and Human Science (MHS)*, (Nagoya, Japan), pp. 39–43, Oct. 1995.

[11] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proc. of the International Conference on Neural Networks (ICNN)*, vol. 4, (Perth, Australia), pp. 1942–1948, Nov. 1995.

[12] M. Løvbjerg, T. K. Rasmussen, and T. Krink, "Hybrid particle swarm optimiser with breeding and subpopulations," in *Proc. of the 3rd Annual Conference on Genetic and Evolutionary Computation (GECCO)*, vol. 24, (San Francisco, CA, USA), pp. 469–476, July 2001.

[13] J. Kennedy, "Small worlds and mega-minds: Effects of neighborhood topology on particle swarm performance," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, vol. 3, (Washington, WA, USA), pp. 1931–1938, July 1999.

[14] M. Clerc, "The swarm and the queen: Towards a deterministic and adaptive particle swarm optimization," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, vol. 3, (Washington, WA, USA), pp. 1951–1957, July 1999.

[15] J. Kennedy and R. C. Eberhart, "A discrete binary version of the particle swarm algorithm," in *Proc. of the IEEE International Conference on Systems, Man and Cybernetics (SMC)*, vol. 5, (Orlando, FL, USA), pp. 4104–4108, Oct. 1997.

[16] M. Rosendo and A. Pozo, "A hybrid particle swarm optimization algorithm for combinatorial optimization problems," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, (Barcelona, Spain), pp. 1–8, July 2010.

[17] M. Rosendo and A. Pozo, "Applying a discrete particle swarm optimization algorithm to combinatorial problems," in *Proc. of the 11th Brazilian Symposium on Neural Networks (SBRN)*, (São Paulo, Brazil), pp. 235–240, Oct. 2010.

[18] L. Junliang, H. Wei, S. Huan, L. Yaxin, and L. Jing, "Particle swarm algorithm based task scheduling for many-core systems," in *Proc. of the 12nd IEEE Conference on Industrial Electronics and Applications (ICIEA)*, (Siem Reap, Cambodia), pp. 1860–1864, June 2017.

[19] E. Ozcan and C. K. Mohan, "Analysis of a simple particle swarm optimization system," in *Proc. of the Intelligent Engineering Systems Through Artificial Neural Networks (ANNIE)*, vol. 8, (Saint Louis, MO, USA), pp. 253–258, Nov. 1998.

[20] E. Ozcan and C. K. Mohan, "Particle swarm optimization: Surfing the waves," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, vol. 3, (Washington, WA, USA), pp. 1939–1944, July 1999.

[21] C. Sun, J. Zeng, J. Pan, S. Xue, and Y. Jin, "A new fitness estimation strategy for particle swarm optimization," *Inform. Sciences*, vol. 221, pp. 355–370, Feb. 2013.

[22] M. Clerc and J. Kennedy, "The particle swarm: Explosion, stability, and convergence in a multidimensional complex space," *IEEE T. Evolut. Comput.*, vol. 6, pp. 58–73, Feb. 2002.

[23] Y. Shi and R. C. Eberhart, "A modified particle swarm optimizer," in *Proc. of the IEEE World Congress on Computational Intelligence (WCCI)*, (Anchorage, AK, USA), pp. 69–73, May 1998.

[24] F. van den Bergh, *An Analysis of Particle Swarm Optimizers*. PhD thesis, University of Pretoria, Pretoria, South Africa, 2002.

[25] S. Sengupta, S. Basak, and R. A. Peters, "Particle swarm optimization: A survey of historical and recent developments with hybridization perspectives," *Mach. Learn. Knowl. Extr.*, vol. 1, pp. 157–191, Oct. 2018.

[26] R. C. Eberhart and Y. Shi, "Tracking and optimizing dynamic systems with particle swarms," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, vol. 1, (Seoul, South Korea), pp. 94–100, May 2001.

[27] Y. Shi and R. C. Eberhart, "Fuzzy adaptive particle swarm optimization," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, vol. 1, (Seoul, South Korea), pp. 101–106, May 2001.

[28] R. C. Eberhart and Y. Shi, "Comparing inertia weights and constriction factors in particle swarm optimization," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, vol. 1, (La Jolla, CA, USA), pp. 84–88, July 2000.

[29] J. Xin, G. Chen, and Y. Hai, "A particle swarm optimizer with multi-stage linearly-decreasing inertia weight," in *Proc. of the 2nd International Joint Conference on Computational Sciences and Optimization (CSO)*, vol. 1, (Sanya, China), pp. 505–508, Apr. 2009.

[30] A. Chatterjee and P. Siarry, "Nonlinear inertia weight variation for dynamic adaptation in particle swarm optimization," *Comput. Oper. Res.*, vol. 33, pp. 859–871, Mar. 2006.

[31] R. C. Eberhart and Y. Shi, "Particle swarm optimization: Developments, applications and resources," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, vol. 1, (Seoul, South Korea), pp. 81–86, May 2001.

[32] R. Kar, D. Mandal, S. Bardhan, and S. P. Ghoshal, "Optimization of linear phase FIR band pass filter using particle swarm optimization with constriction factor and inertia weight approach," in *Proc. of the IEEE Symposium on Industrial Electronics and Applications (ICIEA)*, (Langkawi, Malaysia), pp. 326–331, Sept. 2011.

[33] J. Kennedy and R. Mendes, "Population structure and particle swarm performance," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, vol. 2, (Honolulu, HI, USA), pp. 1671–1676, May 2002.

[34] P. N. Suganthan, "Particle swarm optimiser with neighbourhood operator," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, vol. 3, (Washington, WA, USA), pp. 1958–1962, July 1999.

[35] J. Kennedy, "Stereotyping: Improving particle swarm performance with cluster analysis," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, vol. 2, (La Jolla, CA, USA), pp. 1507–1512, July 2000.

[36] K. Veeramachaneni, T. Peram, C. Mohan, and L. A. Osadciw, "Optimization using particle swarms with near neighbor interactions," in *Proc. of the Genetic and Evolutionary Computation Conference (GECCO)*, (Chicago, IL, USA), pp. 110–121, July 2003.

[37] T. Peram, K. Veeramachaneni, and C. K. Mohan, "Fitness-distance-ratio based particle swarm optimization," in *Proc. of the IEEE Swarm Intelligence Symposium (SIS)*, (Indianapolis, IN, USA), pp. 174–181, Apr. 2003.

[38] F. van den Bergh and A. P. Engelbrecht, "A new locally convergent particle swarm optimiser," in *Proc. of the IEEE International Conference on Systems, Man and Cybernetics (SMC)*, vol. 3, (Yasmine Hammamet, Tunisia), p. 6, Oct. 2002.

[39] R. Brits, A. P. Engelbrecht, and F. van den Bergh, "A niching particle swarm optimizer," in *Proc. of the 4th Conference on Simulated Evolution and Learning (SEAL)*, vol. 2, (Singapore, Singapore), pp. 692–696, Nov. 2002.

[40] N. Higashi and H. Iba, "Particle swarm optimization with Gaussian mutation," in *Proc. of the IEEE Swarm Intelligence Symposium (SIS)*, (Indianapolis, IN, USA), pp. 72–79, Apr. 2003.

[41] H. Tang, X. Yang, and S. Xiong, "Modified particle swarm algorithm for vehicle routing optimization of smart logistics," in *Proc. of the 2nd International Conference on Measurement, Information and Control (ICMIC)*, vol. 2, (Harbin, China), pp. 783–787, Aug. 2013.

[42] A. P. Engelbrecht, "Asynchronous particle swarm optimization with discrete crossover," in *Proc. of the IEEE Symposium on Swarm Intelligence (SIS)*, (Orlando, FL, USA), pp. 1–8, Dec. 2014.

[43] E. S. Peer, F. van den Bergh, and A. P. Engelbrecht, "Using neighbourhoods with the guaranteed convergence PSO," in *Proc. of the IEEE Swarm on Intelligence Symposium (SIS)*, (Indianapolis, IN, USA), pp. 235–242, Apr. 2003.

[44] F. van den Bergh and A. P. Engelbrecht, "Cooperative learning in neural networks using particle swarm optimizers," *South African Comput. J.*, vol. 26, pp. 84–90, Nov. 2000.

[45] F. van den Bergh and A. P. Engelbrecht, "A cooperative approach to particle swarm optimization," *IEEE T. Evolut. Comput.*, vol. 8, pp. 225–239, June 2004.

[46] Z. Zhan, J. Zhang, Y. Li, and H. S. Chung, "Adaptive particle swarm optimization," *IEEE T. Syst. Man Cy. B*, vol. 39, pp. 1362–1381, Dec. 2009.

[47] K. E. Parsopoulos and M. N. Vrahatis, *Particle Swarm Optimization Method for Constrained Optimization Problem*, vol. 76 of *Frontiers in Artificial Intelligence and Applications*, pp. 214–220. Fairfax, VA, USA: IOS Press, 2002.

[48] X. Hu and R. Eberhart, "Solving constrained nonlinear optimization problems with particle swarm optimization," in *Proc. of the 6th World Multiconference on Systemics, Cybernetics and Informatics (SCI)*, vol. 5, (Orlando, FL, USA), pp. 203–206, July 2002.

[49] X. Hu, R. C. Eberhart, and Y. Shi, "Engineering optimization with particle swarm," in *Proc. of the IEEE Swarm Intelligence Symposium (SIS)*, (Indianapolis, IN, USA), pp. 53–57, Apr. 2003.

[50] G. Coath and S. K. Halgamuge, "A comparison of constraint-handling methods for the application of particle swarm optimization to constrained nonlinear optimization problems," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, vol. 4, (Canberra, Australia), pp. 2419–2425, Dec. 2003.

[51] S. He, E. Prempain, and Q. H. Wu, "An improved particle swarm optimizer for mechanical design optimization problems," *Eng. Optimiz.*, vol. 36, pp. 585–605, Oct. 2004.

[52] C.-L. Sun, J.-C. Zeng, and J.-S. Pan, "An improved vector particle swarm optimization for constrained optimization problems," *Inform. Sciences*, vol. 181, pp. 1153–1163, Mar. 2011.

[53] X. Hu and R. C. Eberhart, "Multiobjective optimization using dynamic neighborhood particle swarm optimization," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, vol. 2, (Honolulu, HI, USA), pp. 1677–1681, May 2002.

[54] C. A. Coello Coello and M. Salazar Lechuga, "MOPSO: A proposal for multiple objective particle swarm optimization," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, vol. 2, (Honolulu, HI, USA), pp. 1051–1056, May 2002.

[55] C. A. C. Coello, G. T. Pulido, and M. S. Lechuga, "Handling multiple objectives with particle swarm optimization," *IEEE T. Evolut. Comput.*, vol. 8, pp. 256–279, June 2004.

[56] J. E. Fieldsend and S. Singh, "A multi-objective algorithm based upon particle swarm optimisation, an efficient data structure and turbulence," in *Proc. of the UK Workshop on Computational Intelligence (UKCI)*, (Birmingham, UK), pp. 37–44, Sept. 2002.

[57] S. Mostaghim and J. Teich, "Strategies for finding good local guides in multi-objective particle swarm optimization (MOPSO)," in *Proc. of the IEEE Swarm Intelligence Symposium (SIS)*, (Indianapolis, IN, USA), pp. 26–33, Apr. 2003.

[58] K. E. Parsopoulos, V. P. Plagianakos, G. D. Magoulas, and M. N. Vrahatis, "Stretching technique for obtaining global minimizers through particle swarm optimization," in *Proc. of the Particle Swarm Optimization Workshop*, vol. 29, (Indianapolis, IN, USA), pp. 22–29, Apr. 2001.

[59] K. E. Parsopoulos and M. N. Vrahatis, "Modification of the particle swarm optimizer for locating all the global minima," in *Proc. of the 5th International Conference on Artificial Neural Nets and Genetic Algorithms (ICANNGA)*, (Prague, Czech Republic), pp. 324–327, Apr. 2001.

[60] R. Brits, A. P. Engelbrecht, and F. van den Bergh, "Solving systems of unconstrained equations using particle swarm optimization," in *Proc. of the IEEE International Conference on Systems, Man and Cybernetics (SMC)*, vol. 3, (Yasmine Hammamet, Tunisia), p. 6, Oct. 2002.

[61] K. E. Parsopoulos and M. N. Vrahatis, "On the computation of all global minimizers through particle swarm optimization," *IEEE T. Evolut. Comput.*, vol. 8, pp. 211–224, June 2004.

[62] T. Blackwell and J. Branke, "Multi-swarm optimization in dynamic environments," in *Proc. of the Workshops on Applications of Evolutionary Computation (EvoWorkshops)*, (Coimbra, Portugal), pp. 489–500, Apr. 2004.

[63] J. Kennedy, "The particle swarm: Social adaptation of knowledge," in *Proc. of the IEEE International Conference on Evolutionary Computation (ICEC)*, (Indianapolis, IN, USA), pp. 303–308, Apr. 1997.

[64] I. L. Schoeman and A. P. Engelbrecht, "Using vector operations to identify niches for particle swarm optimization," in *Proc. of the IEEE Conference on Cybernetics and Intelligent Systems (CIS)*, vol. 1, (Singapore, Singapore), pp. 361–366, Dec. 2004.

[65] I. L. Schoeman and A. P. Engelbrecht, "A parallel vector-based particle swarm optimizer," in *Proc. of the 7th International Conference on Adaptive and Natural Computing Algorithms (ICANNGA)*, (Coimbra, Portugal), pp. 268–271, Mar. 2005.

[66] X. Li, "A multimodal particle swarm optimizer based on fitness Euclidean-distance ratio," in *Proc. of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, (London, UK), pp. 78–85, July 2007.

[67] J. Li, D. Wan, Z. Chi, and X. Hu, "An efficient fine-grained parallel particle swarm optimization method based on GPU-acceleration," *Int. J. Innov. Comput. I.*, vol. 3, pp. 1707–1714, Dec. 2007.

[68] X. Li, "Adaptively choosing neighbourhood bests using species in a particle swarm optimizer for multimodal function optimization," in *Proc. of the Genetic and Evolutionary Computation Conference (GECCO)*, (Seattle, WA, USA), pp. 105–116, June 2004.

[69] D. Parrott and X. Li, "A particle swarm model for tracking multiple peaks in a dynamic environment using speciation," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, vol. 1, (Portland, OR, USA), pp. 98–103, June 2004.

[70] D. Parrott and X. Li, "Locating and tracking multiple dynamic optima by a particle swarm model using speciation," *IEEE T. Evolut. Comput.*, vol. 10, pp. 440–458, Aug. 2006.

[71] X. Li, "Niching without niching parameters: Particle swarm optimization using a ring topology," *IEEE T. Evolut. Comput.*, vol. 14, pp. 150–169, Feb. 2010.

[72] C. Yue, B. Qu, and J. Liang, "A multi-objective particle swarm optimizer using ring topology for solving multimodal multi-objective problems," *IEEE T. Evolut. Comput.*, vol. 22, pp. 805–817, Oct. 2018.

[73] R. Mendes, J. Kennedy, and J. Neves, "The fully informed particle swarm: Simpler, maybe better," *IEEE T. Evolut. Comput.*, vol. 8, pp. 204–210, June 2004.

[74] S. Lalwani, H. Sharma, S. C. Satapathy, K. Deep, and J. C. Bansal, "A survey on parallel particle swarm optimization algorithms," *Arab. J. Sci. Eng.*, vol. 44, pp. 2899–2923, Apr. 2019.

[75] M. Gupta and K. Deep, "A state-of-the-art review of population-based parallel metaheuristics," in *Proc. of the World Congress on Nature and Biologically Inspired Computing (NaBIC)*, (Coimbatore, India), pp. 1604–1607, Dec. 2009.

[76] D. Gies and Y. Rahmat-Samii, "Reconfigurable array design using parallel particle swarm optimization," in *Proc. of the IEEE Antennas and Propagation Society International Symposium*, vol. 1, (Columbus, OH, USA), pp. 177–180, June 2003.

[77] S. Baskar and P. N. Suganthan, "A novel concurrent particle swarm optimization," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, vol. 1, (Portland, OR, USA), pp. 792–796, June 2004.

[78] S.-C. Chu and J.-S. Pan, *Intelligent Parallel Particle Swarm Optimization Algorithms*, vol. 22 of *Studies in Computational Intelligence*, pp. 159–175. Berlin, Heidelberg: Springer, 2006.

[79] J.-F. Chang, S.-C. Chu, J. Roddick, and J.-S. Pan, "A parallel particle swarm optimization algorithm with communication strategies," *J. Inf. Sci. Eng.*, vol. 21, pp. 809–818, July 2005.

[80] J. F. Schutte, B. J. Fregly, R. T. Haftka, and A. D. George, "A parallel particle swarm optimizer," tech. rep., University of Florida, Department of Electrical and Computer Engineering, Gainesville, FL, USA, 2003.

[81] J. F. Schutte, J. A. Reinbolt, B. J. Fregly, R. T. Haftka, and A. D. George, "Parallel global optimization with the particle swarm algorithm," *Int. J. Numer. Meth. Eng.*, vol. 61, pp. 2296–2315, Dec. 2004.

[82] G. Venter and J. Sobieszczanski-Sobieski, "Parallel particle swarm optimization algorithm accelerated by asynchronous evaluations," *J. Aeros. Comp. Inf. Com.*, vol. 3, pp. 123–137, Mar. 2006.

[83] B.-I. Koh, A. D. George, R. T. Haftka, and B. J. Fregly, "Parallel asynchronous particle swarm optimization," *Int. J. Numer. Meth. Eng.*, vol. 67, pp. 578–595, July 2006.

[84] A. W. McNabb, C. K. Monson, and K. D. Seppi, "Parallel PSO using MapReduce," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, (Singapore, Singapore), pp. 7–14, Sept. 2007.

[85] I. Aljarah and S. A. Ludwig, "Parallel particle swarm optimization clustering algorithm based on MapReduce methodology," in *Proc. of the 4th World Congress on Nature and Biologically Inspired Computing (NaBIC)*, (Mexico City, Mexico), pp. 104–111, Nov. 2012.

[86] F. Han, W. Cui, G. Wei, and S. Wu, "Application of parallel PSO algorithm to motion parameter estimation," in *Proc. of the 9th International Conference on Signal Processing (SIP)*, (Beijing, China), pp. 2493–2496, Oct. 2008.

[87] Ş. Gülcü and H. Kodaz, "A novel parallel multi-swarm algorithm based on comprehensive learning particle swarm optimization," *Eng. Appl. Artif. Intell.*, vol. 45, pp. 33–45, Oct. 2015.

[88] B. Cao, J. Zhao, Z. Lv, X. Liu, S. Yang, X. Kang, and K. Kang, "Distributed parallel particle swarm optimization for multi-objective and many-objective large-scale optimization," *IEEE Access*, vol. 5, pp. 8214–8221, May 2017.

[89] Y. Lorion, T. Bogon, I. J. Timm, and O. Drobnik, "An agent based parallel particle swarm optimization – APPSO," in *Proc. of the IEEE Swarm Intelligence Symposium (SIS)*, (Nashville, TN, USA), pp. 52–59, Mar./Apr. 2009.

[90] N. Dali and S. Bouamama, "GPU-PSO: Parallel particle swarm optimization approaches on graphical processing unit for constraint reasoning: case of Max-CSPs," *Procedia Comput. Sci.*, vol. 60, pp. 1070–1080, Dec. 2015.

[91] B. Rymut and B. Kwolek, "GPU-supported object tracking using adaptive appearance models and particle swarm optimization," in *Proc. of the International Conference on Computer Vision and Graphics (ICCVG)*, (Warsaw, Poland), pp. 227–234, Sept. 2010.

[92] Y. Zhou and Y. Tan, "GPU-based parallel particle swarm optimization," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, (Trondheim, Norway), pp. 1493–1500, May 2009.

[93] M. M. Hussain, H. Hattori, and N. Fujimoto, "A CUDA implementation of the standard particle swarm optimization," in *Proc. of the 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, (Timisoara, Romania), pp. 219–226, Sept. 2016.

[94] L. Mussi, F. Daolio, and S. Cagnoni, "Evaluation of parallel particle swarm optimization algorithms within the CUDA™ architecture," *Inform. Sciences*, vol. 181, pp. 4642–4657, Oct. 2011.

[95] H. Zhu, C. Pu, K. Eguchi, and J. Gu, "Euclidean particle swarm optimization," in *Proc. of the 2nd International Conference on Intelligent Networks and Intelligent Systems (ICINIS)*, (Tianjin, China), pp. 669–672, Nov. 2009.

[96] H. Zhu, Y. Guo, J. Wu, J. Gu, and K. Eguchi, "Paralleling Euclidean particle swarm optimization in CUDA," in *Proc. of the 4th International Conference on Intelligent Networks and Intelligent Systems (ICINIS)*, (Kunming, China), pp. 93–96, Nov. 2011.

[97] O. Awwad, A. Al-Fuqaha, G. Ben Brahim, B. Khan, and A. Rayes, "Distributed topology control in large-scale hybrid RF/FSO networks: SIMT GPU-based particle swarm optimization approach," *Int. J. Commun. Syst.*, vol. 26, pp. 888–911, July 2013.

[98] Y. Hung and W. Wang, "Accelerating parallel particle swarm optimization via GPU," *Optim. Method. Softw.*, vol. 27, pp. 33–51, Feb. 2012.

[99] P. J. Angeline, "Using selection to improve particle swarm optimization," in *Proc. of the IEEE World Congress on Computational Intelligence (WCCI)*, (Anchorage, AK, USA), pp. 84–89, May 1998.

[100] B. Yang, Y. Chen, and Z. Zhao, "A hybrid evolutionary algorithm by combination of PSO and GA for unconstrained and constrained optimization problems," in *Proc. of the IEEE International Conference on Control and Automation (ICCA)*, (Guangzhou, China), pp. 166–170, June 2007.

[101] J. Jana, M. Suman, and S. Acharyya, "Repository and mutation based particle swarm optimization (RMPSO): A new PSO variant applied to reconstruction of gene regulatory network," *Appl. Soft Comput.*, vol. 74, pp. 330–355, Jan. 2019.

[102] A. Stacey, M. Jancic, and I. Grundy, "Particle swarm optimization with mutation," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, vol. 2, (Canberra, Australia), pp. 1425–1430, Dec. 2003.

[103] M. Imran, H. Jabeen, M. Ahmad, Q. Abbas, and W. Bangyal, "Opposition based PSO and mutation operators," in *Proc. of the 2nd International Conference on Education Technology and Computer (ICETC)*, vol. 4, (Shanghai, China), pp. 506–508, June 2010.

[104] V. Miranda and N. Fonseca, "EPSO: Best-of-two-worlds meta-heuristic applied to power system problems," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, vol. 2, (Honolulu, HI, USA), pp. 1080–1085, May 2002.

[105] G. B. Charles Darwin, *On the Origin of Species*. Oxford World's Classics, New York, NY, USA: Oxford University Press, 2008.

[106] H. Wang, H. Sun, C. Li, S. Rahnamayan, and J.-S. Pan, "Diversity enhanced particle swarm optimization with neighborhood search," *Inform. Sciences*, vol. 223, pp. 119–135, Feb. 2013.

[107] J. Robinson, S. Sinton, and Y. Rahmat-Samii, "Particle swarm, genetic algorithm, and their hybrids: Optimization of a profiled corrugated horn antenna," in *Proc. of the IEEE Antennas and Propagation Society International Symposium*, vol. 1, (San Antonio, TX, USA), pp. 314–317, June 2002.

[108] C. F. Juang, "A hybrid of genetic algorithm and particle swarm optimization for recurrent network design," *IEEE T. Syst. Man Cy. B*, vol. 34, pp. 997–1006, Mar. 2004.

[109] F. Valdez, P. Melin, and O. Castillo, "Evolutionary method combining particle swarm optimization and genetic algorithms using fuzzy logic for decision making," in *Proc. of the IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, (Jeju Island, South Korea), pp. 2114–2119, Aug. 2009.

[110] E. Alba, J. Garcia-Nieto, L. Jourdan, and E. Talbi, "Gene selection in cancer classification using PSO/SVM and GA/SVM hybrid algorithms," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, (Singapore, Singapore), pp. 284–290, Sept. 2007.

[111] Y. Fu, M. Ding, C. Zhou, and H. Hu, "Route planning for unmanned aerial vehicle (UAV) on the sea using hybrid differential evolution and quantum-behaved particle swarm optimization," *IEEE T. Syst. Man Cy-S*, vol. 43, pp. 1451–1465, Nov. 2013.

[112] J. Yang, H. Zhang, Y. Ling, C. Pan, and W. Sun, "Task allocation for wireless sensor network using modified binary particle swarm optimization," *IEEE Sens. J.*, vol. 14, pp. 882–892, Mar. 2014.

[113] G. Tian, Y. Ren, and M. Zhou, "Dual-objective scheduling of rescue vehicles to distinguish forest fires via differential evolution and particle swarm optimization combined algorithm," *IEEE T. Intell. Transp.*, vol. 17, pp. 3009–3021, Nov. 2016.

[114] J. Senthilnath, S. Kulkarni, J. A. Benediktsson, and X. S. Yang, "A novel approach for multispectral satellite image classification based on the bat algorithm," *IEEE Geosci. Remote. S.*, vol. 13, pp. 599–603, Apr. 2016.

[115] R. Storn and K. Price, "Differential evolution: A simple and efficient heuristic for global optimization over continuous spaces," *J. Glob. Optim.*, vol. 11, pp. 341–359, Dec. 1997.

[116] T. Hendtlass, "A combined swarm differential evolution algorithm for optimization problems," in *Proc. of the 14th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE)*, (Budapest, Hungary), pp. 11–18, June 2001.

[117] W.-J. Zhang and X.-F. Xie, "DEPSO: Hybrid particle swarm with differential evolution operator," in *Proc. of the IEEE International Conference on Systems, Man and Cybernetics (SMC)*, vol. 4, (Washington, WA, USA), pp. 3816–3821, Oct. 2003.

[118] B. Luitel and G. K. Venayagamoorthy, "Differential evolution particle swarm optimization for digital filter design," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, (Hong Kong, China), pp. 3954–3961, June 2008.

[119] H. Talbi and M. Batouche, "Hybrid particle swarm with differential evolution for multimodal image registration," in *Proc. of the IEEE International Conference on Industrial Technology (ICIT)*, vol. 3, (Hammamet, Tunisia), pp. 1567–1572, Dec. 2004.

[120] R. Xu, J. Xu, and D. C. Wunsch, "Clustering with differential evolution particle swarm optimization," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, (Barcelona, Spain), pp. 1–8, July 2010.

[121] V. Miranda and R. Alves, "Differential evolutionary particle swarm optimization (DEEPSO): A successful hybrid," in *Proc. of the BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence (BRICS-CCI & CBIC)*, (Ipojuca, Brazil), pp. 368–374, Sept. 2013.

[122] A. Abdullah, S. Deris, S. Z. M. Hashim, M. S. Mohamad, and S. N. V. Arjunan, "An improved local best searching in particle swarm optimization using differential evolution," in *Proc. of the 11th International Conference on Hybrid Intelligent Systems (HIS)*, (Malacca, Malaysia), pp. 115–120, Dec. 2011.

[123] M. G. H. Omran, A. P. Engelbrecht, and A. Salman, "Differential evolution based particle swarm optimization," in *Proc. of the IEEE Swarm Intelligence Symposium (SIS)*, (Honolulu, HI, USA), pp. 112–119, Apr. 2007.

[124] M. Pant, R. Thangaraj, C. Grosan, and A. Abraham, "Hybrid differential evolution: Particle swarm optimization algorithm for solving global optimization problems," in *Proc. of the 3rd International Conference on Digital Information Management (ICDIM)*, (London, UK), pp. 18–24, Nov. 2008.

[125] M. G. Epitropakis, V. P. Plagianakos, and M. N. Vrahatis, "Evolving cognitive and social experience in particle swarm optimization through differential evolution," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, (Barcelona, Spain), pp. 1–8, July 2010.

[126] C. Zhang, J. Ning, S. Lu, D. Ouyang, and T. Ding, "A novel hybrid differential evolution and particle swarm optimization algorithm for unconstrained optimization," *Oper. Res. Lett.*, vol. 37, pp. 117–122, Mar. 2009.

[127] L. Xiao and X. Zuo, "Multi-DEPSO: A DE and PSO based hybrid algorithm in dynamic environments," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, (Brisbane, Australia), pp. 1–7, June 2012.

[128] M. G. Omran, A. P. Engelbrecht, and A. Salman, "Bare bones differential evolution," *Eur. J. Oper. Res.*, vol. 196, pp. 128–139, July 2009.

[129] S. Das, A. Konar, and U. K. Chakraborty, "Annealed differential evolution," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, (Singapore, Singapore), pp. 1926–1933, Sept. 2007.

[130] G. Yang, D. Chen, and G. Zhou, "A new hybrid algorithm of particle swarm optimization," in *Proc. of the International Conference on Intelligent Computing (ICICA)*, (Kunming, China), pp. 50–60, Aug. 2006.

[131] X. H. Wang and J. J. Li, "Hybrid particle swarm optimization with simulated annealing," in *Proc. of the International Conference on Machine Learning and Cybernetics (ICMLC)*, vol. 4, (Shanghai, China), pp. 2402–2405, Aug. 2004.

[132] F. Zhao, Q. Zhang, D. Yu, X. Chen, and Y. Yang, "A hybrid algorithm based on PSO and simulated annealing and its applications for partner selection in virtual enterprise,"

in *Proc. of the International Conference on Intelligent Computing (ICICA)*, (Hefei, China), pp. 380–389, Aug. 2005.

[133] N. Sadati, M. Zamani, and H. R. F. Mahdavian, "Hybrid particle swarm-based-simulated annealing optimization techniques," in *Proc. of the 32nd Annual Conference on IEEE Industrial Electronics (IECON)*, (Paris, France), pp. 644–648, Nov. 2006.

[134] W.-j. Xia and Z.-m. Wu, "A hybrid particle swarm optimization approach for the job-shop scheduling problem," *Int. J. Adv. Manuf. Tech.*, vol. 29, pp. 360–366, June 2006.

[135] S.-C. Chu, P.-W. Tsai, and J.-S. Pan, *Parallel Particle Swarm Optimization Algorithms with Adaptive Simulated Annealing*, vol. 31, pp. 261–279. Berlin, Germany: Springer, 2006.

[136] H.-L. Shieh, C.-C. Kuo, and C.-M. Chiang, "Modified particle swarm optimization algorithm with simulated annealing behavior and its numerical verification," *Appl. Math. Comput.*, vol. 218, pp. 4365–4383, Dec. 2011.

[137] X. Deng, Z. Wen, Y. Wang, and P. Xiang, "An improved PSO algorithm based on mutation operator and simulated annealing," *Int. J. Multimedia Ubiquitous Eng.*, vol. 10, pp. 369–380, Oct. 2015.

[138] X. Dong, D. Ouyang, D. Cai, Y. Zhang, and Y. Ye, "A hybrid discrete PSO–SA algorithm to find optimal elimination orderings for Bayesian networks," in *Proc. of the 2nd International Conference on Industrial and Information Systems (ICIIS)*, vol. 1, (Dalian, China), pp. 510–513, July 2010.

[139] Q. He and L. Wang, "A hybrid particle swarm optimization with a feasibility-based rule for constrained optimization," *Appl. Math. Comput.*, vol. 186, pp. 1407–1422, Mar. 2007.

[140] P. Shelokar, P. Siarry, V. Jayaraman, and B. Kulkarni, "Particle swarm and ant colony algorithms hybridized for improved continuous optimization," *Appl. Math. Comput.*, vol. 188, pp. 129–142, May 2007.

[141] A. Ghodrati and S. Lotfi, "A hybrid CS/GA algorithm for global optimization," in *Proc. of the International Conference on Soft Computing for Problem Solving (SocProS)*, (Roorkee, India), pp. 397–404, Dec. 2011.

[142] X. Shi, Y. Li, H. Li, R. Guan, L. Wang, and Y. Liang, "An integrated algorithm based on artificial bee colony and particle swarm optimization," in *Proc. of the 6th International Conference on Natural Computation (ICNC)*, vol. 5, (Yantai, China), pp. 2586–2590, Aug. 2010.

[143] R. C. Eberhart and X. Hu, "Human tremor analysis using particle swarm optimization," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, vol. 3, (Washington, WA, USA), pp. 1927–1930, Aug. 2002.

[144] A. P. Engelbrecht and A. Ismail, "Training product unit neural networks," 1999.

[145] C. Zhang, H. Shao, and Y. Li, "Particle swarm optimisation for evolving artificial neural network," in *Proc. of the IEEE International Conference on Systems, Man and Cybernetics (SMC)*, vol. 4, (Nashville, TN, USA), pp. 2487–2490, Oct. 2000.

[146] A. Chatterjee, K. Pulasinghe, K. Watanabe, and K. Izumi, "A particle-swarm-optimized fuzzy-neural network for voice-controlled robot systems," *IEEE T. Ind. Electron.*, vol. 52, pp. 1478–1489, Dec. 2005.

[147] V. G. Gudise and G. K. Venayagamoorthy, "Comparison of particle swarm optimization and backpropagation as training algorithms for neural networks," in *Proc. of the IEEE Swarm Intelligence Symposium (SIS)*, (Indianapolis, IN, USA), pp. 110–117, Apr. 2003.

[148] R. Mendes, P. Cortez, M. Rocha, and J. Neves, "Particle swarms for feedforward neural network training," in *Proc. of the International Joint Conference on Neural Networks (IJCNN)*, vol. 2, (Honolulu, HI, USA), pp. 1895–1899, May 2002.

[149] T. Ince, S. Kiranyaz, and M. Gabbouj, "A generic and robust system for automated patient-specific classification of ECG signals," *IEEE T. Bio.-Med. Eng.*, vol. 56, pp. 1415–1426, May 2009.

[150] Y. V. Pehlivanoglu, "A new particle swarm optimization method enhanced with a periodic mutation strategy and neural networks," *IEEE T. Evolut. Comput.*, vol. 17, pp. 436–452, June 2013.

[151] H. Quan, D. Srinivasan, and A. Khosravi, "Short-term load and wind power forecasting using neural network-based prediction intervals," *IEEE T. Neur. Net. Lear.*, vol. 25, pp. 303–315, Feb. 2014.

[152] B. A. Garro, H. Sossa, and R. A. Vazquez, "Design of artificial neural networks using a modified particle swarm optimization algorithm," in *Proc. of the International Joint Conference on Neural Networks (IJCNN)*, (Atlanta, GA, USA), pp. 938–945, June 2009.

[153] B. Al-Kazemi and C. K. Mohan, "Training feedforward neural networks using multi-phase particle swarm optimization," in *Proc. of the 9th International Conference on Neural Information Processing (ICONIP)*, vol. 5, (Singapore, Singapore), pp. 2615–2619, Nov. 2002.

[154] B. Al-Kazemi and C. Mohan, *Discrete Multi-phase Particle Swarm Optimization*, pp. 305–327. Advanced Information and Knowledge Processing, London, UK: Springer, 2005.

[155] M. Conforth and Y. Meng, "Toward evolving neural networks using bio-inspired algorithms," in *Proc. of the International Conference on Artificial Intelligence (IC-AI)*, (Las Vegas, NV, USA), pp. 413–419, July 2008.

[156] M. Hamada and M. Hassan, "Artificial neural networks and particle swarm optimization algorithms for preference prediction in multi-criteria recommender systems," *Informatics*, vol. 5, p. 25, May 2018.

[157] Y. H. Zweiri, J. F. Whidborne, and L. D. Seneviratne, "A three-term backpropagation algorithm," *Neurocomputing*, vol. 50, pp. 305–318, Jan. 2003.

[158] P. Erdogmus, *Particle Swarm Optimization with Applications*. London, UK: IntechOpen, 2018.

[159] D. Gao, X. Li, and H. Chen, "Application of improved particle swarm optimization in vehicle crashworthiness," *Math. Probl. Eng.*, vol. 2019, pp. 1–10, Mar. 2019.

[160] F. Huang, R. Li, H. Liu, and R. Li, "A modified particle swarm algorithm combined with fuzzy neural network with application to financial risk early warning," in *Proc. of the IEEE Asia-Pacific Conference on Services Computing (APSCC)*, (Guangzhou, Guangdong), pp. 168–173, Mar. 2006.

[161] L. T. Bui, O. Soliman, and H. A. Abbass, "A modified strategy for the constriction factor in particle swarm optimization," in *Proc. of the 3rd Australasian Conference on Artificial Life and Computational Intelligence (ACAL)*, (Geelong, Australia), pp. 333–344, Dec. 2007.

[162] A. P. Engelbrecht, *Computational Intelligence: An Introduction*. Chichester, UK: John Wiley & Sons, second ed., 2007.

[163] A. P. Engelbrecht, *Fundamentals of Computational Swarm Intelligence*. Hoboken, NJ, USA: John Wiley & Sons, 2006.

[164] K. Zielinski and R. Laur, "Stopping criteria for a constrained single-objective particle swarm optimization algorithm," *Informatica*, vol. 31, pp. 51–59, Mar. 2007.

[165] S. Abraham, S. Sanyal, and M. Sanglikar, "Particle swarm optimisation based Diophantine equation solve," *Int. J. Bio-Inspir. Com.*, vol. 2, pp. 100–114, Mar. 2010.

[166] O. Pérez, I. Amaya, and R. Correa, "Numerical solution of certain exponential and nonlinear Diophantine systems of equations by using a discrete particle swarm optimization algorithm," *Appl. Math. Comput.*, vol. 225, pp. 737–746, Dec. 2013.

[167] I. Amaya, J. Cruz, and R. Correa, "Real roots of nonlinear systems of equations through a metaheuristic algorithm," *DYNA-Colombia*, vol. 78, pp. 15–23, Dec. 2011.

[168] M. Grailoo, M. Grailoo, and A. Bakhshi, "Solving systems of nonlinear equations with an improved particle swarm optimization," in *Proc. of the 4th IEEE International Conference on Computer Science and Information Technology (ICCSIT)*, vol. 9, (Chengdu, China), pp. 578–582, June 2011.

[169] M. Jaberipour, E. Khorram, and B. Karimi, "Particle swarm algorithm for solving systems of nonlinear equations," *Comput. Math. Appl.*, vol. 62, pp. 566–576, Mar. 2006.

[170] C. P. Salomon, G. Lambert-Torres, L. E. B. da Silva, M. P. Coutinho, and C. H. V. de Moraes, "A hybrid particle swarm optimization approach for load-flow computation," *Int. J. Innov. Comput. I.*, vol. 9, pp. 4359–4372, Nov. 2013.

[171] J. Zhao, J. Sun, C.-H. Lai, and W. Xu, "An improved quantum-behaved particle swarm optimization for multi-peak optimization problems," *Int. J. Comput. Math.*, vol. 88, pp. 517–532, Dec. 2010.

[172] S. Reyes-Sierra, J. Plata-Rueda, and R. Correa-Cely, "Real and/or complex roots calculation of nonlinear equations systems through modified particle swarm optimization," *Ing. Univ.*, vol. 16, pp. 349–362, Dec. 2012.

[173] X. F. Mai and L. Li, "Bacterial foraging algorithm based on PSO with adaptive inertia weigh for solving nonlinear equations systems," *Adv. Mat. Res.*, vol. 655, pp. 940–947, Mar. 2013.

[174] A. M. Ibrahim and M. A. Tawhid, "A hybridization of cuckoo search and particle swarm optimization for solving nonlinear systems," *Evol. Intell.*, vol. 12, pp. 541–561, Dec. 2019.

[175] R. S. Dembo, S. C. Eisenstat, and T. Steihaug, "Inexact Newton methods," *SIAM J. Numer. Anal.*, vol. 19, pp. 400–408, Apr. 1982.

[176] J. M. McNamee and V. Pan, *Numerical Methods for Roots of Polynomials - Part II*, vol. 16 of *Studies in Computational Mathematics*. Waltham, MA, USA: Academic Press, 2013.

[177] L. J. V. Miranda, "PySwarms, a research-toolkit for particle swarm optimization in Python," *J. Open Source Softw.*, vol. 3, p. 433, Jan. 2018.

[178] N. Padhye, K. Deb, and P. Mittal, "Boundary handling approaches in particle swarm optimization," in *Proc. of the 7th International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA)*, (HuangShan, China), pp. 287–298, July 2013.

[179] M. Jamil and X. S. Yang, "A literature survey of benchmark functions for global optimisation problems," *Int. J. Math. Model. Numer. Optim.*, vol. 4, p. 150, July 2013.

[180] W. contributors, "Test functions for optimization. From Wikipedia—The Free Encyclopedia." [Online] `https://en.wikipedia.org/w/index.php?title=Test_functions_for_optimization&oldid=937963949`, accessed on 30 March 2020.

[181] D. H. Ackley, *A Connectionist Machine for Genetic Hillclimbing*, vol. 28 of *Engineering and Computer Science*. Norwell, MA, USA: Kluwer Academic Publishers, 1987.

[182] F. Razavi and F. Jalali-Farahani, "Ant colony optimization: A leading algorithm in future optimization of petroleum engineering processes," in *Proc. of the 9th International Conference on Artificial Intelligence and Soft Computing (ICAISC)*, (Zakopane, Poland), pp. 469–478, June 2008.

[183] F. Hoffmeister and T. Bäck, "Genetic algorithms and evolution strategies: Similarities and differences," in *Proc. of the 1st International Conference on Parallel Problem Solving from Nature (PPSN)*, (Dortmund, Germany), pp. 455–469, Oct. 1990.

[184] L. Han and X. He, "A novel opposition-based particle swarm optimization for noisy problems," in *Proc. of the 3rd International Conference on Natural Computation (ICNC)*, vol. 3, (Haikou, China), pp. 624–629, Aug. 2007.

[185] F. M. Hemez and C. J. Stull, "Optimal inequalities to bound a performance probability," in *Proc. of the 31st International Modal Analysis Conference (IMAC)*, (Garden Grove, CA, USA), pp. 1–15, Feb. 2013.

[186] V. Picheny, T. Wagner, and D. Ginsbourger, "A benchmark of kriging-based infill criteria for noisy optimization," *Struct. Multidiscip. O.*, vol. 48, pp. 607–626, Sept. 2013.

[187] J. D. Schaffer, R. A. Caruana, L. J. Eshelman, and R. Das, "A study of control parameters affecting online performance of genetic algorithms for function optimization," in *Proc. of the 3rd International Conference on Genetic Algorithms (ICGA)*, (Fairfax, VA, USA), pp. 51–60, June 1989.

[188] R. Chiong, *Nature-Inspired Algorithms for Optimisation*, vol. 193 of *Studies in Computational Intelligence*. Berlin, Germany: Springer, 2009.

[189] R. Fisher, *Statistical Methods for Research Workers*. No. 5 in Biological Monographs and Manuals, Edinburgh, UK: Oliver and Boyd, eleventh ed., 1925.

[190] H. Scheffé, *The Analysis of Variance*. Wiley Classics Library, Hoboken, NJ, USA: John Wiley & Sons, 1959.

[191] G. Iversen, H. Norpoth, and H. Norpoth, *Analysis of Variance*. Quantitative Applications in the Social Sciences, Thousand Oaks, CA, USA: SAGE Publications, second ed., 1987.

[192] R Core Team, *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2019.

[193] A. Ratnaweera, S. K. Halgamuge, and H. C. Watson, "Self-organizing hierarchical particle swarm optimizer with time-varying acceleration coefficients," *IEEE T. Evolut. Comput.*, vol. 8, pp. 240–255, June 2004.

[194] A. El-Gallad, M. El-Hawary, A. Sallam, and A. Kalas, "Enhancing the particle swarm optimizer via proper parameters selection," in *Proc. of the IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, vol. 2, (Winnipeg, Canada), pp. 792–797, May 2002.

[195] Y. Dai, L. Liu, and Y. Li, "An intelligent parameter selection method for particle swarm optimization algorithm," in *Proc. of the 4th International Joint Conference on Computational Sciences and Optimization (CSO)*, (Yunnan, China), pp. 960–964, Apr. 2011.

[196] J. C. Bansal, P. K. Singh, M. Saraswat, A. Verma, S. S. Jadon, and A. Abraham, "Inertia weight strategies in particle swarm optimization," in *Proc. of the 3rd World Congress on Nature and Biologically Inspired Computing (NaBIC)*, (Salamanca, Spain), pp. 633–640, Oct. 2011.

[197] Y. He, W. Ma, and J. Zhang, "The parameters selection of PSO algorithm influencing on performance of fault diagnosis," *MATEC Web Conf.*, vol. 63, p. 02019, Jan. 2016.

[198] S. Helwig and R. Wanka, "Theoretical analysis of initial particle swarm behavior," in *Proc. of the 10th International Conference on Parallel Problem Solving from Nature (PPSN)*, (Dortmund, Germany), pp. 889–898, Sept. 2008.

[199] Q. Liu, W. Wei, H. Yuan, Z.-H. Zhan, and Y. Li, "Topology selection for particle swarm optimization," *Inform. Sciences*, vol. 363, pp. 154–173, Oct. 2016.

[200] A. P. Engelbrecht, "Particle swarm optimization: Global best or local best?," in *Proc. of the BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence (BRICS-CCI & CBIC)*, (Ipojuca, Brazil), pp. 124–135, Sept. 2013.

[201] J. Kennedy and R. Mendes, "Neighborhood topologies in fully informed and best-of-neighborhood particle swarms," *IEEE T. Syst. Man Cy. C*, vol. 36, pp. 515–519, July 2006.

[202] M. Clerc, *Combinatorial Problems*, ch. 16, pp. 201–210. Hoboken, NJ, USA: John Wiley & Sons, 2010.

[203] A. S. Mohais, R. Mendes, C. Ward, and C. Posthoff, "Neighborhood re-structuring in particle swarm optimization," in *Proc. of the 18th Australian Joint Conference on Artificial Intelligence (AUS-AI)*, (Sydney, Australia), pp. 776–785, Dec. 2005.

[204] Y. Valle, G. Venayagamoorthy, S. Mohagheghi, J. Hernandez Mejia, and R. G. Harley, "Particle swarm optimization: Basic concepts, variants and applications in power systems," *IEEE T. Evolut. Comput.*, vol. 12, pp. 171–195, Apr. 2008.

[205] P. Moritz *et al.*, "Ray: A distributed framework for emerging ai applications," in *Proc. of the 13rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, (Carlsbad, CA, USA), pp. 561–577, Oct. 2018.

[206] G. Ramadas, E. Fernandes, and A. Rocha, "Finding multiple roots of systems of non-linear equations by a hybrid harmony search-based multistart method," *Appl. Math. Inform. Sci.*, vol. 12, pp. 21–32, Jan. 2018.

[207] R. M. Silva, M. G. Resende, and P. M. Pardalos, "Finding multiple roots of a box-constrained system of nonlinear equations with a biased random-key genetic algorithm," *J. Glob. Optim.*, vol. 60, pp. 289–306, Oct. 2014.

[208] M. J. Hirsch, P. M. Pardalos, and M. G. Resende, "Solving systems of nonlinear equations with continuous GRASP," *Nonlinear Anal.-Real*, vol. 10, pp. 2000–2006, Aug. 2009.

[209] I. Tsoulos and A. Stavrakoudis, "On locating all roots of systems of nonlinear equations inside bounded domain using global optimization methods," *Nonlinear Anal.-Real*, vol. 11, pp. 2465–2471, Aug. 2010.

[210] E. Pourjafari and H. Mojallali, "Solving nonlinear equations systems with a new approach based on invasive weed optimization algorithm and clustering," *Swarm Evol. Comput.*, vol. 4, pp. 33–43, June 2012.

[211] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Math. Control. Signal*, pp. 303–314, Dec. 1989.

[212] K. Hornik, M. Stinchcombe, and H. White, "Multilayered feedforward networks are universal approximators," *Neural Networks*, vol. 2, pp. 359–366, Mar. 1989.

[213] K.-I. Funahashi, "On the approximation realization of continuous mappings by neural networks," *Neural Networks*, vol. 2, no. 3, pp. 183–192, 1989.

[214] M. Chen, Y. Hao, K. Hwang, L. Wang, and L. Wang, "Disease prediction by machine learning over big data from healthcare communities," *IEEE Access*, vol. 5, pp. 8869–8879, Apr. 2017.

[215] A. R. Sharma and P. Kaushik, "Literature survey of statistical, deep and reinforcement learning in natural language processing," in *Proc. of the International Conference on Computing, Communication and Automation (ICCCA)*, (Greater Noida, India), pp. 350–354, May 2017.

[216] G. Hinton *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Proc. Mag.*, vol. 29, pp. 82–97, Nov. 2012.

[217] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Las Vegas, NA, USA), pp. 770–778, June 2016.

[218] H. A. Rowley, S. Baluja, and T. Kanade, "Neural network-based face detection," *IEEE T. Pattern. Anal.*, vol. 20, pp. 23–38, Jan. 1998.

[219] W. G. Hatcher and W. Yu, "A survey of deep learning: Platforms, applications and emerging research trends," *IEEE Access*, vol. 6, pp. 24411–24432, Apr. 2018.

[220] C. Aggarwal, *Neural Networks and Deep Learning: A Textbook.* Cham, Switzerland: Springer, 2018.

[221] A. Oliveira, *The Digital Mind: How Science is Redefining Humanity.* Cambridge, MA, EUA: MIT Press, 2017.

[222] H. Bellen, C. Tong, and H. Tsuda, "100 years of Drosophila research and its impact on vertebrate neuroscience: A history lesson for the future," *Nat. Rev. Neurosci.*, vol. 11, pp. 514–522, Jul. 2010.

[223] E. Durand, *Solutions Numériques des Équations Algébriques, Tome I: Équations du Type $f(x) = 0$; Racines d'un Polynôme.* Paris, France: Masson, 1960.

[224] I. O. Kerner, "Ein Gesamtschrittverfahren zur Berechnung der Nullstellen von Polynomen," *Numer. Math.*, vol. 8, pp. 290–294, 1966.

[225] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Adaptive Computation and Machine Learning, Cambridge, MA, EUA: MIT Press, 2016.

[226] J. Patterson and A. Gibson, *Deep Learning: A Practitioner's Approach*. Sebastopol, CA, USA: O'Reilly Media, 2017.

[227] F. M. Dias, *Técnicas de Controlo Não-Linear Baseadas em Redes Neuronais: Do Algoritmo à Implementação*. PhD thesis, Universidade de Aveiro, Aveiro, Portugal, 2005.

[228] R. Hecht-Nielsen, "Kolmogorov's mapping neural network existence theorem," in *Proc. of the International Conference on Neural Networks (ICNN)*, vol. 3, (San Diego, CA, USA), pp. 11–14, June 1987.

[229] J. Heaton, *Introduction to Neural Networks for Java*. Chesterfield, MO, USA: Heaton Research, second ed., 2008.

[230] J. C. Principe, N. R. Euliano, and W. C. Lefebvre, *Neural and Adaptive Systems: Fundamentals Through Simulations*. Hoboken, NJ, USA: John Wiley & Sons, 1999.

[231] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," in *Proc. of the 7th International Conference on Learning Representations (ICLR)*, (New Orleans, LA, USA), May 2019.

[232] J. Brownlee, "How to configure the number of layers and nodes in a neural network." [Online] `https://machinelearningmastery.com/how-to-configure-the-number-of-layers-and-nodes-in-a-neural-network/`, accessed on 21 February 2020.

[233] A. M. Olveira, *Inductive Learning by Selection of Minimal Complexity Representations*. PhD thesis, University of California, Berkeley, CA, USA, 1994.

[234] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *B. Math. Biophys.*, vol. 5, pp. 115–133, Dec. 1943.

[235] F. Rosenblatt, "The perceptron, a perceiving and recognizing automaton," Tech. Rep. 85-460-1, Cornell Aeronautical Laboratory, Jan. 1957.

[236] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychol. Rev.*, vol. 65, pp. 386–408, Nov. 1958.

[237] F. F. Rosenblatt, "Principles of neurodynamics. perceptrons and the theory of brain mechanisms," *Am. J. Psychol.*, vol. 76, pp. 705–707, Dec. 1963.

[238] A. G. Ivakhnenko and V. G. Lapa, *Cybernetic Predicting Devices*. New York, NY, USA: CCM Information Corporation, 1973.

[239] M. Minsky and S. A. Papert, *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, EUA: MIT Press, expanded ed., 1988.

[240] S. Linnainmaa, "The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors," Master's thesis, University of Helsinki, Helsinki, Finland, 1970.

[241] P. Werbos, *Beyond regression: New tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, Cambridge, MA, USA, 1974.

[242] P. J. Werbos, "Applications of advances in nonlinear sensitivity analysis," in *System Modeling and Optimization*, vol. 38 of *Lecture Notes in Control and Information Sciences*, pp. 762–770, Berlin, Germany: Springer, 1982.

[243] D. E. Rumelhart and J. L. McClelland, *Learning Internal Representations by Error Propagation*, vol. 1 of *Explorations in the Microstructure of Cognition*, pp. 318–362. Cambridge, MA, EUA: MIT Press, 1987.

[244] K. Fukushima, "A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biol. Cybernetics*, vol. 36, pp. 193–202, Apr. 1980.

[245] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten Zip code recognition," *Neural Comput.*, vol. 1, pp. 541–551, Dec. 1989.

[246] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *P. Natl. Acad. Sci. USA*, vol. 79, pp. 2554–2558, Apr. 1982.

[247] W. Little, "The existence of persistent states in the brain," *Math. Biosci.*, vol. 19, pp. 101–120, Feb. 1974.

[248] D. H. Ballard, "Modular learning in neural networks," in *Proc. of the 6th National Conference on Artificial Intelligence (AAAI)*, (Seatle, WA, USA), pp. 279–284, July 1987.

[249] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen," Master's thesis, Technical University of Munich, Munich, Germany, 1991.

[250] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, pp. 1735–1780, Nov. 1997.

[251] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, pp. 273–297, Sept. 1995.

[252] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE T. Signal Proces.*, vol. 45, pp. 2673–2681, Nov. 1997.

[253] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, pp. 504–507, Apr. 2006.

[254] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Proc. of the 20th Annual Conference on Neural Information Processing Systems (NIPS)*, (Vancouver, Canada), pp. 153–160, Dec. 2006.

[255] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep, big, simple neural nets for handwritten digit recognition," *Neural Comput.*, vol. 22, pp. 3207–3220, Dec. 2010.

[256] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proc. of the 26th Annual International Conference on Machine Learning (ICML)*, (Montreal, Canada), pp. 873–880, June 2009.

[257] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *Proc. of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, (Barcelona, Spain), pp. 1237–1242, July 2011.

[258] D. Ciregan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Providence, Rhode Island), pp. 3642–3649, June 2012.

[259] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proc. of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, (Fort Lauderdale, FL, USA), pp. 315–323, Apr. 2011.

[260] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. of the 26th Annual Conference on Neural Information Processing Systems (NIPS)*, (Lake Tahoe, NV, USA), pp. 1097–1105, Dec. 2012.

[261] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Proc. of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, (Vancouver, Canada), pp. 6645–6649, May 2013.

[262] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, pp. 1929–1958, Jan. 2014.

[263] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. of the 3rd International Conference on Learning Representations (ICLR)*, (San Diego, CA, USA), May 2015.

[264] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Proc. of the 28th Annual Conference on Neural Information Processing Systems (NIPS)*, (Montreal, Canada), pp. 2672–2680, Dec. 2014.

[265] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. of the 31st Annual Conference on Neural Information Processing Systems (NIPS)*, (Long Beach, CA, USA), pp. 5998–6008, Dec. 2017.

[266] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," May 2018. arXiv:1810.04805.

[267] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. of the 12nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, (Savannah, GA, USA), pp. 265–283, Nov. 2016.

[268] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. of the 33rd Conference on Neural Information Processing Systems (NeurIPS)*, (Vancouver, Canada), pp. 8024–8035, Dec. 2019.

[269] J. M. McNamee, *Numerical Methods for Roots of Polynomials - Part I*, vol. 14 of *Studies in Computational Mathematics*. Amsterdam, Netherlands: Elsevier, 2007.

[270] B. Sendov, A. Andreev, and N. Kjurkchiev, "Numerical solution of polynomial equations," in *Handbook of Numerical Analysis*, vol. 3, pp. 625–778, Amsterdam, Netherlands: Elsevier, 1994.

[271] M. Petković, *Point Estimation of Root Finding Methods*, vol. 1933 of *Lecture Notes in Mathematics*. Berlin, Germany: Springer, 2008.

[272] R. Hormis, G. Antoniou, and S. Mentzelopoulou, "Separation of two-dimensional polynomials via a sigma-pi neural net," in *Proc. of the IASTED International Conference on Modelling, Simulation, and Optimization (MSO)*, (Pittsburgh, PA, USA), pp. 304–306, Apr. 1995.

[273] S. J. Perantonis, N. Ampazis, S. J. Varoufakis, and G. Antoniou, "Factorization of 2-D polynomials using neural networks and constrained learning techniques," in *Proc. of the IEEE International Symposium on Industrial Electronics (ISIE)*, vol. 3, (Guimarães, Portugal), pp. 1276–1280, July 1997.

[274] D.-S. Huang and M. Zhao, "A neural network based factorization model for polynomials in several elements," in *Proc. of the 5th International Conference on Signal Processing Proceedings and 16th World Computer Congress (WCC - ICSP)*, vol. 3, (Beijing, China), pp. 1617–1622, Aug. 2000.

[275] D.-S. Huang and Z. Chi, "Neural networks with problem decomposition for finding real roots of polynomials," in *Proc. of the IEEE International Joint Conference on Neural Network (IJCNN)*, (Washington, WA, USA), pp. A25–A30, July 2001.

[276] D.-S. Huang and Z. Chi, "Finding complex roots of polynomials by feedforward neural networks," in *Proc. of the IEEE International Joint Conference on Neural Network (IJCNN)*, (Washington, WA, USA), pp. A13–A18, July 2001.

[277] D.-S. Huang, H. I. Horace, C. L. Ken, Z. Chi, and H.-S. Wong, "A new partitioning neural network model for recursively finding arbitrary roots of higher order arbitrary polynomials," *Appl. Math. Comput.*, vol. 162, pp. 1183–1200, Mar. 2005.

[278] D.-S. Huang, H. H. Ip, Z. Chi, and H. Wong, "Dilation method for finding close roots of polynomials based on constrained learning neural networks," *Phys. Lett. A*, vol. 309, pp. 443–451, Mar. 2003.

[279] D.-S. Huang, H. H. Ip, and Z. Chi, "A neural root finder of polynomials based on root moments," *Neural Comput.*, vol. 16, pp. 1721–1762, Aug. 2004.

[280] D.-S. Huang, H. H.-S. Ip, K. C. K. Law, and Z. Chi, "Zeroing polynomials using modified constrained neural network approach," *IEEE T. Neural Networ.*, vol. 16, pp. 721–732, May 2005.

[281] B. Mourrain, N. Pavlidis, D. Tasoulis, and M. Vrahatis, "Determining the number of real roots of polynomials through neural networks," *Comput. Math. Appl.*, vol. 51, pp. 527–536, Feb. 2006.

[282] X. Zhang, D. Zhu, and W. Hu, "Finding multiple real roots by neural networks based on complete discrimination system of polynomial," in *Proc. of the IEEE Conference on Cybernetics and Intelligent Systems (CIS)*, (Chengdu, China), pp. 236–241, Sept. 2008.

[283] M. Das and P. Seal, "Polynomial real roots finding using feed forward neural network: A simple approach," in *Proc. of the IEEE National Conference on Computing and Communication Systems (NCCCS)*, (Durgapur, India), pp. 1–4, Nov. 2012.

[284] A.-L. Cauchy, *Cours d'Analyse de l'École Royale Polytechnique.* Cambridge Library Collection - Mathematics, New York, NY, USA: Cambridge University Press, 2009.

[285] A. Terui and T. Sasaki, "Durand–Kerner method for the real roots," *Jpn. J. Ind. Appl. Math.*, vol. 19, p. 19, Feb. 2002.

[286] P. Fraigniaud, "The Durand–Kerner polynomials roots-finding method in case of multiple roots," *BIT*, vol. 31, pp. 112–123, Mar. 1991.

[287] P. B. Harrington, "Sigmoid transfer functions in backpropagation neural networks," *Anal. Chem.*, vol. 65, pp. 2167–2168, Aug. 1993.

[288] K. L. Priddy and P. E. Keller, *Artificial Neural Networks: An Introduction*, vol. TT68 of *Tutorial Texts in Optical Engineering.* Bellingham, WA, USA: SPIE Press, 2005.

[289] K. Levenberg, "A method for the solution of certain nonlinear problems in least squares," *Q. Appl. Math.*, vol. 2, pp. 164–168, July 1944.

[290] D. W. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *J. Soc. Ind. Appl. Math.*, vol. 11, pp. 431–441, June 1963.

[291] M. T. Hagan and M. B. Menhaj, "Training feedforward networks with the Marquardt algorithm," *IEEE T. Neural Networ.*, vol. 5, pp. 989–993, Nov. 1994.

[292] M. T. Hagan, H. B. Demuth, M. H. Beale, and O. De Jesús, *Neural Network Design.* Stillwater, OK, USA: Martin Hagan, second ed., 2014.

[293] J. Heaton, *Artificial Intelligence for Humans, Volume 3: Deep Learning and Neural Networks.* Chesterfield, MO, USA: Heaton Research, 2015.

[294] A.-L. Cauchy, *Exercices de Mathématiques*. Whitefish, MT, USA: Kessinger Publishing, 2010.

[295] H. Yu and B. Wilamowski, *Levenberg-Marquardt Training*, vol. 5 of *Intelligent Systems*, ch. 12, pp. 12–1–12–16. Boca Raton, FL, USA: CRC Press, second ed., 2011.

[296] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," Sept. 2012. arXiv:1209.5145.

[297] Q. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Ng, "On optimization methods for deep learning," in *Proc. of the 28th International Conference on Machine Learning (ICML)*, pp. 265–272, Jan. 2011.

Appendix A

Particle Swarm Optimisation Initialisation Scheme

Algorithm 1 A PSO initialisation scheme for both gbest and lbest models.

for $i \leftarrow 1, l$ **do** ▷ For each particle...
  $\vec{x}_0^{\,i} \leftarrow U(\vec{x}_{\min}, \vec{x}_{\max}) \in \mathbb{R}^d$ ▷ Generate a random initial position
  $\vec{p}_0^{\,i} \leftarrow \vec{x}_0^{\,i}$ ▷ Set the best personal position to its initial position
  $\vec{n}_{best} \leftarrow \min(N_i)$ ▷ Find the position of the best particle in the neighbourhood
  if $f(\vec{n}_{best}) < f(\vec{p}_0^{\,i})$ **then** ▷ If the neighbourhood's best position is better than the current personal best
    $\vec{l}_0^{\,i} \leftarrow \vec{n}_{best}$ ▷ Accept the neighbourhood's best position
  else
    $\vec{l}_0^{\,i} \leftarrow \vec{p}_0^{\,i}$ ▷ Otherwise, set the neighbourhood's best position to the current personal best position
  end if
  $\vec{V}_0^{\,i} \leftarrow U(\vec{x}_{\min}, \vec{x}_{\max}) \in \mathbb{R}^d$ ▷ Generate a random initial velocity
end for

Appendix B

Particle Swarm Optimisation Algorithm Flowchart

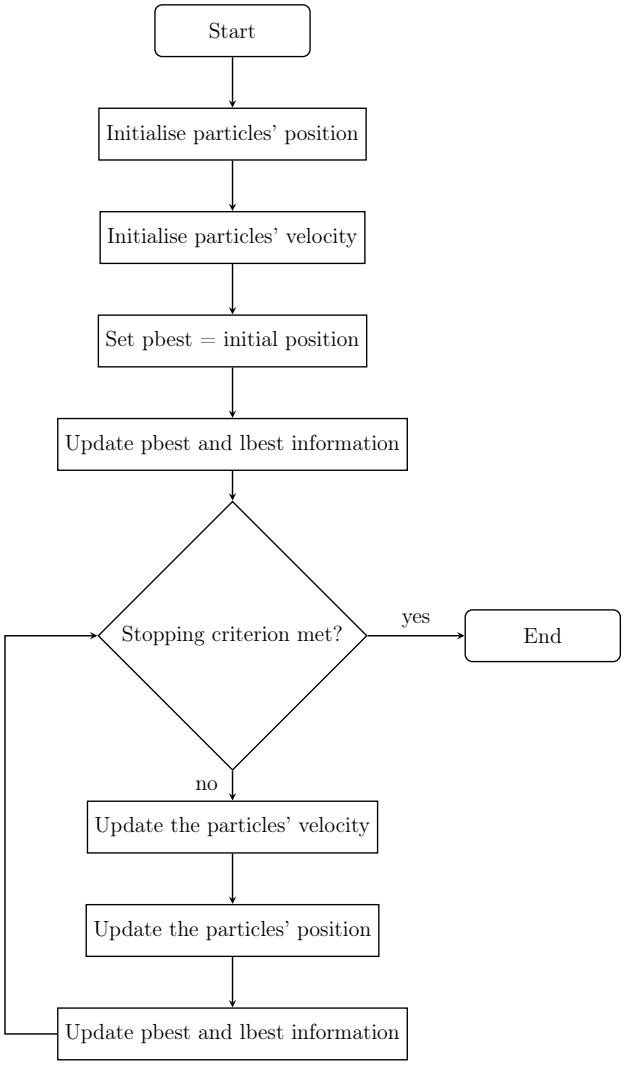


Figure B.1: Flowchart of the PSO algorithm.

Appendix C

Particle Swarm Optimisation Cycle

Algorithm 2 The optimisation cycle of the PSO algorithm for root-finding.

while $t <$ `maximum_number_of_iterations` **and** *not* `precision_stop` **do**
  $t \leftarrow t + 1$ ▷ Increment the number of iterations...
  for $i \leftarrow 1, s$ **do** ▷ For each particle...
    for $j \leftarrow 1, d$ **do** ▷ For each dimension, compute velocity
      $\varphi_1 \leftarrow U(0,1)$ ▷ Generate two random numbers
      $\varphi_2 \leftarrow U(0,1)$
      $\vec{V}^{\,i}_{t+1}[j] \leftarrow \omega \vec{V}^{\,i}_{t}[j] + \varphi_1 R_1{}^{i}_{t}(\vec{p}^{\,i}_{t}[j] - \vec{x}^{\,i}_{t}[j]) + \varphi_2 R_2{}^{i}_{t}(\vec{g}_{t}[j] - \vec{x}^{\,i}_{t}[j])$
    end for
    $\vec{x}^{\,i}_{t+1} \leftarrow \vec{x}^{\,i}_{t} + \vec{V}^{\,i}_{t+1}$ ▷ Compute next position
    $\vec{x}^{\,i}_{t+1} \leftarrow$ `check_bounds(`$\vec{x}^{\,i}_{t+1}$`)` ▷ Change, if needed, the next position according to the search space's bounds
    $fit^{\,i}_{t+1} \leftarrow |f(\vec{x}^{\,i}_{t+1})|$ ▷ Compute the cost
    if $fit^{\,i}_{t+1} < f(\vec{p}^{\,i}_{t})$ **then** ▷ If necessary, update personal best position
      $\vec{p}^{\,i}_{t} \leftarrow \vec{x}^{\,i}_{t+1}$
      $\vec{n}_{best} \leftarrow \min(N_i)$ ▷ Find the position of the best particle in the neighbourhood
      if $f(\vec{n}_{best}) < f(\vec{p}^{\,i}_{t})$ **then** ▷ If the neighbourhood's best position is better than the current personal best
        $\vec{l}^{\,i}_{t} \leftarrow \vec{n}_{best}$ ▷ Accept the neighbourhood's best position
      else
        $\vec{l}^{\,i}_{t} \leftarrow \vec{p}^{\,i}_{t}$ ▷ Otherwise, set the neighbourhood's best position to the current personal best position
      end if
    end if
  end for
  $\hat{y} \leftarrow \min(\vec{p}_{\,t})$ ▷ Find the position of the best particle in the swarm
  if $\hat{y} \leq \epsilon$ **then** ▷ Check the precision stop criterion
    `precision_stop` $\leftarrow$ `True`
  end if
end while

Appendix D

Test Functions

D.1 Ackley Function

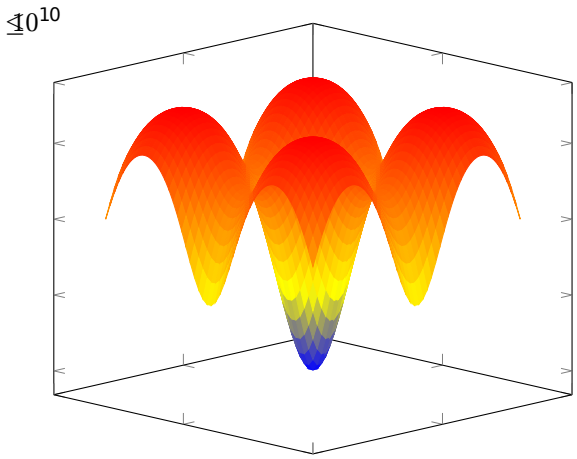


Figure D.1: A 2D graphical representation of the Ackley function.

- Graphical representation: Figure D.1
- Expression:

$$f(x_1, \cdots, x_d) = -20exp\left(-0.2\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(2\pi x_i)\right) + 20 + e \tag{D.1}$$

- Number of dimensions used: 30
- Search domain: $-5 \leq x_i \leq 5$
- Root at: $(0, \cdots, 0)$

D.2 Rastrigin Function

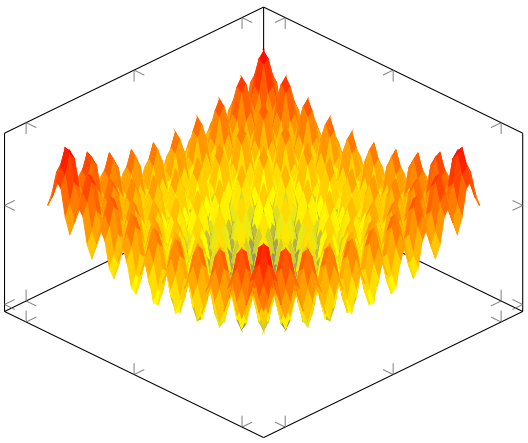


Figure D.2: A 2D graphical representation of the Rastrigin function.

- Graphical representation: Figure D.2
- Expression:

$$f(x_1, \cdots, x_d) = 10d + \sum_{i=1}^{d} \left(x_i^2 - 10\cos(2\pi x_i) \right) \tag{D.2}$$

- Number of dimensions used: 30
- Search domain: $-5.12 \leq x_i \leq 5.12$
- Root at: $(0, \cdots, 0)$

D.3 Rosenbrock Function

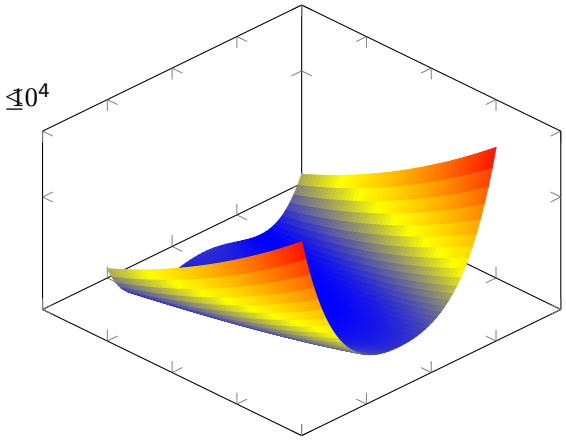


Figure D.3: A 2D graphical representation of the Rosenbrock function.

- Graphical representation: Figure D.3
- Expression:

$$f(x_1, \cdots, x_d) = \sum_{i=1}^{d-1} \left(100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2 \right) \tag{D.3}$$

- Number of dimensions used: 30
- Search domain: $-2.048 \leq x_i \leq 2.048$
- Root at: $(1, \cdots, 1)$

D.4 Schaffer Function No. 2

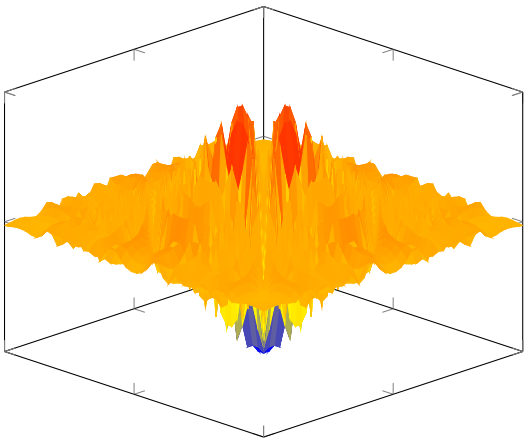


Figure D.4: A 2D graphical representation of a Schaffer function no. 2.

- Graphical representation: Figure D.4
- Expression:

$$f(x, y) = 0.5 + \frac{\sin^2(x^2 - y^2) - 0.5}{[1 + 0.001 \times (x^2 + y^2)]^2} \tag{D.4}$$

- Number of dimensions: 2
- Search domain: $-100 \leq x, y \leq 100$
- Root at: $(0, 0)$

D.5 Sphere Function

- Graphical representation: Figure D.5

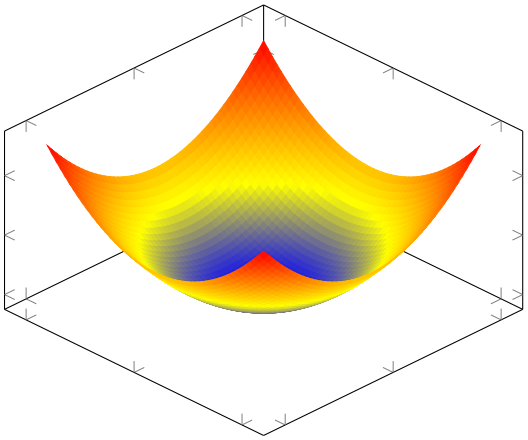


Figure D.5: A 2D graphical representation of the sphere function.

- Expression:

$$f(x_1, \cdots, x_d) = \sum_{i=1}^{d} x_i^2 \tag{D.5}$$

- Number of dimensions used: 30
- Search domain: $-5.12 \leq x, y \leq 5.12$
- Root at: $(0, \cdots, 0)$

Appendix E

Swarm Architectures

E.1 All-Connected-To-All

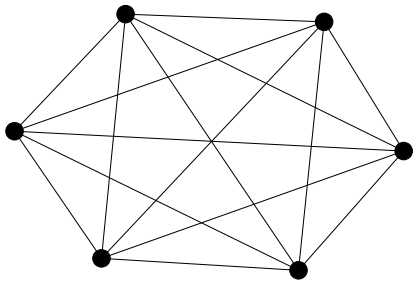
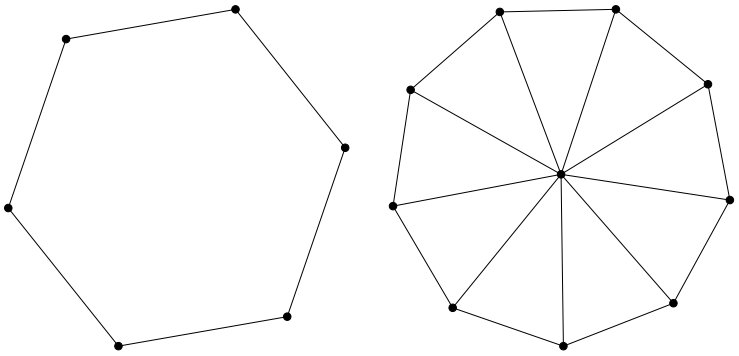


Figure E.1: An example of a graphical representation of an all-connected-to-all communication structure, also known as gbest model, with six particles.

E.2 Ring and Pyramid



(a) An example of a graphical representation of a ring communication structure, with six particles.

(b) An example of a graphical representation of a pyramid communication structure, with ten particles.

Figure E.2: An example of the mesh and toroid communication structures.

E.3 Random

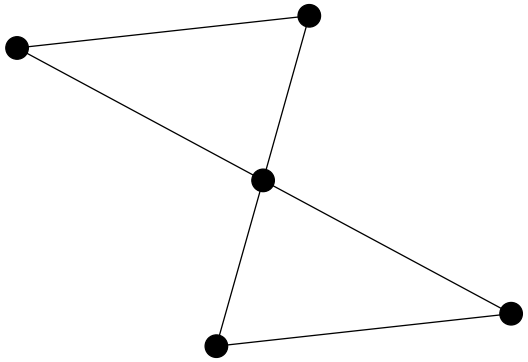
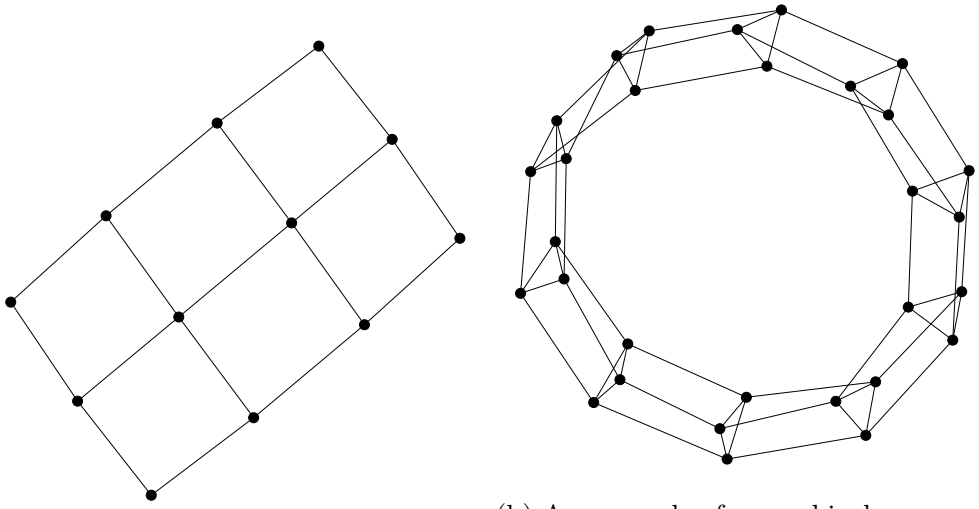


Figure E.3: An example of a graphical representation of a random communication structure, with five particles.

E.4 Mesh and Toroid



(a) An example of a graphical representation of a mesh communication structure, with twelve particles.

(b) An example of a graphical representation of a toroid communication structure, also known as von Neumann architecture, with 30 particles.

Figure E.4: An example of the mesh and toroid communication structures.

E.5 Star

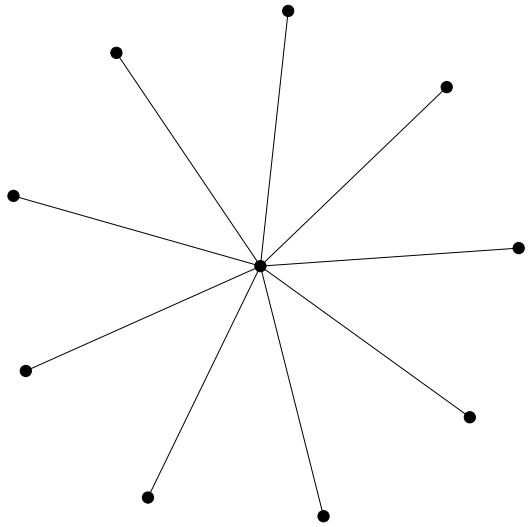


Figure E.5: An example of a graphical representation of a star communication structure, also knows as wheel architecture, with ten particles.

Appendix F

MRF–PSO Algorithm Flowchart

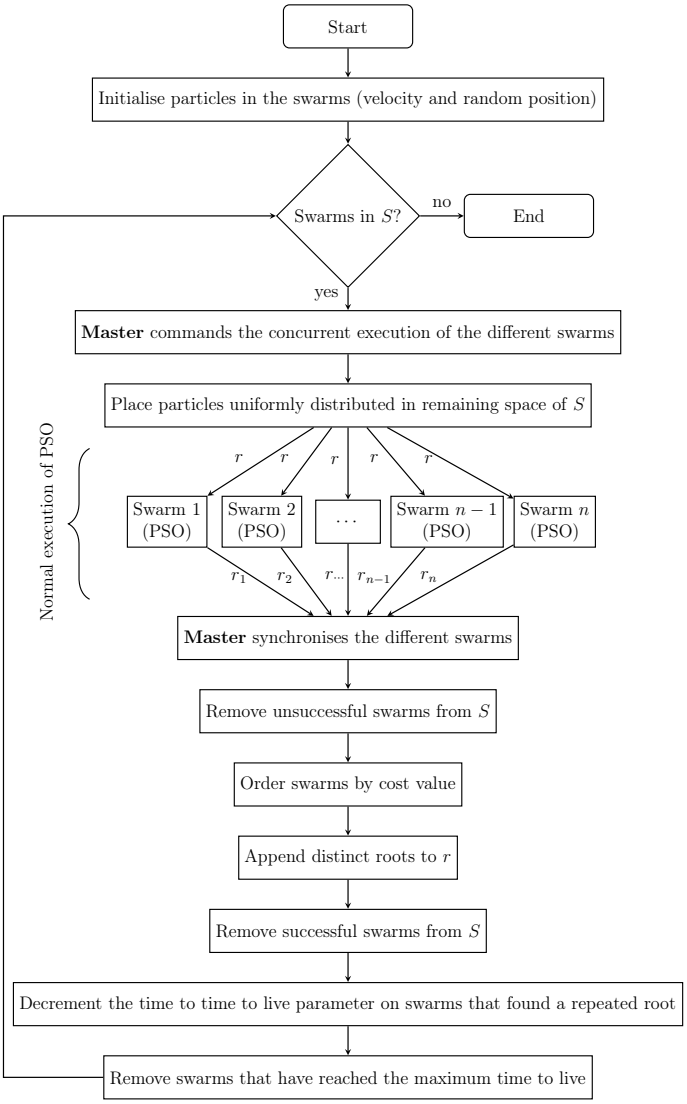


Figure F.1: Flowchart of the MRF-PSO algorithm.

Appendix G

Data Sets

G.1 Input Data Set for Training the ANN 5 for Computing the Real Roots

Table G.1: Head of the input data set for training the ANN 5 for computing only the real roots.

a_0	a_1	a_2	a_3	a_4	a_5
-0.0452	-0.0216	0.6994	-0.7907	-0.7857	1
-0.0075	-0.0660	0.0624	0.9778	1.8083	1
0.0007	0.0045	-0.0694	-0.4979	-0.2036	1
-0.0936	0.1581	0.4911	-0.8276	-0.5898	1
⋮	⋮	⋮	⋮	⋮	⋮

G.2 Output Data Set for Training the ANN 5 for Computing the Real Roots

Table G.2: Head of the output data set for training the ANN 5 for computing only the real roots.

α_1	α_2	α_3	α_4	α_5
0.8984	0.5802	-0.2214	-0.9046	0.4330
0.2375	-0.1197	-0.5886	-0.6316	-0.7060
0.8643	-0.4946	-0.1466	0.0997	-0.1192
0.5233	-0.5453	0.6685	0.6728	-0.7295
⋮	⋮	⋮	⋮	⋮

G.3 Input Data Set for Training the ANN 5 for Computing the Real and Complex Roots

Table G.3: Head of the input data set for training the ANN 5 for computing both real and complex roots.

a_0	a_1	a_2	a_3	a_4	a_5
0.6489	0.5608	0.0764	0.9170	0.6737	0.0856
0.1398	0.8675	0.6737	0.7915	0.6805	0.3244
0.8254	0.8206	0.5198	0.7785	0.8128	0.0229
0.1884	0.4211	0.5824	0.0200	0.9023	0.7577
0.4004	0.5791	0.8399	0.8190	0.9042	0.2391
⋮	⋮	⋮	⋮	⋮	⋮

G.4 Output Data Set for Training the ANN 5 For Computing the Real and Complex Roots

Table G.4: Head of the output data set for training the ANN 5 for computing both real and complex roots.

$\mathbf{Re}(\alpha_1)$	$\mathbf{Im}(\alpha_1)$	$\mathbf{Re}(\alpha_2)$	$\mathbf{Im}(\alpha_2)$	$\mathbf{Re}(\alpha_3)$	$\mathbf{Im}(\alpha_3)$	$\mathbf{Re}(\alpha_4)$	$\mathbf{Im}(\alpha_4)$	$\mathbf{Re}(\alpha_5)$	$\mathbf{Im}(\alpha_5)$
-6.1207	0	-1.8031	0	-0.8277	0	0.4406	-0.7973	0.4406	0.7973
-1.1952	-0.8191	-1.1952	0.8191	-0.1822	0	0.2375	-1.0345	0.2375	1.0345
-34.5703	0	-0.8836	-0.4828	-0.8836	0.4828	0.4004	-0.9324	0.4004	0.9324
-1.4203	0	-0.3379	-0.3584	-0.3379	0.3584	0.4527	-0.7189	0.4527	0.7189
-2.9534	0	-0.6260	-0.5992	-0.6260	0.5992	0.2114	-0.8430	0.2114	0.8430
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Appendix H

Results of the Neural Network-Based Approach for Approximating the Roots of Polynomials Using Particle Swarm Optimisation with the Parameters Found in Chapter 3

Table H.1: Capacity of the networks to generalise the outputs using PSO with the parameters found in Chapter 3. Results are given in terms of MSE.

Degree	**Real roots**	**Complex roots**	**Complex roots (polar coordinates)**
5	0.2090	0.6461	0.8774
10	0.3622	1.8710	1.8418
15	0.7587	2.6139	2.3439
20	0.9031	2.8137	1.9875
25	2.5435	3.5868	3.0355