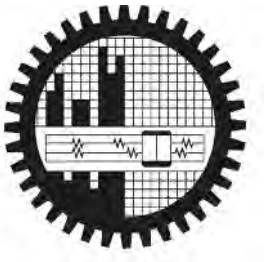
CONTINUOUS OPTIMIZATION WITH EVOLUTIONARY AND SWARM INTELLIGENCE ALGORITHMS

By

MOHAMMAD SHAFIUL ALAM



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY
DHAKA, BANGLADESH

Ph.D. Thesis

Continuous Optimization with Evolutionary and Swarm Intelligence Algorithms

Submitted By

Mohammad Shafiul Alam

Ph.D. Candidate
Department of Computer Science and Engineering
Student Id: 1009054002P

A thesis submitted to the Department of Computer Science and Engineering in partial fulfillment of the requirements for the degree of
Doctor of Philosophy (Ph.D.)

Supervised by

Dr. Md. Monirul Islam

Professor, Department of CSE, BUET

The thesis titled "**Continuous Optimization with Evolutionary and Swarm Intelligence Algorithms**" submitted by Mohammad Shafiul Alam, Roll No. 1009054002 P, Session October 2009, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, is accepted as satisfactory in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Ph.D.). Examination held on September 22, 2013.

BOARD OF EXAMINERS

1. _____
   Dr. Md. Monirul Islam — Chairman
   Professor, Department of CSE, BUET — (Supervisor)

2. _____
   Head, Department of CSE, BUET — Member (Ex-Officio)

3. _____
   Dr. M. Kaykobad — Member
   Professor, Department of CSE, BUET

4. _____
   Dr. Md. Saidur Rahman — Member
   Professor, Department of CSE, BUET

5. _____
   Dr. Mohammad Mahfuzul Islam — Member
   Professor, Department of CSE, BUET

6. _____
   Dr. M. Sohel Rahman — Member
   Professor, Department of CSE, BUET

7. _____
   Dr. Md. Monirul Islam — Member
   Assistant Professor, Department of CSE, BUET

8. _____
   Prof. Dr. Chowdhury Mofizur Rahman — Member (External)
   Pro-Vice Chancellor, United International University
   House #80, Road #8A, Satmasjid Road, Dhanmondi, Dhaka 1209

9. _____
   Dr. Latifur R. Khan — Member (External)
   Professor, Department of Computer Science, University of Texas at Dallas 800 W Campbell Rd, Richardson, TX 75083-0688, USA

DECLARATION

I, hereby, declare that the work presented in this thesis is the outcome of the investigation performed by me under the supervision of Dr. Md. Monirul Islam, Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka. I also declare that no part of this thesis and thereof has been or is being submitted elsewhere for the award of any degree or diploma.

Signature

(Mohammad Shafiul Alam)

To
My Mom, Dad, Grandma,
Shabnam and Nusrat

Acknowledgements

I would like to express my deep gratitude to my supervisor, Dr. Md. Monirul Islam, Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, for his continuous support, advice and care. His patience, guidance, continuous encouragement, constant supervision, constructive criticism, reading many inferior drafts and correcting them at all stages have made it possible to complete this thesis. Without his continuous guidance and support, I would not be able to achieve what I have done in this thesis.

I also wish to express my special thanks to each and every member of my Doctoral Committee for their helpful advice, valuable suggestions and great questions and comments. Their suggestions helped me not only to improve the quality of my thesis, but also to equip me with new knowledge and experience to do my research better along the course of this thesis.

I also owe special thanks to my two external examiners — Prof. Dr. Chowdhury Mofizur Rahman, Pro-Vice Chancellor, United International University, Bangladesh and Prof. Dr. Latifur R. Khan, Department of Computer Science, University of Texas at Dallas, USA, for spending the time and undertaking the pains to examine my thesis and for giving me so many helpful suggestions and comments. I am truly thankful to them for their invaluable support and suggestions, especially during the last phase of my works.

I am also especially grateful to Prof. Dr. Kazuyuki Murase, University of Fukui, Japan, with whom I have had the privilege of working with on a number of research papers and he has co-authored most of my publications. His excellent guidance, insights and support have been invaluable for me during each of my publications as well as during every phase of writing this dissertation.

I would also like to thank all the faculty members and staff of the CSE Department of BUET, for their various support and constant cooperation.

Finally, I express my sincere and deepest gratitude to Almighty Allah, for every moment and every achievement in my life. Then, a million thanks to the members of my family, especially my Mom, Dad and my late Grandma, who undertook so much pain and hardship only to raise and provide for me, since the very moment I have been on this planet. I know I would never be able to return their endless love, care and affection for me. I also miss my younger sister, Nusrat who shares so many similarities with me, but today she is so far from me — thousands of miles away, across Pacific and Atlantic. Nusrat, I miss you every single day. And, my loving wife — Shabnam, who was my oxygen, my life support system and the burning and chilling inspiration for our every moment together. I am also eagerly waiting, soon to see the sweet face of my unborn baby. Shabnam, I love you so much and thank you for such a wonderful gift for me. Without the unconditional love, support and sacrifices of my Dad, Mom, Grandma, Shabnam and Nusrat, it would not be possible for me to complete this long journey. So, I dedicate this thesis to every one of them.

Abstract

Continuous optimization generally refers to the task of finding values for a set of continuous variables that optimizes a given objective function. The problem of continuous optimization appears frequently in every field of science and engineering, and there exist several paradigms of algorithms that deal with the continuous optimization problem. However, in comparison to other analytical, single state and local search based algorithms, the evolutionary and swarm intelligence algorithms show more resilience against local optima and premature convergence, especially when dealing with complex, high dimensional, multimodal problems. This is because both the evolutionary and swarm intelligence algorithms maintain a whole population of candidate solutions that provide diversity and explorative search capacity against locally optimal points. However, some experimental studies also reveal that sometimes the population of candidate solutions may lose its diversity too soon and the entire population may prematurely converge around the locally optimal points. The aim of this thesis is the study and development of novel evolutionary and swarm intelligence algorithms for continuous optimization problems that try to balance between global explorations and local exploitations and to maintain sufficient amount of population diversity to avoid premature convergence. Along the course of this thesis, we have developed two novel evolutionary algorithms and three improved swarm intelligence algorithms, which include the Recurring Two Stage Evolutionary Programming (RTEP), the Diversity Guided Evolutionary Programming (DGEP), the ABC with Self-adaptive Mutation (ABC-SAM), ABC with Improved Explorations (ABC-IX) and ABC with Adaptive Explorations and Exploitations (ABC-AX^2). They employ techniques like dynamic adaptation and self-adaptation (e.g., ABC-SAM and ABC-AX^2), hybridization with other meta-heuristic techniques for more explorations (e.g., ABC-IX), recurring alternations between complementary explorations and exploitations (e.g., RTEP) and automatic control of mutation step size using population diversity information (e.g., DGEP). We have also carried out intensive experimental studies on each of these algorithms to better understand how they work, how their components, control parameters and the proposed techniques affect their performance, final solution quality, convergence speed, population diversity and explorative search capacity. Each of our newly introduced algorithms is tested and evaluated on as many as 55 benchmark problems on continuous optimization from two different benchmark suites. Experimental studies show that the performance of the proposed algorithms is significantly better than many other relevant state-of-the-art evolutionary and swarm intelligence algorithms, which empirically establishes the effectiveness of our proposed techniques for the continuous optimization problems.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Acronyms and Definitions

ABC Artificial Bee Colony Algorithm

ABC-AX² Artificial Bee Colony Algorithm with Adaptive Explorations and Exploitations

ABC-IX Artificial Bee Colony Algorithm with Improved Explorations

ABC-SAM Artificial Bee Colony Algorithm with Self-Adaptive Mutation

ACO Ant Colony Optimization

AIS Artificial Immune System

ALEP Evolutionary Programming with Adaptive Lévy Mutation

BA Bees Algorithm

BCO Bee Colony Optimization

BF Bacterial Foraging

BFO Bacterial Foraging Optimization

BGA Breeder Genetic Algorithm

BSO Bee Swarm Optimization

CABC Cooperative Artificial Bee Colony Algorithm

CEP Classical Evolutionary Programming

ChABC Chaotic Artificial Bee Colony Algorithm

CLPSO Comprehensive Learning Particle Swarm Optimization Algorithm

CMAES Covariance Matrix Adaptation Evolution Strategy

CPSS Cross-Generational Probabilistic Survival Selection

CS Cuckoo Search Algorithm

DABC Artificial Bee Colony Algorithm with Diversity Strategy

DCGA Diversity Control Oriented Genetic Algorithm

DE Differential Evolution

DGA Dual Genetic Algorithm

DGEP Diversity Guided Evolutionary Programming

DGM Diversity Guided Mutation

DMS-PSO Dynamic Multi-Swarm Particle Swarm Optimization Algorithm

DPGA Dual Population Genetic Algorithm

EABC Elitist Artificial Bee Colony Algorithm

ED Euclidean Distance

FA Firefly Algorithm

FE Function Evaluations

FEP Fast Evolutionary Programming

FGPG Fitness Gain Per Generation

EP Evolutionary Programming

ES Evolution Strategy

GA Genetic Algorithm

GABC Gbest Guided Artificial Bee Colony Algorithm

GAUG Genetic Algorithm with Unexpressed Genes

GP Genetic Programming

GSO Glowworm Swarm Optimization

HJABC Hook Jeeves Artificial Bee Colony Algorithm

IFEP Improved Fast Evolutionary Programming

IMGA Island Model Genetic Algorithm

IWD Invasive Weed Optimization Algorithm

LSRCMA Real Coded Memetic Algorithm with Adaptive Local Search

MABC Modified Artificial Bee Colony Algorithm

MCN Maximum Cycle Number

MSO Multi-Swarm Optimization

NFE Number of Function Evaluations

NFE_{max} Maximum Number of Function Evaluations

NSDE Differential Evolution with Neighborhood Search

PBX Position-Based Crossover

PDGA Primal Dual Genetic Algorithm

PS-EA Particle Swarm Inspired Evolutionary Algorithm

PSO Particle Swarm Optimization Algorithm

PSO-RDL Particle Swarm Optimization with Recombination by Dynamic Linkage Discovery

RCMA Real Coded Memetic Algorithm

RCMA-XHC Real Coded Memetic Algorithm with Crossover Hill Climbing

RFD River Formation Dynamics Algorithm

RTEP Recurring Two Stage Evolutionary Programming

RTS Restricted Tournament Search

SA Simulated Annealing

SADE Self-Adaptive Differential Evolution

SD Standard Deviation

SDS Stochastic Diffusion Search

SI Swarm Intelligence

SIA Swarm Intelligence Algorithm

SOA Swarm Based Optimization Algorithm

SPP Self-Propelled Particles Algorithm

SS-ADC Scatter Search with Adaptive Diversity Control

STEP Sequential Two Stage Evolutionary Programming

XHC Crossover Hill Climbing

Chapter 1

Introduction

1.1 Introduction

Evolutionary and swarm intelligence algorithms are a class of meta-heuristic algorithms that take their inspirations directly from nature. Evolutionary algorithms (EAs) are stochastic search methods that use the computational models of natural evolutionary processes to simulate the evolution of a population of candidate solutions in order to find an optimal solution to a problem, particularly an optimization problem. Swarm intelligence algorithms (SIAs) are optimization algorithms that employ a swarm (i.e., population) of decentralized, self-organized agents and model some means of communication and information sharing among them to materialize a co-operative distributed search towards some optimal solution. Since its advent during mid-seventies, the EA has evolved into several major branches, such as genetic algorithm (GA) [1], genetic programming (GP) [2], evolutionary programming (EP) [3] and evolution strategy (ES) [4]. The swarm intelligence algorithms have also appeared in many different forms, such as the ant colony optimization (ACO) [5], particle swarm optimization (PSO) [6], bacterial foraging [7], artificial bee colony (ABC) algorithm [8] and their many variants [9]. During the last few decades, both EAs an SIAs have been extensively employed to solve wide and diverse range of problems, such as continuous optimization (e.g., [10], [11]),

discrete optimization (e.g., [12], [13]), constrained optimization (e.g., [14], [15]), multi-objective optimization (e.g., [16], [17]), design optimization (e.g., [18], [19]), learning decision rules (e.g., [20]), optimizing object recognition model (e.g., [21]), evolving fuzzy rules (e.g., [22]), training neural networks (e.g., [23], [24]), design of digital IIR filter [25], PID controller [26], parameter optimization of milling processes [27] and so on [9].

The scope of this thesis is the study of EAs and SIAs for solving continuous optimization problems. The research area of continuous optimization has been very active and dynamic, especially over the last few decades, because it has numerous applications in widely diverse fields, such as engineering, mathematics, sciences, business and even social sciences. This has led to different kinds of deterministic, stochastic and meta-heuristic algorithms [28] for continuous optimization. Among the stochastic and meta-heuristic approaches, the EAs and SIAs have gradually become very popular within the research community. This is because both these family of algorithms offer several distinct advantages over the gradient based exact or direct search methods [29], such as their global search capability, robustness against local optima, parallelism, ease of implementation and no requirement of a differentiable or continuous objective function. However, current methods still find many difficulties in solving complex, high-dimensional and real-world problems. As a result, the excitement of developing new improvements and designing new heuristics for the EAs and SIAs that can better address the challenging issues of continuous optimization problems does still draws the deep interests of many of today's researchers.

1.2 Continuous Optimization Problem

Many real-world problems can be formulated as optimization problems of the parameters that assume values from the continuous domain, i.e., the continuous optimization problems. A continuous optimization problem can be formalized as follows.

$$\underset{\boldsymbol{x}}{\text{minimize}}\ f(\boldsymbol{x})$$

$$\text{subject to:}\ \ \boldsymbol{x} \in S$$

Here, the goal is to find a vector $\boldsymbol{x}_{min} \in S$ such that $f(\boldsymbol{x}_{min}) \leq f(\boldsymbol{x})$ for all $\boldsymbol{x} \in S$, where the *search space* $S \subseteq R^n$ is a bounded subset of R^n and the *objective function,* $f : S \rightarrow R$ is an n-dimensional real valued function that is to be optimized over its parameter $\boldsymbol{x}$. Each element x_i of the vector $\boldsymbol{x}$ is a real-valued variable: $\boldsymbol{x} = [x_1, x_2, \ldots, x_n]^T \in R^n$.

The task of continuous optimization is generally referred with many different names, such as real-parameter optimization [30], function optimization [31] and numeric optimization [32]. However, all of them actually refer to the general task of finding a solution across a real valued,

(usually) multi-dimensional search space such that the solution gives the best value, i.e., minimum or maximum value, of an objective function, depending on whether it is a minimization or maximization task. This solution should have not only the best objective function value around its local neighborhood, but also the best objective value over all the feasible solutions across the entire search space.

The problem of continuous optimization arises in many different forms — it can be a constrained or unconstrained, single or multi-objective problem. However, studying many different variants of the problem with numerously many variants of EAs and SIAs would be an extremely broad topic to be covered. Therefore, we have limited the scope of our thesis only to the study of single objective, unconstrained continuous global optimization problems using evolutionary and swarm intelligence algorithms. Problems belonging to this subset have the following characteristics — the decision variables assume values from continuous, real-valued domains, there are no constraints on the values that the decision variables can assume (except their explicit boundary constraints) and there is only a single objective function that has to be either minimized or maximized. For many complex real world problems, the search space is multimodal, with the number of optima often being exponential to the number of dimensions. Large number of real world problems, found in many practical applications, exhibits these characteristics. To cite only a few examples, we can include acoustics equipment design [33], aerospace engineering [34], astronomy and astrophysics [35], cancer therapy planning [36], chemical process modeling [37], data analysis [38], fluid mechanics [39], economic and financial forecasting [40], hydro-geology [41], industrial process design [42], laser equipment design [43], model fitting to data (calibration) [44], machine learning [45], molecular biology [46], optimization in numerical mathematics [30], optimal operation of closed engineering or other systems [47], pattern recognition [45], portfolio management [48], potential energy models in computational physics and chemistry [43], the problems of atomic or molecular clusters and crystals [49], process control [42], protein folding [50], robot design and manipulations [51], systems of nonlinear equations and inequalities [52], waste water treatment systems management [53] and many others [9]. However, with the gradually widespread recognition of EAs and SIAs to solve complex, real world continuous optimization problems, they have been increasingly applied on many of these problems, as well as many other new application domains.

1.3 Motivations and Objectives

Both evolutionary and swarm intelligence algorithms maintain a population of candidate solutions that is expected to be improved gradually through successive perturbation and selection operations until they reach sufficiently close to the globally optimum point. But practical experiences [54], [55] often show that the evolving population loses its diversity and explorative capacity too soon and the candidate solutions may prematurely get trapped around the locally optimal points of the fitness landscape. This general problem of premature convergence often originates from the lack of balance between global explorations and local exploitations during the optimization process. There exist a number of research works (e.g., [55], [58]–[61], [69]) that try to achieve a balance between explorations and exploitations and to maintain sufficient amount of population diversity for avoiding premature convergence. However, despite the significant progress made by EAs and SIAs, there are still several important research issues and challenges that have yet to be fully addressed.

The prime motivation of this thesis is to concentrate on designing new and improved evolutionary and swarm intelligence algorithms for continuous optimization that will address the conflicting goals of global explorations and local exploitations, which is often considered a key factor for the good performance of the algorithm, in terms of both convergence speed and the quality of the final solution. Exploration refers to the capability of the algorithm to examine new, unvisited search regions more extensively, which is necessary for robustness of the algorithm against the locally optimal points. In contrast, exploitation refers to concentrating the search on local neighborhoods of the already found solutions, which often leads to increased convergence speed, but possibly towards some locally optimal point. Therefore, balancing explorations with exploitations may lead to both increased convergence speed with improved solution quality, which is the prime concern of most EAs and SIAs. Therefore, the main objective of our study is to introduce novel evolutionary and swarm intelligence based algorithms for the continuous optimization problem that try to balance between global explorations and local exploitations and maintain sufficient population diversity in order to avoid the problems of premature convergence and fitness stagnations. The issues of explorations vs. exploitations and population diversity vs. convergence are often closely related, and addressing these issues is often considered essential for both improved final solution quality and higher convergence speed of an algorithm. Along the course of this thesis, we plan to develop a number of improved evolutionary and swarm intelligence algorithms that would address these closely related issues. More specifically, the aim of this thesis is to accomplish the following objectives and to achieve the following possible outcomes.

a) Most existing works (e.g., [56], [57], [62]–[68]) consider explorations and exploitations to be conflicting objectives, so they attempt to increase either the exploitative or the explorative capacity of the algorithm, either for faster, but possibly premature convergence or for slower, but global convergence avoiding the locally optimal points. However, explorations and exploitations are not always conflicting objectives; rather they can be complementary operations to each other. For example, some exploitation is always necessary after exploring to a new search region in order to realize the potentials of the newly explored solutions. Also, long exploitations can lead to getting trapped around the locally optimal points, so some successive explorative operations might help to break free from any locally optimal point. Based on such complementary properties of exploitations and explorations, we want to develop new evolutionary and swarm intelligence algorithms that would try to balance explorations with exploitations for faster convergence and improved final solution quality.

b) We want to introduce a multi-stage framework for evolutionary and swarm intelligence algorithms, each stage with its own explorative and/or exploitative operators to accomplish its explorative/exploitative objective. The complementary stages may be executed repeatedly, alternating again and again, in order to balance between explorations and exploitations. Such a design is motivated by considering the complementary, rather than conflicting, properties of explorations and exploitations. We intend to investigate and analyze how the repeated alternation between explorations and exploitations, possibly with some adaptive strategy might distribute the conflicting goals of explorations and exploitations across the iterations of an evolutionary and swarm intelligence algorithm.

c) An adequate amount of diversity within the population of candidate solutions is often considered necessary to avoid the problem of premature convergence and to continue the search space explorations [71]. Most existing algorithms (e.g., [70]–[74]) try to promote population diversity by altering their selection and/or perturbation operators. However, none of these algorithms use the population diversity information to control and guide the optimization process. We intend to introduce a novel evolutionary or swarm intelligence algorithm that will not only try to preserve the population diversity, but also try to make use of the existing diversity information to adaptively control the selection and perturbation operators for better optimization. Since the existing population diversity is often a good measure of the maturity of the ongoing optimization process, using the diversity information to adaptively guide the selections and/or perturbation operators may improve the global convergence speed and the final solution quality of the algorithms.

d) Most evolutionary and swarm intelligence algorithms (e.g., [56], [57], [60]–[68]) are designed to be either more explorative or more exploitative. For example, the recently introduced [75] artificial bee colony algorithm is aligned towards more exploitations than

explorations, like many other EAS and SIAs (e.g., [64]–[66], [68]), while there also exist some EAs and SIAs (e.g., [56], [57], [60]–[62]) that are biased towards more explorations. Along the course of this thesis, we plan to design some more explorative variants of ABC and EP-based algorithms by employing a number of techniques, such as recurring alternations, adaptation, self-adaptation and hybridization with other existing algorithms. We also plan to find out whether and by how much these techniques may improve their performance. Study, analysis and experiments with the algorithms we intend to develop may help other researchers design novel and more improved variants of the evolutionary and swarm intelligence algorithms.

e) For the purpose of evaluation and comparison of the proposed evolutionary and swarm intelligence algorithms, we intend to use two different benchmark suites on continuous optimization problems — the standard benchmark suite [55]–[67] and the recently proposed CEC2005 benchmark suite [76] introduced in the special session on real-parameter optimization at the 2005 IEEE Congress on Evolutionary Computation. Both the suites include several continuous problems with different complexity. The standard benchmark suite has 30 different functions, including both low and high dimensional, unimodal and multimodal, separable and non-separable functions. However, the functions in the CEC2005 suite are more challenging, including many shifted, scaled, rotated, expanded and hybrid composite functions. Using both the benchmark suites, the proposed algorithms will be compared with other existing state-of-the-art EAs and SIAs.

1.4 Main Contributions

Based on the above-mentioned motivations and objectives, we have developed a number of evolutionary and swarm intelligence algorithms (chapters 3–8), employing novel techniques that try to balance between global explorations and local exploitations. The major contributions of this thesis can be summarized in the following few points.

a) Based on the complementary characteristics of explorative and exploitative operations during optimization, we have developed, in chapter 3, the Recurring Two-Stage Evolutionary Programming (RTEP) — a novel evolutionary algorithm that try to maximize both the degree of explorations and exploitations by regularly alternating between two different ‘modes’ (i.e., stages) of execution — one explorative and the other one exploitative. Each stage has its own explorative/exploitative operators and objectives. A number of control parameters allow the user to control the degree of explorations and exploitations during the optimization procedure. Evaluation of RTEP on the benchmark problems shows that RTEP performs at least equally well, and often significantly better than many other existing evolutionary algorithms.

b) The existing amount of genetic diversity within the population of candidate solutions can often indicate the maturity of the ongoing optimization process. But how effectively this diversity information can be used to control and guide the evolutionary mutation and selection operations? Our next algorithm — Diversity Guided Evolutionary Programming (DGEP), proposed in chapter 4, is based on the concept of employing population diversity information to control the evolutionary procedure. DGEP introduces Diversity Guided Mutation (DGM) — a novel mutation scheme that tries to ensure both global explorations and local exploitations by controlling the mutation step size using the diversity information of the candidate solutions across the search space. DGEP also makes some diversity preserving measures to maintain an adequate amount of population diversity that assists the proposed DGM mutation scheme. The effectiveness of DGEP and DGM is empirically established through its performance comparison with some other recent and relevant evolutionary and swarm intelligence algorithms.

c) The magnitude of perturbations on a candidate solution affects the degree of explorations and exploitations around it. Adapting the perturbation step size, separately for each candidate solution of the population, can have a positive influence on the performance of the algorithm by adaptively controlling both explorations and exploitations around each candidate solution. Our next algorithm — Artificial Bee Colony algorithm with Self–Adaptive Mutation (ABC-SAM), proposed in chapter 5, tries to control and adapt the mutation step size for the basic ABC algorithm. Experimental studies (chapter 8) on ABC-SAM reveal that the self-adaptation of mutation step size can often improve the performance of the basic ABC algorithm, especially when sufficient degree of explorations is ensured by the proposed self-adaptation scheme.

d) Some of the single state meta-heuristic algorithms [77] (e.g., simulated annealing, tabu search, iterated local search) can provide a control over the degree of explorations around a single candidate solution. Hybridizing such an algorithm with the population based meta-heuristic algorithms (e.g., EAs and SIAs) may provide a better control over the explorative and exploitative characteristics of the algorithms. Based on this idea, we have introduced (chapter 6) the ABC with Improved Explorations (ABC–IX) — a novel ABC-variant that tries to increase the explorative capacity of the basic ABC algorithm by hybridizing ABC with the simulated annealing algorithm. ABC-IX also incorporates a strategy for self-adaptive perturbation rate to control and customize the perturbation rate, separately for each candidate solution in order to produce better offspring solutions from the existing ones. Experimental

results (chapter 8) on ABC-IX show that both the techniques (hybridization and self-adaptation) contribute significantly and synergistically to improve the performance of ABC-IX in comparison to the basic ABC algorithm and several other recent variants of the evolutionary and swarm intelligence algorithms.

e) The basic ABC algorithm is inherently biased towards more exploitations by its very design [78]. Our next algorithm — ABC-AX^2 is an improved variant of the basic ABC algorithm that incorporates three different control parameters, separately within each candidate solution, that try to adaptively control and customize the degree of explorations and exploitations, separately around each candidate solution of the population. ABC-AX^2 employs three adaptive and self-adaptive techniques that gradually adjust the values of the three control parameters for more effective explorations and exploitations. The experimental results (chapter 8) show that all three control parameters, with their adaptive and self-adaptive strategies, affect the performance of the algorithm in a positive and synergistic way. Both the convergence speed and the final solution quality of ABC-AX^2 are often found better than ABC-SAM, ABC-IX and several other state-of-the-art improved variants of the basic ABC algorithm.

f) In chapter 8, we have conducted an in-depth experimental study on each of the algorithms developed along the course of this thesis, analyzing several aspects of the algorithms, such as the role and effects of their control parameters, the influence and contribution of the proposed techniques on the performance of the algorithms, the contribution and synergy (if any) of the proposed improvements, their impact on the convergence speed, final solution quality, population diversity, perturbation success rates and so on. We have also demonstrated that our proposed algorithms might be improved further by incorporating some simple adaptive, self-adaptive and/or hybrid techniques, as exhibited by a newly introduced adaptive variant — the Adaptive RTEP (ada-RTEP) in chapter 8.

g) The performance of each algorithm we have developed is evaluated and tested on as many as 55 benchmark problems on continuous optimization, including 30 standard benchmark functions [55]–[67], as well as 25 recent functions introduced in the special session on real-parameter optimization at the 2005 IEEE Congress on Evolutionary Computation [76]. The benchmark functions include both unimodal and multimodal, separable and non-separable, low-dimensional and high-dimensional, shifted, scaled, rotated, expanded and hybrid composite functions. Very few works have ever been tested and evaluated on such a wide and diverse range of benchmark problems.

1.5 Organization of the Thesis

The rest of this thesis is organized as follows. Part-II (Foundation) includes the chapter 2 which briefly introduces the fundamentals of evolutionary and swarm intelligence algorithms, with their many different variants, models, operators and processes. The essential terms related to EAs and SIAs are explained with figures and examples. Chapter 2 also presents the strengths, limitations and applications of EAs and SIAs, how they are employed to solve the continuous optimization problems and a brief literature survey on how they deal with some challenging research issues, such as avoiding premature convergence and balancing explorations with exploitations. Also, an overview of two different benchmark suites consisting of total 55 different benchmark functions on continuous optimization is also presented in this chapter (section 2.17, Tables 2.3–2.4), which is used extensively throughout the rest of this thesis (i.e., chapters 3 to 8) to evaluate and experiment with each of our proposed algorithms — the RTEP, DGEP, ABC-SAM, ABC-IX and ABC-AX^2.

The part-III (Proposed Algorithms) of our thesis spans the next five chapters (the chapters 3 to 7), each one of which develops a new evolutionary or swarm intelligence algorithm, trying to tweak and improve the degree of explorations and exploitations of an existing EA or SIA, and compares the results with relevant other state-of the-art EAs and SIAs. For example, chapter 3 introduces RTEP — a new EP-based algorithm that tries to balance between global explorations and local exploitations by periodically alternating between complementary explorative and exploitative operations. Chapter 4 presents DGEP — another novel EP-based algorithm that tries to achieve more effective mutations by using the population diversity information to adaptively control the mutation step size. Each of chapters 5, 6 and 7 tries to develop an improved variant of the Artificial Bee Colony (ABC) algorithm. Chapter 5 introduces ABC-SAM that tries to increase the explorative capacity of the basic ABC algorithm by employing a self-adaptive mutation scheme. Chapter 6 introduces ABC-IX, which is another improved ABC-variant that hybridizes ABC with a simulated annealing based probabilistic selection and a self-adaptive perturbation rate in order to control and improve the explorative capacity of ABC. Chapter 7 introduces ABC-AX^2 that controls and customizes the degree of explorations and exploitations, separately for each candidate solution, by incorporating three additional control parameters within each candidate solution and by employing three adaptive and self-adaptive rules that gradually adjust the values of these control parameters.

The next chapter (chapter 8) makes several detailed experiments on each of the five algorithms developed along the course of this thesis, (i.e., RTEP, DGEP, ABC-SAM, ABC-IX, ABC-AX^2) to investigate the role and contribution of the proposed improvements, the synergy and justifications of the techniques introduced, the roles and effects of the algorithm specific

control parameters, how they affect the population diversity, explorative capacity, perturbation success rate and so on. Chapter 8 also exhibits that incorporating some additional techniques of adaptation and self-adaptation can often improve the results of most EAs and SIAs, as demonstrated by a newly developed algorithm variant in this chapter — ada-RTEP.

In the next chapter (chapter 9), we draw conclusions by presenting a summary of the algorithms developed so far, the major contributions of our thesis and suggest a number of possible future research directions for the continuation of our works. Then, the Appendix A presents the two suites of benchmark functions for the continuous optimization problem that have been used all through this document to evaluate and compare each of our proposed algorithms, followed by the Appendix B, which briefly lists the published and submitted papers based on parts of this dissertation. Finally, we have included the complete bibliography for this work at the end of this document.

Chapter 2

Evolutionary and Swarm Intelligence Algorithms

2.1 Introduction

To handle difficult real world problems, scientists and researchers have long been delving into natural processes and beings. It is often found that optimization is at the core of many natural processes, such as Darwinian evolution. Many creatures in nature are also found to be engaged in varieties of distributed optimization tasks through their food foraging and social group behavior, such as swarm of ants and bees, flocks of birds, schools of fishes and so on [79]. During the last few decades, researchers have introduced several models of nature-inspired search and optimization algorithms that have been successfully employed to a wide range of diversely varied problems, ranging from pure theoretical and scientific studies to practical, industrial and commercial applications. Such wide, diverse and growing applicability of the nature-inspired algorithms is a clear indication of their remarkable power and potentials to tackle complex real world problems.

The field of nature and bio-inspired optimization algorithms mainly constitutes two different families of algorithms — the evolutionary computing algorithms and the swarm intelligence based algorithms. Both the families of algorithms share many common features and both are centered on the problems of search and optimization. However, they are not equivalent, and each has its own distinctive features and characteristics. In this chapter, we present the essential terms, notions and concepts related to EAs and SIAs as well as their strengths, limitations, applications and brief literature survey on how they address the challenging research issues, such as avoiding premature convergence, preserving population diversity and balancing explorations with exploitations during the optimization process.

2.2 Organization of the Chapter

The rest of this chapter is organized as follows. Sections 2.3–2.5 explain the biological basis behind EAs, present the basic terms and notions related to EAs and provide a brief overview of a generic, typical EA. Section 2.6 mentions the six main components of EAs and explains each one of them through the subsections 2.6.1–2.6.6. Section 2.7 explains how to solve a continuous optimization problem using an EA, using two different examples. Sections 2.8–2.10 present the overview of SIAs, their key properties and principles. Section 2.11 provides several examples of swarm intelligence (SI) based algorithms. Section 2.12 presents, in details, the Artificial Bee Colony (ABC) algorithms, which is a recently introduced SI based algorithm. Sections 2.13–2.14 explain, in details, the major strengths and limitations of EAs and SIAs. The next two sections — 2.15 and 2.16 present a brief literature survey on EA and ABC about how they deal with the problems of premature convergence, fitness stagnation and the exploration vs. exploitation dilemma. Finally, section 2.17 briefly presents an overview of two different benchmark suites on continuous optimization that we have thoroughly used in this thesis (chapters 3 to 8) to test and evaluate each of our proposed algorithms and to compare them with many other relevant state-of-the-art EAs and SIAs.

2.3 Biological Basis of EA

In this section some biological terms, related with EA, are defined. This will be helpful to have some comprehension of the biological processes upon which EAs are based on.

Every living organism consists of one or more cells. Inside each cell, there is a nucleus, which is known as the central part of the cell. The nucleus of every cell of an organism contains the same set of *chromosomes*. Chromosomes are strings of DNA and serve as a model or 'blue print' of the whole organism. Blocks of DNA within a chromosome are known as *genes*. Each gene encodes a particular *trait*, for example color of eyes or hair. Each gene is located at a particular position in the chromosome. This location is the identity of the gene, and determines the trait to which it is related. The collection of all the genetic materials within all the chromosomes is called *genome*. A specific set of genes in genome is called *genotype*. The genotype is directly related with the organism's *phenotype*, i.e., its physical and mental characteristics, such as hair color, personality, complexion etc. When two organisms mate, their r*eproduction* makes some shuffling of genetic materials of chromosomes from both the parents. A pair of chromosomes exchange genetic information and produce offspring that contain a combination of information from each parent. This is the *recombination* operation, which is often referred to as *crossover*. Random effects are usually involved in the selection of parents and in the process of shuffling of genes among the chromosomes. Usually organisms with higher

fitness get better chance of mating and surviving. EAs usually use some function of the fitness measure to select individuals probabilistically to undergo the mating and reproduction procedure using the genetic operations such as crossover/recombination and mutation.

Evolution requires some amount of diversity to work appropriately. In nature, an important source of diversity is *mutation,* which changes a randomly selected gene in the chromosome by a random amount. In an EA, a large amount of diversity is usually introduced at the start of the algorithm, by randomizing the genes across the population. However, this diversity may fall rapidly during the next generations because both recombination and selection operations are usually diversity decreasing operations. Thus, the importance of mutation, which introduces further diversity while the algorithm is running, cannot be overemphasized. However, some researchers like crossover/recombination as the main search operator and prefer mutation as a background operator to reintroduce some of the original diversity that may have been lost, while others view mutation as playing the dominant role in the evolutionary process. For example, in the EP-based algorithms mutation is the sole evolutionary search operator, with no recombination or crossover operation.

2.4 Basic EA Terminology

Since EA is inspired by the concepts of natural genetics and the Darwinian theory of evolution, they use lots of terms from biology and genetics. Since these terms are prevalent and ubiquitous across the literature on evolutionary computation, we briefly present them in Table 2.1.

Table 2.1: Common terms used in evolutionary computation

Term	Meaning in the context of EA
individual	A candidate solution to the problem at hand
population	A set of *individuals* (i.e., candidate solutions) maintained by an EA
genome	Often used interchangeably with the term *individual*. Actually *genome* is the data structure of an *individual*, which is used during genetic operations, such as *crossover* and *mutation*
genotype	same as the term *genome*
chromosome	same as the *genotype* or *genome*, but represented as a fixed-length vector
parent and child (offspring)	A parent (i.e., existing *individual* or candidate solution) is perturbed by genetic operations (i.e., mutation, crossover/recombination) to produce a new candidate solution (*child* or *offspring*)
gene	A *gene* is a particular slot position in a *chromosome*. In other words, each *chromosome* is a list (sequence) of *gene values*.
allele	a particular setting of a *gene*

Table 2.1 (continued): Common terms used in evolutionary computation

Term	Meaning in the context of EA
phenotype	The *phenotype* of an *individual* is a measurement of how it performs during its *fitness assessment*
fitness	The *fitness* of an *individual* is a measurement of its quality as a candidate solution to the problem at hand
fitness assessment	The task of computing the *fitness* of an *individual*
fitness landscape	quality function; shows the fitness of every possible *individual*
generation	One complete cycle (iteration) of *fitness assessment*, *breeding* and *re-insertion* to form the *population* for the next cycle. The term generation may also refer to the *population* produced in each such cycle
selection	choosing individuals for *breeding*, based on their *fitness*
breeding or mating	The procedure of producing one or more *children* (new candidate solutions) from a pool of *parents* (i.e., existing candidate solutions) through a process of *selection* and genetic operations (*crossover*/*recombination* and/or *mutation*)
recombination or crossover	A form of genetic operation related to sexual *breeding*. A typical *recombination* or *crossover* operation *selects* two *individuals* (parents), combines or swaps their information in some way to produce (usually) two new individuals (offspring).
mutation	A form of genetic operation that selects an *individual* (candidate solution) and randomly perturbs it to form a new *individual*.
re-insertion	Constructing the *population* of the next *generation* from the union of the parent and offspring solutions.

2.5 EA Overview

Evolutionary algorithm is an iterative, stochastic search and optimization technique based on the concepts of the Darwinian theory of evolution. EA maintains a *population* of *individuals* or *chromosomes* (i.e., candidate solutions) that are selected using Darwinian 'laws of natural selection' with 'survival of the fittest' and updated using operators borrowed from natural genetics like *crossover*, *recombination* and *mutation*. EA is an iterative algorithm — it runs generation by generation. In each generation, a typical EA employs selection, crossover/recombination and/or mutation operations to produce a pool of new candidate solutions (i.e., offspring) from the existing solutions (i.e., parents). From the union of the parents and offspring, the selection and/or reinsertion policy tries to keep the better candidate solutions and wipe out the low quality individuals during constituting the next generation population. Crossover and recombination are like 'sexual breeding' operations that produce new candidate solutions by combining information from two (or, more) different existing solutions, while mutation is like an 'asexual' genetic operation that randomly alters 'gene' value(s) of an individual.

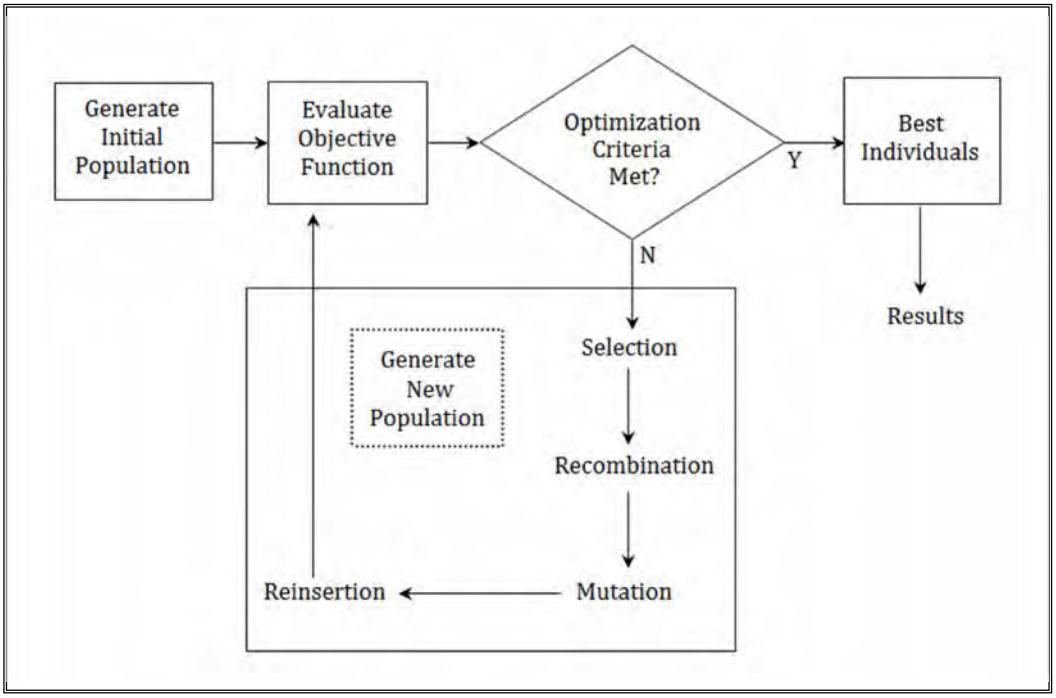


Figure 2.1: Block diagram of a typical EA

Algorithm 2.1: A typical EA

1. *Initialization*. Generate a random population of *n* chromosomes (individuals). Each chromosome is a candidate solution for the problem.
2. *Fitness Evaluation*. Evaluate the fitness $f(\boldsymbol{x})$ of each chromosome $\boldsymbol{x}$ of the population.
3. *Generate new population*. Create a new population by repeating the following steps until the new population is complete.
    - 3.1 *Selection*. Select two parent chromosomes from the population according to their fitness values (the better the fitness, the bigger chance to be selected).
    - 3.2 *Recombination*. Combine the gene values of the two parents to produce new offspring.
    - 3.3 *Mutation*. With a mutation probability, mutate the new offspring at each gene/locus (position in its chromosome).
    - 3.4 *Reinsertion.* Either accept or reject the new offspring in a new population.
4. *Test for termination*. If the stopping criteria are satisfied, then stop and return the best solution in the current generation population. Otherwise, continue.
5. *Loop back*. Go to the step 2 to continue the evolution with the new population of candidate solutions.

Figure 2.2: Pseudocode of a typical EA

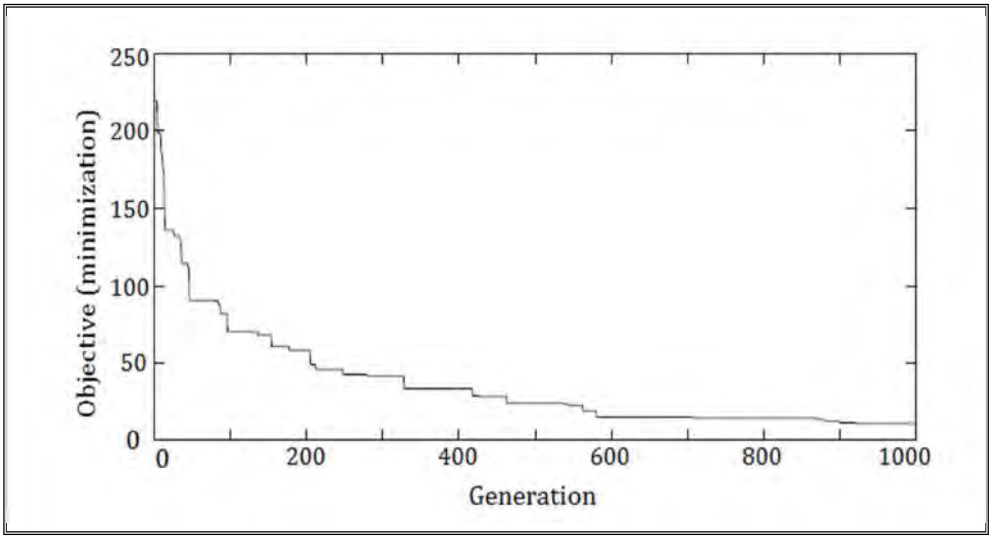


Figure 2.3: Gradual improvement of the objective value during the run of a typical EA. Here, the objective was the minimization of a continuous function.

In each generation of a typical EA, there are four operations — selection, crossover (or, recombination), mutation and re-insertion. Selection picks the individuals that go through the genetic perturbations by crossover (or, recombination) and mutation operations, followed by the re-insertion scheme that selects high quality individuals for the next generation and weeds out the low quality solutions. The selection and re-insertion operations are usually based on fitness and hence responsible for search space exploitations, while the perturbation operations (crossover/recombination and mutation) usually perform the explorations of the search regions. Fig. 2.1 shows the structure of a typical evolutionary algorithm, while Fig. 2.2 presents the pseudocode of a typical EA. The algorithm starts with a population of individuals, each representing a candidate solution of the problem at hand. Each generation of solutions passes through the evolutionary process of selection, crossover/recombination and/or mutation operations to generate new offspring solutions. The next generation is constructed by selecting good quality individuals from the union of the parents and offspring. This process continues again and again until some stopping criteria are met, which may be based on some predefined maximum number of generations, some maximal number of fitness evaluations or reaching some desired fitness or objective function value. During the run, the fitness value of the best found individual usually improves over time, generation by generation. This fitness improvement may stagnate at the intermediate locally optimal points or during the end of the run, as demonstrated in Fig. 2.3. In the ideal case, the fitness stagnation would happen with the successful finding of the global optimum. However, stagnation may also occur around a strong locally optimal point, which can lead to premature convergence of the optimization procedure with insufficient solution quality, which is one of the major problems of EAs (and SIAs, too).

2.6 Components of EA

EAs have a number of component procedures and operators that must be described to have a proper comprehension of the algorithm. When solving a particular problem using EA, each of these components is required to be specified precisely in order to describe the EA. The essential components of an EA are as follows.

- Representation (Encoding)
- Fitness (or, evaluation) function
- Selection
- Recombination/Crossover
- Mutation
- Reinsertion

2.6.1 Representation (Encoding)

Representation means to represent or encode a candidate solution of the actual problem as a data object inside a computer program, suitable to be manipulated by the simulated evolutionary computation. A data object, representing a potential solution of the actual problem, is called a 'chromosome', 'genotype', 'genome' or 'individual' interchangeably, while the actual physical solution in the problem domain is referred as 'phenotype'. A candidate solution may be encoded as a collection (e.g., vector, list) of attributes (parameters or search variables) within a chromosome or individual. Each attribute inside a chromosome is called a 'gene'. An appropriate encoding for the parameters in the genotype depends on the problem at hand. However, encoding with binary bit string is very common, because almost all problem parameters can be encoded using binary representations. The genetic operations often depend on the encoding of the chromosome, so special encoding may necessitate designing specialized mutation and crossover/recombination operators. Although there exist many EA variants with many different encodings and operators, the two most commonly used encodings are for numeric search domain and permutation domain. Most of the problems handled by EAs appear from these two domains. Numeric domains cover all the problems where the goal is to find a numerical vector (i.e., real-valued vector) that optimizes an objective function value. Most of the applications of EA come from this domain, so there already exist a lot of works addressing the issues of encodings and operators for numeric domains. The two key encodings for numeric domain are the *binary string* (*bit string*) *encoding* and the *value encoding*.

In *binary string encoding*, every chromosome is a string of bits, 0 or 1. For example, a chromosome **A** may be represented as: **1101010100100101111**. However, in many problems of the numeric domain, *value encoding* is used, which directly encodes the chromosome as a

sequence of its attribute values. For example, if each attribute is a real number, the chromosome is represented as a string (or, vector) of real values. Values can be anything connected to the problem, from numbers or characters to some complicated structures or objects. Some examples of value encoding are shown below.

Chromosome ***A***	white, gray, gray, black, brown
Chromosome ***B***	b a c d c c a b g a c a d b d d c c b
Chromosome ***C***	0.5498 1.5329 2.1092 9.2143 0.2241

Another form of representation is the *permutation encoding*, which is mostly used in ordering (or, permutation) problems, such as the traveling salesman problem or task ordering problem. In permutation encoding, every chromosome is a string of numbers, each number representing a position in a sequence. For example, a chromosome ***D*** may have the permutation encoding: 7 2 4 1 6 5 9 3 8, where the original *sequence* has 9 different members. For the traveling salesman problem, the above chromosome ***D*** may represent the order at which the cities are travelled, such as the city #7 been travelled first, followed by the city #2, then city #4, and so on. Another special kind of representation, which is used with genetic programming [2], is the *tree encoding*. In tree encoding, each chromosome represents a computer program or expression that is evolved through the evolutionary process. In this scheme, every chromosome is a tree of some objects, such as instructions, or procedures of a programming language.

2.6.2 Fitness (Evaluation) Function

The fitness function (or, evaluation function) performs the role to define how 'well' each individual chromosome is carrying out as a possible solution of the actual problem. Actually, it is a particular type of objective function, but is called 'fitness function' because it assigns a level of fitness to each individual. This fitness measure summarizes how close is an individual to achieve the objective of the given problem at hand. The selection and reinsertion phases usually depend significantly on the fitness values of the individuals assigned by the fitness function. As an example, suppose, we want to find the value of x within the domain of 8-bit integers so that the value of the objective function $f(x)=x^2$ is maximized. Here, the phenotype search space contains all possible integers within the range. The evolution will start with a limited number of chromosomes, sampled over the range of 8-bit integers. If we use a binary encoding, then a chromosome 00010100 will represent a phenotype integer 20. To measure the fitness of the chromosome, the fitness function may simply compute the square of the corresponding phenotype: $20^2 = 400$. The more the value of the square, the better the chromosome is. Thus, the evaluation function builds a bridge from the genotype space of the simulated evolution towards

the phenotype space of the actual problem domain. The direction of the evolution depends entirely on how the fitness function interprets the fitness and evaluates the chromosomes.

Designing a workable fitness function may not be straightforward. Even though it is no longer the human designer, but the computer, that comes up with the final design, it is the human designer who has to design the fitness function. If the fitness function is not designed appropriately, the algorithm may converge to undesired solutions or even fail to converge at all. Besides, the fitness function should be designed such a way that it not only closely expresses the goal of the human designer, but also can be computed quickly with small computational expense. The execution speed of an EA or SIA is directly dependent on how efficiently the fitness function can be computed. In some instances, *fitness approximation* is allowed instead of exact fitness computation. This reason might be an extremely high computational need of fitness evaluation, or the lack of an appropriate and known model of the fitness function. In some cases, it may become completely impossible to even guess the definition of the fitness function (e.g., the aesthetics of a fashion product or the taste of a coffee). The interactive genetic algorithm is a family of EA that handle this difficulty by employing external agents (e.g., humans, polls, online forums) for fitness evaluations [80].

2.6.3 Selection

Inspired by the 'laws of natural selection' and 'survival of the fittest' in the Darwinian theory of evolution, a typical EA selects a number of individuals (parents) from the population to constitute a mating pool, where they go through the genetic procedures of crossover/recombination with each other to reproduce a number of new individuals (offspring). Selection is usually based on fitness to provide the fitter individuals with better chance of mating, reproduction, and survival in order to simulate the Darwinian evolutionary principle of survival of the fittest.

There are a number of standard techniques to make the selection of individuals (parents). For example, the roulette wheel selection, rank based selection, tournament selection, steady state selection and scaling selection. In *Roulette Wheel Selection,* parents are selected in proportion to their fitness. The better an individual, the more likely it is to be selected. Consider a roulette wheel where all the individuals are placed. The slice of every individual is as large as (or, proportional to) the fitness of the individual, assigned by the fitness function. The wheel is rotated at a random pace and a marble is thrown into it to select a chromosome. Chromosomes with higher fitness and occupying more area on the roulette wheel will have better chance to be selected. If we need N individuals, the marble is thrown N times; each time it returns an individual.

When the best individual of the population is exceptionally better than the other individuals, the *roulette wheel selection* will cause problems by selecting the best individual several times, and filling the populations with its multiple copies. For example, if the best chromosome's fitness covers 70% of the area of the roulette wheel, then the other chromosomes will have few chances to be selected. *Rank based selection* eliminates such problem. Rank based selection first ranks all the individuals of the population based on their objective function value. Then every chromosome receives fitness from its ranking, not from their actual objective values. Selection is made from these rank based fitness values.

In *tournament selection,* a number (say, T) of individuals is chosen uniformly at random from the population and the individual with the best fitness of this group is selected as a parent. This process is repeated as many times as individuals are needed as parents. The parameter T takes values ranging from 2 to N, where N is the number of individuals in the population. Large values of T (e.g., $T=10$) intensifies the degree of exploitations because it increases the likelihood of picking from the very fit individuals during each selection, while smaller values of T (e.g., $T=2$) makes the algorithm more explorative, because it picks both high and low quality individuals with significant probability, allowing some diversity to persist throughout the entire course of the evolution, which may be necessary to avoid premature convergence for some problems.

Another commonly used selection scheme is the *Scaling Selection*. As the average fitness of the population increases gradually, the scaling selection scheme ensures that the strength of the selective pressure also increases and the fitness function gradually becomes more and more discriminating with the ongoing generations. This method can be helpful in making the best selection later on when all individuals have relatively high fitness values and only small differences in their fitness values distinguish them one from another.

Another selection scheme — *Steady State Selection* ensures that most of the chromosomes of the current generation survive to the next generation. Such a selection scheme usually exhibits steady improvement of fitness values, and avoids wild oscillations in the average fitness values. In each generation, some good (or, best) chromosomes are selected for mating. The offspring replace some bad (or, worst) chromosomes from the population. The rest of the population survives to the new generation. The opposite of Steady State Selection is the *Generational selection* — the offspring of the parent individuals selected from i-th generation become the entire next (i.e., $i+1$-th) generation. No individuals are retained between the generations.

A term closely connected with selection is *Elitism.* Elitism is a method which safeguards the best part of the population. During each generation, the best chromosome or a pre-defined number (say, k) of best chromosomes are copied to the new population. Then any selection scheme completes the rest of the selection. The parameter k is called the *elite size*. Elitism has shown good performance with a number of problems [68], [162], because it protects the best solutions. However, both steady state selection and too strong elitism (i.e., large value of the elite size k) can turn an EA into too much exploitative by filling the population only by the best individual and its offspring solutions.

2.6.4 Recombination/Crossover

Recombination is the process of generating new individuals (i.e., offspring candidate solutions) by combining the information of two or more existing individuals (parent candidate solutions). Each individual contains a number of attributes (i.e., parameters or search variables). Recombination is done by combining the attribute values of the parents. There are several ways of recombination. The representation (encoding) of the parents play an important role in determining the method of recombination to be applied on the parents.

For individuals with real valued encoding of the attributes, several variants of recombination is defined. For example, *line recombination*, *extended line recombination*, *intermediate recombination*. In intermediate recombination, the attribute variables of the offspring are randomly chosen somewhere around and between the parents' attribute variables. Offspring are produced according to the following rule.

$$Var_i^o = Var_i^{P1} * \sim_i + Var_i^{P2} * (1 - \sim_i); \qquad \sim_i \in [-d,\ 1+d]; \qquad i \in \{1,\ 2,\ \ldots, n\} \qquad (2.1)$$

$\alpha_i \in [-d, 1+d]$ is generated uniform at random; $d = 0.25$; α_i is generated anew for each i.

Here, n is the number of variables in each individual (which is usually same as the dimensionality of the problem), α_i is a scaling factor chosen uniformly at random from the interval $[-d, 1+d]$. If d is set to 0, offspring are always generated at the intermediate region of the parents. Thus, over the generations, the area spanned by the offspring gradually reduces than the area of the parents. From statistical studies, an appropriate value of $d = 0.25$ has been chosen, which ensures that the area spanned by the offspring does not shrink by successive recombination operations over the generations. The difference between intermediate recombination and the *line* (or, *extended line*) *recombination* is that the latter always produces the offspring solutions on the line (or, extended line) connecting both the parent solutions, while the former one (i.e., intermediate recombination) produces the offspring at any point inside the hypercube (or, extended hypercube, based on the value of d in eq. (2.1)) surrounding the parents.

When parents have binary encoding, their recombination constitutes a special case, known as *crossover*. During crossover, a random bit position, k is selected from the range [1…n]. Then, the pair of mating parent exchanges all their bits starting from the k-th position. The random bit position k is called the *crossover point* and selected anew for each crossover operation. Depending on the number of crossover points, there exist *single-point*, *two-point* and *multi-point crossover*. An example of single point crossover is shown in the following Fig. 2.4.

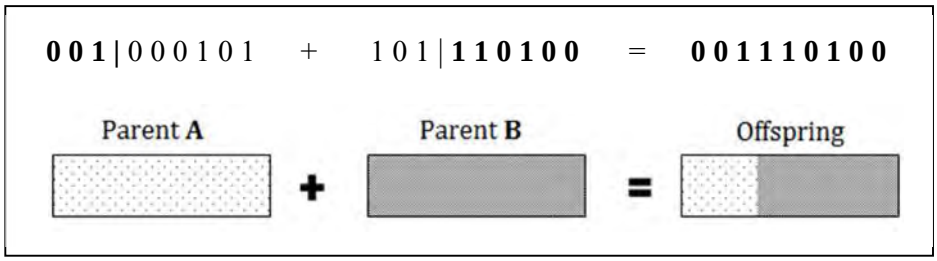


Figure 2.4: Single-point crossover

For two-point crossover, two crossover points are selected at random, and binary string from the beginning of chromosome to the first crossover point is copied from one parent, the part from the first to the second crossover point is copied from the second parent and the rest is copied from the first parent. This is illustrated with an example in the following Fig. 2.5.

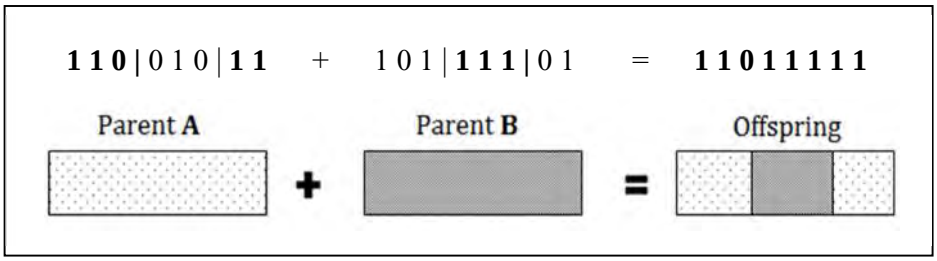


Figure 2.5: Two-point crossover

Another form of crossover is *uniform crossover* in which bits are randomly copied from the first or from the second parent, based on a crossover probability p_c. The following Fig. 2.6 shows this kind of crossover using an example.

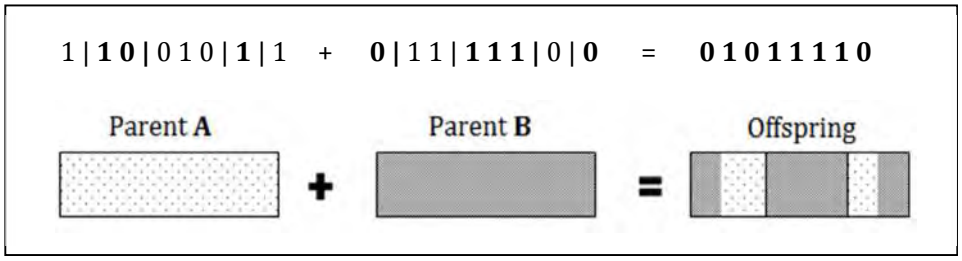


Figure 2.6: Uniform crossover

2.6.5 Mutation

Mutation means randomly altering the chromosomes. In EP-based algorithms, mutation is considered the sole genetic operator, while other EA families employ both the recombination and mutation operations. Two parameters are associated with mutation — *mutation step* and *mutation rate*. Mutation step controls the amount of variation incurred by the mutation. Mutation rate controls the probability of an attribute (gene) within a chromosome being mutated. Usually mutations are applied with small mutation steps and low mutation rate. Offspring are probabilistically mutated after being created by the recombination step. Like recombination, there exist several standard ways of mutation. The encoding of the chromosome determines the possible means of mutation that may be applied on it. If the chromosome has *value encoding*, with real values for the attributes, then mutation means adding randomly created real values with the attributes. For binary encoding, mutation means the flipping of the bits, since every bit has to be either 0 or 1.

The mutation rate usually depends on the problem at hand. For many problems, the mutation rate is made inversely proportional to the number of variables. The more attributes (genes or search parameters) an individual has, the smaller is set the mutation probability. There exist a number of research works [81]–[83] that try to find the optimal mutation rate and step size for a problem. However, a mutation rate of $1/n$ (n: number of genes within a chromosome) has been reported to exhibit satisfactory results for a wide range of problems. With this mutation rate, only one variable per individual is altered. Thus, the mutation rate is become independent of the population size. Another strategy suggested in [83] is to set higher mutation rates at the beginning of the evolution, and declining this rate with the increasing generations, which has shown an acceleration of the search process for many problems.

The optimal size for the mutation step is usually difficult to realize. It always depends on the problem at hand and often varies during the ongoing optimization process. Small mutation steps are usually successful, while larger steps, when successful, produce good results much quicker. For this reason, a good mutation operator should produce both small and large step-sizes in suitable proportions. Such a mutation operator, as proposed and employed in [81], [84] is specified the following eq. (2.2).

$$Var_i^{Mut} = Var_i + s_i * r_i * a_i \tag{2.2}$$

$i \in \{1, 2, \ldots, n\}$ uniform at random

$s_i \in \{-1, +1\}$ uniform at random

$r_i = r.\ domain_i$; r : mutation range (standard: 10%)

$a_i = 2^{-u.\,k}$; $u \in [0, 1]$ uniform at random, k: mutation precision.

This mutation is able to generate most points in the hypercube defined by the domain of the attributes (search variables) of the individuals. Most mutated individuals tend to be near the parent individual. Only some mutated individuals will be far away from the parent. Thus, the probability of small step-sizes is greater than larger steps, which is proper for most problems.

For binary valued individuals, mutation means the flipping of the attribute (i.e., gene) values, because every gene can have either of only two states — 0 and 1. Thus, the size of the mutation step is always 1. For every individual, the gene value to be mutated is chosen mostly uniform at random. The following example in Fig. 2.7 shows how a binary mutation alters a chromosome with 10 bits. Here, mutation randomly flips the bit at position 6 from 0 to 1.

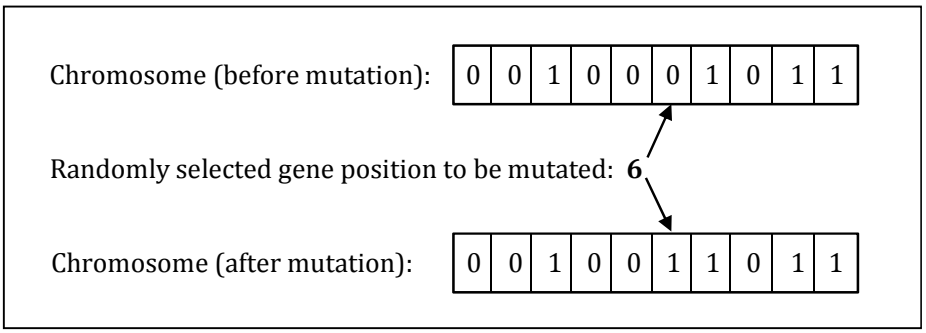


Figure 2.7: Bit-flip mutation

In order to mutate real variables, it is possible to adapt the direction and step-size to conduct a more effective search process. These methods are from evolutionary strategies [4] and also from evolutionary programming [3]. The extensions of these methods and new developments include several new schemes, such as — adaptation of n (number of search variables) step-sizes but no direction [88] [89], adaptation of n step-sizes and only one direction [89] and adaptation of n step-sizes and n directions [90].

2.6.6 Reinsertion

Reinsertion is the process of constructing the next generation population from the union of the parents and offspring. There exist several schemes, each with its own merits and demerits. In *pure reinsertion* [3], the number of offspring reproduced is equal to the number of parents and the offspring population replaces the parent population entirely. Thus, each individual, even the best one, lives for only a single generation. In *uniform reinsertion* [3], fewer offspring are produced than parents and offspring replace parents uniformly at random. As a result, better individuals may be replaced by weaker offspring. In *elitist reinsertion* [3], the worst individuals of the current population are replaced by the new offspring, which means the elitist reinsertion can become very exploitative. Another scheme, *fitness based reinsertion* [3] produces more offspring than needed, and they are inserted into the population based on their fitness. A

number of reinsertion schemes may be combined to construct a new scheme. For example, the *elitist* scheme combined with *fitness-based reinsertion* prevents the best individuals from being lost, and it is recommended for many problems. In this scheme, a given number of the least fit parents are replaced by the same number of the fittest offspring. Some EAs [91] consider not only the fitness value, but also the diversity of a candidate solution to allow it to the next generation. A more diverse individual promotes more search space explorations and helps the algorithm avoid premature convergence to any locally optimal point.

In addition to all the global reinsertion schemes described above, reinsertion may also be based on local policies. In local reinsertion, individuals are considered within its bounded neighborhood only. The reinsertion of an offspring usually takes place in exactly the same neighborhood from where it is selected. This preserves the locality of the genetic information. Examples of some local reinsertion policies are as follows.

- Insert every offspring and replace weakest individuals in the neighborhood.
- Insert offspring fitter than weakest individual in the neighborhood and replace the weakest individuals in the neighborhood.
- Insert offspring fitter than the parent and replace the parent.
- Insert every offspring and replace individuals in the neighborhood randomly.
- Insert offspring fitter than weakest individual in the neighborhood replacing parent.
- Insert offspring fitter than the weakest individual in the neighborhood and replace individuals in the neighborhood uniformly at random.

During the reinsertion step, a number of better parent solutions may be replaced by some worse offspring. However, this does not cause trouble for most problems, since if the inserted offspring are extremely bad, they are very likely to be replaced with new, better offspring solutions in the next generation.

2.7 Continuous Optimization with EA

An EA always works with a set of *individuals* (chromosomes), representing candidate solutions of the problem. EA also needs an *evaluation function* (or, *fitness function*) that can assign a fitness value (i.e., a quality measure) to every individual of the population. EA moves from generation to generation, performing *selection* and *reproduction* of the individuals. The *selection* operation (described briefly in section 2.6.3) selects some individuals (parents) for *reproduction* by using the *recombination* and *mutation* operations to produce a number of new candidate solutions (offspring). There exist several *recombination* and *mutation* operations in the literature, some of which are briefly described in the earlier sections 2.6.4 and 2.6.5, respectively. From the union of parents and offspring, the *reinsertion* operator (section 2.6.6)

selects some individuals to constitute the next generation population. To employ an EA on a continuous optimization problem, we have to define at least the following five fundamental components of EA — the chromosome representation, the fitness (or, evaluation) function, the selection procedure, the reproduction operators (mutation and crossover/recombination) and the reinsertion policy, each of which is briefly described in the following paragraphs.

Representation (encoding): For the problem of continuous optimization, it is common to use either *binary string representation* or the *real-valued vector representation* (section 2.6.1). However, for numerical function optimization, fixed-length real-valued vector representation is more common. Here, each individual (chromosome) $\boldsymbol{x}$ is represented as a fixed-length vector of floating point numbers, i.e., $\boldsymbol{x}=[x_1, x_2, \ldots, x_D]^T$, where D is the number of search variables (the dimensionality of the problem).

Fitness (Evaluation) function: For a maximization problem, where F is the objective function to be maximized, the fitness of an individual $\boldsymbol{x}$ can be estimated simply by using the value of $F(\boldsymbol{x})$. However, for a minimization problem, the same value might be used, but only after inversion or negation, i.e., $fitness(\boldsymbol{x})=1.0/F(\boldsymbol{x})$ or $-F(\boldsymbol{x})$ might be used. In case $F(\boldsymbol{x})$ can take both positive and negative values, we estimate the fitness of $\boldsymbol{x}$ for a minimization problem using (2.3).

$$fitness(\boldsymbol{x}) = \begin{cases} \dfrac{1}{1+F(\boldsymbol{x})} & \text{if } F(\boldsymbol{x}) \geq 0 \\ 1+\left|F(\boldsymbol{x})\right| & \text{otherwise} \end{cases} \tag{2.3}$$

Selection: The selection operation is usually based on the fitness values of the individuals, though some other criteria (e.g., diversity) may also be considered, as in [88]. A selection operator has its own particular effect on the exploitative/explorative characteristics of the EA, so it should be carefully chosen, considering the explorative/exploitative requirements of the problem at hand. The Roulette wheel selection, developed by Holland in 1975 [1], is one of the earliest and commonly used selection operator, which can be directly used for the continuous optimization problem. The other selection operators, briefly described in the section 2.6.3, can also be used directly, without any alteration, for the continuous optimization problem.

Recombination/Crossover: The representation (encoding) of the individuals usually directs the choice of the recombination operator. For the *binary string representation*, the classical crossover operator is the simple crossover [89]. The *n-point crossover* [90] and the *uniform crossover* [91] may also be used. For *real-valued vector representation*, numerous recombination/crossover operators have been developed, such as the Flat crossover [92], Simple crossover [93], [94], Arithmetical crossover [94], BLX-α crossover [95], Linear crossover [93], Discrete crossover [81], Extended Line crossover [81], Extended Intermediate crossover [81], Wright's Heuristic crossover [96], Linear BGA crossover [97], Fuzzy Connectives based crossover [98] and so on.

Mutation: The choice of an appropriate mutation operator largely depends on the representation (encoding) of the individuals. For *binary string representation*, the simple *bit flip mutation* (section 2.6.5) is often used. For *real-valued vector representation*, several mutation operators have been designed and tested, such as the Random mutation [94], Non-uniform mutation [94], Creep mutations [99], [100], Mühlenbein mutation [81], Discrete Modal mutation [102], Continuous Modal mutation [101] and so on.

Reinsertion: For the continuous optimization problem, any of the local or global reinsertion policies (section 2.6.6) can be used. However, for improved performance, we should select the reinsertion policy only after cautiously considering its explorative/exploitative properties, its synergy with the selection, crossover/recombination and mutation operations, and the explorative/exploitative requirements of the problem at hand.

Now we present two examples of how to employ an EA on the continuous optimization problems. In the first example, we show only one iteration of an EA that tries to maximize the function $f(x) = x^2$. The EA in this example (Table 2.2) uses binary string representation, roulette wheel (i.e., fitness proportional) selection, single point crossover and bit-flip mutation. The control parameters of the EA are set as follows — the crossover probability = 0.50, mutation probability = 0.01 and population size, $n = 4$. Because of the low mutation probability on this small population size, no mutation has been taken place in the current iteration, as shown in Table 2.2. The execution of the single generation is divided into two stages — the selection stage (columns 2–4 in Table 2.2) and the recombination stage (columns 5–6). During the selection procedure, column 2 computes the fitness of each individual $\boldsymbol{x}_i$, column 3 calculates their relative probability to be selected and column 4 shows the number of times each individual is actually selected by the roulette wheel selection scheme. During crossover, the column 5 shows the crossover point by using the '|' symbol. The bottom row shows that all three measures of the population fitness (the worst, average and best fitness values) have improved significantly, even within this single generation. This indicates the excellent capacity of an EA to produce sufficiently good quality solutions within fixed, finite execution time.

Table 2.2: A single iteration of an evolutionary algorithm using binary string representation, roulette wheel selection and single point crossover operation.

Selection Procedure				**Crossover/Recombination**		
Population Member ($\boldsymbol{x_i}$)	$fitness(\boldsymbol{x_i})$	$\frac{fitness(\boldsymbol{x}_i)}{\sum_{k=1}^{n} fitness(\boldsymbol{x}_k)}$	Copies Selected	Mating Pool	New Population	Fitness (New)
0110	36	0.19	1	01|10	0101	25
0010	4	0.02	0	10|01	1010	100
1001	81	0.43	2	1001	1001	81
1000	64	0.34	1	1000	1000	64
Worst	4					25
Average	46.25					67.50
Best	81					100

Our next example of employing an EA to solve the continuous optimization problem is the Classical Evolutionary Programming (CEP), which belongs to the evolutionary programming family. We have implemented CEP according to [102], which is described in the following steps. We have also compared the results of CEP with most of our own algorithms that are proposed along the chapters 3 to 8. Like other EP-based algorithms, CEP does not use any recombination operator; rather, it uses mutation as its sole genetic variation operator. CEP mutates every individual of the current population to produce a new offspring, then selects the better n individuals from the union of $2n$ parents and offspring by using a tournament based global reinsertion scheme. The following steps are performed in every generation of CEP.

- Generate an initial population of n individuals. Each individual $\boldsymbol{I}$ is represented as a pair of real valued vectors $(\boldsymbol{x}_i, \boldsymbol{\eta}_i)$, for $i = 1, 2, \ldots, n$; Here, $\boldsymbol{x}_i$'s are objective variables and $\boldsymbol{\eta}_i$'s are standard deviations for Gaussian mutations. Each $\boldsymbol{x}_i$ (and $\boldsymbol{\eta}_i$) has D components, where D is the dimensionality of the problem. Each component of $\boldsymbol{x}_i$, for $i = 1, 2, \ldots, n$, is generated uniformly at random within its search domain. All the components of $\boldsymbol{\eta}_i$, for $i = 1, 2, \ldots, n$, are initialized to some moderate value (e.g., 3.0), as is done in [57].
- Calculate fitness value of each individual $(\boldsymbol{x}_i, \boldsymbol{\eta}_i)$ based on objective function value $F(\boldsymbol{x}_i)$.
- **Mutation step:** Mutate each individual $(\boldsymbol{x}_i, \boldsymbol{\eta}_i)$, for $i = 1, 2, \ldots, n$, to create an offspring $(\boldsymbol{x}_i', \boldsymbol{\eta}_i')$ — that is, for $j = 1, 2, \ldots, D$,

$$x_i'(j) = x_i(j) + \eta_i(j) N_j(0,1) \tag{2.4}$$

$$\eta_i'(j) = \eta_i(j) \exp\left(\tau' N_j(0,1) + \tau N_j(0,1)\right) \tag{2.5}$$

  Here, $x_i(j)$, $x_i'(j)$, $\eta_i(j)$, and $\eta_i'(j)$ are the j-th component of the vectors $\boldsymbol{x}_i$, $\boldsymbol{x}_i'$, $\boldsymbol{\eta}_i$ and $\boldsymbol{\eta}_i'$, respectively. $N_j(0,1)$ is a normally distributed one-dimensional random number with mean = 0 and standard deviation = 1. Subscript j in $N_j(0,1)$ indicates the random number is generated anew for each value of j. The factor τ and τ' are set to $\left(\sqrt{2\sqrt{n}}\right)^{-1}$ and $\left(\sqrt{2n}\right)^{-1}$
- Calculate the fitness of each offspring produced by the previous mutation step.
- Conduct a pair wise tournament based competition over the union of parents and offspring. For each individual, q opponents are picked uniformly at random from all the other parents and offspring. If the individual's fitness is not less than its opponent in the pair wise competition, it receives a 'win'.
- Select the n individuals from the union of parents and offspring that have received the highest number of 'win's. They become the parents for the next generation.
- Stop if some stopping criterion (e.g., predefined maximum number of generations) is fulfilled. Otherwise, go back to the mutation step to start another CEP iteration.

2.8 Swarm Intelligence Based Algorithms

Swarm of insects that show social behavior, such as ants, bees, wasps and termites, have fascinated scientists and researchers as well as naturalists and poets for hundreds of years. Each member of a swarm, e.g., an ant or a bee, seems to perform very simple and basic operations, but together all their activities seamlessly integrate in an unsupervised and distributed way, without any central control, and yet lead to extraordinary results, such as the construction of an amazing ant colony, bee hive, termite nest or finding a set of optimal routes to good quality food sources over large distances and across lots of obstacles from their nests. Swarm intelligence is the branch of science that studies such emergent collective intelligence from the groups of decentralized, self-organized simple agents, in both natural and artificial systems. In particular, swarm intelligence based algorithms focus on the communal behavior and collective intelligence that may result from the local interactions and self-organizations of the individuals with each other and with their environment. The motivation of SI based algorithms lies in numerous natural systems, like the colonies of ants and termites, hives of bees, schools of fish, flocks of birds and herds of land animals. In the following sections, we briefly explain the properties and principles of SI, followed by a number of examples of SI based algorithms, their notable strengths and limitations, a detailed overview of the Artificial bee colony (ABC) algorithm, which is a recently introduced [75] SI based algorithm and a brief literature survey on a few existing improved variants of the ABC algorithm.

2.9 Properties of Swarm Intelligence Algorithms

A typical swarm intelligence based system is a multi-agent system that shows self-organized behavior based on the local interactions of the agents among themselves and with the surrounding environment. The self-organization behavior, usually combined with some significant degree random fluctuations, often leads to the emergence of globally intelligent behavior, in a distributed style, without the need of any central coordination. A typical swarm intelligence based system can be characterized by the following distinct properties.

- A swarm is composed of many agents (individuals).
- The individuals are usually homogeneous or nearly homogeneous in nature. They are either identical or belong to only a few closely related typologies.
- The individuals interact with each other and with the environment. The interactions are based on only a few simple behavioral rules that exploits some local information that they exchange, either directly or via some medium or environment (e.g., stigmergy).
- The overall behavior of the swarm system is a direct result of the interactions of the self-organized individuals with each other and with their environment. This group

behavior shows the emergence of intelligent behavior from the collaborative, distributed, self-organized behavior of the individual swarm members.

The distinguished property of a swarm intelligence based algorithm is its ability to show collective, coordinated, intelligent group behavior without the presence of any central or external coordinator, which results from the interactions of spatially distributed individuals that exchange important local information among themselves to organize their behavior based on some simple, basic rules. Also, some randomness and stochastic behavior should also be present in each individual of the swarm, which may depend on the information it receives from the neighboring individuals and also from the surrounding environment. These properties make it possible to design swarm intelligence based algorithms that are significantly scalable, parallel, and fault-tolerant.

- Scalability is possible in a swarm intelligence based system, because the interactions are only among the neighboring individuals, so the total number of interactions don't increase severely with the number of individuals in the swarm.
- Parallelism is possible in swarm intelligence based algorithms, because the swarm individuals can simultaneously perform different actions in different regions of the search space, without waiting for each-other or without the need for any central control.
- Since every individual swarm member is autonomous, decentralized and self-organized by using local information only, they can easily aggregate to form a fault tolerant system. The individuals are homogeneous or nearly homogeneous, so they are easily interchangeable — a failed individual can easily be replaced by another one, thus making a highly fault-tolerant system.

2.10 Principles of Swarm Intelligence Algorithms

In swarm based algorithms, the local interactions among simple self-organized agents lead to the development of collective intelligence at the global level. It has been showed [103] that such self-organization is the key feature of a swarm system, from which results a global (macroscopic) level intelligent response from the local (microscopic) level interactions. In [103], the self-organization in a swarm system is characterized through four characteristics.

(i) Positive feedback from the agents — It is often a simple behavioral rule that helps to create a collective and convenient behavioral structure. For example, recruitment and reinforcement of the insects, such as depositing pheromones along the trails by the ants, following nearby ants or their pheromone trails, dancing of bees after returning to hive after they find some good quality food source (i.e., nectar) — all these are the examples of positive feedback.

(ii) Negative feedback from the failures of the agents — This counterbalances the positive feedback, which is necessary to stabilize the collective search pattern. Examples of negative feedback in natural swarms include the evaporation of pheromones from longer, unattractive ant trails, the abandonment of a food source by the bees when its nectar amount exhausts, or avoiding food sources that gets too much crowded from the competition of many other foragers. In such situations, a negative feedback helps to stabilize the system.

(iii) Some random fluctuations that act as a source of creativity and innovation and helps avoid the local minima. Some examples of random fluctuations include random walks by ants, random task switching among the foraging ants, random explorations by scout bees, some random low-rate errors made by each swarm member and so on. Randomness is always important to break free from any locally optimal point and it also promotes the discovery of new, innovative solutions.

(iv) Multiple interactions among the swarm individuals — This usually occurs automatically from the interaction among agents and their positive and negative feedbacks, because the agents in the swarm use the information from each-other to self-organize their own behavior. Gradually, the information, data and their impact spread throughout the swarm.

For the emergence of swarm intelligence, the following five requirements have been identified in [104] that are to be satisfied by the swarm behavior.

(i) The proximity principle — The swarm individuals should be able to do simple, basic space and time computations.

(ii) The quality principle — The swarm should be able to respond to quality factors in the environment such as the quality of food sources or the safety of a location.

(iii) The principle of diverse response — The swarm should not allocate all of its resources along excessively narrow channels and it should distribute resources into many nodes.

(iv) The principle of stability — The swarm should not change its mode of behavior upon every fluctuation of the environment.

(v) The principle of adaptability — The swarm must be able to change behavior mode when the investment in energy is worth the computational price.

In some recent works (e.g., [105], [106]), it is emphasized that an intelligent division of labor and performing specialized tasks simultaneously by specialized agents are also important for the emergence of swarm intelligence. The division of labor is also seen in nature in many social insects, such as the different species of leaf-cutter ants performing different tasks or the different groups of bees (e.g., the employed, onlooker and scout bees in the ABC algorithm [75]) performing differentiated foraging tasks.

2.11 Examples of Swarm Intelligence Algorithms

Over the last few years, the field of swarm intelligence has been very active, producing several algorithms that mimic the intelligent swarm behaviors found in the natural world. From these algorithms, we briefly present only a few in the following paragraphs. A more detailed survey on this topic can be found in [79].

Ant Colony Optimization: Ant Colony Optimization (ACO) [5] is based on the intelligent ant swarm behavior to find an (or, a set of) optimal route(s) from their nest to some distant food source(s). ACO-based algorithms are suitable for the optimization problems that need to find routes from a source (or, initial state) to a destination (or, goal state). In ACO, artificially simulated ant agents search across a parameter space that represents the search space of all possible solutions to find some optimal solution. To do so, the artificial ants put pheromones, like natural ants, along their way from their nest to the resources. Their search behavior is also randomly affected by the pheromones laid down by other ants. Better trails have smaller lengths, so they require less time to travel and hence have higher density of pheromones, which in turn attracts more ants to deposit pheromones along them. Gradually an optimal or near-optimal trail is found by such distributed, self-organized behavior and interactions among the simulated ants.

Artificial Bee Colony algorithm: The Artificial Bee Colony (ABC) algorithm is a recently introduced [75] population based meta-heuristic algorithm that mimics the intelligent food foraging behavior of the honey bees found in nature. There exist three categories of honey bees — the employed, onlooker and scout bees. The ABC algorithm uses the same three groups of bee agents for differentiated degree of explorations and exploitations of the search space. The employed and onlooker bees exploit the already found food positions (i.e., candidate solution), while the scout bees randomly explore the search space for newer food sources. Based on the interaction among the onlooker and employed bees (recruitment), self-organization of each bee agent and an intelligent division of work by the different groups of bees, the ABC algorithm shows very good performance on a wide range of optimization problems, including both discrete (e.g., [13]) and continuous (e.g., [24]) problems.

Artificial Immune Systems algorithms: The Artificial Immune Systems (AIS) algorithms are a class of swarm intelligence algorithms that are based on the principles and processes found in the immune systems of the vertebrate and mammalian species. AIS mimics some of the characteristics of the immune system, such as learning and memory. Some examples of AIS based algorithms include the Clonal Selection algorithms [107], Negative Selection algorithms [108], Immune Network algorithms [109] and Dendritic Cell algorithms [110].

Bacterial Foraging Optimization: The Bacterial Foraging Optimization (BFO) [7] is a novel swarm intelligence based algorithm that is inspired by the food-seeking and reproductive behavior of common bacteria such as E. coli. A common bacterium can perform a number of operations, such as swim towards nutrient, tumble to change directions and reproduce into two identical bacteria. The action that a bacterium takes depends probabilistically on its surrounding environment and its neighboring bacteria. The BFO is a probabilistic and population based metaheuristic that model such bacteria behavior across the parameter space to solve many search and optimization problems [111].

Bat Algorithm: The Bat Algorithm (BA) [112] has been motivated by the intelligent echolocation behavior of certain species of bats. BA employs a strategy that mimics the frequency-tuning based control of the loudness and rate of emission of sound wave pulses by bats in order to balance between global explorations and local exploitations of the search space.

Cuckoo Search: The Cuckoo Search (CS) [113] is a recent algorithm based on the brooding behavior of cuckoos. Some cuckoo species use other host birds for hatching their eggs and raising their offspring. In CS, such brooding behavior is hybridized with perturbation steps produced from Lévy distribution. The more explorative Lévy distribution makes the CS more robust than some other swarm intelligence based algorithms, such as PSO and ABC, as demonstrated in [113].

Differential Search Algorithm: The Differential Search Algorithm (DSA) [114] is another recent SI-based algorithm, motivated by the self-organized migration behavior of some super-organisms. The performance of DSA is evaluated on the numerical optimization problems and compared with some other recent evolutionary and swarm intelligence based algorithms, such as ABC, JDE, JADE, SADE, EPSDE, GSA, PSO and CMA-ES [114].

Firefly Algorithm: The Firefly Algorithm (FA) [115] mimics the swarming and re-grouping behavior of fireflies. Each firefly has some flashing capability and the intensity of its light can make it more or less attractive to the other fireflies. Using its light intensity, a firefly can attract other fireflies to form a subgroup or sub-swarm where the other fireflies crowd around the locally best firefly (i.e., the firefly with maximum flash intensity within its neighborhood). This makes FA particularly suitable for the multimodal optimization problems.

Glowworm Swarm Optimization: The Glowworm Swarm Optimization (GSO) [116] is another swarm intelligence algorithm that models the self-organization and swarming behavior of the glowworms. Each agent in GSO moves across a search space and carry a luminescence amount called luciferin. The fitness of the current search space location of each particular glowworm is represented by its luciferin amount. Each glowworm probabilistically selects a subset of its neighboring glowworms. Then it moves towards any of its neighbors that has a higher luciferin

value than itself. Such selections and movements, based only on local information, enable the swarm of glowworms to partition into disjoint subgroups. Each subgroup gradually converges to a locally optimal point. The GSO algorithm is particularly suitable for simultaneous discovery of multiple optimal points of the multimodal functions [116]–[118].

Multi-Swarm Optimization: The Multi-Swarm Optimization (MSO) [119] is a novel variation of the particle swarm optimization (PSO) algorithm to more effectively deal with multimodal functions. MSO employs multiple sub-swarms to search through different regions of the search space. A specific diversification method makes decisions about where and when to initiate the sub-swarms. Employing multiple swarms ensure better search space explorations, which is especially suitable for multi-modal problems with many local optima.

Particle Swarm Optimization: The Particle Swarm Optimization (PSO) [6] algorithm is motivated by the social behavior of bird flocking and fish schooling. In PSO, each candidate solution is represented as a particle or point in the *D*-dimensional search space. During initialization, each particle is usually created at a random location of the *D*-dimensional space with a random velocity along a random direction. Each particle can also communicate with its neighboring particles and possibly with the best-so-far found particle. Particles then fly through the solution space and often accelerated towards those particles within their neighborhood which have better fitness values. The velocity, acceleration and direction information of each particle may also be perturbed randomly and possibly using some key information, such as the best-so-far location seen by each particle or its neighbors or the global best particle found so far. There exist several variants of the PSO algorithm (e.g., [18], [69], [74]) and they have been applied on wide and diverse range of optimization problems [6].

The above list is no way an exhaustive or comprehensive list of the existing swarm intelligence based algorithms. There exist many other SI based algorithms, such as the Gravitational Search Algorithm [120], Central Force Optimization [121], Intelligent Water Drops algorithm [122], Altruism algorithm [123], Magnetic Optimization algorithm [124], Krill Herd algorithm [125], River Formation Dynamics [126], Self-Propelled Particles [127], Stochastic Diffusion Search (SDS) [128], Invasive Weed Optimization [129] and so on [79].

2.12 The Artificial Bee Colony (ABC) Algorithm

The ABC algorithm is based on the intelligent food foraging behavior of honey bees. Tereshko has developed a model of the foraging behavior of a honeybee colony [173]–[175], which identifies three essential components (food sources, employed bees and unemployed bees) and two modes of behavior (recruitment to a food source and abandonment of a source). With these components and modes of behavior, Tereshko [173]–[175] demonstrated the automatic

emergence of distributed, collective swarm intelligence. The ABC algorithm is based on the same model of bee behavior [75]. In the ABC algorithm, the position of a food source represents a candidate solution to the optimization problem and the nectar amount of the food source corresponds to the quality (i.e., fitness value) of the associated candidate solution. ABC employs three different groups of bee agents — the employed, onlooker and scout bees. An employed bee always forages in the vicinity a food source (i.e., candidate solution) that is previously visited by itself. During each foraging attempt, it produces a modification on the food source position (i.e., candidate solution) in its memory depending on the local information (visual information) and evaluates the nectar amount (i.e., fitness value) of the newly found food source (i.e., new candidate solution). If the nectar amount of the new one is higher than that of the previous one, the employed bee memorizes the new position and forgets the old one. Otherwise, it keeps the position of the previous one in its memory. Each employed bee performs a special dance, known as the 'waggle dance' after it returns to the bee hive. The waggle dance contains important piece of information about the quality (fitness) of the food source (candidate solution) it has just found. The onlooker bees (i.e., unemployed foragers) wait around the 'dance floor' of the hive and watch the waggle dances of the employed bees. Each onlooker bee selects any of the employed bees to follow and then forages in the vicinity of the food source of the selected employed bee. The probability of an employed bee to be selected (by any onlooker bee) is proportional to the quality of its food source. As in the case of the employed bees, each onlooker bee also produces a modification on the food position and checks the nectar amount of the new position either to accept or reject it. If the quality of a food source declines because of foraging and drops below some threshold value, the employed bee assigned to it becomes a 'scout' bee. The scout bees randomly explore the search space to discover new food sources. For every food source, there is only one employed bee and (possibly) a number of onlooker bees.

In the original implementation of the ABC algorithm [75], half of the colony consists of employed bees, while the other half is the onlooker bees. The number of food sources (i.e., candidate solutions being exploited) is kept equal to the number of employed bees in the colony. Scout bees are created only when it is necessary, i.e., when a particular food source/candidate solution fails to improve for an unacceptably long period of time, indicating possible stagnation at some locally optimal point. However, in each cycle, no more than one scout bee is initiated, which limits the degree of random explorations of the algorithm. After a scout bee discovers a food source with sufficiently good quality, it turns into an employed bee. A brief outline of the algorithm is presented below (Fig. 2.8).

Algorithm 2.2: Standard Artificial Bee Colony (ABC) Algorithm

1. Initialize Population.
2. **repeat**
3. Send the employed bees to forage around their food sources.
4. For each onlooker bee, pick an employed bee, based on its nectar quality.
5. Send the onlooker bees to forage around their selected employed bees.
6. Send the scout bees to search for new food sources.
7. Memorize the best food source position found so far.
8. **until** some predefined stopping criteria are met.

Figure 2.8: Brief pseudocode of the standard ABC algorithm

As Fig. 2.8 presents, the ABC is an iterative algorithm, it makes progress cycle (iteration) by cycle. Each cycle of the search consists of four steps — (i) sending the employed bees onto their food sources and evaluating their nectar amounts, (ii) sharing the nectar information with each onlooker bee, allowing it to pick a food source to forage, (iii) sending the onlooker bees to their selected food sources for foraging, and (iv) sending the scout bees to random locations to possibly find new food sources. During initialization, a set of food sources is randomly selected across the entire search area and their nectar amounts (i.e., fitness values) are determined. Then ABC executes cycle by cycle, where each cycle constitutes three different stages with different sets of operations — the employed bee stage, the onlooker bee stage and the scout bee stage, each one of which is described in the following paragraphs.

Employed bee stage: This stage mimics the foraging by the employed bees in the vicinity of their current food source positions. The position of each food source is updated in this stage. Suppose, an employed bee is currently positioned at a food source position $\boldsymbol{x_i}$. The employed bee searches in the vicinity of $\boldsymbol{x_i}$ by producing a new trial food position $\boldsymbol{v_i}$ around $\boldsymbol{x_i}$ using (2.6).

$$v_{ij} = x_{ij} + \varphi_{ij}\left(x_{kj} - x_{ij}\right) \tag{2.6}$$

Here, $j \in \{1, 2, \ldots, D\}$ and $k \in \{1, 2, \ldots, SN\}$ are randomly picked indices, D is the number of search dimensions, SN is the number of employed bees (or, food positions) and φ_{ij} is a uniform random value from [-1, 1]. Thus, the new trial solution $\boldsymbol{v_i}$ is produced from $\boldsymbol{x_i}$ by perturbing one of its randomly picked parameters (i.e., x_{ij}) and using the information of another randomly picked candidate solution $\boldsymbol{x_k}$. If $\boldsymbol{v_i}$ has higher 'fitness' value than the original solution $\boldsymbol{x_i}$, then $\boldsymbol{x_i}$ is discarded and replaced by $\boldsymbol{v_i}$. For the problem of function optimization, where F is the function to be minimized, ABC can compute the 'fitness' of a candidate solution $\boldsymbol{x_i}$ by using the same (2.3), as mentioned earlier, but repeated again here for the ease of the reader.

$$fitness(\boldsymbol{x}) = \begin{cases} \dfrac{1}{1+F(\boldsymbol{x})} & \text{if } F(\boldsymbol{x}) \geq 0 \\ 1+\left|F(\boldsymbol{x})\right| & \text{otherwise} \end{cases} \tag{2.3}$$

Onlooker bee stage: In this stage, each onlooker bee first randomly selects a particular employed bee to follow, then forages only in the vicinity of its food source. Suppose, p_i is the probability that the employed bee with food source position $\boldsymbol{x_i}$ would be selected by an onlooker bee, which is computed by ABC using (2.7).

$$p_i = \frac{fitness(\boldsymbol{x_i})}{\sum_{n=1}^{SN} fitness(\boldsymbol{x_n})} \tag{2.7}$$

This makes the probability p_i to be proportional to $fitness(\boldsymbol{x_i})$, ensuring that the probability of picking a food source is kept proportional to its quality. Similar to the employed bees, each onlooker bee also employs (2.6) to produce a trial food source $\boldsymbol{v_i}$ in the vicinity of its current food position $\boldsymbol{x_i}$. If $\boldsymbol{v_i}$ has better fitness value than the old food position $\boldsymbol{x_i}$, then $\boldsymbol{x_i}$ is replaced by $\boldsymbol{v_i}$. Otherwise, $\boldsymbol{x_i}$ is retained and $\boldsymbol{v_i}$ is discarded.

Scout bee stage: If a particular food source position $\boldsymbol{x_i}$ has not been improved over an unusually long period of time (i.e., last *limit* cycles), then it is presumed to be stuck at a locally optimal point. If this happens, the ABC algorithm abandons $\boldsymbol{x_i}$ and the bee employed to $\boldsymbol{x_i}$ now becomes a scout bee that is placed at random across the search space using (2.8), where $j = 1, 2, \ldots, D$ and $[min_j, max_j]$ is the search space along the j-th dimension.

$$x_{ij} = min_j + rand(0,1) * (max_j - min_j) \tag{2.8}$$

Incorporating all of above stages and operations, the following pseudocode presents a detailed step-by-step description of the standard ABC algorithm.

Step 1) Generate an initial population of SN individuals. Each individual $\boldsymbol{x_i}$ is a food source (i.e., candidate solution) that has D parameters, where D is the dimensionality of the problem. Also, initialize the cycle counter C as $C = 1$.

Step 2) Evaluate the fitness of each individual candidate solution.

Step 3) Each employed bee, placed at a food source that is different from others, search in the neighborhood of its current position to find a better food source. To accomplish this, the ABC algorithm generates a new solution $\boldsymbol{v_i}$ around each employed bee $\boldsymbol{x_i}$ using (2.6), which is repeated below, for the ease of the reader.

$$v_{ij} = x_{ij} + \varphi_{ij}\left(x_{kj} - x_{ij}\right) \tag{2.6}$$

Here, $k \in \{1, 2, \ldots, N_{emp}\}$ and $j \in \{1, 2, \ldots, D\}$ are randomly chosen indices, N_{emp} is number of employed bees (or, food sources) and $¤_{ij}$ is a uniform random value produced from [-1, 1].

Step 4) Compute the fitness of both $\boldsymbol{x_i}$ and $\boldsymbol{v_i}$. Apply greedy selection scheme to pick the better one of them to be included into the population and discard the other one.

Step 5) Calculate and normalize the selection probability value p_i for each food source $\boldsymbol{x_i}$ using (2.7), which is repeated again below for the ease of the reader. The value of p_i acts as the selection probability of the food source $\boldsymbol{x_i}$ for the fitness proportional selection in step (6).

$$p_i = \frac{fitness(\boldsymbol{x_i})}{\sum_{n=1}^{SN} fitness(\boldsymbol{x_n})} \tag{2.7}$$

Step 6) Assign each onlooker bee to a food source position $\boldsymbol{x_i}$ at random with probability = p_i. This ensures a fitness proportional selection (and exploration) of the food sources by the onlooker bees.

Step 7) Produce new food position $\boldsymbol{v_i}$ for each onlooker bee by using (2.6), with $\boldsymbol{x_i}$ in (2.6) now being the current food source position to which this onlooker bee is assigned and $\boldsymbol{x_k}$ in (2.6) being another food source picked uniformly at random.

Step 8) Evaluate the fitness of $\boldsymbol{x_i}$ and $\boldsymbol{v_i}$. Apply greedy selection between them, i.e., if the new trial food position $\boldsymbol{v_i}$ has higher fitness value, then accept it and discard the original solution $\boldsymbol{x_i}$. Otherwise, retain $\boldsymbol{x_i}$ and discard $\boldsymbol{v_i}$.

Step 9) If a particular food position $\boldsymbol{x_i}$ has not been improved for an unacceptably long period of time, say the previous *limit* attempts with (2.6), then it is assumed to be stuck at a strong local optimum and is selected for abandonment. Replace this food position by placing a scout bee at a food source placed uniformly at random over the search space using (2.8), which is repeated again below for the ease of the reader. For $j = 1, 2, \ldots, D$, repeat the following (2.8).

$$x_{ij} = min_j + rand(0,1) * (max_j - min_j) \tag{2.8}$$

Step 10) Set the iteration (i.e., cycle) counter $C = C + 1$. Also, keep track of the best food source position (i.e., candidate solution) found so far.

Step 11) Check for termination. If the best solution found is acceptable or a predefined maximum number (say, *MCN*) of cycles have elapsed, then stop and return the best solution found so far. Otherwise go back to step 3 and repeat again.

We can extend the brief pseudocode (Fig. 2.8) of the standard ABC algorithm by following the more detailed steps described above. Fig. 2.9 presents a more detailed pseudocode for the standard ABC algorithm. Here, each cycle (i.e., iteration) of the ABC algorithm consists of foraging by the employed bees (steps 4–5, Fig. 2.9), then foraging by the onlooker bees (steps 7–9), followed by the placement of the scout bees (step 10).

Algorithm 2.3: Artificial Bee Colony (ABC) Algorithm

1: Initialize a pool of *SN* food source positions (candidate solutions) $\boldsymbol{x_i}$, for i = 1, 2, …, *SN*. Each $\boldsymbol{x_i}$ is a vector of *D* parameters: $\boldsymbol{x_i} = [x_{i1}, x_{i2}, \ldots, x_{iD}]^T$

2: Evaluate the fitness of each food source position using (2.3).

3: **repeat**

4: For each employed bee, perturb its current food source position $\boldsymbol{x_i}$ to produce a new food source position $\boldsymbol{v_i}$ by using (2.6).

5: Evaluate each new solution $\boldsymbol{v_i}$ by using (2.3). If $\boldsymbol{v_i}$ has higher fitness than $\boldsymbol{x_i}$, then accept $\boldsymbol{v_i}$ to replace $\boldsymbol{x_i}$. Otherwise, discard $\boldsymbol{v_i}$.

6. Calculate the probability value p_i associated with each food position $\boldsymbol{x_i}$ using (2.7).

7: For each onlooker bee, assign it to a food source position $\boldsymbol{x_i}$, proportionally based on the probability p_i.

8: For each onlooker bee, perturb the food source position of its employed bee $\boldsymbol{x_i}$ to produce a new food source position $\boldsymbol{v_i}$ by using (2.6).

9: Evaluate each new solution $\boldsymbol{v_i}$ using (2.3). If $\boldsymbol{v_i}$ has higher fitness value than $\boldsymbol{x_i}$, then accept $\boldsymbol{v_i}$ to replace $\boldsymbol{x_i}$. Otherwise, discard $\boldsymbol{v_i}$.

10: If a food source has not improved during the last *limit* cycles, then abandon it and replace it with a new randomly placed scout bee with its food source $\boldsymbol{x_i}$ produced by using (2.8).

11: Memorize the best food source position found so far.

12: Set cycle counter $C = C + 1$.

13: **until** C = Maximum cycle number (*MCN*).

14: **return** the best food source position (i.e., candidate solution) found so far.

Figure 2.9: Detailed pseudocode of the standard ABC algorithm

2.13 Strengths of EAs and SIAs

Both EAs and SIAs belong to the population-based, stochastic, meta-heuristic family of algorithms and they share many strengths and advantages over other analytical, deterministic, direct-search and single-state algorithms. Their most prominent strengths and advantages are listed in the following paragraphs.

(i) Both EAs and SIAs are inherently parallel in nature, because they have to maintain a population/swarm of candidate solutions. The candidate solutions can produce multiple offspring to explore the solution space in multiple directions simultaneously. If a particular path reaches some dead end or a poor local optimum, it can be easily discarded and search can continue with more efforts along the more promising paths.

(ii) EAs and SIAs are not just parallel hill-climbers; rather they are much better than that. Each candidate solution can share information with other solutions and affect how they do the hill-climbing in the fitness landscape. The better solutions can share their information with the weaker ones, through the recombination and perturbation operations, thus causing them to move towards the good quality search regions.

(iii) EAs and SIAs can often produce good quality solutions by searching only a small fraction of the search space. This is why they are particularly effective on the problems with very large search space — too large for any exhaustive search. The source of this strength is their inherent parallelism and their ability to implicitly evaluate many schemas (i.e., patterns within candidate solutions) simultaneously, as demonstrated in [131] with an example of the Schemata theorem [130].

(iv) EAs and SIAs are especially suited for non-linear problems with extremely large search space. Most real-world problems are non-linear, where the search variables are not independent of one another — changing a single variable may have ripple effects on the others and can affect the objective value in an unpredicted way. Nonlinearity results in an exponential increase of the search space. Since EAs and SIAs can usually find good quality solutions by searching only a small fraction of the search space, and within a reasonable amount of time, they are often a good choice for the nonlinear problems.

(v) EAs and SIAs are well suited for problems with complex fitness landscape where the fitness function may be discontinuous, noisy or multimodal (having many local optima). Many real-world problems have a huge search space with several local optima where the search algorithm may get stuck with premature convergence. Both EAs and SIAs have proven their effectiveness at escaping the locally optimal points and locating the neighborhood of the global optimum, even in a very complex fitness landscape [132].

(vi) EAs and SIAs are exceptionally good for multi-objective and multi-parameter optimization [133]. They can handle large number of parameters and work on many objectives simultaneously. Their use of parallelism enables them to produce multiple good solutions to the same problem, possibly with one candidate solution optimizing one objective and another solution optimizing a different one [48].

(vii) EAs and SIAs often come up with novel and unconventional solutions to a problem. This is because they don't use any prior knowledge or domain-specific information of the problem during their search. They are like 'blind watchmakers' [134], because they simply make random changes to their candidate solutions and then use the fitness function to determine whether those changes produce any improvements. An interesting example of novel design from the EA is [130], where the EA came up with a high-performance jet engine turbine design that was three times better than a human-designed configuration and 50% better than a configuration designed by an expert system. To find the design, the EA successfully searched across a huge solution space with more than 10^{387} possible solutions, which clearly indicates the strength of EAs on large and complex optimization problems.

2.14 Limitations of EAs and SIAs

Despite their many strengths and advantages, as presented in the previous section 2.13, both EAs and SIAs do have some limitations, too. However, these limitations are mostly well-addressed by the researchers and many of them have firm roots in their biological and natural inspirations. Some of their limitations are as follows.

Need for an appropriate representation: During solving any problem with EA or SIA, the first step is to represent a candidate solution as an individual of the population or the swarm. This requires an effective and appropriate representation of the problem. The representation must be robust and fault-tolerant — it has to ensure that random mutation or perturbations of the existing candidate solutions can't produce invalid or inconsistent candidate solutions. One simple way to ensure this is to employ a repair operator (e.g., [135]) after each perturbation that ensures the validity of the newly perturbed solution, possibly by doing some extra tasks.

Need for a well-designed fitness function: The fitness function should be thoughtfully designed such that higher fitness values are assigned only to those candidate solutions that are actually better solutions to the given problem at hand. If the fitness function is not designed properly, the EA or SIA may be unable to find a solution or even worse, may wind up solving a completely wrong problem, an example of which can be found in [136], where the objective was to design a circuit that can produce an oscillating signal. After the EA finishes its execution using the improper fitness function, a circuit is evolved that really outputs an oscillating signal, but not producing by itself, rather by acting as a radio receiver to pick up and relay an oscillating signal from the nearby electronic devices, which was highly undesirable.

High computational complexity: For complex problems, the repeated fitness evaluations may become computationally prohibitive and may limit the applicability of EAs and SIAs. In many real world problems, the fitness evaluation is computationally very expensive. For example, in some structural optimization problems, a single function evaluation may require hours, or even days of simulations. For such instance, EAs or SIAs may not be a good choice. However, in such cases, an EA or SIA may use approximate fitness evaluations, which is less computation intensive than the exact fitness evaluations.

Limitation with dynamic data: Both EAs and SIAs have inherent limitations to deal with dynamic data sets. This is because the population or swarm usually converges to the best solutions of the already seen data, which may no longer be the best solution (or, even a good solution) for the newly arrived dynamic data points.

Limitation with decision problems: For decision problems, the fitness measure of a candidate solution is just a right/wrong measure, so the fitness value is either 1 or 0. This makes EAs or SIAs to be ineffective for such problems, because there is no fitness hill to climb, rather steep fitness cliffs with no gradient direction to follow.

Limitation with deceptive functions: Another type of problem where both EAs and SIAs face difficulty is the problems with 'deceptive' fitness functions [131]. For these problems, the gradient directions often take the algorithm farther from the global optimum. This can happen when a strong local optimum is located far from the narrow global optimum and the gradients mostly point towards that local optimum. For such problems, an EA or SIA is likely to find the strong local optimum instead of the global optimum.

Limitation with fine tuning: Both EA and SIA are good to produce decent near-optimum solutions within bounded computation time and resource. Usually they can quickly reach close to the peak of a good local or the global optimal hill of the fitness landscape. But precisely pinpointing the peak is usually difficult for them and can be done only by random mutations or perturbations (i.e., by pure chance only). However, for most real world situations, a near-optimum solution is good enough, given a bounded, reasonable amount of time and efforts.

Premature convergence: One crucial issue for the success of both EAs and SIAs is the *premature convergence* [3], [131], which means that the entire population of candidate solutions has converged to one or a few locally optimal points, failing to locate the global optimum. As a result, the algorithm returns a suboptimal solution which may be nowhere around the globally optimum solution. This usually happens when one or a few candidate solutions suddenly become exceptionally fitter than the rest of the population, causing the selection operator to pick them several times for reproduction, which may gradually fill the entire population only by their descendants. This drastically drives down the diversity of the population, because the entire population now represents only a single (or, a few) locally optimal point(s) around those exceptionally fit candidate solutions. Premature convergence is followed by fitness stagnation for many successive iterations. In the ideal case, such fitness stagnation of the population coincides with the successful discovery of the global optimum, which is highly desirable. But stagnation may also occur with premature convergence, because all the individuals have become so similar that the crossover/recombination operation can't produce any new or better candidate solutions. At this point, mutation is the only way to break the premature convergence and explore other areas of the search space. But mutation often requires a number of random downhill steps [82] to get away from the peak of the locally optimal hill where the entire population has converged, which may not be allowed by the selection operator that tries to prematurely dismiss those downhill steps produced by mutation operation.

The likelihood of premature convergence often depends on the characteristics of the fitness landscape — some functions may provide an easy ascent to the globally optimum peak of the fitness landscape, while some other may allow easier rides to the locally optimal points. This may happen more easily with the deceptive problems [131, p.125] or with the problems that have wide, strong locally optimal points far from the global optimum. The problem of premature

convergence is common to arise with EAs and SIAs, and this is particularly common with small population/swarm size, because the possibility of large improvements by a mutation on a particular individual is high with small population size and it may cause the individual to become dominant over the rest of the small population. It is interesting to note that premature convergence does occur in nature, which is called 'genetic drift' by the biologists. This is nothing unusual with the real evolution, because evolution has no commitment to find the global best solution; rather it can find locally optimal solutions that are just good enough. However, premature convergence in nature is less common, because the most beneficial mutations on living things usually produce small, incremental fitness improvements, allowing very little chance to an individual to be dominant over the rest of the population.

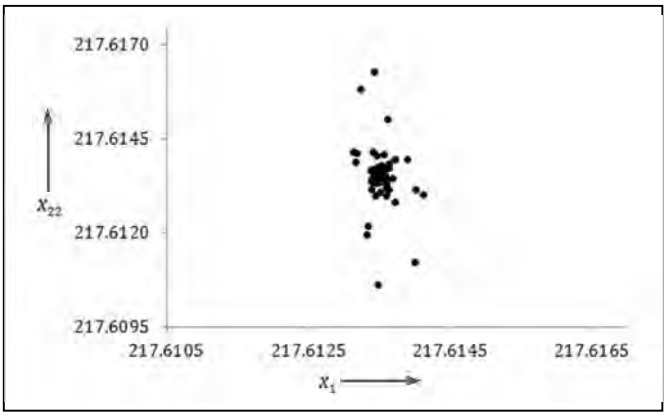


Figure 2.10: Projection of a prematurely converged population of CEP [102] on the dimensions x_1 and x_{22} at the generation number 1000. The scale along x_1 and x_{22} is magnified to show how intensely all the individuals have converged into a very narrow region of the search space.

Fig. 2.10 presents a pathological example of premature convergence, where almost all the individuals have lost their diversity and became too similar. This is an experiment using Classical Evolutionary Programming (CEP) [3] on a 30D benchmark problem. Fig. 2.10 shows the values of the individuals after 1000 generations along two search dimensions — x_1 and x_{22}, with the scale along both the dimensions had been much magnified. Since all the individuals have become too similar to each other, crossover/recombination among them can produce offspring only like themselves. This is why some amount of genetic diversity must be present throughout the entire optimization procedure to avoid the premature convergence. In other words, sufficient amount of population diversity ensures sufficient explorative capacity of the population that helps to avoid premature convergence. In the literature, there exist several works (e.g., [59]–[69], [137]–[172]), from both the evolutionary and swarm intelligence algorithm families, that try to deal with the problems of premature converge. Most of these algorithms either try to maintain sufficient amount of population diversity or try to tweak the explorative and exploitative properties of the algorithm to avoid the problem. A brief literature

survey on the closely related issues of premature convergence, population diversity and explorations vs. exploitations by EAs and SIAs is presented in the following sections 2.15–2.16.

2.15 Literature Survey on EA

This section does not provide any comprehensive survey of EAs on the continuous optimization problem, which would be too extensive a subject to be covered here. Instead, this section presents a brief survey of many existing EAs that try to deal with the problems of premature convergence and fitness stagnation, primarily by tweaking the degree of explorations and/or exploitations of their operators and by maintaining an adequate amount of genetic diversity within the evolving population. Significant amount of research has already been done on this issue, e.g., [137]–[172]. We reviewed a number of representative research works on this issue and categorized them into five different categories, as explained in the following subsections.

2.15.1 Dynamic Operator and Parameter Control

The convergence of the population is affected by the synergy of different evolutionary operators, such as the crossover and mutation and the values of different parameters, like the population size, crossover rate, mutation rate and step size. These operators and parameter settings can be dynamically adjusted to guide the evolutionary process, as demonstrated by several previous works employing deterministic control [137], [138], adaptive control [139], and self-adaptive parameter control [140]. Among them, the simplest scheme is the deterministic control where some predetermined policy controls the operator and parameter settings. Due to the fixed preset policy, it may fail to cope with the dynamic scenario that evolves continuously during the evolution. A better alternative is the adaptive control methods that exploit feedback from the population to control the policy. A good example is the Diversity Guided Evolutionary Algorithm (DGEA) [141]. DGEA applies diversity-decreasing operators (selection and recombination) as long as the diversity is above a certain threshold, d_{low}. When the diversity drops below d_{low} the algorithm switches to a diversity-increasing operator (mutation) until another threshold, d_{high} is reached. The remaining scheme, self-adaptive parameter control methods encode the control parameter values, e.g., mutation rate and step size, as genes in chromosomes and make them evolve with other regular genes. This ensures a self-adaptive and distributed, rather than central control over the parameter values and the evolutionary process itself adapts and finds its way towards an appropriate settings. But the effectiveness of self-adaptive control has not yet been demonstrated on wide range of problems.

2.15.2 Complex Population Structures

In simple GA, any chromosome can mate with any other, so an exceptionally superior quality gene may spread rapidly throughout the entire population. Such high gene flow may cause loss

of genetic diversity which eventually leads to premature convergence. To address this problem, some algorithms have adopted complex population structures or multiple sub-populations, e.g., the diffusion model [142, C6.3], Island-model GA (IMGA) [142, C6.4], [143] multinational GA [72], religion-based EA [144], forking GA [145], bi-objective multi-population algorithm [10], variable island GA [143] and dual population GA [146]. Periodic migration of chromosomes between the population structures, e.g., islands, nations and religions, facilitates exchange of useful information. The semi-isolated and spatially separated nature of the sub-populations lowers gene flow and promotes overall population diversity. The number of sub-populations is either fixed, as in [142], [146], or variable allowing suitable merging and splitting of the sub-populations, as in [72], [143]. Also, the different evolving sub-populations may have the same evolutionary objective, as in [72], [142] or completely different objectives, as in [146]. However, the convergence rate and effectiveness of these algorithms is sensitive to several parameters, such as migration rates and size, migration policy, number and topology of the population structures. A number of research works, such as [147], [148], [149], [150], have made both theoretical and experimental study on the effect of these parameter.

2.15.3 Specialized Selection Operators

An effective way to avoid premature convergence is to control the strength of selection, so that excessively fit candidate solutions don't get too great of an advantage. Some selection operators, such as the Rank, Scaling and Tournament selections (section 2.6.3) are easier ways to achieve this. Another selection scheme to ensure this is the Boltzmann selection [131], where the strength of selection increases gradually in a manner similar to the temperature variable in simulated annealing. Besides, some algorithms, such as crowding [151] and fitness sharing [152]–[154], alter the selection operator in a way that penalizes similar chromosomes and promotes diverse ones. This eventually forms multiple diverse species within the population, each evolving to a different peak of the fitness landscape. The Crowding scheme randomly selects a number, say *CF* (crowding factor) of chromosomes, from current population after producing an offspring, *O*. The chromosome that is most similar to *O* is then replaced by it. This reduces the possibility of hosting similar chromosomes within the population. Several variants of crowding exist in the literature, such as deterministic crowding [155], [156], probabilistic crowding [157], restricted tournament selection (RTS) [158].

Another approach — fitness sharing [152]–[154] is founded upon the idea that there is only limited and fixed amount of 'resources' (i.e., fitness value) available at each neighborhood of the fitness landscape. All the chromosomes occupying the same neighborhood have to share the resource/fitness value. Fitness sharing transforms their raw fitness values in such a way that the larger the density of chromosomes in a region, the more penalized their fitness values become. This discourages similar chromosomes within the population and promotes diversity.

Another example of employing specialized selection operator is the Diversity Control Oriented Genetic Algorithm (DCGA) [159]. DCGA introduces cross-generational probabilistic survival selection (CPSS), a novel selection scheme that calculates the survival probability of each chromosome based on its distance from the current best chromosome. To select a chromosome for the next generation, CPSS considers not only its fitness but also its dissimilarity with the current best chromosome in a way that promotes useful genetic diversity.

2.15.4 Specialized Variation Operators

The genetic variation operators, i.e., recombination and mutation, have been altered in several ways to tweak their capacity of explorations and exploitations. For example, Real Coded Memetic Algorithm (RCMA) with XHC [160] employs crossover hill climbing, which is an exploitative crossover operator that performs effective local tuning of the chromosomes to improve them. The memetic algorithm component of RCMA ensures high level of population diversity to provide a reliable ground on which the more exploitative crossover operator performs its fine tuning. Some evolutionary systems, such as EP and ES [4], [161] schemes employ mutation as their only variation operator. A number of innovative distributions have been proposed for mutation in EP, such as Lévy distribution [56], Cauchy distribution [57], a combination of Cauchy and Gaussian distributions [57]. These distributions introduce relatively large variations to produce offspring which increases the exploration ability of the mutation operator. Larger variations around the parent chromosomes contribute to more population diversity and better resilience against local optima and premature convergence. Two other algorithms, RCMA with Adaptive Local Search (LSRCMA) [162] and Differential Evolution with Neighborhood Search (NSDE) [163], [164] make attempts to combine the benefits of more exploitative local search and neighborhood search techniques with more explorative memetic algorithm and differential evolution respectively. Another approach, Covariance Matrix Adaptation Evolution Strategy (CMAES) [165] continuously adapts the mutation step size in such a way that the likelihood of producing better offspring is maximized. A proper balance between exploitations and explorations is essential to maintain necessary genetic diversity, avoid premature convergence and achieve adequate convergence speed, as demonstrated in [160].

2.15.5 Memory-Based Algorithms

Memory-based algorithms make use of an additional explicit or implicit memory in order to maintain more genetic information that provides the population with additional genetic diversity. Examples of memory-based algorithms include GA with unexpressed genes (GAUG) [166], the Diploid GA [167], [168], Dual GA (DGA) [169], [170], the Primal-Dual GA

(PDGA) [171], [172] and so on. The biological inspiration behind memory based algorithms is that most beings in nature have a large number of 'repressed' genes in their chromosomes with relatively small number of dominant genes that are expressed in their characteristics. The repressed genes store large amount of latent information that are used for producing offspring and thus act as a source of population diversity for the next generations. GAUG uses haploid (single strand) chromosomes, but includes a number of unexpressed genes that are used for preserving diversity. Diploid GAs use diploid (double strand) chromosomes and follow some predetermined dominance rules that decide which genes will be expressed and used for fitness evaluation. The remaining genes are used for providing diversity to produce offspring. Both DGA and PDGA employ haploid chromosomes, but the chromosomes may be interpreted in a complemented way to provide additional diversity. In PDGA, a less fit chromosome may be interpreted both as original and as complemented, and the original one is replaced by the complemented one if the latter gives better fitness evaluation. Memory based algorithms are more suitable for dynamic optimization problems because the usage of additional memory for extra diversity makes it easier to adapt to external and unprecedented environmental changes.

2.16 Literature Survey on ABC

This section is not a complete survey on the swarm intelligence algorithms in general, or the ABC algorithm in particular. In this section we first make a brief survey on the swarm intelligence algorithms that are based on the intelligent honey bees behavior, then narrow down our focus only on the subset of these algorithms that deal with the continuous optimization problems, including ABC, then briefly review some of the improved ABC-variants that try to deal with the problems of premature convergence and fitness stagnation by tweaking the explorative and/or exploitative characteristics of the standard ABC algorithm.

In the literature, there exist different models that try to simulate the specific food foraging behavior of honey bee swarms. Such models have been applied for solving both combinatorial and continuous optimization problems (e.g., [173]–[193]). For example, Tereshko and Loengarov [173] have modeled a bee colony as a dynamic system of autonomous robots, each with a small memory and some capability to adapt its behavior based on the information acquired from the environment. Results [173] demonstrate that these insect-like robots, by mimicking bee behavior, are quite successful in real robotic tasks. Tereshko, Loengarov and Lee [174], [175] have built a foraging model including components such as food source, committed foragers (i.e., bees), and uncommitted foragers and identified some patterns of bee-like behavior that are found responsible for the emergence of collective intelligence. Wedde et

al. [176] developed a novel routing algorithm, i.e., BeeHive, based on a dynamically communicative model that emulates the intelligent behavior of honey bees. In this model, bee agents travel through the network regions to collect, store and share several pieces of information about the network state in order to update the local routing tables. Another metaheuristic, Bee Colony Optimization (BCO) [177], has been employed for solving combinatorial problems, both deterministic as well as dynamic instances with uncertainty. The intelligent swarm behavior of bees is also demonstrated to be useful in solving complex transportation problems [178]–[180]. Drias et al. [181] introduced a metaheuristic, called Bees Swarm Optimization (BSO) and applied it to solve the maximum weighted satisfiability (max-sat) problem. Benatchba et al. [182] introduced a metaheuristic derived from the reproduction process of honey bees to solve the 3-sat problem. Fathian et al. [183] have developed a novel two-stage algorithm for cluster analysis using the mating behavior of honey bees. Algorithms inspired by bee behavior have also been employed for solving Traveling Salesman Problem [184], Generalized Assignment Problem [185], Job shop scheduling [186], Dynamic allocation of internet servers [187] and so on.

Most of the works mentioned in the previous paragraph are focused on discrete and combinatorial problems. Relatively few works exist in literature for continuous optimization problems. However, there exist at least three different algorithms on continuous optimization based on the intelligent bee behavior, which are — the Virtual Bee Algorithm (VBA) [188], Bees Algorithm (BA) [189], [190] and Artificial Bee Colony (ABC) Algorithm [11], [75], [191]–[193]. Yang et al. developed the VBA and applied it on numeric function optimizations. Yang demonstrated the capability of the algorithm by employing it on functions with two parameters only. In VBA, a swarm of virtual bees randomly explore the search space following a communication model that mimics the bees' behavior in nature. Bees that find some good quality nectar (i.e., candidate solution) that is better than some predefined threshold usually share this information with other bees by waggle dance upon returning to the hive. The optimal solution is identified from the intensity of the interactions among the bees. Results show that VBA is more effective and better resilient against local optima in comparison to genetic algorithm. The Bees Algorithm (BA) was designed by Pham et al. [189], [190] which also mimics the food foraging behavior of honey bees. BA requires a number of parameters to be set and tries to combine a kind of neighborhood search with random search that is suitable for both combinatorial and continuous problems. However, both VBA and BA are mostly tested on low dimensional problems with fewer search parameters and they are often outperformed by the ABC algorithm, which has demonstrated excellent performance on both low and high

dimensional functions [11]. ABC is first introduced by Karaboga in 2005 [75], then further improved by Basturk and Karaboga [191]—[193]. They have compared the performance of ABC with that of genetic algorithm (GA) [194], differential evolution (DE) [195], PSO [196] and particle swarm inspired evolutionary algorithm (PS-EA) [197] using a number of test problems on continuous function optimization, where ABC has significantly outperformed all the other algorithms on most of the functions. Pawar et al. [198]–[200] employed the ABC algorithm to some mechanical engineering problems, including multi-objective optimization of electro-chemical machining process parameters, optimization of process parameters of the abrasive flow machining process and the milling process. To maximize the exploitation capacity of the onlooker bees, Tsai et al. [201] introduced the Newtonian law of universal gravitation into the basic ABC algorithm to introduce the Interactive ABC (IABC) algorithm. Baykasoglu et al. [185] hybridized some techniques, e.g., shift neighborhood search and greedy randomized adaptive search heuristics and applied it to the generalized assignment problem. ABC is also extended for constrained optimization problems [192], training neural networks [202], data analysis [203], clustering [204], structural analysis [205], design of digital IIR filters [25], PID controller [26], software testing [206], multi-objective optimization [17] and so on.

The ABC algorithm has been extended and improved in many ways, most of which try to improve either its explorative characteristics (for better strength against premature convergence and fitness stagnation) or its exploitative properties (for better convergence speed). Here, we briefly present some of the recent and improved ABC-variants that are focused on tweaking the explorative/exploitative characteristics of the basic ABC algorithm. For example, the cooperative ABC (CABC) [60] is a cooperative and explorative variant of ABC. CABC has been introduced in two different versions — CABC_S and CABC_H. In order to perform more explorations, CABC_S decomposes the search space into multiple sub-spaces and employs different bee colonies to search and explore the different sub-spaces. The other variant, CABC_H tries to perform more exploitations than CABC_S by repeatedly alternating between explorative CABC_S and exploitative ABC. Another variant — ABC with diversity strategy (DABC) [61] tries to maintain sufficient level of population diversity for conducting more explorations by alternating between two different perturbation schemes — one explorative, and the other one exploitative. DABC regularly measures the existing population diversity d and employs either its explorative or exploitative perturbation based on the current value of d. Another explorative ABC-variant is Chaotic ABC (ChABC) [62] which employs chaotic search behavior during perturbations to produce new food positions from the existing ones. Chaotic dynamics are produced by the logistic equations (eq. (4)–(7) in [62]) which provide a simple mechanism to

escape from local minima and avoid premature convergence. Fenglei et al. [63] introduced a novel control mechanism of the locally optimal solutions to improve the explorative search capacity of the ABC algorithm. The Gbest-guided ABC (GABC) [64] is an exploitative ABC-variant that tries to improve the convergence speed of ABC by using the information of the global best solution found so far into the perturbation scheme (2.6) of the basic ABC. The Hooke Jeeves ABC (HJABC) [65] is a hybrid ABC-variant that intensifies the degree of exploitations by hybridizing basic ABC with an efficient local search technique (i.e., the Hooke Jeeves pattern search). Qingxian and Haijun [66] introduced a modified initialized scheme and replaced the roulette wheel selection of basic ABC algorithm by Boltzmann selection scheme for improving the exploitations and convergence speed of ABC. Quan and Shi [67] introduced a novel search iteration operator based on the fixed point theorem of contractive mapping which is reported to improve the exploitations and convergence speed of ABC. The Elitist ABC (EABC) [68] is another exploitative ABC-variant that hybridizes ABC with two different local search operators to intensify the degree of exploitations around the best candidate solution found so far.

A significant limitation of most of the improved ABC-variants described so far is that they try to improve either the explorative or the exploitative (but not both) characteristics of the basic ABC algorithm. The exploitative improvements are usually based on intensifying local search around the best-so-far solutions (e.g., [64], [65], [68]) and hybridizing efficient local search operators with ABC (e.g., [65], [67]), while the explorative improvements are usually based on more population diversity (e.g., [60], [61]) and more explorative initialization, selection and/or perturbation operations (e.g., [62], [66]). Along the course of this thesis, we have developed three improved variants of the standard ABC algorithm that try to balance between explorations and exploitations by employing techniques like adaptation, self-adaptation and hybridization. A detailed description of our three improved ABC-variants is presented in the upcoming chapters 5–7.

2.17 Benchmark Problems on Continuous Optimization

In order to evaluate a newly introduced algorithm and to compare it with other existing algorithms on the continuous optimization problem, a number of benchmark problems have been proposed and employed by the researchers [57], [76]. Along the course of this thesis, we have developed a number of improved EP and ABC-based algorithms, as described in details along the chapters 3 to 7, each one of which is evaluated and compared using two different suites of benchmark problems, consisting of a total of 55 benchmark problems on numerical function optimization. No other work, as to our best knowledge, has been tested and evaluated

on such a wide range of benchmark problems. Each of our algorithms is first tested on the standard benchmark suite, which consists of 30 standard continuous functions for numerical optimization. Each function in this suite is well studied and has been used to evaluate many other existing evolutionary and swarm intelligence algorithms, e.g., [10], [11], [55]–[69], [146], [160]–[164], [196]. Later, each of our algorithms is also evaluated on a recent benchmark suite consisting of 25 functions for numeric optimization, introduced in the special session on real parameter optimization at the 2005 IEEE Congress on Evolutionary Computation (CEC-2005), held on 2-4 September 2005 at Edinburgh, UK. These CEC2005 suite functions are relatively more complex, because they include many shifted, rotated, scaled, expanded and hybrid composite functions. A brief summary of these functions is presented in this section (Tables 2.3 and 2.4). Further details on each of these functions can be found in the Appendix A, and also in [11], [65] and [76].

The standard benchmark suite (Table 2.3) contains both unimodal (f_1–f_9) and multimodal (f_{10}–f_{30}), separable (e.g., f_1, f_3) and non-separable (e.g., f_2, f_4), high dimensional (f_1–f_{18}) and low dimensional (f_{19}–f_{30}) functions. A function is called multimodal if it has multiple local optima. To optimize such a function, the search algorithm must possess both exploitative and explorative characteristics so that it can explore the locally optimal points without being trapped anywhere and thus keep exploring further for better unvisited regions of the search space. Some of the multimodal functions can have hundreds of local optima, even when the dimensionality is just two or three (e.g., Rastrigin function f_{10}, as demonstrated in Fig. 2.11). The number of local optima usually increases exponentially with the number of dimensions. This often makes the minimization of high dimensional multimodal functions extremely difficult. Although the Rosenbrock function f_7 is usually considered to be a unimodal function, as in [11], [56], there is some evidence (e.g., [228], [229]) that it contains several minima in high dimensional instances. The global minimum is situated inside a narrow and almost flat valley (Fig. A.1.5 in the Appendix A). Finding the valley is straightforward, but precisely pinpointing the global minimum in the almost flat valley is extremely difficult, because the flat region does not provide any useful gradient direction pointing towards the global minimum. The Ackley function f_{13} has an exponential term with cosine sum which issues numerous local minima surrounding one narrow basin for the global minimum. The Griewank function f_{14} has a product term implementing interdependence among all the variables, so any technique trying to optimize each variable separately is bound to failure for this function. The complexity of the Schwefel function f_{12} is due to having a strong second-best local minimum which is very far from the global minimum of the search space. The low dimensional functions f_{19}–f_{30} have dimensionality

$D \leq 10$, and most of them contain only a small number of local minima. But their locally minimal points usually have large distances among themselves over the parameter space (e.g., Michalewicz function f_{29}, as shown in Fig. A.1.13 in the Appendix A). This makes their optimization difficult for many algorithms, because escaping from their strong locally optimal valleys usually requires relatively large, more explorative perturbations on the candidate solutions. A brief summary on the standard benchmark suite is presented in Table 2.3.

Table 2.3: The standard benchmark suite functions for the evaluation and comparison of each of our proposed algorithms. Here, *D*: dimensionality of the function, *S*: search space, f_{min}: function value at global minimum, *C*: function characteristics with values — *U*: Unimodal, *M*: Multimodal, *S*: Separable and *N*: Non-Separable.

No	Function	*S*	*C*	f_{min}	Formulation
f_1	Sphere	$[-100, 100]^D$	*US*	0	$f(\boldsymbol{x}) = \sum_{i=1}^{D} x_i^2$
f_2	Schwefel 2.22	$[-10, 10]^D$	*UN*	0	$f(\boldsymbol{x}) = \sum_{i=1}^{D} \lvert x_i \rvert + \prod_{i=1}^{D} \lvert x_i \rvert$
f_3	Schwefel 2.21	$[-10, 10]^D$	*US*	0	$f(\boldsymbol{x}) = \max_i \{\lvert x_i \rvert, 1 \leq i < D\}$
f_4	Schwefel 1.2	$[-100, 100]^D$	*UN*	0	$f(\boldsymbol{x}) = \sum_{i=1}^{D} \left(\sum_{j=1}^{i} x_j \right)^2$
f_5	Powell	$[-4, 5]^D$	*UN*	0	$f(\boldsymbol{x}) = \sum_{i=1}^{D/k} \left(x_{4i-3} + 10x_{4i-2} \right)^2 + 5\left(x_{4i-1} - x_{4i} \right)^2 + \left(x_{4i-2} - x_{4i-1} \right)^4 + 10\left(x_{4i-3} - x_{4i} \right)^4$
f_6	Dixon-Price	$[-10, 10]^D$	*UN*	0	$f(\boldsymbol{x}) = \left(x_1 - 1 \right)^2 + \sum_{i=2}^{D} i\left(2x_i^2 - x_{i-1} \right)^2$
f_7	Rosenbrock	$[-30, 30]^D$	*UN*	0	$f(\boldsymbol{x}) = \sum_{i=1}^{D-1} \left[100\left(x_{i+1} - x_i^2 \right)^2 + \left(x_i - 1 \right)^2 \right]$
f_8	Step	$[-100, 100]^D$	*US*	0	$f(\boldsymbol{x}) = \sum_{i=1}^{D} \left(\lfloor x_i + 0.5 \rfloor \right)^2$
f_9	Quartic	$[-1.28, 1.28]^D$	*US*	0	$f(\boldsymbol{x}) = \sum_{i=1}^{D} i x_i^4 + \text{random}[0,1)$
f_{10}	Rastrigin	$[-5.12, 5.12]^D$	*MS*	0	$f(\boldsymbol{x}) = \sum_{i=1}^{D} \left[x_i^2 - 10\cos(2\pi x_i) + 10 \right]$
f_{11}	Non-continuous Rastrigin	$[-5.12, 5.12]^D$	*MS*	0	$f(\boldsymbol{x}) = \sum_{i=1}^{D} \left[y_i^2 - 10\cos(2\pi y_i) + 10 \right]$ where $y_i = \begin{cases} x_i & \lvert x_i \rvert < 0.5 \\ 0.5 * \text{round}(2x_i) & \lvert x_i \rvert \geq 0.5 \end{cases}$
f_{12}	Schwefel	$[-500, 500]^D$	*MS*	-12569.5	$f(\boldsymbol{x}) = -\sum_{i=1}^{D} x_i \sin\left(\sqrt{\lvert x_i \rvert} \right)$
f_{13}	Ackley	$[-32, 32]^D$	*MN*	0	$f(\boldsymbol{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} x_i^2} \right) - \exp\left(\frac{1}{D}\sum_{i=1}^{D} \cos(2\pi x_i) \right) + 20 + e$
f_{14}	Griewank	$[-600, 600]^D$	*MN*	0	$f(\boldsymbol{x}) = \frac{1}{4000}\sum_{i=1}^{D} x_i^2 - \prod_{i=1}^{D} \cos\left(\frac{x_i}{\sqrt{i}} \right) + 1$
f_{15}	Alpine	$[-10, 10]^D$	*MS*	0	$f(\boldsymbol{x}) = \sum_{i=1}^{D} \lvert x_i \sin(x_i) + 0.1x_i \rvert$

Table 2.3 (Continued): The standard benchmark functions for the evaluation and comparison of each of our proposed algorithms.

No	Function	*S*	*C*	f_{min}	Formulation
f_{16}	Weierstrass	$[-0.5, 0.5]^D$	*MS*	0	$f(\boldsymbol{x})=\sum_{i=1}^{D}\left(\sum_{k=0}^{kmax}\left[a^k\cos\left(2\pi b^k(x_i+0.5)\right)\right]\right)-D\sum_{k=0}^{kmax}\left[a^k\cos\left(2\pi b^k 0.5\right)\right];\ a=0.5, b=3, kmax=20$
f_{17}	Penalized	$[-50, 50]^D$	*MN*	0	$f(\boldsymbol{x})=\frac{\pi}{D}\left[10\sin^2(\pi y_1)+(y_D-1)^2\right]+\sum_{i=1}^{D}u(x_i,10,100,4)+\frac{\pi}{D}\sum_{i=1}^{D-1}(y_i-1)^2\left(1+10\sin^2(\pi y_{i+1})\right), y_i=1+\frac{1}{4}(1+x_i)$
f_{18}	Penalized2	$[-50, 50]^D$	*MN*	0	$f(\boldsymbol{x})=0.1\left[\sin^2(\pi x_1)+(x_D-1)^2\left(1+\sin^2(2\pi x_D)\right)\right]+\sum_{i=1}^{D}u(x_i,5,100,4)+0.1\sum_{i=1}^{D-1}(x_i-1)^2\left(1+\sin^2 3\pi x_{i+1}\right)$
f_{19}	Foxholes	$[-65.536, 65.536]^D$	*MS*	1	$f(\boldsymbol{x})=\left[1/500+\sum_{j=1}^{25}\left(1.0/j+\sum_{i=1}^{2}(x_i-a_{ij})^6\right)\right]^{-1}$
f_{20}	Kowalik	$[-5, 5]^D$	*MN*	3.07e-04	$f(\boldsymbol{x})=\sum_{i=1}^{11}\left(a_i-x_1\left(b_i^2+b_i x_2\right)/\left(b_i^2+b_i x_3+x_4\right)\right)^2$
f_{21}	Six Hump Camel Back	$[-5, 5]^D$	*MN*	-1.0316	$f(\boldsymbol{x})=4x_1^2-2.1x_1^4+(1/3)x_1^6+x_1x_2-4x_2^2+4x_2^4$
f_{22}	Branin	[-5, 10] x [0, 15]	*MS*	0.398	$f(\boldsymbol{x})=\left(x_2-\frac{5.1}{4\pi^2}x_1^2+\frac{5}{\pi}x_1-6\right)^2+10\left(1-\frac{1}{8\pi}\right)\cos(x_1)+10$
f_{23}	Hartman3	$[0, 1]^D$	*MN*	-3.86	$f(\boldsymbol{x})=-\sum_{i=1}^{4}c_i\exp\left[-\sum_{j=1}^{D}a_{ij}\left(x_j-p_{ij}\right)^2\right];\ D=3,6$
f_{24}	Hartman6	$[0, 1]^D$	*MN*	-3.32	
f_{25}	Shekel5	$[0, 10]^D$	*MN*	-10.15	$f(\boldsymbol{x})=-\sum_{i=1}^{m}\left[(x-a_i)(x-a_i)^T+c_i\right]^{-1};\ m=5,7,10$
f_{26}	Shekel7	$[0, 10]^D$	*MN*	-10.40	
f_{27}	Shekel10	$[0, 10]^D$	*MN*	-10.55	
f_{28}	Fletcher-Powell	$[-\pi, \pi]^D$	*MN*	0	$f(\boldsymbol{x})=-\sum_{i=1}^{D}(A_i-B_i)^2; A_i=\sum_{j=1}^{D}\left(a_{ij}\sin\alpha_j+b_{ij}\cos\alpha_j\right), B_i=\sum_{j=1}^{D}\left(a_{ij}\sin x_j+b_{ij}\cos x_j\right)$
f_{29}	Michalewicz	$[0, \pi]^D$	*MS*	-9.66015	$f(\boldsymbol{x})=-\sum_{i=1}^{D}\sin(x_i)\left(\sin\left(ix_i^2/\pi\right)\right)^{2m};\ m=10$
f_{30}	Langerman	$[0, 10]^D$	*MN*	-1.4	$f(\boldsymbol{x})=-\sum_{i=1}^{10}c_i\exp\left(-\frac{1}{\pi}\sum_{j=1}^{D}(x_j-a_{ij})^2\right)\cos\left(\pi\sum_{j=1}^{D}(x_j-a_{ij})^2\right)$

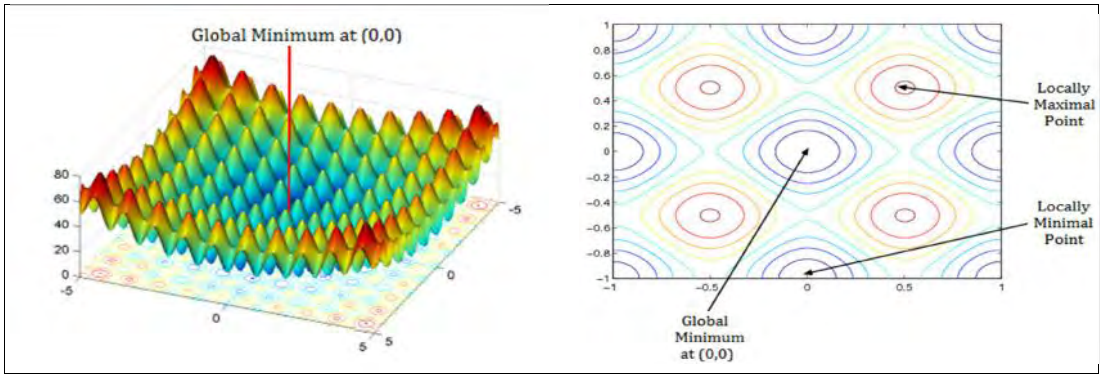


Figure 2.11: 3D surface plot (on the left) and 2D contour plot (right) of 2D Rastrigin function

In addition to the standard benchmark suite functions, as described above, we have also employed the recently introduced CEC2005 benchmark suite [76] to evaluate and compare each of our proposed algorithms. The CEC2005 suite consists of 25 functions of different complexity, including many shifted, rotated, scaled, expanded and hybrid composite functions. The first five functions (F1−F5) are unimodal, while the remaining twenty (F6-F25) are multimodal functions. For almost all of these functions, their global minimum is shifted randomly along each dimension, and their coordinate space is often rotated to make their optimization even more difficult. The suite also contains expanded functions, functions with noise, functions without bounds, non-continuous functions, functions with the global minimum on the bounds and functions with a narrow basin for the global minimum. The functions F15−F25 are the hybrid composite functions that are composed of several other standard benchmark functions to make them more challenging to minimize. A Gaussian distribution is used to combine the constituent benchmark functions to blur the individual function's characteristics. A brief summary on the CEC2005 benchmark suite is presented in Table 2.4. A more detailed description on each of the CEC2005 functions can be found in the Appendix A and in [76].

Table 2.4: The CEC2005 benchmark suite functions. Here, D is the dimensionality of the function, S is the search space and f_{min} is the function value at the global minimum.

No	Function	S	f_{min}	Function Characteristics
F_1	Shifted Sphere Function	$[-100, 100]^D$	−450	Unimodal, Shifted, Separable, Scalable
F_2	Shifted Schwefel's Problem 1.2	$[-100, 100]^D$	−450	Unimodal, Shifted, Non-separable, Scalable
F_3	Shifted Rotated High Conditioned Elliptic Function	$[-100, 100]^D$	−450	Unimodal, Shifted, Rotated, Non-separable, Scalable
F_4	Shifted Schwefel's Problem 1.2 with Noise	$[-100, 100]^D$	−450	Unimodal, Shifted, Non-separable, Scalable, Noise in Fitness
F_5	Schwefel's Problem 2.6 with Global Optimum on Bounds	$[-100, 100]^D$	−310	Unimodal, Non-separable, Scalable, Global Optimum on the Bounds
F_6	Shifted Rosenbrock Function	$[-100, 100]^D$	390	Multimodal, Shifted, Non-separable, Scalable, Very Narrow Valley from Local Optimum to Global Optimum
F_7	Shifted Rotated Griewank Function without Bounds	Unbounded, but Initialize Population in $[0, 600]^D$	−180	Multimodal, Shifted, Rotated, Non-separable, Scalable
F_8	Shifted Rotated Ackley's Function with Global Optimum on Bounds	$[-32, 32]^D$	−140	Multimodal, Shifted, Rotated, Non-separable, Scalable, Global Optimum on the Bound
F_9	Shifted Rastrigin's Function	$[-5, 5]^D$	−330	Multimodal, Shifted, Separable, Scalable

Table 2.4 (continued): The CEC2005 benchmark suite functions. Here, D is the dimensionality of the function, S is the search space and f_{min} is the function value at the global minimum.

No	Function	S	f_{min}	Function Characteristics
F_{10}	Shifted Rotated Rastrigin Function	$[-5, 5]^D$	−330	Multimodal, Shifted, Rotated, Non-separable, Scalable
F_{11}	Shifted Rotated Weierstrass Function	$[-0.5, 0.5]^D$	90	Multimodal, Shifted, Rotated, Non-separable, Scalable, Continuous but not Differentiable on Most Points
F_{12}	Schwefel's Problem 2.13	$[-\pi, \pi]^D$	−460	Multimodal, Non-separable, Shifted, Scalable
F_{13}	Shifted expanded Griewank plus Rosenbrock Function	$[-3, 1]^D$	−130	Multimodal, Shifted, Rotated, Non-separable, Scalable, Expanded
F_{14}	Shifted Rotated Scaffer's F6 Function	$[-100, 100]^D$	−300	Multimodal, Shifted, Rotated, Non-separable, Scalable, Expanded
F_{15}	Hybrid Composition of Rastrigin, Weierstrass, Griewank, Ackley and Sphere Functions	$[-5, 5]^D$	120	Multimodal, Shifted, Scalable, Hybrid, Separable Near Global Optimum
F_{16}	Rotated Version of the Hybrid Composition Function F_{15}	$[-5, 5]^D$	120	Multimodal, Shifted, Rotated, Non-separable, Scalable, Hybrid
F_{17}	F_{16} with Noise in Fitness	$[-5, 5]^D$	120	Multimodal, Shifted, Rotated, Non-separable, Scalable, Hybrid, Gaussian Noise in Fitness
F_{18}	Rotated Hybrid Composition of Ackley, Sphere, Rastrigin, Weierstrass and Griewank Functions	$[-5, 5]^D$	10	Multimodal, Shifted, Rotated, Non-separable, Scalable, Hybrid
F_{19}	Rotated Hybrid Composition Function F_{18} with Narrow Basin Global Optimum	$[-5, 5]^D$	10	Multimodal, Shifted, Rotated, Non-separable, Scalable, Hybrid, Narrow Basin for Global Optimum
F_{20}	Rotated Hybrid Composition Function F_{18} with Global Optimum on the bounds	$[-5, 5]^D$	10	Multimodal, Shifted, Rotated, Non-separable, Scalable, Hybrid
F_{21}	Rotated Hybrid Composition of F_{13}, F_{14}, Rastrigin, Weierstrass, Griewank Functions	$[-5, 5]^D$	360	Multimodal, Shifted, Rotated, Non-separable, Scalable, Hybrid
F_{22}	Rotated Hybrid Composition Function F_{21} with High Condition Number Matrix	$[-5, 5]^D$	360	Multimodal, Shifted, Rotated, Non-separable, Scalable, Hybrid, Global Optimum on the Bound
F_{23}	Non-Continuous version of the Rotated Hybrid Composition Function F_{21}	$[-5, 5]^D$	360	Multimodal, Non-continuous, Non-separable, Shifted, Rotated, Scalable, Hybrid, Global Optimum on Bound
F_{24}	Rotated Hybrid Composition of $F_{13\text{-}14}$, Sphere, Ackley, Rastrigin, Griewank, Non-Continuous F_{14} and Rastrigin, High Conditioned Elliptic Functions	$[-5, 5]^D$	260	Multimodal, Shifted, Rotated, Non-separable, Scalable, Hybrid
F_{25}	Rotated Hybrid Composition Function F_{24} , but without Bounds	Unbounded, but Initialize Population in $[2, 5]^D$	260	Multimodal, Shifted, Rotated, Non-separable, Scalable, Hybrid, Search Space Unbounded

Chapter 3

Recurring Two-Stage Evolutionary Programming

3.1 Introduction

The necessity of search space explorations to avoid premature convergence is emphasized in the previous sections 2.14–2.15. This chapter introduces recurring two-stage evolutionary programming (RTEP) — a novel evolutionary algorithm that tries to balance the explorative and exploitative features of the conventional evolutionary algorithms. RTEP is based on repeated and alternated execution of two different stages, namely the exploration and exploitation stages, each stage with its own mutation operator, selection strategy and explorative/exploitative objective. This is significantly different from most other existing evolutionary algorithms (e.g., [11]–[27]) which typically follow a single stage execution model with no explicit attempts to balance between explorations and exploitations. In this chapter, we have presented both analytical and experimental studies on RTEP to evaluate and compare its performance as well as to understand the role and necessity of its repeated and alternated explorative and exploitative operations.

3.2 Organization of the Chapter

The rest of the chapter is organized as follows. Section 3.3 describes the proposed algorithm — RTEP in details, along with necessary pseudocode, flowchart of the algorithm and analysis and explanation of its different stages with their recurring design. Section 3.4 explains the differences of the proposed RTEP algorithm from most other existing evolutionary systems. A brief theoretical analysis on RTEP is presented in the next section 3.5. Section 3.6 evaluates the performance of RTEP on two different benchmark suites of continuous optimization problems, compares its results with several other recent and relevant algorithms and makes a brief discussion on their results. Finally, section 3.7 concludes with a summary of this chapter and provides a few suggestions on further research with RTEP.

3.3 The Proposed Algorithm — RTEP

Most evolutionary algorithms (e.g., [11]–[27]) make no explicit attempts to maintain a proper balance between the explorative and exploitative operations. Exploitation is usually carried out implicitly by fitness based selection pressure, while exploration is conducted by the genetic operators, especially by the mutation operation. Such implicit explorations and exploitations, without any explicit control over them by the algorithm, often lead to quick loss of population diversity and stagnation around the local optimal points of the fitness landscape [54]. There also exist a number of studies (e.g., [56]–[66], [160]–[163]) that try to balance between explorations and exploitations. However, most of these works are aligned either towards more explorations (e.g., [56], [57], [60]–[63]) or towards more exploitations (e.g., [64]–[68], [160], [162], [163]). This is because most algorithms consider explorations and exploitations to be conflicting objectives, so they try to increase either the exploitative or the explorative capacity of the operators, either for faster, but possibly premature convergence or for slower, but more global convergence avoiding local optima.

Some recent studies (e.g., [207]–[209]) reveal that the explorations and exploitations during an evolutionary optimization are not always conflicting objectives; rather they may be complementary to each other. For example, some exploitation is always necessary after exploring to a new search region in order to realize the potentials of the newly explored solutions. Also, long exploitations can lead to getting trapped around the locally optimal points, so some successive explorative operations might help to break free from the local optima. The motivation behind RTEP is to exploit such complementary characteristics of explorations and exploitations. RTEP is based on the proper utilization of the complementary properties of exploitations and explorations to improve the optimization procedure for better solution quality and faster convergence. To ensure a proper balance between exploitations and explorations, RTEP repeatedly switches and alternates to and from its explicitly explorative and exploitative operations during the course of evolution. The exploration stage employs mutations with large step size to reach the unexplored regions of the search space. The exploitation stage employs small mutation steps to exploit and hill-climb the locally optimal hills around each point. All these operations are designed explicitly (rather than implicitly) either for explorations or for explorations, which is motivated by observing the following important facts.

- Exploration is a non-local operation. Therefore, genetic operations involving distant and dissimilar genes may help the individuals to break free from local optima and can lead the search process to unexplored regions of the search space.

- After exploring to a new region, some exploitative steps are often necessary to realize the potentials of the newly explored regions and to avoid the early rejection of the new, promising solutions.
- Exploitation is a local operation. Therefore, genetic operations involving similar genotypes (i.e., neighboring individuals in the search space) and allowing only uphill moves are relevant to locate and pinpoint the optimal points of each search region/neighborhood.

A recurring two-stage evolutionary approach is adopted for RTEP in an attempt to balance the conflicting goals of evolution, i.e., exploration and exploitation. This approach employs one stage for global exploration and another stage for local exploitation. The two stages execute in a recurring fashion, one after another, again and again. This regular switching to and from both the stages avoids the problem of making a perfect decision of permanent switch from explorations to exploitations. The regularly alternating stages make it possible to gracefully distribute the conflicting goals of exploration and exploitation across the generations of the evolution. The exploration stage employs Gaussian mutation with a large standard deviation, which is set from the distance of two dissimilar individuals along the search dimension currently being mutated. The same mutation with a small standard deviation, set from the distance between two similar individuals along the current search dimension, is used during the exploitation stage. Like any evolutionary process, RTEP starts with some initial population of individuals. Each individual represents a candidate solution to the particular problem at hand. The solutions may be values of a set of variables that optimize a function, process, plan, design, a set of strategies, or any entity that need to be optimized. During each generation, RTEP employs either explorative or exploitative mutation operators on the individuals to produce some offspring individuals. The offspring are inserted into the population depending on the re-insertion scheme based on the current explorative or exploitative stage objective. The major steps of RTEP can be described as follows.

Step 1) Generate an initial population of M individuals. Each individual $\boldsymbol{x}_i$, i = 1, 2, …, M, is represented as a real valued vector with D components, where D is the dimensionality of the problem. $\boldsymbol{x}_i = \{x_i(1), x_i(2), \ldots, x_i(D)\}$. Each component of $\boldsymbol{x}_i$, for i = 1, 2, …, M, is generated uniformly at random within its domain, i.e., $x_i(j) = Uniform\_Random \sim (min_j, max_j)$, for j = 1, …, D.

Step 2) Initialize the parameters K_1 and K_2 within some certain range. They define how many generations the exploration and exploitation stages shall each continue before switching to the other.

Step 3) Calculate the fitness value of each individual $\boldsymbol{x}_i$, i = 1, 2, …, M, based on the objective function value. If the best fitness value of the population is acceptable, then STOP and return the best solution found so far. Otherwise, CONTINUE.

Step 4) Repeat steps (5) to (8) for K_1 generations. This constitutes a single pass of the exploration stage.

Step 5) For each individual $\boldsymbol{x}_i$, i = 1, 2, …, M, find a set of M individuals that have maximum Euclidean distance from $\boldsymbol{x}_i$ in the D-dimensional search space. This is the set of 'strangers' for $\boldsymbol{x}_i$. Pick a stranger uniformly at random from this set for $\boldsymbol{x}_i$.

Step 6) Create M offspring by applying mutation on each individual $\boldsymbol{x}_i$, i = 1, 2, …, M, of the population. Each individual $\boldsymbol{x}_i$ creates a single offspring $\boldsymbol{x}_i'$ by the following procedure.

$$n_i = 1 + r_i \bmod D \tag{3.1}$$

$$\textbf{for } t = 1 \textit{ to } n_i \textbf{ do } \; x_i'(r_t) = x_i(r_t) + \pm_i(r_t) N_t(0,1) \tag{3.2}$$

Here, (3.1) picks a uniform random integer value for n_i from {1, 2, …, D}. n_i denotes the number of parameters (i.e., components) of $\boldsymbol{x}_i$ out of its D parameters that would be mutated by the subsequent **for** loop in (3.2). Each iteration of the **for** loop picks a random parameter of $\boldsymbol{x}_i$, uniformly from its D parameters, which is denoted as $x_i(r_t)$ in (3.2), then applies Gaussian mutation on it with mean = 0 and standard deviation (SD) set from the distance between $\boldsymbol{x}_i$ and one of its strangers, selected at step (5), along this randomly chosen parameter. To further clarify the notations, the r_i in (3.2) is a uniform random integer, r_t is a random parameter (i.e., gene value) that is being mutated with each iteration of the **for** loop, $N_t(0,1)$ is a normally distributed random number with mean = 0 and SD = 1 generated anew in each iteration of the **for** loop, and $\pm_i(r_t)$ is the SD for mutating the r_t-th component of $\boldsymbol{x}_i$. This SD value (i.e., $\pm_i(r_t)$) is set as the Euclidean distance along the r_t-th component of $\boldsymbol{x}_i$ and its selected stranger (say, $\boldsymbol{x}_j$), which makes $\pm_i(r_t) = \left| x_i(r_t) - x_j(r_t) \right|$. Thus, the component $\pm_i(r_t) N_t(0,1)$ in (3.2) produces a random value from a Gaussian distribution with mean = 0 and SD = $\pm_i(r_t)$, where $\pm_i(r_t)$ is set as the distance between the current individual $\boldsymbol{x}_i$ and one of its randomly chosen stranger in step (5) along the component currently being mutated (i.e., r_t -th component).

Step 7) Compute the fitness value of each offspring $\boldsymbol{x}_i'$, i = 1, 2, …, M. Select M individuals from parents and offspring for the next generation. If the fitness value of $\boldsymbol{x}_i'$ is at least equal to its parent $\boldsymbol{x}_i$, then discard $\boldsymbol{x}_i$ and select $\boldsymbol{x}_i'$ for the next generation. Otherwise, discard $\boldsymbol{x}_i'$.

Step 8) If the best fitness value found so far is acceptable or the maximum number of generations has been reached, then STOP. Otherwise, CONTINUE.

Step 9) Repeat steps (10) to (13) for K_2 generations. This constitutes a single pass of the exploitation stage.

Step 10) For each individual $\boldsymbol{x}_i$, i = 1, 2, …, M, find a set of M individuals that have minimum Euclidean distance from $\boldsymbol{x}_i$ in the D dimensional search space. This is the set of 'neighbors' for $\boldsymbol{x}_i$. Pick a neighbor uniformly at random from this set for $\boldsymbol{x}_i$.

Step 11) Create M offspring in the same way as described earlier in step (6). However, neighbors are involved here, instead of the strangers. More specifically, the SD (= $\pm_i(r_t)$) of the Gaussian mutation using (3.2) is now set from the distance along the corresponding (i.e., r_t-th) component between the current individual $\boldsymbol{x}_i$ and one of its randomly picked neighbor (rather than one of its strangers, as done by the exploration stage in step (6)).

Step 12) Compute the fitness value of each new offspring $\boldsymbol{x}_i'$, i = 1, 2, …, M, produced in the previous step (11). If the fitness value of the offspring solution $\boldsymbol{x}_i'$ is better than the parent $\boldsymbol{x}_i$, then discard the parent $\boldsymbol{x}_i$ and select the newer $\boldsymbol{x}_i'$ for the next generation. Otherwise discard $\boldsymbol{x}_i'$. In this way, select M individuals from the union of M parents and their M offspring for the next generation.

Step 13) If the best fitness value found so far by the population is acceptable or a pre-defined maximum number of generations has been reached, then STOP. Otherwise, go back to step (4) to start over another pass of exploration and exploitation stages.

Fig. 3.1 presents the flowchart of RTEP, which shows that the essence of RTEP is its three components, i.e., the exploration stage, the exploitation stage and their repeated and recurring execution. Next, the pseudocode of RTEP is presented in Fig. 3.2, which is followed by a detailed description of the exploration stage (subsection 3.3.1), the exploitation stage (subsection 3.3.2) and the justification for the recurring execution of the exploration and exploitation stages (subsection 3.3.3) for RTEP.

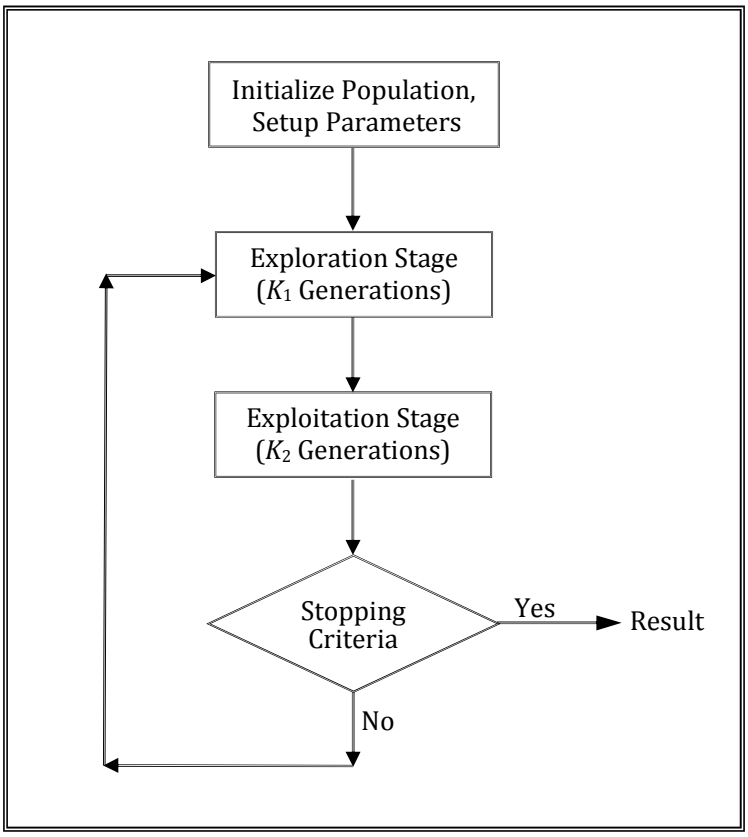


Figure 3.1: Flowchart of RTEP

3.3.1 Exploration Stage

This stage facilitates the exploration of wider regions of a search space so that the chance of finding a good near optimum solution by exploration is increased. Since exploration is a non-local operation, a mutation operation that can produce farther offspring and increase the population diversity is suitable for this stage. In RTEP, the Euclidean distance between the genotype of $\boldsymbol{x}_i$ and one of its strangers along a randomly selected search dimension is used as the SD to mutate $\boldsymbol{x}_i$ along that dimension. To pick a stranger of the current individual $\boldsymbol{x}_i$, RTEP determines the M individuals across the population whose genotypes have the largest Euclidean distance from the genotype of $\boldsymbol{x}_i$ and then picks one of them, say $\boldsymbol{x}_i$, uniformly at random. Since the individuals $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are relatively far apart in the search space, the large Euclidean distance between them could be used as the SD for the explorative mutation.

Algorithm 3.1: Recurring Two-Stage Evolutionary Programming (RTEP)

begin

1. Generate the Initial Population. Initialize Parameters K_1 and K_2

2. **for** K_1 generations **do** **[Exploration Stage]**

Update the set of strangers S_i for every candidate solution $\boldsymbol{x}_i$

for every individual $\boldsymbol{x}_i$ **do**

n_i = Random integer picked uniformly from {1, 2, …, D}

$\boldsymbol{x}_j$ = A stranger of $\boldsymbol{x}_i$, picked uniformly at random from the stranger set S_i

Offspring $\boldsymbol{x}_i' = \boldsymbol{x}_i$

for t = 1 to n_i **do**

r_t = An attribute (gene) of $\boldsymbol{x}_i$, picked uniformly at random from {1, 2, …, D}

Compute the standard deviation of Gaussian mutation: $\sigma_i(r_t) = \left| x_i(r_t) - x_j(r_t) \right|$

Mutate the r_t-th gene of $\boldsymbol{x}_i$ as: $x_i'(r_t) = x_i(r_t) + \sigma_i(r_t) N_t(0,1)$

enddo

if the fitness value of the offspring $\boldsymbol{x}_i'$ is at least equal to the fitness of the parent $\boldsymbol{x}_i$, **then** set $\boldsymbol{x}_i = \boldsymbol{x}_i'$ to accept $\boldsymbol{x}_i'$ and discard the parent $\boldsymbol{x}_i$. Otherwise, discard $\boldsymbol{x}_i'$.

enddo

enddo

3. **if** the best solution found is acceptable or the maximum number of generations has been elapsed **then** conclude RTEP and **GOTO** step 6. **Else** CONTINUE.

4. **for** K_2 generations **do** **[Exploitation Stage]**

Update the set of neighbors, T_i every candidate solution, $\boldsymbol{x}_i$

for every individual, $\boldsymbol{x}_i$ **do**

n_i = Random integer picked uniformly from {1, 2, …, D}

$\boldsymbol{x}_j$ = A neighbor of $\boldsymbol{x}_i$, picked uniformly at random from the neighbor set T_i

Offspring $\boldsymbol{x}_i' = \boldsymbol{x}_i$

for t = 1 to n_i **do**

r_t = An attribute (gene) of $\boldsymbol{x}_i$, picked uniformly at random from {1, 2, …, D}

Compute standard deviation of Gaussian mutation as: $\sigma_i(r_t) = \left| x_i(r_t) - x_j(r_t) \right|$

Mutate the r_t-th gene of $\boldsymbol{x}_i$ as: $x_i'(r_t) = x_i(r_t) + \sigma_i(r_t) N_t(0,1)$

enddo

if fitness($\boldsymbol{x}_i'$) > fitness($\boldsymbol{x}_i$) **then** $\boldsymbol{x}_i = \boldsymbol{x}_i'$

enddo

enddo

5. **if** the best solution found is acceptable or the maximum number of generations has been elapsed **then** conclude RTEP and **GOTO** step 6. **Else** return to step 2 to start another cycle of exploration and exploitation stages.

6. **return** the best individual found so far and conclude RTEP

end

Figure 3.2: Pseudocode of RTEP

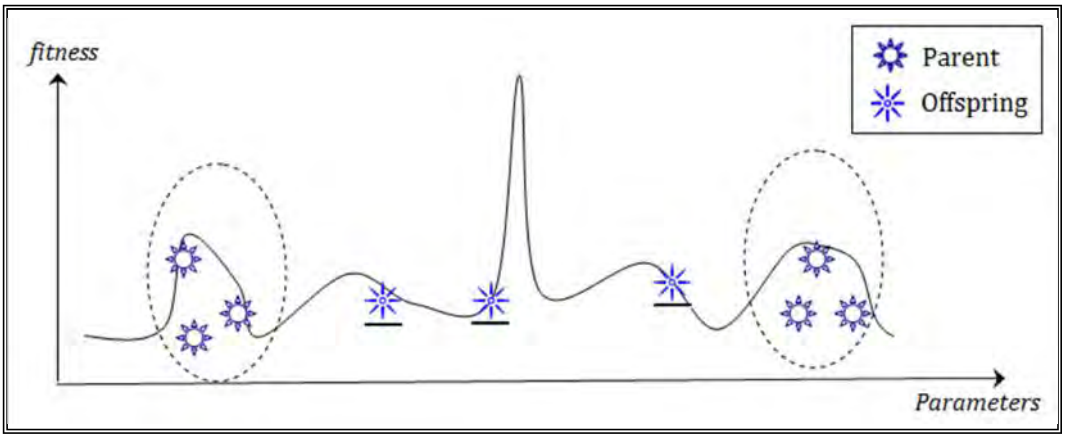


Figure 3.3: Exploration through the fitness landscape

The scenario presented in Fig. 3.3 exemplifies how mutation involving the distance of dissimilar individuals may facilitate explorations. The oval boundaries in Fig. 3.3 represent two groups of individuals that are far apart in the search space and hence 'stranger's to each-other. When RTEP mutates an individual $\boldsymbol{x}_i$ from any of the groups using the Euclidean distance between the genotype of $\boldsymbol{x}_i$ and one of its stranger (i.e., another individual from the other group), some offspring (marked as underlined stars) may be produced in between the two groups. At first, they might seem to be quite similar to their parent $\boldsymbol{x}_i$ in terms of fitness, but a small amount of subsequent hill climbing steps by the exploitative operations of the subsequent exploitation stage (subsection 3.3.2) might reach them to the narrow global optimum (i.e., the narrow peak or maximum of the fitness landscape). Thus exploration of the intermediate search space becomes possible because of combining information from the dissimilar and distant individuals across the search space.

3.3.2 Exploitation Stage

The evolutionary approach may discover some promising regions by executing exploration operations several times. But it is often necessary to realize the potentials of the newly discovered regions before further explorations. To achieve this objective, RTEP executes the exploitation stage after the completion of the exploration stage. The aim of the exploitation stage is to reach peaks of the different explored regions so that the optimum solution, if exists in close proximity, can easily be found.

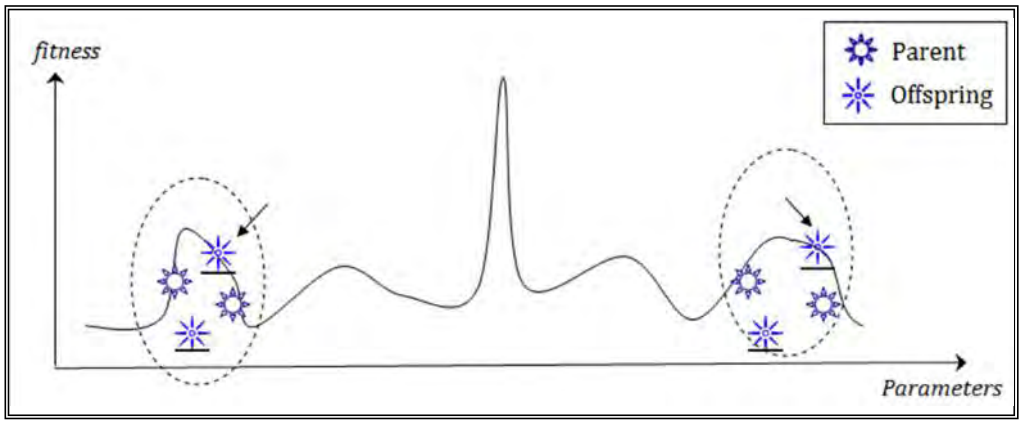


Figure 3.4: Exploitation of the fitness landscape to reach the locally optimal peaks

The exploitation stage uses the distances among similar genotypes (i.e., neighbors) to induce small, exploitative mutations on the candidate solutions. To find a neighbor for the current individual (i.e., candidate solution) $\boldsymbol{x}_i$, RTEP determines M other individuals across the population that have the minimum Euclidean distance of their genotypes from the genotype of $\boldsymbol{x}_i$, and one of them is picked at random as the neighbor of $\boldsymbol{x}_i$. The mutation used for exploitation is the same Gaussian mutation as used for exploration stage by employing the (3.1) and (3.2). However, the only difference is that the Euclidean distance (ED) between the neighbor individuals is used as the standard deviation (SD) for the Gaussian distribution, instead of the distance between two strangers, as used by the exploration stage. It is expected that the neighbors would have much smaller ED, resulting in a small SD that is suitable for producing small, exploitative perturbation steps from the Gaussian distribution.

The scenario presented in Fig. 3.4 demonstrates two different neighborhoods, marked by the oval boundaries. As each of the four parent individuals is exploitatively mutated using its distance from its neighbor (i.e., another individual from the same neighborhood), the four offspring individuals (marked by underline) are likely to be produced within the same local neighborhood. However, the re-insertion policy during the exploitation stage is based on fitness improvement, so only the two fitter offspring (marked by the arrows) are allowed to enter into the population replacing the worse parents. This shows that both the neighborhoods move closer to the peaks of their local fitness hills, which is the objective of the exploitation stage.

3.3.3 Recurring Approach

It is well known that pure EAs are not suitable for fine tuning a search in complex search spaces, and the hybridization of EAs with other methods can greatly improve the search efficiency [210], [211]. A number of approaches have been proposed in the literature which uses GAs [211] for exploration and local search methods for exploitation. According to [212],

the following four issues must be addressed when exploration and exploitation operations are executed separately and then combined in one algorithm. First, when and where should a local search method be applied within the evolutionary cycle? Second, which individuals in the population should be improved by the local search, and how should they be chosen? Third, how much computational effort should be allocated to each local search? Fourth, how can genetic operators be best integrated with the local search in order to achieve a synergistic effect? To address these questions, a number of heuristics and parameters may need to be employed in any classical EA. However, this requires a user to have rich prior knowledge, which often does not exist for complex real-world problems. Hence, a scheme that does not employ many heuristics and user-specified parameters is clearly preferred. The regularly repeated and alternating execution of exploration and exploitation operations on all individuals of the population could be a simple solution for the first three questions, which is adopted in RTEP. Since RTEP uses only mutation for both explorations and exploitations, the problem of integrating different methods or operators does not arise. The solution quality and convergence characteristics of RTEP on a wide range of benchmark functions (sections 3.6, section 8.3, 8.8) show the effectiveness of the aforementioned intuition.

As mentioned previously, RTEP uses the genotype distance between dissimilar and similar individuals along the selected components as the SD in mutating the corresponding components of the individuals in order to realize exploration and exploitation objectives. This makes the exploration and exploitation operations self-adaptive. The degree of exploration is high at the beginning of the evolutionary process and gradually decreases as the process progresses. A similar scenario occurs for exploitative mutations, which start with a medium step size and become very much fine-tuned during the late generations. These are very much possible due to using the distance as the SD of mutation, because distances among individuals are usually high in early generations and gradually drops reflecting the maturity of the search process.

3.4 Differences of RTEP with Other Existing Works

RTEP differs from most other EP-based [3] approaches (e.g., [56], [57]) and memetic algorithms (e.g., [160], [162]) in a number of ways. First, RTEP emphasizes repeated and alternated objective-oriented mutation and selection operations for achieving global exploration and local exploitation goals. The essence of this approach is that when an evolutionary process is trapped into a deep local optimum or finds a very promising region, the repeated execution of objective-oriented operations helps to handle the situation effectively. Since exploration and exploitation operations are executed alternatively in RTEP, there is no need to make a “perfect switching” from one operation to another. The conflicting goals of exploration and exploitation

are expected to be distributed automatically across the generations of the recurring operations. This approach is different from the one used in EP, ES and memetic algorithms. Most EP and ES-based approaches do not execute exploration and exploitation operations separately. Rather, they often use a single stage execution model with self-adaptation rules (e.g., [56], [57], [213], [214]). Even recently introduced and more sophisticated ES schemes, like Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [215], follow a single stage execution model and are usually focused on facilitating more successful mutations by adapting the mutation step size effectively and more or less ignoring the entirely different and often conflicting goals of exploration and exploitation that arise again and again throughout the evolutionary search process. On the other hand, many memetic algorithms, though they take different measures for the conflicting explorative and exploitative goals, usually execute the exploration and exploitation stages only once, one after the other, but not repeatedly (e.g., [212], [216]). Most multimodal benchmark functions have numerous local optima, so repeating the exploration and exploitation stages only once, rather than repeatedly as RTEP does, has the inherent danger of being trapped in local optima, failing to reach the neighborhood of the global optimum.

Second, RTEP uses only the mutation operator for both exploration and exploitation. Although there are many algorithms that use GAs [1] for exploration and local search methods or a specially designed operator for exploitation (see the review paper [212] and the references therein), RTEP is to our knowledge the first population based metaheuristic algorithm that uses only mutation both for exploration and exploitation. Both explorative and exploitative objectives can be achieved using mutation with a large and a small step size, respectively. There are some mutation-only algorithms that we have used later in this chapter for comparison with RTEP. However, they either do not separate exploitations from explorations (e.g., classical EP [3], improved fast EP (IFEP) [57], adaptive EP with Lévy mutation (ALEP) [56]), or use some specialized operator other than mutation (e.g., crossover hill climbing (XHC) used by real-coded memetic algorithm (RCMA) [160], neighborhood search operator used by NSDE [163], adaptive local search operator used by LSRCMA [162]) for exploitations.

Third, RTEP is also significantly different from Simulated Annealing (SA) [233], which also uses large and small step sizes for explorations and exploitations, respectively. SA is a single state algorithm, i.e., it always maintains a single candidate solution. Besides, SA controls its step sizes probabilistically with little or no control by the user. In contrast, RTEP maintains a population of candidate solutions and the user can control, more or less, the degree of explorations and exploitations by setting the control parameters K_1 and K_2. Besides, RTEP always ensures some degree of explorations, even during the final generations, while SA usually becomes too much exploitative, making only uphill moves, during its final iterations.

Fourth, RTEP uses the distance between similar and dissimilar individuals to employ exploitative and explorative mutation, respectively. Some other existing works, like DE [163], [164] also employ the distance between individuals for mutation. However, RTEP is still significantly different from DE because of its recurring nature of explorations and exploitations. DE does not consider exploitations and explorations separately and does not follow any recurring behavior. RTEP has been compared with NSDE [163], which is a recently introduced more sophisticated variant of DE. Results on recent benchmark functions, presented later in this chapter (Tables 3.6 and 3.7), exhibit that RTEP often performs better than NSDE, which indicates the effectiveness of the recurring explorations and exploitations of RTEP over the traditional single stage execution model of the DE variants.

Fifth, RTEP has been theoretically analyzed and empirically tested on as many as 55 benchmark test functions, consisting of 30 standard test functions ([55]–[68], [162], [164], [196]) and 25 test functions introduced very recently at CEC2005 [76]. Few evolutionary approaches have been tested on a similar range of benchmark problems with different characteristics. Results show that RTEP often produces better solutions than most other algorithms on most of the functions.

3.5 Analysis of RTEP

The aim of this section is to analyze RTEP based on search bias and exploration/exploitation operations. When a search operator (e.g., crossover or mutation) is applied to the individuals, some offspring are more likely to be generated than others. This tendency is called search bias and has great impact on the performance of EAs. The search bias includes search step size and search directions. Since RTEP uses only mutation for both explorations and exploitations, the following analysis presented is carried out only for mutation. The standard deviation (SD) used in (3.2) determines the search step size of mutation. The probability density function of the Gaussian distribution used in the mutation is:

$$f_{(0,\pm^2)}(x) = \frac{1}{\pm\sqrt{2®}}\ e^{\frac{-x^2}{2\pm^2}} \tag{3.3}$$

where 0 is the expected value; and σ is the SD. RTEP uses the distance between two individuals along a selected search dimension as the σ in (3.3) to mutate the corresponding gene (i.e., component) of the current individual (i.e., chromosome). However, how does σ affect the mutation step size? To find the analytical relationship between σ and step size, we can generalize the analysis method for the mean search step size in [57]. The expected value of mutation step size for producing offspring is computed as follows.

$$\mathrm{E}\left[|x|\right] = \int_{-\infty}^{+\infty} |x| \frac{1}{\pm\sqrt{2\pi}} e^{\frac{-x^2}{2\pm^2}} dx$$
$$= 2\int_{0}^{+\infty} |x| \frac{1}{\pm\sqrt{2\pi}} e^{\frac{-x^2}{2\pm^2}} dx$$
$$= \pm\sqrt{\frac{2}{\pi}} \tag{3.4}$$

Equation (3.4) tells us that the mean step size is directly proportional to the standard deviation σ. To visualize the effect of σ, the Gaussian probability density function used for introducing mutation steps is plotted for different values of σ in the same scale (Fig. 3.5). It shows that for a large σ, the density function distribution exhibits a central maximum with a long flat tail which is more likely to introduce large variations (i.e., longer jumps) for producing offspring and thus facilitating global explorations. Similarly, a large central maximum with a small flat tail, which is obtained for smaller values of σ, is suitable for producing small steps around the mean, which is necessary for local exploitations. Thus, an optimal value of σ is necessary to facilitate proper exploration and exploitation. However, a suitable value for σ is problem dependent. Even for a single problem, a separate σ may be required for each component (i.e., search dimension) of the individuals during different stages of the evolutionary process. A large σ is beneficial when the distance between the neighborhood of the optimal point and the current search point is larger than σ [57]. As the global optimum is unknown, adapting σ during the course of evolution becomes necessary. However, the self-adaptation scheme of σ described in [3], [102] is partial and often does not work satisfactorily well [217].

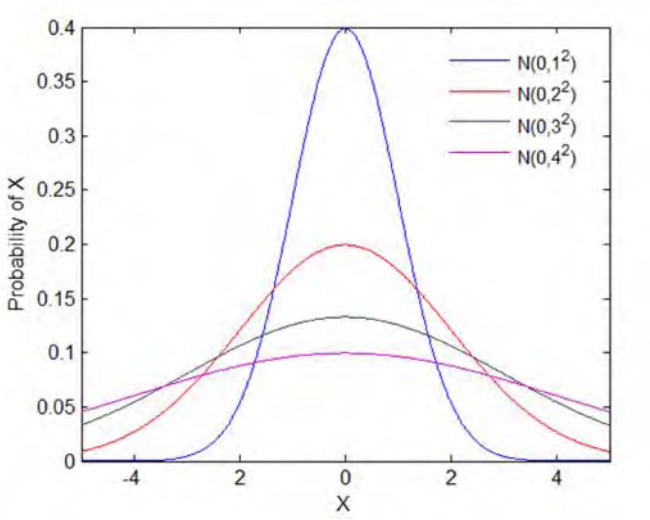


Figure 3.5: Gaussian distributions with mean=0 standard deviations set to 1 to 4

In addition to the search step size, its precise direction is also important. If an optimal step size is found but the right search direction is unknown, the evolutionary process is likely to face difficulty. To address this issue, RTEP incorporates the directional information in mutation by using the differences between corresponding components of two similar or dissimilar individuals. More particularly, RTEP randomly picks n_i (out of D) components of the current individual $\boldsymbol{x}_i$ and each component is mutated using the distance along this dimension between $\boldsymbol{x}_i$ and a similar individual (i.e. neighbor, for exploitations) or a dissimilar individual (i.e. stranger, for explorations). This is illustrated in the pseudocode of RTEP (the inner **for** loops of step 2 and step 4 in Fig. 3.2). Using multiple components provides direction information along each component and the offspring is likely to be produced in between the parents. Such an approach has several advantages. First, it provides an effective search direction by which mutation may produce offspring that can be non-dominated or dominated by individuals in the current population. This would increase the efficacy of exploration or exploitation. Second, it makes mutation operations self-adaptive without using any adaptation scheme. The individuals in a population are usually widely spread over the entire search space at the beginning of the evolutionary process. As the evolution progresses, the population converges around the optimal solutions, and the distance among individuals tend to decrease. This means mutation SD tends to be larger during early generations and smaller near the end of the evolutionary process. Hence, the incorporation of such self-adaptive directional information in mutation is inherently suitable for the search process. Third, such an approach relieves the users of the burden of specifying initial standard deviations for mutation.

Now, we analyze the effect of repeated alteration of explorations and exploitations. Let σ_1 and σ_2 be the standard deviations used by mutation for explorations and exploitations, respectively, with $\sigma_1 \geq \sigma_2$. We denote $\boldsymbol{x}^0$ as a parent individual and $\boldsymbol{x}^g$ as its offspring obtained after g successive generations. The expected value of total variation $\boldsymbol{x}^g - \boldsymbol{x}^0$, introduced by the successive explorative mutations for K_1 generations or exploitative mutations for K_2 generations or explorations followed by exploitations are given by following equations.

$$E(d_1) = \sum_{i=1}^{g} \pm_1(i)\sqrt{\frac{2}{\pi}} = gE(\pm_1)\sqrt{\frac{2}{\pi}} = cgE(\pm_1) \tag{3.5}$$

$$E(d_2) = \sum_{i=1}^{g} \pm_2(i)\sqrt{\frac{2}{\pi}} = gE(\pm_2)\sqrt{\frac{2}{\pi}} = cgE(\pm_2) \tag{3.6}$$

$$E(d_3) = \sum_{i=1}^{g/2} \pm_1(i)\sqrt{\frac{2}{\circledR}} + \sum_{i=1}^{g/2} \pm_2(i)\sqrt{\frac{2}{\circledR}}$$
$$= \frac{g}{2}\left[E(\pm_1) + E(\pm_2)\right]\sqrt{\frac{2}{\circledR}}$$
$$= cg\frac{E(\pm_1) + E(\pm_2)}{2} \quad (3.7)$$

where $E(d_1)$, $E(d_2)$, and $E(d_3)$ are the expected value of total variations by successive explorations, successive exploitations, and explorations followed by exploitations, respectively. $E(\sigma_1)$ and $E(\sigma_2)$ are the mean step length of explorative and exploitative mutations. Since $E(\sigma_1)$ is larger than $E(\sigma_2)$, $E(d_1)$ is larger than $E(d_3)$, while $E(d_2)$ is smaller than $E(d_3)$. Hence, the repeated execution of exploration (or, exploitation) operations introduces more (or, less) variations than that introduced by exploration followed by exploitation. It may be argued that employing a single-stage EA with expected step size $E(\sigma_3) = (E(\sigma_1) + E(\sigma_2))/2$ would achieve a similar effect. However, finding an appropriate value for σ_3 by evaluating the population is not easy, while finding suitable σ_1 and σ_2 is fairly straightforward to determine using the distance between strangers and neighbors. The search space of most benchmark multimodal functions possesses alternating peaks and valleys, so periodically alternating between explorations and exploitations has an intuitive appeal to prove effective for advancing the search by alternating downhill and uphill moves across the search space. Experimental results presented in a later chapter (Tables 8.11–8.12, Fig. 8.4 in chapter 8) demonstrate that executing explorative and exploitative stages sequentially (i.e., one after another) fail to yield sufficiently good results in comparison to the proposed recurring approach with alternating explorations and exploitations. Since the depths of peaks and valleys in the fitness landscape are not known in advance, the repetition of explorative and exploitative steps for some length (i.e., K_1 and K_2, respectively) seems to be a good choice for automatically handling the locally optimal points and the promising new regions, respectively of the search space.

3.6 Evaluation of RTEP on Benchmark Functions

This section presents the evaluation and performance comparison of RTEP which would help to achieve a better understanding of how the repetitions and alternations of the explorative and exploitative operations of RTEP, based on their different mutation strengths, can influence and improve the performance of the evolutionary algorithms. We have tested RTEP on two different suites of benchmark functions — the standard benchmark suite consisting of 30 standard functions ([11], [162]–[164], [196]) and the recently introduced CEC2005 benchmark suite [76]. Both the suites are presented briefly in section 2.17, and more elaborately in the Appendix A.

3.6.1 Standard Benchmark Functions

In this section, the standard benchmark suite, consisting of 30 benchmark test functions, is used to evaluate and experiment with RTEP. Each function is briefly introduced in Table 2.3 and section 2.17. More information on these functions, including their analytical forms and 3D surface plots can be found in the Appendix A. Based on their properties, the functions can be divided into three groups, namely — the unimodal functions (i.e., the functions f_1–f_9 in Table 2.3 that have no local minima aside from their single global minimum), the high dimensional multimodal functions (i.e., the functions f_{10}–f_{18} that have exponentially many locally minimal points in addition to a single global minimum) and the low dimensional multimodal functions (i.e., the functions f_{19}–f_{30} that have dimensionality ≤ 10 and possess only a few local minima).

3.6.2 Results of RTEP on Standard Benchmark Functions

For the results in Table 3.1, the parameters of RTEP are set as follows. The population size M is set to 50. The neighborhood size M is set at 10. Three different sets of values are used to test the effect of the length of exploration and exploitation stages, i.e., K_1 and K_2, respectively. They are set as $(K_1, K_2) = (1, 1)$, $(2, 4)$ and $(4, 8)$. The number of function evaluations (FEs) is set to be 150,000 for the high dimensional functions f_1–f_{18} and to be 10,000 for the low dimensional functions f_{19}–f_{30}. All these values are chosen to make a fair comparison with some of the existing previous works, e.g., CEP [102], ALEP [56], IFEP [57] and RMEA [218].

Table 3.1 shows the mean error of RTEP on the 30 standard test functions over 50 independent runs. The numbers inside the parentheses next to RTEP indicate the values of K_1 and K_2 used in the evaluation. For each function, the best (i.e., lowest) mean error value is shown in boldface font. Fig. 3.6 shows the convergence characteristics of RTEP for several functions, each with three different settings for (K_1, K_2), as $(K_1, K_2) = (1, 1)$, $(2, 4)$ and $(4, 8)$. The following observations can easily be made from the results in Table 3.1 and Figs. 3.6 and 3.7.

a) First, RTEP with different values for (K_1, K_2) have reached the proximity of the global minimum (i.e., mean error = 0) for almost all the functions. This signifies the essence of recurring exploration and exploitation operations for improving the performance of EAs based on mutation. The mutation of RTEP involves directional information and simple selection strategies, provides effective exploration and exploitation, which is evident from the convergence characteristics shown in Fig. 3.6. It is seen that RTEP with different values for K_1 and K_2 have achieved nearly log-linear convergence and reached sufficiently close to the global minima very consistently for almost all the functions.

Table 3.1: Performance of RTEP on the 30 standard benchmark functions, for different values of the parameters K_1 and K_2. Results have been averaged over 50 independent runs. The best results are marked with boldface font. A '+' or '–' in the *t*-test between algorithms *X* versus *Y* indicate that *X* is significantly better or worse, respectively than *Y* with 95% certainty, while a '≈' means that the difference is not statistically significant.

Function	Mean Error ± Standard Deviation			*t*-Test (RTEP vs.)	
	RTEP (1,1)	RTEP (2,4)	RTEP (4,8)	RTEP (2,4) vs. (1,1)	RTEP (4,8) vs. (1,1)
f_1	3.1e–18 ± 3.4e–18	7.5e–18 ± 4.4e–18	**2.4e–20 ± 7.4e–21**	≈	+
f_2	9.8e–08 ± 6.0e–08	1.7e–09 ± 1.5e–09	**2.9e–12 ± 6.8e–13**	≈	+
f_3	**1.0e+00 ± 4.5e–01**	1.6e+00 ± 6.2e–01	1.9e+00 ± 1.4e+00	–	–
f_4	7.2e–14 ± 2.4e–14	2.4e–15 ± 6.2e–16	**2.1e–15 ± 4.1e–16**	+	+
f_5	4.9e–01 ± 8.4e–02	2.6e–03 ± 7.7e–04	**2.0e–03 ± 4.9–e04**	+	+
f_6	2.8e–01 ± 3.3e–02	1.3e–01 ± 3.9e–02	**1.2e–01 ± 3.5e–02**	+	+
f_7	1.5e+01 ± 7.0e+00	**1.1e+00 ± 9.0e–01**	1.5e+00 ± 6.6e–01	+	+
f_8	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	≈	≈
f_9	5.3e–34 ± 1.5e–35	**8.0e–38 ± 6.4e–39**	2.5e–34 ± 1.4e–35	+	+
f_{10}	1.0e–12 ± 7.5e–13	2.5e–14 ± 5.0e–15	**1.9e–14 ± 6.1e–15**	+	+
f_{11}	2.1e–03 ± 5.7e–04	2.9e–07 ± 5.1e–08	**1.1e–06 ± 9.1e–08**	+	+
f_{12}	4.3e+03 ± 8.5e+02	7.1e+02 ± 4.9e+02	**3.6e+02 ± 9.9e+01**	+	+
f_{13}	6.1e–09 ± 3.5e–09	**2.0e–11 ± 6.5e–12**	2.4e–09 ± 9.2e–10	+	+
f_{14}	3.4e–17 ± 9.0e–18	**2.7e–25 ± 6.3e–26**	8.4e–20 ± 3.3e–20	+	+
f_{15}	3.0e–09 ± 9.2e–10	7.8e–10 ± 9.4e–11	**2.9e–12 ± 4.1e–13**	+	+
f_{16}	4.8e–06 ± 3.5e–06	2.2e–07 ± 9.1e–08	**6.1e–08 ± 2.1e–08**	+	+
f_{17}	1.7e–10 ± 5.4e–11	3.2e–13 ± 8.5e–14	**1.7e–13 ± 2.7e–14**	+	+
f_{18}	9.2e–03 ± 3.1e–03	**7.1e–08 ± 7.3e–09**	7.2e–05 ± 2.2e–05	+	+
f_{19}	0.002 ± 7.0e–04	0.002 ± 5.8e–04	0.002 ± 3.3e–05	≈	≈
f_{20}	0.0007 ± 2.9e–04	**0.0004 ± 6.2e–07**	0.0008 ± 3.6e–04	+	≈
f_{21}	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	≈	≈
f_{22}	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	≈	≈
f_{23}	0.04 ± 0.03	0.03 ± 0.02	**0.00 ± 0.00**	≈	+
f_{24}	0 ± 0	0 ± 0	0 ± 0	≈	≈
f_{25}	0.89 ± 0.21	0.55 ± 0.08	**0.39 ± 0.10**	+	+
f_{26}	0.42 ± 0.16	**0.38 ± 0.12**	0.40 ± 0.12	≈	≈
f_{27}	0.27 ± 0.13	0.25 ± 0.10	**0.22 ± 0.10**	≈	≈
f_{28}	0.93 ± 0.32	0.85 ± 0.20	**0.42 ± 0.11**	≈	+
f_{29}	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	≈	≈
f_{30}	0.48 ± 0.16	0.29 ± 0.09	**0.26 ± 0.08**	+	+

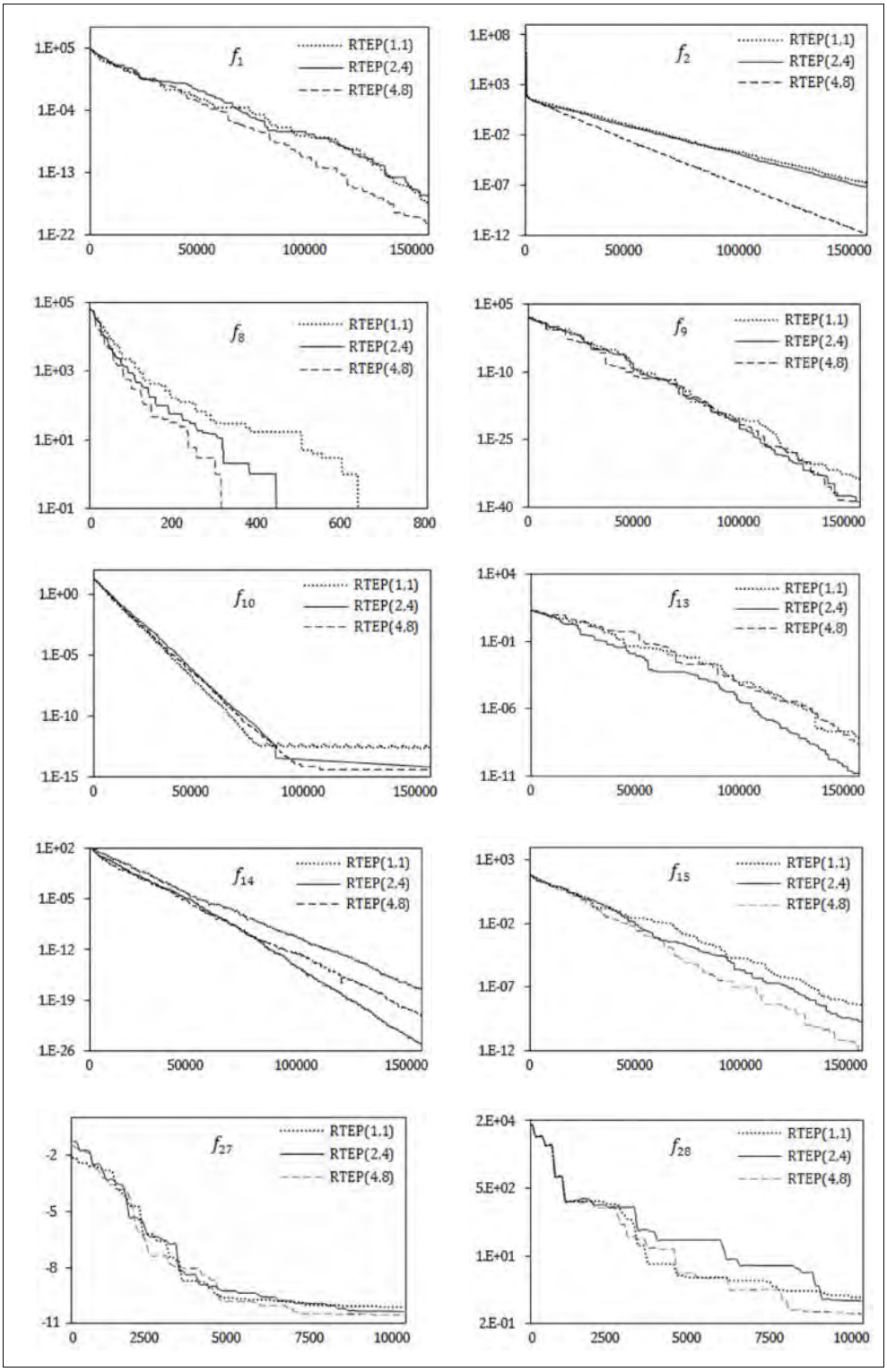


Figure 3.6: The Convergence characteristics of RTEP(1,1), RTEP(2,4) and RTEP(4,8) on four unimodal (f_1, f_2, f_8, f_9) and six multimodal (f_{10}, f_{13}–f_{15}, f_{27}, f_{28}) functions. The vertical axis shows the function value, while the horizontal axis shows the number of function evaluations.

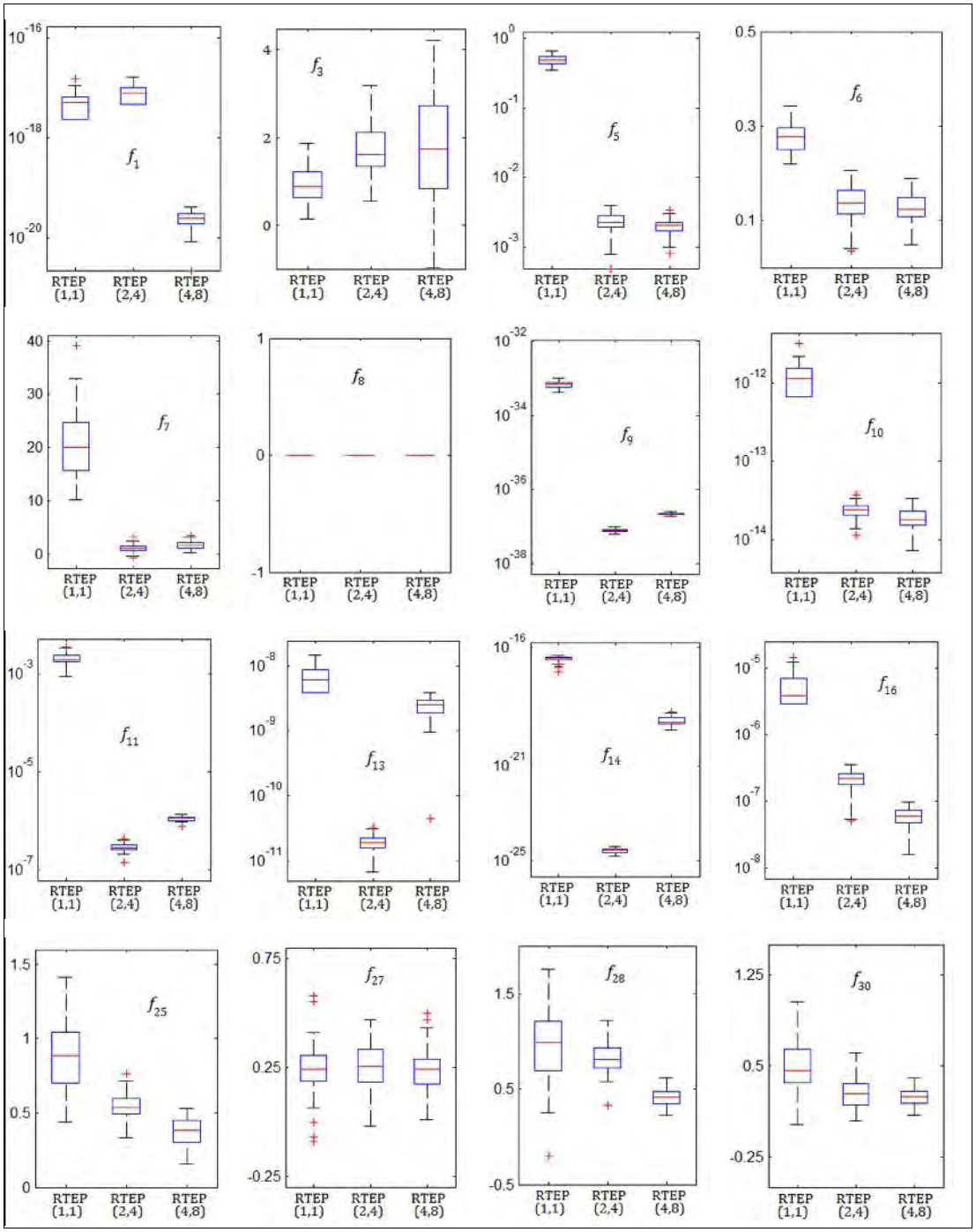


Figure 3.7: Box plots showing the distribution of the final results by the RTEP variants — RTEP(1,1), RTEP(2,4) and RTEP(4,8), over their 50 independent runs. In each box plot, the vertical box corresponds to the results from the 25th percentile to the 75th percentile, the central red dash marks their mean value, the upper and lower whiskers (i.e., the $\top$ and $\perp$ symbols) show the entire range of values, and the red '+' symbols outside the range of whiskers show the outliers.

b) Second, both RTEP(2,4) and RTEP(4,8) performed significantly better than RTEP(1,1) on as many as 17 and 20 functions, respectively, while RTEP(1,1) outperformed RTEP(2,4) and RTEP(4,8) only on two functions each (f_1, f_3 and f_3, f_{20}, respectively). This indicates the necessity of executing exploration and exploitation operations at some length. The *t*-test shows that both RTEP(2,4) and RTEP(4,8) performed better than RTEP(1,1) on all of the nine high-dimensional multimodal functions, which are considered the most challenging family of functions to optimize. Since the low-dimensional functions, f_{19}–f_{30} are relatively easier to optimize, RTEP(1,1) showed similar performance to RTEP(2,4) and RTEP(4,8) on most of these functions.

c) Third, the convergence characteristics of RTEP with different values for K_1 and K_2 seem to be quite similar for some of functions (e.g., f_1, f_9, f_{13}, f_{27}) along the entire evolutionary process (Fig. 3.6). However, RTEP(4,8) shows the best overall convergence characteristics, in terms of both convergence speed (e.g., f_1, f_2, f_8, f_{13}–f_{15}, f_{28}) and the final solution quality (e.g., f_1, f_2, f_{10}, f_{15}, f_{27},). However, some little oscillations are observed at the later stage of the evolutionary process, (e.g., f_{13}, f_{15}). This may be due to the non-optimal setting of K_1 and K_2, and also because of the exploration stage involving mutations with large step sizes.

d) Fourth, the low standard deviation of the results in Table 3.1 by the RTEP variants indicates that RTEP is very consistent and robust, across their 50 independent runs, for almost all the functions. On average, the magnitude of the standard deviation is only around 38%, 32% and 28% of the mean results from RTEP(1,1), RTEP(2,4) and RTEP(4,8), respectively. The distribution of the results by the three RTEP variants over their 50 different runs can be viewed as box plots, as shown by Fig. 3.7 for several randomly selected functions. In each box plot, the vertical box corresponds to the results from the 25th percentile to the 75th percentile, the central red dash marks the mean, the upper and lower whiskers (the $\top$ and $\perp$ symbols, respectively) show the range of values, and the red '+' symbols outside the range (whiskers) show the outliers. The squeezed size of most of the boxes, as well as the very few occurrence of the outliers for most of the functions in Fig. 3.7 (except for only a few box plots, e.g., RTEP(4,8) in f_3) indicates the high degree of consistency, stability and robustness of the RTEP variants for any standard benchmark function in f_1–f_{30}.

3.6.3 Comparison of RTEP with Existing Works

Many approaches exist in the literature against which one could compare the present work. However, it is infeasible and unnecessary to conduct an exhaustive comparison with all algorithms. The aim of our comparison here is to understand the strengths and weaknesses of RTEP. Since the proposed algorithm uses only mutation as a variation operator and executes exploration/exploitation operations separately, we have considered IFEP [57], ALEP [56], CEP [102], RCMA with XHC [160] and recurring multistage EA (RMEA) [218] for comparison. In addition, RTEP is also compared with some basic and representative evolutionary and swarm intelligence algorithms, such as GA [11], DE [195], PSO [196], ABC [75] and a hybrid algorithm (i.e., CLPSO [69]). RMEA is a previous research work by us that is founded on a similar philosophy of RTEP, but a number of key differences make RTEP perform significantly better than RMEA (Table 3.3). IFEP [57] and ALEP [56] are both based on mutation only, while RCMA [160] employs both mutation and crossover for exploration and exploitation, respectively. CLPSO [69] is a hybrid algorithm that combines machine learning techniques with PSO, but does not consider considers explorations and exploitations separately. In the next subsection (section 3.6.4), RTEP is also compared with NSDE and LSRCMA, both of which alter their mutation operation by incorporating specialized exploitative operators for better local exploitation around the candidate solutions.

At first, RTEP is compared with GA [11], DE [195], PSO [196] and ABC [75] in Table 3.2. All these algorithms have two parameters in common — the population size μ and the total number of function evaluations (FE), which are set as $\mu = 50$ and FE = 500,000 for all the functions in Table 3.2. The other algorithm specific parameters are given below.

GA Settings: We have used binary coded standard GA with fitness scaling, seeded selection, random selection, crossover, mutation and elitism. Single point crossover operation with the rate of 0.8 and bit flip mutation with mutation rate of 0.01 is used in the experiments. As the selection operator, stochastic uniform sampling technique has been used. Generation gap is the proportion of the population that is to be replaced in each generation, which is set to 0.9 in this comparison.

DE Settings: The standard DE has two parameters — the scaling factor F and the crossover rate CR. In DE, the parameter F operates as the magnification factor of the differential variation between two candidate solutions, while CR controls how much new information is incorporated into the new trial solutions, which in turn affects the change of the diversity of the population. We have used $F = 0.5$ and $CR = 0.9$, as recommended in [195].

Table 3.2: Comparison of RTEP variants with GA [11], PSO [196], DE [195] and ABC [11] on the 17 standard benchmark functions. Results have been averaged over 50 independent runs. The best results are marked with boldface font.

No.	D	f_{min}	Mean Error					
			GA	**PSO**	**DE**	**ABC**	**RTEP (2,4)**	**RTEP (4,8)**
f_1	30	0	1.1e+03	**0**	**0**	**0**	**0**	**0**
f_2	30	0	11.02	**0**	**0**	**0**	**0**	**0**
f_4	30	0	7.4e+03	**0**	**0**	**0**	**0**	**0**
f_5	24	0	9.70	1.1e–04	2.2e–07	3.1e–03	9.5e–13	**2.7e–15**
f_6	30	0	1.2e+03	0.6666	0.6666	**0**	**0**	**0**
f_7	30	0	1.9e+05	15.08	18.20	0.088	4.6e–03	**1.6e–03**
f_8	30	0	1.2e+03	**0**	**0**	**0**	**0**	**0**
f_9	30	0	1.8e–01	1.2e–03	1.4e–03	3.00e–03	**0**	**0**
f_{10}	30	0	52.92	43.97	11.72	**0**	**0**	**0**
f_{12}	30	–12569.48	8.8e+02	5.7e+03	2.3e+03	**0**	**2.9e–01**	6.6e–01
f_{13}	30	0	14.67	0.16	**0**	**0**	**0**	**0**
f_{14}	30	0	10.63	0.017	0.0015	**0**	**0**	**0**
f_{17}	30	0	13.38	0.021	**0**	**0**	**0**	**0**
f_{18}	30	0	125.06	7.7e–03	2.2e–03	**0**	**0**	5.1e–10
f_{28}	10	0	29.57	1364.45	781.55	8.23	0.32	**0.10**
f_{29}	10	–9.66015	0.16	5.65	0.069	**0**	**0**	**0**
f_{30}	10	–1.4	0.76	1.39	0.35	0.97	**0.06**	0.16

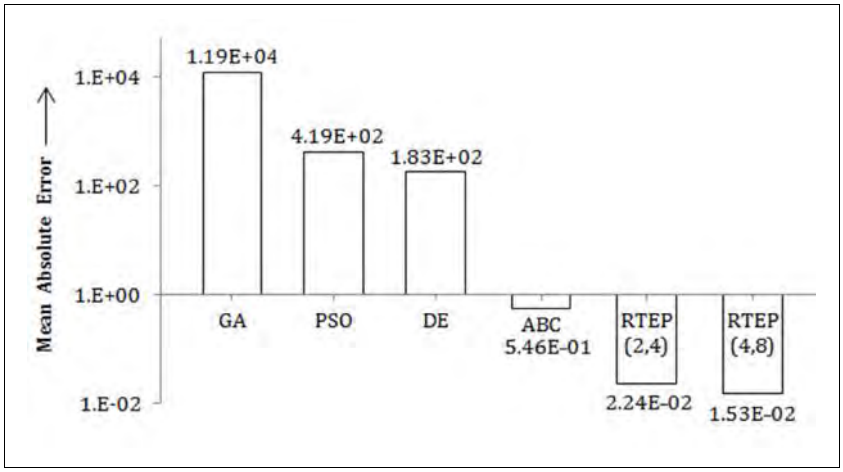


Figure 3.8: Comparison among the RTEP, GA [11], PSO [196], DE [195] and ABC [11] based on their mean absolute errors on the standard benchmark functions. The RTEP variants exhibit the best performance, i.e., smallest mean absolute error values.

PSO Settings: The PSO [196] has three control parameters, in addition to the population size and function evaluations, which are — the cognitive component ¤1, social component ¤2, and the inertia weight *w*, which are set to 1.8, 1.8 and 0.6, respectively, as suggested in [196].

ABC Settings: In addition to the common parameters (population size and maximum number of function evaluations), the basic ABC algorithm has one more control parameter — *limit*, which is set as *SN* * *D*, as recommended in [11]. Here, *SN* is the number of food sources or employed bees and *D* is the dimensionality of the problem.

Table 3.2 compares RTEP(2,4) and RTEP(4,8) with GA [11], PSO [196], DE [195] and ABC [11] on a total of 17 standard benchmark functions from f_1–f_{30} that have dimensionality $D \geq 10$. The following points summarize the results.

GA vs. RTEP: Both RTEP(2,4) and RTEP(4,8) perform significantly better than GA on all (i.e., 17 out of 17) functions.

PSO vs. RTEP: The performance of RTEP is either better (i.e., 13 out of 17) or at least equally good (i.e., remaining four functions) on all the functions.

DE vs. RTEP: RTEP outperforms DE on as many as 11 out of the 17 functions. For the remaining six functions, both the algorithms show similar performance by reaching the global minimum.

ABC vs. RTEP: For most of the functions (10 out of 13), ABC and RTEP variants show similar performance. However, for the seven remaining function, the RTEP variants outperform ABC on five, while ABC performs better only on the remaining two (i.e., f_{12} and f_{18}). Show, the overall performance of RTEP is better than the basic ABC algorithm.

An overall evaluation of the algorithms can be made by using their mean absolute error over all the functions. To calculate the mean absolute errors, the total absolute error over all the functions is computed for all the algorithms by summing their individual errors on each function. Then the total absolute error is divided by the number of functions (i.e., 17 for the results in Table 3.2). Fig. 3.8 shows that RTEP(4,8) has the lowest mean absolute error, followed by RETP(2,4) and the basic ABC algorithm, while the remaining algorithms (i.e., GA, PSO and DE) demonstrate much larger error values.

Next, we compare RTEP against CEP [102], ALEP [56], IFEP [57], RMEA [218] and RCMA with XHC [160] in the Tables 3.3 and 3.4. Like RTEP, both ALEP [56] and IFEP [57] use only mutations for producing offspring. IFEP mixed (rather than switched) Cauchy and Gaussian mutations in one algorithm. This algorithm generated two candidate offspring from each parent — one by Cauchy mutation and another by Gaussian mutation. The better candidate is then chosen to enter into the population. ALEP, on the other hand, generated four candidate

offspring from each parent by using Lévy mutation with four different distributions. It has been shown that ALEP [56] and IFEP [57] performed better than either their non-adaptive versions or the CEP [102]. RCMA with XHC [160] executed exploration and exploitation operations separately and combined them in one algorithm. This algorithm uses position-based crossover (PBX) [160] and breeder genetic algorithm (BGA) mutation [81] for explorations. It employs a negative assortative mating strategy for selecting two parents to perform crossover in order to introduce population diversity. RCMA with a specialized crossover operator, XHC [160], has been shown to perform better than all other variants. We executed RTEP for the same number of FEs as for RMEA [218], ALEP [56] and IFEP [57] in Table 3.3 and RCMA with XHC [160] in Table 3.4. According to [214], we have implemented CEP [102] and executed it for the same population size and the same number of function evaluations. Table 3.3 and 3.4 compares the algorithms on 12 and 5 functions, respectively. This is because the results of ALEP [56], IFEP [57] and RCMA with XHC [160] are directly taken from the corresponding references, where their results are not available for the rest of the functions.

Results in Table 3.3 show that RTEP(2,4) outperforms RMEA on 11 out of the 12 functions. The only function where RMEA performed slightly better is f_3, but the performance difference does not seem statistically significant. Moreover, RTEP is significantly better than IFEP on all (i.e., seven out of seven) the functions reported in [57], while it outperforms ALEP on 10 out of 11 functions, and also outperforms CEP on all of the 12 functions. RTEP is also compared with RCMA-XHC on five functions with dimensionality=25, and for 100000 FEs, as suggested in [160]. Table 3.4 shows that RCMA-XHC [160] outperformed RTEP(2,4) only on one unimodal function f_1, while RTEP performed better than RCMA on rest of the functions, i.e., unimodal functions f_3 and f_7 and multimodal functions f_{10} and f_{14}. Although we could not perform *t*-test due to lack of availability of the standard deviation values for the results of RCMA-XHC, it is easily apparent that the performance difference is significantly better by RTEP for the functions f_3, f_7, f_{10}, and f_{14}.

As Table 3.3 shows, among the counterparts of RTEP, the best results are achieved by RMEA [218], which is previous research work by us and shares some degree of similarity with the currently proposed algorithm — RTEP. However, there also exists significant amount of dissimilarity between RMEA [218] and RTEP. For example, RMEA [218] uses fitness values to measure the similarity or dissimilarity between individuals in order to estimate the neighbors and strangers for a candidate solution. This is based on the assumption that fitness similarity (or dissimilarity) can generally be accounted for a relative similarity (or dissimilarity) between the genotypes. However, such an assumption might not be true. The search space for high-dimensional multimodal functions is usually extremely large, and it is commonly observed

that genetically diverse individuals can have quite similar fitness values. The converse may not be so common for continuous functions, but still two genetically very similar individuals might have quite different fitness values. Thus RMEA [218], by selecting neighbors or non-neighbors merely by fitness values may inadvertently pick inappropriate individuals which would fail to induce the intended exploitations or explorations. RTEP, on the other hand, employs genotype distance to appropriately select neighbors and non-neighbors. This selection operation is consistent with its variation operation (i.e., mutation) which employs the genotype distances

Table 3.3: Comparison among RTEP(2,4), RMEA [218], ALEP [56], IFEP [57] and CEP [102] for 12 standard benchmark functions. Results have been averaged over 50 independent runs. The best results are marked with boldface fonts. The '+' indicates that RTEP(2,4) is significantly better than the compared algorithm with 95% certainty, while '≈' means that the difference is not statistically significant.

Function	Mean Error					*t*-Test (RTEP vs.)	
	RTEP (2,4)	RMEA	IFEP	ALEP	CEP	RMEA	ALEP
f_1	**7.5e–18**	1.05e–17	4.16e–05	6.32e–04	9.1e–04	≈	+
f_4	2.4e–15	**2.21e–15**	–	4.28e–02	2.1e+02	≈	+
f_7	**1.1e+00**	1.52e+01	–	4.34e+01	8.6e+01	+	+
f_{10}	**2.5e–14**	1.74e–08	–	5.85e+00	4.3e+01	+	+
f_{12}	**2.9e–07**	9.26e–04	8.87e–02	–	4.0e+01	+	–
f_{13}	**2.0e–10**	5.08e–06	4.83e–03	1.90e–02	1.5e+00	+	+
f_{14}	**2.7e–25**	6.41e–20	4.53e–02	2.40e–02	8.7e–02	+	+
f_{17}	**3.2e–13**	1.72e–08	–	6.00e–06	4.8e–01	+	+
f_{18}	**7.1e–08**	9.29e–05	–	9.80e–05	8.9e–02	+	+
f_{25}	**0.55**	0.81	4.08	1.03	2.85	≈	+
f_{26}	0.38	1.41	3.41	**0.26**	0.78	+	≈
f_{27}	**0.25**	0.94	1.71	0.62	0.85	+	≈

Table 3.4: Comparison between RTEP and RCMA-XHC [160] on five benchmark functions with dimensions=25. Results have been averaged over 50 independent runs. The best results are marked with boldface fonts. The '+', '≈' and '–' indicate that RTEP(2,4) is significantly better, similar and worse, respectively than its counterpart, RCMA-XHC [160].

Function	Mean Error		*t*-Test
	RTEP(2,4)	RCMA	RTEP(2,4) vs. RCMA
f_1	4.72e–21	**6.5e–101**	–
f_3	**4.13e–17**	3.81e–07	+
f_7	**1.04e+00**	2.2e+00	≈
f_{10}	**8.79e–15**	1.4e+00	+
f_{14}	**6.29e–28**	1.3e–02	+

too in order to pick an appropriate mutation step size, either exploitative or explorative. Another difference between RMEA and RTEP is that RMEA uses a single fitness-based greedy re-insertion policy which always accepts better offspring only, no matter whether the current stage is the exploration or exploitation stage. In contrast, RTEP uses different re-insertion strategies, consistently based on the current explorative/exploitative stage. Another difference between RTEP and RMEA is that, during each mutation, RMEA mutates every individual D different times, each time randomly picking one of its D gene values. This might cause one gene to be mutated several times, which is likely to destroy the behavioral link between the parent and the offspring. In contrast, RTEP mutates a random number, say r, of genes of every individual. A small value of r is more likely to preserve close behavioral link between the parent and the offspring (which would be better for exploitations), while moderate or large values of r would facilitate better search space explorations. Thus, the mutation operation of RTEP possesses both exploitative and explorative features, while the mutation of RMEA makes it off-balance towards more explorations. The significant performance difference between RTEP and RMEA, as illustrated in Table 3.3, indicates the better effectiveness of the proposed RTEP algorithm over its counterpart, RMEA.

The previous comparisons suggest that RTEP is better than its counterparts that either use mutations only or execute exploration and exploitation operations separately. It is interesting to investigate the performance of RTEP with an approach that neither uses mutation nor considers the exploration and exploitation operations explicitly. One such approach is the CLPSO [69], which is an improved variant of the PSO algorithm [196]. CLPSO uses a novel learning strategy in which all other particles' historical best information is used to update a particle's velocity to move the search process forward. It has demonstrated better performance

Table 3.5: Comparison between RTEP and CLPSO [69] on six standard benchmark functions. Results have been averaged over 30 independent runs. The best results are marked with boldface font. The '+', '≈' and '–' indicate that RTEP(2,4) is significantly better, similar and worse, respectively than its counterpart, CLPSO [69].

Function	**Mean Error**		***t*-Test**
	RTEP(2,4)	**CLPSO**	**RTEP(2,4) vs. CLPSO**
f_1	**3.05e–21**	4.38e–14	+
f_7	**8.11e–01**	2.08e+01	+
f_{10}	**1.05e–15**	4.75e–10	+
f_{12}	**2.92e–11**	4.34e–10	+
f_{13}	7.54e–13	**0.00e+00**	–
f_{14}	**3.19e–27**	3.11e–10	+

than other variants of PSO for wide range of complex functions. Since the number of FEs used by CLPSO [69] has been 200000, RTEP(2,4) is re-implemented for the same FEs. Table 3.5 presents results for RTEP(2,4) and CLPSO [69] on six standard benchmark functions over 30 independent runs. Results show that RTEP(2,4) has performed better than CLPSO on two unimodal and three multimodal functions, while CLPSO outperformed RTEP(2,4) on only one multimodal function (f_{13}). The performance differences are clearly significant and mostly (5 out of 6 functions) better for RTEP.

3.6.4 RTEP on CEC2005 Benchmark Functions

RTEP has also been evaluated on the recently introduced CEC2005 benchmark suite [76]. This new suite includes 25 functions with more complexity, including many shifted, rotated, expanded and hybrid composite functions. A brief overview on these functions is presented in Table 2.4 and section 2.17. More detailed description on each function can be found in [76] and also in the appendix A. In all our comparisons, the dimensionality of these functions is set to 30 and the FE is set to be 3.0e+05. These settings are for a fair comparison with previous algorithms, such as the RCMA with adaptive local search (LSRCMA) [162] and DE with neighborhood search (NSDE) [163].

The mean error values over 25 independent runs on each function by RTEP, LSRCMA [162] and NSDE [163] are presented in Tables 3.6 and 3.7. Results indicate that RTEP achieves performance comparable to and often better than the other two algorithms. The following points summarize our observations on their performance comparison.

- In case of the five unimodal functions F1–F5, RTEP is found to be significantly better than both of LSRCMA and NSDE on three functions, while it is outperformed by them only on one (f_4) and two functions (f_1, f_4), respectively.
- For the relatively more complex multimodal functions F6–F14, the superiority of RTEP is clearly visible. RTEP is found to perform significantly better than the other two algorithms on six (out of nine) functions, while it is outperformed by them on only two functions (f_6 and f_8), with similar performance on the remaining one (f_{14}).
- The results in Table 3.7 indicate that the performance of all three algorithms is somewhat compromised for the hybrid composite functions. However, RTEP still performs significantly better than LSRCMA and NSDE on six and five (out of 11) functions, while it shows similar or worse performance for the remaining few functions. In short, RTEP shows better performance on more functions than either of NSDE and LSRCMA.

Table 3.6: Comparison among RTEP, LSRCMA [162] and NSDE [163]on five unimodal and nine multimodal functions introduced at CEC2005 [76]. All the results have been averaged over 25 independent runs. The best results are marked with boldface font. Function properties are expressed by *S*: Shifted, *R*: Rotated, *N*: Non-separable. The '+' and '–' indicate that RTEP(2,4) is significantly better and worse, respectively than the compared algorithm, while '≈' means that the difference is not statistically significant.

Function	Mean Error			*t*-Test	
	RTEP (2,4)	LSRCMA	NSDE	RTEP vs. LSRCMA	RTEP vs. NSDE
$f_1\,(S/-/-)$	7.62e–09	9.36e–09	**0.00e+00**	≈	–
$f_2\,(S/-/N)$	**2.44e–11**	8.71e–06	5.62e–08	+	+
$f_3\,(S/R/N)$	**9.25e+01**	8.77e+05	6.40e+05	+	+
$f_4\,(S/-/N)$	8.99e+02	3.96e+01	**9.02e+00**	–	–
$f_5\,(-/-/N)$	**7.64e+00**	2.18e+03	1.56e+03	+	+
$f_6\,(S/-/N)$	5.01e+02	4.95e+01	**2.45e+01**	–	–
$f_7\,(S/R/N)$	**2.85e–05**	1.32e–02	1.18e–02	+	+
$f_8\,(S/R/N)$	1.47e+02	**2.07e+01**	2.09e+01	–	–
$f_9\,(S/-/-)$	**3.69e–04**	6.80e–01	7.96e–02	+	+
$f_{10}\,(S/R/N)$	**5.92e+00**	9.05e+01	4.29e+01	+	+
$f_{11}\,(S/R/N)$	**7.39e+00**	3.11e+01	1.41e+01	+	+
$f_{12}\,(S/-/N)$	**4.73e+02**	4.39e+03	6.59e+03	+	+
$f_{13}\,(S/-/N)$	**1.54e–01**	3.96e+00	1.62e+00	+	+
$f_{14}\,(S/R/N)$	**8.94e+00**	1.25e+01	1.31e+01	≈	≈

Table 3.7: Comparison among RTEP, LSRCMA [162] and NSDE [163] on the 11 hybrid composite functions introduced at CEC2005 [76]. Results have been averaged over 25 independent runs. The best results are marked with boldface font. The '+' and '–' indicate that RTEP(2,4) is significantly better and worse, respectively than the compared algorithm, while '≈' means the difference is not statistically significant.

Function	Mean Error			*t*-Test	
	RTEP (2,4)	LSRCMA	NSDE	RTEP vs. LSRCMA	RTEP vs. NSDE
f_{15}	9.05e+02	**3.56e+02**	3.64e+02	≈	–
f_{16}	**5.84e+01**	3.26e+02	6.90e+01	+	+
f_{17}	**2.38e+01**	2.79e+02	1.01e+02	+	+
f_{18}	**7.41e+01**	8.77e+02	9.04e+02	–	–
f_{19}	9.96e+02	**7.81e+02**	9.65e+02	+	+
f_{20}	9.02e+02	**4.79e+02**	9.11e+02	–	–
f_{21}	5.11e+02	**5.80e+02**	**5.84e+02**	+	+
f_{22}	**1.42e+02**	9.08e+02	8.89e+02	–	–
f_{23}	**8.01e+01**	5.59e+02	5.34e+02	+	+
f_{24}	7.98e+02	**2.00e+02**	**2.00e+02**	+	+
f_{25}	**1.01e+02**	4.11e+02	6.60e+02	+	+

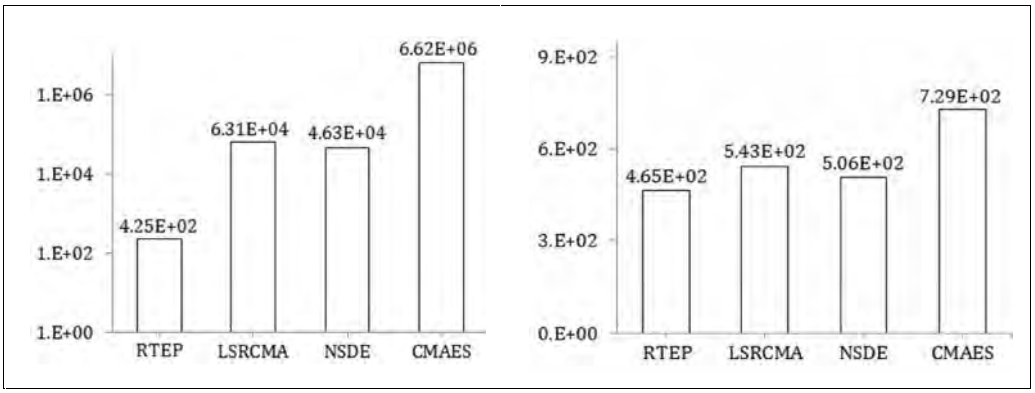


Figure 3.9: Comparison of NSDE [163], LSRCMA [162], CMAES [165] and RTEP (2,4), based on the mean absolute errors for the CEC2005 benchmark functions F1-F14 (on the left) and the hybrid composition functions F15-F25 (on the right).

Fig. 3.8 shows the mean absolute error of NSDE, LSRCMA and RTEP, separately for the non-composite functions F1–F14 and the more complex, hybrid composition functions F15–F25. Here, RTEP shows the minimum mean absolute error, in comparison to both NSDE and LSRCMA. Thus we can conclude that RTEP shows the overall better performance than both of its counterparts — NSDE and LSRCMA.

There may be two reasons why RTEP is a bit inefficient on hybrid composition functions. First, the fixed strategy used in RTEP that executes repeatedly exploration and exploitation operations for a fixed number of iterations before alternating to the other operation may not work well for composition functions. An adaptive approach that can dynamically change this number during the course of evolution may be more appropriate than the fixed strategy, particularly for complex search spaces. Second, the directional information used in RTEP is taken by considering two neighboring or distant individuals. It could be better if the information of more individuals or the characteristics of the search space around the current points could be used in the mutations. The incorporation of these ideas could be a topic for our future study.

3.6.5 Discussion on Results of RTEP

This section briefly explains why the performance of RTEP is better than most other algorithms in comparison for most of the benchmark functions. First, RTEP emphasizes performing global exploration and local exploitation not only separately but also adaptively. The utilization of the distance of dissimilar or similar individuals in mutation of RTEP clearly reflects such emphasis, because such distances among individuals are usually high during the early generations, but gradually drops with the ongoing evolutionary process. RMEA [218], though attempts to perform recurring explorations and exploitations, its inappropriate selection scheme of

strangers and neighbors based on phenotype distance (rather than genotype distance) is likely to fail to bring about the desired explorations and exploitations. ALEP [56], IFEP [57], and CEP [102] do not separate exploration and exploitation operations; rather, IFEP and ALEP primarily emphasize producing good offspring. The emphasis on only good solutions may reduce the population diversity resulting in poor overall performance. Both RCMA with XHC [160] and LSRCMA [162] perform explorations and exploitations separately. Exploration is carried out by using PBX [160] and BGA mutation [81], while exploitation is conducted either by a special crossover operator (for RCMA with XHC) or using an adaptive local search operator (for LSRCMA). A special neighborhood search operator is also used for mutations in NSDE [163]. The problem of using different operators lies in ensuring their synergistic effect [212]. CLPSO [69] is a learning approach that does not employ exploration and exploitation operations separately. Although it utilizes the best information of all particles to update the velocity of any one particle, it may still trap into local optima due to the inherent problem of a learning scheme.

Second, mutation in RTEP does not produce offspring blindly, but rather utilizes the information of other individuals to produce objective-oriented offspring. The mutation produces offspring in such way that an offspring either facilitates the exploration of wider and unvisited regions of the search space or the exploitation of the local neighborhoods. However, the mutation in ALEP [56], IFEP [57], and CEP [102] does not use the information of other individuals and produces offspring blindly. The consequence of blind mutation is that the offspring produced may be dominated by individuals in the current population. Both RCMA with XHC [160] and LSRCMA [162] also use blind mutations and crossover for explorations. NSDE [163] tries to guide the mutation operations by controlling the step size, but with no direction information. Although RMEA [218] employs direction information for appropriate mutation step size, it often destroys the behavioral link between the parent and the offspring which makes the algorithm more explorative than exploitative resulting in deteriorated performance in comparison to RTEP.

Third, RTEP uses two different simple selection strategies to select offspring for the next generation. During exploration, our algorithm allows an offspring for the next generation if the offspring is potentially more promising than the parent, which allows slightly worse (in terms of fitness) offspring to enter into the population, replacing its parent. However, only the better offspring is allowed during exploitation stage. These selection strategies match the exploration/exploitation objective of an evolutionary process. A tournament-based selection scheme is used in ALEP [56], IFEP [57], and CEP [102]. For each parent or offspring, q opponents are chosen uniformly at random for pairwise fitness comparison from all the parents and offspring. However, the value of q affects the population diversity. A large value of q

corresponds to high selection pressure, so the probability of the fittest individual being selected multiple times becomes high, resulting in loss of population diversity. Both RCMA with XHC [160] and LSRCMA [162] allow only better offspring for both explorations and exploitations, which is more likely to reduce the population diversity and may lead to premature convergence.

3.7 Conclusion and Future Research Directions

Evolutionary systems based on mutation have been introduced to the scientific community for around four decades [1]. However, most mutation-based algorithms (e.g., [11], [56], [57], [146], [162]—[164]) use a single-stage execution model to tackle the conflicting goals of evolution, i.e., the exploration and exploitation. These algorithms mostly rely on increasing the exploration capability of the mutation operation, although both exploration and exploitation are necessary during the evolution. Improving the capability of one operation at the expense of another becomes a crucial decision due to the unknown scenario at different stages of evolution. This chapter introduces RTEP — a recurring explorative/exploitative scheme based on mutation to unravel the conflicting goals of evolution in finding a good near-optimum solution for complex problems. RTEP adopts repeatedly alternated execution of exploration and exploitation operations during the evolution. RTEP uses Gaussian mutation in its both the recurring stages for producing offspring. Global exploration and local exploitation are encouraged through the use of mutation involving directional information and appropriate selection strategies. The distance between two dissimilar or similar individuals is used in RTEP as the SD of Gaussian mutation to explore the search space globally or locally. In the experiments section (chapter 8, sections 8.3, 8.8), we have carried out extensive experiments to evaluate how well RTEP can perform compared to some other existing evolutionary algorithms. In most of the cases, RTEP clearly outperformed most other algorithms in comparison.

RTEP involves two user specific parameters, K_1 and K_2, which are the durations of the exploration and exploitation stages, respectively. In a later chapter (Table 8.11, chapter 8) it is demonstrated that small values work sufficiently well for K_1 and K_2 for all the benchmark functions. However, results usually start to deteriorate with increasing values of K_1 and K_2 (Tables 8.11–8.12, chapter 8). This happens because the recurring nature of RTEP starts to diminish with larger lengths of its recurring stages. Thus, the key ingredient for good results from RTEP is short stage lengths with frequent alternation of the stages. Any small value for (K_1,K_2) would likely to perform good enough. For example, all three settings of RTEP with (K_1,K_2) = (1,1), (2,4) and (4,8) have produced satisfactory results in Table 3.1. Since real-world problems vary wildly from each other, it would be inappropriate for us to suggest some optimal

choice as the values of (K_1,K_2). Instead, what we suggest is to try with a number of small values for K_1, K_2 when the user does not have sufficient prior knowledge about the problem characteristics.

There are several future research directions suggested by this chapter. First, a scheme to make the length of exploration and exploitation stages, i.e., K_1 and K_2 adaptive could be devised. Second, since the framework presented by RTEP is generic enough, it could be effectively extended many other existing algorithms. Although RTEP employs only Gaussian mutation and makes use of no crossover/recombination operator, it is not a rigid design requirement of RTEP. In fact, RTEP provides a very generic framework and any mutation and/or recombination or crossover scheme could be incorporated within it. There exist lots of genetic operators whose strengths have already been demonstrated, e.g., SBX crossover and polynomial mutation [219]. Any such operator might be incorporated into RTEP, while the only requirement is to design both "explorative" and "exploitative" variants of that operator. It is open to the researchers how they define the explorative and exploitative variants, which would not be difficult in most situations. For example, every EA has to maintain a population of potential solutions, so it may readily introduce the participation of dissimilar and similar individuals in some way to define the "explorative" and "exploitative" versions of its variation operators. Third, RTEP has been applied mainly to continuous parameter optimization problems. It would be interesting to study how well RTEP performs for other problems, especially the discrete and real world ones. Fourth, as RTEP showed excellent capacity to locate the global optimum, one interesting idea would be to hybridize RTEP with other existing algorithms. For example, RTEP could be employed on a problem that is partially solved by another algorithm while the global minimum is still unknown, yet the algorithm has reached fitness stagnation. In this situation, RTEP could be employed, starting with the candidate solutions produced by the previous algorithm, and it would be interesting to find whether or not RTEP could break the fitness stagnation, improve the final solution quality and locate the global optimum of the search space.

Chapter 4

Diversity Guided Evolutionary Programming

4.1 Introduction

The necessity of preserving adequate amount of population diversity to ensure more search space explorations and to avoid premature convergence is emphasized in the previous section 2.14. There have been several studies (e.g., [141]–[172]), as briefly presented and categorized in the previous section 2.15, that have focused to devise techniques to preserve sufficient amount of population diversity throughout the evolutionary process. Since mutation is the major operator in many evolutionary systems, such as evolutionary programming and evolutionary strategies, a number of research works have also been done (e.g., [140], [165], [220]–[223]) for the elegant control and adaptation of the mutation step size to properly traverse across the locally optimal points and to locate the global optimum point. This chapter introduces Diversity Guided Evolutionary Programming (DGEP), a novel algorithm that tries to combine the best of both these research directions (i.e., research along diversity-preserving evolutionary algorithms and research for designing novel mutation strategies). DGEP incorporates Diversity Guided Mutation, a novel mutation scheme that controls and guides the mutation step size using the existing population diversity information. It also takes some extra diversity preservative measures to maintain an adequate amount of population diversity that assists the proposed mutation scheme. In this chapter, we have tested and evaluated DGEP on a wide range of continuous benchmark functions and compared the results with a number of existing state-of-the-art evolutionary and swarm intelligence algorithms.

4.2 Organization of the Chapter

The rest of this chapter is organized as follows. Section 4.3 describes the proposed algorithm DGEP in details, along with its central component — the diversity guided mutation (DGM) scheme with the pseudocode. A brief analysis of its strengths and how it avoids premature convergence are also presented in this section. The next section 4.4 highlights the differences between the proposed algorithm — DGEP and most other existing evolutionary algorithms. Sections 4.5–4.6 perform the evaluations of DGEP using two different benchmark suites on the continuous optimization problem and compare the performance of DGEP with several other relevant evolutionary and swarm intelligence algorithms. Finally, the section 4.7 concludes the chapter by leaving a few suggestions on future research directions.

4.3 The Proposed Algorithm — DGEP

Since mutation is the sole variation operator in many evolutionary systems, including EP and ES [4], [161] schemes, much research, both practical, e.g., [220] and theoretical, e.g., [221] has been done for improving the effectiveness of mutation as a search mechanism. A small mutation step size provides better search stability, but is likely to get trapped into locally optimal points while a large mutation step size is better immune against local optima, but the search may be unstable with unacceptably slow or no convergence to the global optima. This is why most existing works [140], [165], [222], [223] try to adapt the mutation rate and step size dynamically in order to achieve adequate convergence rate and to avoid premature convergence to local optima. However, none of the existing schemes, to our best knowledge, makes use of population diversity information to guide the mutation step size. However, the existing amount of population diversity can often be a good indicator of the maturity of the ongoing optimization process. This chapter introduces Diversity Guided Evolutionary Programming (DGEP), which is based on the central idea of making use of the population diversity information to induce more effective mutations. DGEP is based on Diversity Guided Mutation (DGM), a novel mutation scheme that employs the diversity information for the automatic adaptation of the mutation step size to avoid premature convergence to local optima. The balance between the exploitative and explorative features of the mutation operation is sought by employing diversity information existent at two different granularities: micro and macro. At the micro level, diversity among genetically very similar chromosomes is used to induce small mutation steps that are suitable for exploitations of the existing chromosomes, while at the macro level the diversity across two random chromosomes is used to induce relatively large variations that are suitable for search space explorations. To assist the proposed

diversity based mutation operator with additional diversity, the selection operator of DGEP is also modified. In addition to fitness based selection, the selection operator also looks for chromosomes that are either duplicates or have failed to improve for a long time and replace them with randomly placed chromosomes in a way that promotes population diversity.

DGEP differs from the Classical Evolutionary Programming (CEP) [3], [102] at two different points. First, DGEP introduces and incorporates Diversity Guided Mutation (DGM), a novel mutation scheme that uses population diversity information to adapt the mutation step size. Second, DGEP employs a few simple diversity preserving schemes to assist the DGM scheme with additional amount of diversity, because sufficient diversity is required by DGM to perform both exploitative and explorative search around the current chromosomes. DGM makes use of the population diversity information at two different levels: diversity among very similar chromosomes is used for exploitative mutations on the existing solutions while diversity among dissimilar chromosomes across the population is used for explorative mutations to perform the search space explorations. After random initialization and during early generations, both these diversity values (say, div_{low} and div_{high}) are usually large. As the population converges around the locally optimum points of the search space, div_{low} drops rapidly, but div_{high} usually remains sufficiently high because of the existence of several locally optimal points within the population. This high amount of diversity among non-neighbors facilitates large mutations and thus helps avoid premature convergence around any locally optimum point. In the ideal case, the entire population gradually converges towards a single global optimum which causes both div_{low} and div_{high} to decrease, allowing the algorithm to converge. In addition to the DGM scheme for more effective mutations, DGEP also employs some diversity preserving mechanisms to help the DGM scheme, such as the elimination of identical or nearly duplicate chromosomes, detection of the chromosomes that have reached fitness stagnation for an unacceptably long time period and replacing such chromosomes by randomly produced ones or by chromosomes produced from a series of hill climbing steps inter-mixed with diversity guided mutations. The details of the DGEP algorithm can be presented in the following steps.

Step 1) Generate an initial population of M chromosomes. Each chromosome, $\boldsymbol{C}$ is represented as a pair of real valued vectors, $(\boldsymbol{x}_i, \boldsymbol{\eta}_i)$, for $i = 1, \ldots, M$; $\boldsymbol{x}_i$ is the vector composed of the parameter values being optimized and $\boldsymbol{\eta}_i$ is the standard deviation vector to perform Gaussian mutations as hill climbing steps on $\boldsymbol{x}_i$, if necessary (step 7). Each $\boldsymbol{x}_i$ (and $\boldsymbol{\eta}_i$) has D components: $\boldsymbol{x}_i = [x_i(1), , x_i(2), \ldots, x_i(D)]^T \in R^n$ and $\boldsymbol{\eta}_i = [\eta_i(1), \eta_i(2), \ldots, \eta_i(D)]^T \in R^n$, D being the dimensionality of the problem. Each component of $\boldsymbol{x}_i$, for $i = 1, \ldots, M$, is generated uniformly at random within its domain (i.e., $x_i(j) = Uniform\_Random \sim (min_j, max_j)$, for $j = 1, \ldots, D$). All the

components of $\boldsymbol{\eta}_i$, for $i = 1, \ldots, M$, are initially set to some moderate value (e.g., 3.0), as suggested in [102]. The iteration counter, t is initialized to 1.

Step 2) Evaluate the fitness of each chromosome $\boldsymbol{x}_i$. Compute the selection probability p_i for each chromosome, $\boldsymbol{x}_i$ using $p_i = \frac{fitness(\boldsymbol{x}_i)}{\sum_{i=1}^{M} fitness(\boldsymbol{x}_i)}$. This also normalizes p_i values into [0, 1].

Step 3) Repeat steps 4-5 for M times. This constitutes the child population, $C(t)$ from the parent population, $P(t)$.

Step 4) Select a chromosome, $\boldsymbol{x}_i$ based on its normalized fitness value, p_i. We employed traditional roulette wheel selection scheme which ensures fitness proportional selection. However, any other fitness based selection scheme can also be adopted.

Step 5) Apply the DGM scheme to mutate the selected chromosome, $\boldsymbol{x}_i$ to produce an offspring chromosome, $\boldsymbol{x}_i'$. Details of DGM are explained later with its pseudocode (Fig. 4.1). Insert $\boldsymbol{x}_i'$ into the child population, $C(t)$.

Step 6) Merge the parent population, $P(t)$ and child population, $C(t)$. Sort the chromosomes according to their fitness values and select the best M chromosomes to constitute the next generation candidate population, $Q(t)$.

Step 7) Check for chromosomes trapped into deep local optima. If a particular chromosome, $\boldsymbol{x}_i$ in $Q(t)$ has not been improved for several (say, u) consecutive iterations, then allow $u/2$ DGM mutations randomly intermixed with $u/2$ hill climbing steps by Gaussian mutations using $N(0, \boldsymbol{\eta}_i)$ distribution (as done using eq. (1) and (2) in [57]). If this leads to fitness improvement, replace the original chromosome with the newly produced chromosome. Otherwise eliminate both chromosomes from the population.

Step 8) Elimination of duplicate chromosomes. If the Euclidean distance between the genotypes of two chromosomes, $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ is less than a certain fraction, l of the average Euclidean distance among all the genotypes across the population, then $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are considered as duplicates. Select the one with lower fitness value and eliminate it from the population.

Step 9) If the current number of chromosomes is less than M in the candidate population, $Q(t)$ then introduce new chromosome, $\boldsymbol{x}_i$ placed uniformly at random across the search domain using: $x_{ij} = min_j + rand(0,1) * (max_j - min_j)$ for $j = 1, \ldots, n$; Repeat this step until $Q(t)$ consists of M chromosomes. Now, $Q(t)$ becomes the parent population, $P(t)$ for the next iteration.

Step 10) Keep track of the best chromosome found so far. Also, increase the iteration counter, t by setting $t = t + 1$.

Step 11) Check for termination. If the best chromosome found so far is acceptable or the iteration (generation) counter, t exceeds some predefined maximum number of iterations, stop the process and return the best chromosome found so far. Otherwise go back to step 2 and repeat again.

The essence of DGEP is the DGM scheme (step 5) for mutation. Also, DGM is supported by some simple diversity preserving schemes, as described in steps (7) and (8). Before proceeding to the details of the DGM scheme, we address a number of issues and present some implementation details in the following points.

(a) In step 1, the initial values of standard deviations, i.e., $\boldsymbol{\eta}_i$'s are all set to 3.0. This is not ad-hoc, rather only to be identical with the initial parameter settings of CEP [102], [57], IFEP [57] and ALEP [56] for a fair performance comparison. Though such a choice of initial values does not consider the search space along each dimension, the self-adaptive mechanism for standard deviations (in step (7) of DGEP, following eq. (1) and eq. (2) in [57] for CEP and IFEP, eq. (23) in [56] for ALEP) automatically adapts each $\boldsymbol{\eta}_i(j)$ separately and makes it suitable for an effective search around the corresponding individual, $\boldsymbol{x}_i$ along the dimension, j.

(b) In step 2, how to compute the fitness of an individual, $\boldsymbol{x}_i$? Since the objective function, f is to be minimized, so smaller values of $f(\boldsymbol{x}_i)$ should translate into higher fitness values. Also, both positive and negative values of $f(\boldsymbol{x}_i)$ need to be considered. In our implementation, we have employed the following formula to compute the fitness value of an individual, $\boldsymbol{x}_i$ from its objective value, $f(\boldsymbol{x}_i)$.

$$fitness(\boldsymbol{x}_i) = \begin{cases} \dfrac{1}{1+f(\boldsymbol{x}_i)} & \text{if } f(\boldsymbol{x}_i) \geq 0 \\ 1+|f(\boldsymbol{x}_i)| & \text{otherwise} \end{cases} \tag{4.1}$$

(c) Since each mutation operation by the DGM scheme (step 5) possesses both exploitative and explorative potentials, so we are quite indiscriminate to pick the selection operators (in steps 4 and 6), without expecting any special exploitative or explorative pressure from the selection operators. For parent selection, we have employed roulette wheel selection (i.e., fitness proportional selection) in step 4 and for survivor selection, we employ truncation selection (step 6). Both the selection operators are a bit exploitative in order to maintain good convergence speed. However, the choice of these selection operators is not an essential component of DGEP and any other selection operator might be used. One advantage with both these operators is that they don't require any user-specific parameter which might involve problem specific knowledge. Some other popular selection operators, e.g., the

tournament selection, can also be employed. However, tournament selection requires a parameter (e.g., the tournament size) which needs to be set carefully and can affect the exploitative-explorative bias of the algorithm.

(d) A prolonged fitness stagnation of an individual may be caused not only by a deep local optimum, but also a wide, flat plateau of the search space. To detect such a situation, our implementation equips each individual with a counter that keeps track of how many consecutive mutation attempts fail to improve the individual. If the counter exceeds a maximum allowed value (say, *u)*, then we assume that the chromosome is stuck either at a deep local optimum or within a wide flat plateau of the search space. Then we initiate the sequence of DGM mutations and hill climbing steps by Gaussian mutations, as mentioned in step 7 of DGEP. Our implementation of step 7 operates like this: let $\boldsymbol{x}_i$ be the individual stuck at a local optimum or a flat plateau. Then repeat the following operations on $\boldsymbol{x}_i$ for u times: Randomly flip a fair, unbiased coin. If it comes up with a head, then mutate $\boldsymbol{x}_i$ by the DGM mutation scheme and accept the offspring solution to replace the parent $\boldsymbol{x}_i$. Otherwise (i.e., the flip comes up with a tail), apply Gaussian mutation on $\boldsymbol{x}_i$ by using the self-adaptive mechanism with $\boldsymbol{\eta}_i(j)$'s, in the same was as in eq. (1) and (2) of [57]. While the DGM mutation is always accepted, the Gaussian mutation is accepted only when it leads to fitness improvement; hence we call it hill climbing steps by Gaussian mutations. Please note again that, with each Gaussian mutation, the $\boldsymbol{\eta}_i(j)$'s (i.e., the standard deviations of Gaussian distribution, maintained separately for each individual $\boldsymbol{x_i}$ and for each of its dimension, j) also go through the self-adaptation scheme, as described by eq. (2) in [57]. Thus the sequence of perturbations attempt to break free the individual and improve its current situation, no matter whether its fitness stagnation is due to some strong local optima or due to a flat plateau of the search space.

(e) Function evaluations (FEs) are often the most prohibitive and limiting section of evolutionary algorithms. In our implementation of DGEP, we employ a population of size of 50 which is half the population size of CEP and IFEP, because each DGM mutation involves two (rather than one) function evaluations for the two different mutations (i.e., exploitative and explorative). Thus, DGEP still uses roughly the same number of FEs as in CEP, IFEP and ALEP. Also, with half the population size, the computational cost of average Euclidean distance across the population (required in step 8) will be much smaller than using the full size population. Our naïve implementation for step 8 requires $\Theta(M^2)$ extra computations, where M is our reduced population size (i.e., 50). However, it might be further reduced by using a more thoughtful implementation.

(f) In every generation, DGEP updates the average Euclidean distance across the population and also updates the pairwise Euclidean distances for the individuals. With this information ready at hand, implementing step 8 requires only the value of the parameter, l. The specific value of l, along with the other parameter values of DGEP in our implementation is provided in the experiments section 4.5.

(g) In every generation, DGEP computes the pairwise distance between individuals and uses this distance to pick the neighbors. For each individual, $\boldsymbol{x}_i$ its neighbors are selected to be the $|N|$ individuals that have the smallest Euclidean distance from it. As the individuals are continuously evolving and changing, neighbors of an individual may vary with generations, bur neighborhood size $|N|$ is kept constant all through the evolution. The value of $|N|$ in our implementation, along with other parameter values, is specified in the subsequent experiments section 4.5.

(h) DGEP eliminates duplicate individuals in step 8, which are replaced by random individuals in step 9. But how can we ensure that the step 9 of DGEP is not producing duplicate individuals again? In most evolutionary algorithms, the strong locally (or, globally) optimal points of the fitness landscape usually draws most of the individuals, so the population may be filled with individuals that are duplicates or nearly duplicates of some other individuals around the same optimal point. But replacing such an individual with a random individual, as is done in steps 8–9 of DGEP, has very little or virtually no possibility to produce a duplicate individual. This is because, for each of the benchmark functions, the search space is high dimensional and the domain along each dimension is continuous. For example, consider a problem with dimensionality $D = 30$ or 60. For such a problem, even if the domain for each dimension is discrete and just $\{0, 1\}$, the possibility of producing an individual that is identical to any individual of a population of size $= 100$ is merely $100/2^{30}$ or $100/2^{60}$, which is very close to 0.

The details of DGM scheme with its pseudocode and a brief analysis on how DGEP tries to balance between exploitations and explorations by employing population diversity information and how it achieves better immunity against premature convergence is presented the following sub-sections. Then, we have made a number of points where DGEP differs from most other existing works in literature.

4.3.1 Diversity Guided Mutation (DGM)

The central component of the proposed DGEP system is the DGM scheme for mutation. In order to mutate a particular chromosome $\boldsymbol{x_i}$, DGM randomly picks two chromosomes, one from its neighbors (i.e., most similar chromosomes) and the other from the rest of the population. Suppose d_1 and d_2 be the distance values of these two chromosomes from $\boldsymbol{x_i}$. Since d_1 is the distance between two neighbors, it is expected to be pretty small, while d_2 being the distance between two non-neighbor chromosomes across the population is expected to be rather large in comparison to d_1 (unless the entire population has converged). DGM employs d_1 to produce a small, exploitative variation on $\boldsymbol{x}_i$ while d_2 is used to bring large, explorative variations. Based on the fitness value of both these newly produced offspring of $\boldsymbol{x}_i$, only one is selected to compete for insertion into the next generation population. In order to assist the DGM mutation scheme, an adequate amount of diversity is sought by adopting some simple techniques, such as elimination of duplicate chromosomes by randomly produced chromosomes, replacement of chromosomes stuck at strong local optima or wide flat plateau by multiple hill climbing steps randomly intermixed with the DGM mutations, as described in steps (7) and (8) of DGEP. The details of DGM are presented in the pseudocode in Fig. 4.1.

DGM involves three individuals to produce an offspring. There might be some difference of opinions on whether the proposed DGM scheme can rightfully be called a mutation operator, because it requires and recombines information from two other individuals, while a traditional mutation operator usually does not make use of any extra information from other individuals. However, as is found in the entire differential evolution (DE) literature, a DE mutation operator is also based on using information from three other individuals from the population. In a DE mutation, a direction vector is computed from one individual to another, which is then added to a third individual. The DGM scheme is quite similar to the DE mutation, so we don't find any problem to use the term 'mutation' for it which is completely consistent with the DE tradition. Apart from the similarity with the DE mutation, there is also significant difference between DGM and DE. Usually DE is centered around distances in gene level, while DGEP is based on distances among chromosome (i.e., individual) level. While the DE mutation does not show any concern for exploitation or exploration and maintains no relation with the existing population diversity, DGM puts its active effort to balance exploitation with exploration by estimating the existing population diversity across similar and dissimilar individuals and employing the information for exploitative and explorative mutations, respectively. These two different mutations with widely different step sizes are actually merged to constitute the composite DGM mutation scheme. DGM produces much improved results than the recent DE variants, like NSDE [163], [164], as is revealed by our evaluation studies on DGEP (e.g., Tables 4.10–4.11).

Algorithm 4.1: Diversity Guided Mutation (x): *Returns* $\boldsymbol{x'}$ *produced by diversity guided mutation on* $\boldsymbol{x}$

1. N_x: a set of chromosomes that have minimum distance from the chromosome, $\boldsymbol{x}$
2. Pick two chromosomes $\boldsymbol{y}$ and $\boldsymbol{z}$ uniformly at random from N_x and $P(t) - N_x$ respectively
3. Compute d_1, d_2: distance of the point, $\boldsymbol{x}$ from the points, $\boldsymbol{y}$ and $\boldsymbol{z}$ respectively
4. A: a set of genes of $\boldsymbol{x}$ picked at random for mutation
5. For each gene, j in A

   begin

   $K = Uniform\_Random \sim [0,3]$

   $child_1.j = Uniform\_Random \sim [\, x.j - K*d_1, x.j+K*d_1 \,]$

   $child_2.j = Uniform\_Random \sim [\, x.j - K*d_2, x.j+K*d_2 \,]$

   end
6. Apply greedy selection between $child_1$ and $child_2$ to select the fitter one. Let, $\boldsymbol{x'}$ be the selected offspring.
7. **return** $\boldsymbol{x'}$

Figure 4.1: Pseudocode for DGM

For most of the benchmark functions on which DGEP and the other algorithms have been tested, the search spaces along all the dimensions (i.e., the domains of values for all the genes) are equal and identical. This is why the adjustment of the genes by DGM uses the same amplitude of step size for all the genes. This may not be a good approach in many real world problems where the domains of the values corresponding to different genes are widely different. In practical applications it is possible to have one design variable in [0,1] while another one within [0, 1000]. However, such a situation can be easily handled by a simple linear scaling of all the domains to the same size. Before applying the DGM mutation, every gene value, x_i with domain [min_i, max_i] can be preprocessed by a simple linear transformation function, g that transforms each dimension to [0, 1] by $g(x_i) = (x_i - min_i)/(max_i - min_i)$.

In the pseudocode of DGM (Fig. 4.1), K operates as a random scaling factor of the distance between individuals in the DGM mutation scheme. As we have found with some experimentations, a random value of K within some small range (e.g., $Uniform\_Random \sim [0, 3]$ as implemented in DGM) often provides better results than a fixed or larger value of K. The primary objective of exploration and exploitation is conducted by the distances among neighbors and non-neighbors (i.e., d_1 and d_2 in the pseudocode), not by K. The role of K is essentially a bit of further improvements of the results, as we have been found with some experimental studies.

4.3.2 Diversity Guided Explorations and Exploitations

In this section we present how DGM tries to balance between explorations and exploitation using population diversity information. Mutations with small step size usually induce small amount of variations to the parent chromosomes and thus exploits existing solutions until the population gets stuck at local optima. In contrast, mutations with large step size are more likely to escape from local optima and thus better suited for search space explorations, but may lack the stability to perform in-depth tuning of the existing solutions and thus may fail to locate or may oscillate around the global optimum. Some existing works try to control mutation step size following some rigid adaptation strategy [4], [161] or using different distributions [56], [57] or outcomes of previous mutations [165]. In contrast, DGM adapts the mutation step size by picking distance values from the population. Exploitation is conducted by picking two very similar chromosomes and employing the small genotype distance between them which ensures a small step size that is suitable for exploitation. If this distance value is d_1, the mutation step size is generated uniformly at random from $[0, K*d_1]$. Exploration is carried out following the same way, but the distance between two non-neighbors, say d_2, is employed and the mutation step size is generated uniformly at random from $[0, K*d_2]$. The pair of distance values, d_1 and d_2 acts as sampled, approximate diversity of neighbors and non-neighbors across the population. Thus the distribution and distances of neighbors and non-neighbors from a particular chromosome $\boldsymbol{x}_i$ controls the degree of exploitations and explorations around $\boldsymbol{x}_i$. During the early generation both the neighbors and non-neighbors would be far apart from each other, which would make the distance values d_1 and d_2 to be large in most instances and the evolutionary search would be mostly explorative. As the population evolves, chromosomes would approach the local peaks with the formation of several neighbor groups within the population. However, considering a high dimensional multimodal problem with number of local optima much higher than the number of evolving chromosomes (which is usually the case), the distance between different neighborhoods would still be significantly large. Thus, the inter neighborhood diversity d_2 would be sufficiently high to carry out explorations in parallel to the exploitations with small intra-neighborhood diversity d_1. Unless the entire population converges to a single global optimum, both explorations and exploitations would continue and guide the evolution to better solutions. In addition to the mutation operator, the selection operator of DGEP also possesses both explorative and exploitative features. While the fitness based selection exerts mainly exploitative pressure on the evolution, the elimination of duplicate chromosomes and other chromosomes that get trapped at local optima (i.e., no fitness improvement for a long time) put explorative pressure on the evolutionary search. The synergy of these explorative and exploitative forces ensures better balance and higher amount of population diversity in comparison to classical evolutionary approaches, as demonstrated later (Table 8.14, chapter 8).

Balancing exploitation and exploration has also been addressed by some other works, such as RCMA [160], LSRCMA [162] and NSDE [163], [164]. All these works differentiate the degree of exploitations and exploration on different chromosomes based on their fitness values. This probably makes them more exploitative than explorative. DGEP does not differentiate the degree of exploitations and explorations on the chromosome level; rather it attempts both explorative and exploitative variation from each chromosome. Also, unlike RCMA [160], LSRCMA [162] and NSDE [163], [164], the selection operator of DGEP puts some explorative pressure with elimination of duplicate and stagnant chromosomes. Thus both the evolutionary operators of DGEP, i.e. selection and mutation, exhibit awareness towards explorations, exploitations and population diversity to avoid premature convergence.

To some extent, the degree of exploration that DGEP is able to conduct might be controlled by K, as found in the pseudocode of DGM (step 5 in Fig. 4.1). K acts as a scaling factor of the distance, say d, between neighbors or non-neighbors. For example, consider an optimization problem with only one parameter. A chromosome with a parameter value v is mutated uniformly at random within the range $[v - K*d, v + K*d]$. The standard deviation of the mutation perturbation would be $\frac{K*d}{\sqrt{3}}$, which is proportional to both K and d. Thus, larger values of K (and d) are likely to provide better explorations. However, more exploration is not always helpful, especially once the neighborhood of the global optima is found or faster convergence is sought. Instead, what is always essential is a proper balance between explorations and exploitations. With our scheme of choosing d from two different ranges of values (exploitative and explorative), we leave up K to a simple random value. With some experiments, it is observed that a uniform random value of K within some small range, e.g., [0, 3.0], produces sufficiently good results and is often better than a fixed value of K. This is why K is simply picked uniformly at random from [0, 3.0], which is quite arbitrary and meant not to be optimal. With such simple choice of K, exploitation and exploration is mainly carried out by choosing between d_1 and d_2, i.e., the exploitative and explorative distances to guide the mutation step size.

4.3.3 Avoiding Premature Convergence

To avoid premature convergence, the search operators should have some capacity to escape from the locally optimal points of the search space. In the following two scenarios, we depict how DGEP possesses an inherent immunity against local optima and can conduct search until the population is driven to the globally optimum point. For the ease of visualization, a maximization problem with only one parameter is considered. Scenario #1 (Fig. 4.2) depicts a situation where the search process has failed to spot the globally maximum point and the entire population has converged to two local maxima. In this situation, the average distance among

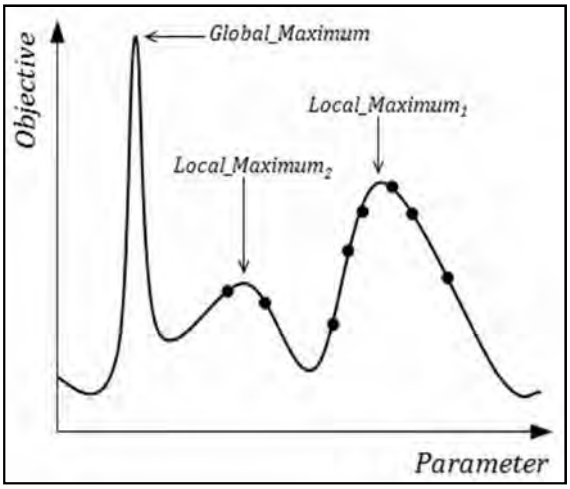


Figure 4.2: Scenario #1 — the entire population has converged to a few locally optimal hills, but missing the narrow global optimum.

neighbors would be pretty small and help the chromosomes hill climb towards the locally optimal objective values, *Local_Maximum*$_1$ and *Local_Maximum*$_2$, as depicted in Fig. 4.2. If the two peaks are well-separated with significant amount of distance between them, it would be quite difficult for traditional mutation operators to have sufficiently large step size in order to escape from the local maxima. However, the higher the distance between the two neighborhoods, the more explorative mutation step size DGEP would pick from their distances which makes it easier for DGEP to break free from the local maxima. Once the vicinity of the global maximum is found, both the intra-neighborhood and inter-neighborhood distances start to drop and allow DGEP to converge to the global maximum.

The worst possible scenario that can appear during evolutionary search is the scenario #2 (Fig. 4.3) where the entire population has converged to a single locally maximal point, far from the narrow global maximum. If the distance to *Global_Maximum* is significantly large, it would be extremely difficult to improve the scenario by using any simple, e.g., [137] or adaptive, e.g., [140], [222], mutation scheme. Use of specialized selection operator, e.g., [155], [224], [225] or probability distributions, e.g., [56], [57] for mutation would not help in such a situation. This is because the appropriately large mutation step size in such situations cannot be derived from the population information and the traditional fitness based selection operator discards the newly produced offspring that may be actually closer to the *Global_Maximum* but have lower objective value. However, DGEP still possess some probability to improve this situation because of its specialized selection operator. As the entire population climbs towards the single local maximum that entraps the entire population (i.e., *Local_Maximum*$_1$), some chromosomes would be more or less duplicates of each other. Also, some chromosomes, after reaching very close to the objective value of *Local_Maximum*$_1$ (Fig. 4.3) would get into fitness stagnation for a long

time. This triggers the selection operator, as described in steps (7) and (8) of DGEP, to replace these chromosomes by randomly producing chromosomes across the search space and applying a sequence of diversity guided mutations intermixed with hill climbing Gaussian mutation steps before exerting selection pressure on them. Random placement of the offspring chromosomes is likely to induce large variations from the current population points, while the sequence of hill climbing and mutation steps provide them with good survival probability when compared to the existing chromosomes. As a result, some of the new chromosomes may reach some other locally maximal points and turn this scenario into the previous scenario #1 (i.e., the population contains multiple locally maximal points), which is much easier to deal with. Thus DGEP ensures better probability that the search for global maximum continues avoiding permanent premature convergence around the local maxima.

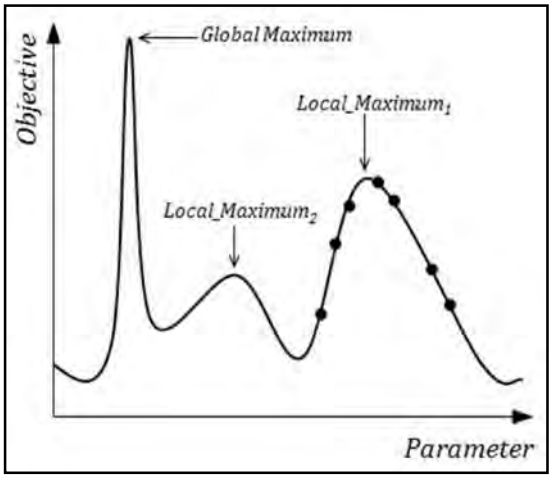


Figure 4.3: Scenario #2 — the entire population has converged to a single locally maximal hill, far from the narrow global maximum.

4.4 Differences of DGEP with Other Existing Works

DGEP differs from most other existing evolutionary algorithm approaches in a number of ways. First, it neither tries to find an optimal mutation rate and step size (e.g., [137]) nor tries to adapt their values using some adaptation rule (e.g., [140], [222]). It rather uses existing distances among chromosomes as a measure of diversity across neighbors and non-neighbors of the population and guides the mutation operation with this information. Mutation step size is not explicitly controlled by any rigid adaptation formula, rather becomes guided by the distances between chromosomes within the search space. As the fitness landscape may have complex and highly non-linear characteristics, any fixed adaptation rule based on generation number (e.g., similar to evolutionary schemes [140], [222]) is likely to fail to respond to the continuously evolving optimization scenario.

Second, DGEP tries to balance the exploitative and explorative search requirements during each mutation by producing two different offspring from each chromosome: one with small step size to facilitate exploitation around the parent chromosome and the other with large enough step size for search space exploration. Most existing approaches don't use such explicit measure to balance between exploitations and explorations. The use of two widely different ranges of mutation variations provides DGEP an inherent immunity against local minima.

Third, most existing approaches (e.g., [151]–[165]) tweak either the selection operator or the variation operator of the standard EA to promote diversity and prevent premature convergence, while DGEP alters both its variation and selection operators. While the DGM mutation scheme plays the key role of employing diversity information carrying out effective mutations, the selection operator also assists it by promoting diversity with the elimination of the chromosomes that are either duplicates or strongly trapped in some local optima. Altering only one operator (e.g., selection) and leaving the other (i.e., mutation) unchanged may allow the operators to play reverse roles in the evolution and nullify each other's effect resulting in unacceptably slow convergence speed and poor solution quality.

Fourth, some approaches, e.g., [141], [146], [159], attempt to estimate the population diversity using some diversity metric and try to make key decisions based on this diversity value. Since there is no generally accepted metric to measure population diversity, DGEP uses very general all-purpose Euclidean distance to measure the distances between the genotypes of the chromosomes. Using simple metric provides a way for plain and simple interpretation and analysis of the operations.

Fifth, unlike evolutionary systems those employ complex procedures to promote population diversity, e.g., island [142], nation [72], religion [144], immigration [226], reserve population [146], switching to and from different genetic operations [141], DGEP employs very simple diversity preserving measures. DGEP eliminates only those chromosomes that don't contribute significantly in the evolutionary search, i.e., chromosomes that are duplicates or almost identical to some other existing chromosome and the chromosomes that have not been improved for a long time. However, in the latter case, DGEP makes a series of variation operations to improve it before discarding it completely. Simple measures for diversity preservation make the system open for clear interpretation and easier analysis.

4.5 Evaluation of DGEP on Benchmark Functions

This section evaluates the performance of DGEP on a number of benchmark functions and compares it with several other existing evolutionary systems. Many evolutionary systems exist in the literature against which we could compare DGEP. However, since DGEP uses mutation as the sole variation operator and makes use of population diversity information to adjust the

mutation variation and to balance between exploitations and explorations, we primarily consider Classical EP (CEP) [3], Improved Fast EP (IFEP) [57], Adaptive EP with Lévy Mutation (ALEP) [56], Island Model GA (IMGA) [142], [143], GA with Restricted Truncation Selection (RTS) [158], Dual Population Genetic Algorithm (DPGA) [146], the Real Coded Memetic Algorithm (RCMA) with Crossover Hill Climbing (XHC) [160], Comprehensive Learning Particle Swarm Optimizer (CLPSO) [69], RCMA with Adaptive Local Search (LSRCMA) [162], Differential Evolution with Neighborhood Search (NSDE) [163], [164] and Covariance Matrix Adaptation Evolution Strategy (CMAES) [165] for comparison. Like DGEP, both ALEP [56] and IFEP [57] use only mutation for producing offspring. IFEP [57] has mixed, rather than switched, Cauchy and Gaussian mutations in one algorithm. This algorithm generates two candidate offspring from each parent: one by Cauchy mutation and one by Gaussian mutation. The better candidate is then chosen by IFEP [57] as the offspring. ALEP [56], on the other hand, generates four candidate offspring from each parent by Lévy mutation with four different distributions. It has been shown that ALEP [56] and IFEP [57] perform better than either their non-adaptive versions or the classical EP (CEP) [3]. IMGA [142], [143] uses multiple sub-populations following the island model with periodic exchange of individuals between the islands according to some predetermined migration policy. The isolated nature of the islands helps the populations evolve separately which lowers gene flow and promotes population diversity. RTS [158] is a crowding method that is somewhat different from the standard crowding and usually produces better results than all other variants of crowding. DPGA [146] maintains a reserve population in addition to the main population for promoting diversity. Both the populations evolve through generations, but with completely different objectives. While the main population evolves to optimize fitness value, the reserve population evolves with the purpose of providing useful diversity around the best chromosomes of the main population. The distance value between the two populations is mostly adapted for exploitations, but switched to explorations when the main population gets trapped into local optima for several generations. RCMA with XHC [160] executes explorative and exploitative operations separately and combines them in one algorithm. It uses PBX crossover [160] and BGA mutation [81] for exploration. RCMA employs a negative assortative mating strategy for selecting two parents to perform crossover in order to introduce population diversity. RCMA with a specialized crossover operator, XHC [160] has been shown to perform better than all other variants. NSDE [163], [164] and LSRCMA [162] tries to balance between exploitations and explorations by combining the benefits of exploitative local search or neighborhood search with their more explorative component of differential evolution or memetic algorithm, respectively.

4.5.1 DGEP on Standard Benchmark Functions

At first, a set of 30 standard benchmark functions, as introduced in section 2.17 and summarized in Table 2.3, is used to evaluate and compare the performance of DGEP. Based on their properties, the standard functions can be divided into three groups: the functions with no local minima (f_1–f_9, Table 2.3), many local minima (f_{10}–f_{18}), and only a few local minima (f_{19}–f_{30}). Further details on these benchmark functions with their analytical forms, characteristics and figures can be found in the Appendix A.

For the evaluations of DGEP (i.e., Tables 4.1– 4.9), the population size was set to 50 and the number of function evaluations (FEs) was set to 150,000 for the high-dimensional functions f_1–f_{18} and 10,000 for the low dimensional functions, f_{19}–f_{30}. These values are chosen to make a fair comparison with the other works in comparison. The remaining parameters of DGEP are u (maximum number of generations without fitness improvement before discarding a chromosome), l (ratio of genotype distances to determine duplicates), $|N|$ (number of neighbors for each chromosome) and $|A|$ (number of genes to be mutated by the DGM scheme). Except with some extreme choice of values, the performance of DGEP is not very sensitive to these parameters. For the results in Table 4.1, we have employed DGEP with four different 'moderate' settings of these parameter values — DGEP^1 with u = 50, l = 0.01, $|N|$ = 5 and $|A|$ = 1, DGEP^2 with u = 30, l = 0.03, $|N|$ = 10 and $|A|$ = 3, DGEP^3 with u = 70, l = 0.005, $|N|$ = 4 and $|A|$ = 5, DGEP^4 with u = 40, l = 0.01, $|N|$ = 6 and $|A|$ = 4. These values are quite arbitrary and meant not for optimum. The performance of DGEP is not much sensitive to these parameter values, unless some extreme choice of values. According to [214], CEP has been implemented for the same population size and number of function evaluations as of DGEP.

Table 4.1 shows the mean best results of DGEP on the 30 standard benchmark functions. DGEP has been employed with four different parameter settings, as denoted by DGEP^1, DGEP^2, DGEP^3 and DGEP^4, and the parameter values are just as specified in the previous paragraph. Each DGEP variant has made 50 different runs on each function and the mean of the best results found over the 50 different runs are presented in Table 4.1. The following observations can easily be made from the results in Table 4.1 and Figs. 4.4 and 4.5.

a) First, all four DGEP variants, each one with its different parameter settings, have reached the sufficiently close to the global minimum (i.e., mean error = 0) for almost all the functions (Table 4.1). This indicates the effectiveness of the proposed DGEP algorithm over the wide and diverse range of its parameter values.

b) Fig. 4.4 shows that the mean absolute error, achieved by each DGEP variant, is quite low. However, among the four DGEP variants, the minimum magnitude of the mean absolute error is achieved by DGEP[1], so it will be simply referred as DGEP in all the subsequent experiments of this chapter (Tables 4.2–4.11, Figs. 4.5–4.8). As Fig. 4.5 shows, DGEP (i.e., DGEP[1]) has achieved nearly log-linear convergence and reached sufficiently close to the global minimum, very consistently for all the functions in Fig. 4.5. Many EP and ES schemes fail to do this [161], especially for the multimodal functions f_{10}–f_{30}. In comparison to CEP, the proposed DGEP scheme shows faster convergence speed and better final solution quality for all the functions (Fig. 4.5, Table 4.3).

c) The low standard deviation of the results in Table 4.1 by the DGEP variants indicates that DGEP is very consistent and robust, across their 50 independent runs, for almost all the functions. On average, the magnitude of the standard deviation is only around 27%, 33%, 30% and 23% of the mean results from DGEP[1], DGEP[2], DGEP[3] and DGEP[4], respectively.

Table 4.1: Performance of the DGEP variants on the 30 standard benchmark functions, each variant using a different setting of the parameter values. Results have been averaged over 50 independent runs. The best results are marked with boldface font, if not identical with the results from the other algorithms.

No	f_{min}	**DGEP[1]**		**DGEP[2]**		**DGEP[3]**		**DGEP[4]**	
		Mean Error	Std Dev	Mean Error	Std Dev	Mean Error	Std Dev	Mean Error	Std Dev
f_1	0	5.42x10^{-8}	1.48x10^{-8}	2.46x10^{-7}	1.36x10^{-7}	8.16x10^{-8}	6.77x10^{-9}	**3.32x10^{-9}**	**9.40x10^{-10}**
f_2	0	**6.64x10^{-12}**	**3.61x10^{-12}**	9.07x10^{-12}	8.61x10^{-12}	1.32x10^{-11}	1.16x10^{-11}	5.54x10^{-11}	6.56x10^{-12}
f_3	0	**1.06**	**0.32**	3.16	0.56	2.39	0.44	4.80	1.78
f_4	0	4.13x10^{-8}	9.25x10^{-9}	6.79x10^{-7}	8.84x10^{-8}	**2.05x10^{-8}**	**7.12x10^{-9}**	7.40x10^{-8}	3.11x10^{-8}
f_5	0	**2.48x10^{-4}**	**1.33x10^{-4}**	6.24x10^{-4}	2.97x10^{-4}	4.54x10^{-4}	1.05x10^{-4}	2.71x10^{-4}	1.53x10^{-4}
f_6	0	**4.12x10^{-2}**	**7.97x10^{-3}**	4.54x10^{-2}	9.75x10^{-3}	1.01x10^{-1}	3.78x10^{-2}	1.55x10^{-1}	4.06x10^{-2}
f_7	0	1.28	0.76	3.19	0.84	**1.25**	**0.48**	2.85	1.08
f_8	0	0	0	0	0	0	0	0	0
f_9	0	1.94x10^{-12}	8.23x10^{-13}	**6.58x10^{-13}**	**9.76x10^{-14}**	3.10x10^{-12}	7.67x10^{-13}	2.99x10^{-12}	6.03x10^{-13}
f_{10}	0	**1.56x10^{-12}**	**5.12x10^{-13}**	9.63x10^{-12}	5.65x10^{-12}	2.81x10^{-12}	8.42x10^{-13}	9.37x10^{-12}	1.61x10^{-12}
f_{11}	0	2.85x10^{-5}	4.12x10^{-6}	9.35x10^{-6}	1.88x10^{-6}	**8.62x10^{-7}**	**3.15x10^{-7}**	4.40x10^{-6}	1.56x10^{-6}
f_{12}	–12569.5	**2.10**	**1.38**	5.65	2.02	15.15	6.90	4.53	0.95
f_{13}	0	2.47x10^{-16}	6.83x10^{-17}	7.73x10^{-17}	3.05x10^{-17}	4.60x10^{-17}	8.24x10^{-18}	**8.04x10^{-18}**	**1.32x10^{-18}**

Table 4.1 (continued): Performance of DGEP variants on the 30 standard benchmark functions.

No	f_{min}	DGEP[1]		DGEP[2]		DGEP[3]		DGEP[4]	
		Mean Error	Std Dev	Mean Error	Std Dev	Mean Error	Std Dev	Mean Error	Std Dev
f_{14}	0	**7.52×10^{-14}**	**2.54×10^{-14}**	2.02×10^{-12}	9.18×10^{-13}	4.64×10^{-11}	2.03×10^{-11}	7.93×10^{-12}	2.54×10^{-12}
f_{15}	0	9.53×10^{-13}	2.50×10^{-13}	7.82×10^{-11}	4.46×10^{-11}	7.58×10^{-12}	3.11×10^{-12}	**6.50×10^{-13}**	**2.11×10^{-13}**
f_{16}	0	**5.50×10^{-8}**	**8.21×10^{-9}**	6.44×10^{-6}	1.21×10^{-6}	6.52×10^{-7}	8.57×10^{-8}	5.72×10^{-8}	8.40×10^{-9}
f_{17}	0	**3.32×10^{-12}**	**1.94×10^{-13}**	2.65×10^{-10}	1.80×10^{-10}	8.82×10^{-11}	4.94×10^{-11}	7.71×10^{-12}	7.99×10^{-13}
f_{18}	0	**1.74×10^{-4}**	**2.65×10^{-5}**	2.25×10^{-4}	1.80×10^{-4}	1.75×10^{-4}	2.65×10^{-5}	1.98×10^{-4}	2.04×10^{-5}
f_{19}	1	**0.02**	**0.01**	0.05	0.03	0.05	0.03	0.04	0.02
f_{20}	3.07×10^{-4}	2.17×10^{-4}	8.2×10^{-5}	2.57×10^{-4}	8.10×10^{-4}	**2.25×10^{-5}**	**9.01×10^{-6}**	2.67×10^{-4}	7.2×10^{-5}
f_{21}	−1.0316	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f_{22}	0.398	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f_{23}	−3.86	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f_{24}	−3.32	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f_{25}	−10.15	0.28	0.11	0.35	0.08	**0.21**	**0.07**	0.42	0.09
f_{26}	−10.40	0.05	1.1×10^{-2}	**0.03**	**1.2×10^{-2}**	0.15	6.0×10^{-2}	0.08	2.2×10^{-2}
f_{27}	−10.54	**0.04**	**8.8×10^{-3}**	0.09	3.3×10^{-2}	0.18	7.8×10^{-2}	0.10	4.0×10^{-2}
f_{28}	0	**0.38**	**9.3×10^{-2}**	0.45	1.2×10^{-1}	0.39	1.4×10^{-1}	0.50	1.9×10^{-1}
f_{29}	−9.66	0	0	0	0	0.01	2.8×10^{-3}	0	0
f_{30}	−1.4	0.18	5.5×10^{-2}	0.23	4.2×10^{-2}	**0.16**	**7.0×10^{-2}**	0.33	1.1×10^{-1}

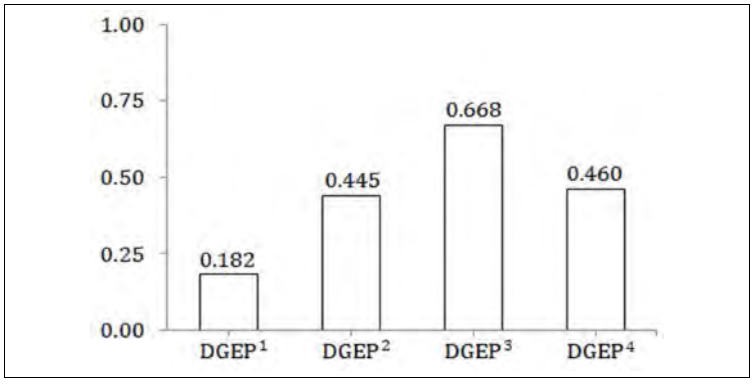


Figure 4.4: The mean absolute error values of the DGEP variants on the standard benchmark functions. All the DGEP variants exhibit sufficiently small error values, with DGEP[1] showing the least mean absolute error.

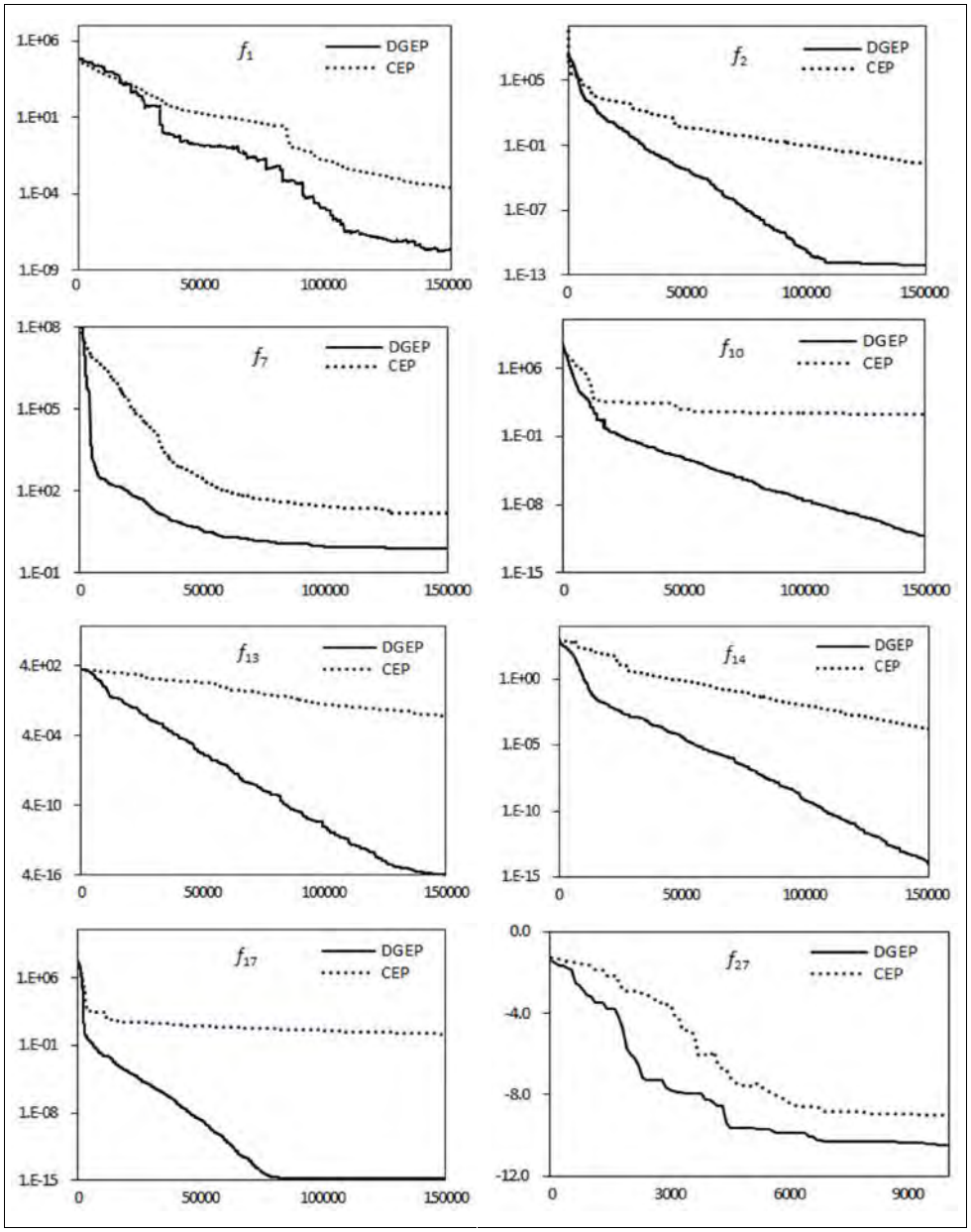


Figure 4.5: Convergence characteristics of DGEP and CEP [102] on the unimodal functions f_1, f_2, f_7, multimodal high-dimensional functions f_{10}, f_{13}, f_{14}, f_{17} and the low-dimensional function f_{27}. The vertical axis shows the function value, while the horizontal axis shows the number of function evaluations.

Table 4.2 presents the results of DGEP along with three other algorithms — CEP [102], ALEP [56] and IFEP [57]. The results of ALEP and IFEP are reported only for the 12 and 7 functions, as available from [56] and [57], respectively. The remaining missing results are shown with a '–' in Table 4.2. DGEP outperforms both ALEP and IFEP on almost all these available functions, so we can just guess about somewhat similar performance on the rest of the functions. The results of DGEP have been averaged over 50 independent runs, as has been done

and reported for CEP [102], ALEP [56] and IFEP [57]. Fig. 4.5 shows the convergence characteristics of DGEP and CEP [102] on several functions in terms of the mean best fitness value. It is evident from the Fig. 4.5 that DGEP achieves nearly log-linear convergence until it has reached sufficiently close to the global minimum. The evolutionary process progresses smoothly, almost no where getting stuck at the intermediate local minima until it reaches very close proximity of the global minimum, while many evolutionary algorithms (e.g., CEP) expend significant amount of time being stuck at several intermediate local minima [161]. As obvious from Table 4.2, the performance of DGEP is remarkably better than the other three algorithms. The *t*-test confirms with at least 95% certainty that the improvement of DGEP is statistically significant for most of the functions. DGEP is significantly better than IFEP [57] on most (i.e., six out of seven) of the functions, while IFEP is better on the remaining one function only. Furthermore, DGEP has outperformed both CEP (12 out of 12) and ALEP (11 out of 12) on all or most of the test functions. Thus the overall performance of DGEP is much better than all three of its counterparts. For more comparison between CEP and DGEP, we have re-implemented CEP, using the same settings as mentioned in [3], and results are compared in Table 4.3, which shows that DGEP significantly outperforms CEP on most (27 out of 30) of the functions, while both the algorithms perform equally well on the remaining three.

Table 4.2: Comparison among DGEP, ALEP [56], IFEP [57] and CEP [102] on 12 standard benchmark functions. Results have been averaged over 50 independent runs. The best results are marked with boldface font. The '+' indicates that DGEP is significantly better than the compared algorithm with at least 95% certainty, while '≈' means that the difference is not statistically significant.

Function	Mean Error				*t*-Test (DGEP vs.)	
	DGEP	IFEP	ALEP	CEP	IFEP	ALEP
f_1	5.42e–08	4.16e–05	6.32e–04	2.2e–04	+	+
f_3	**4.13e–08**	–	4.18e–02	5.0e–02	-N/A-	+
f_7	1.28	–	4.34e+01	6.17e+01	-N/A-	+
f_{10}	**1.56e–12**	–	5.85e+00	8.9e+01	-N/A-	+
f_{12}	2.10	**8.87e–02**	1.00e+02	4.6e+03	–	-N/A-
f_{13}	**2.47e–16**	4.83e–03	1.90e–02	9.2e+00	+	+
f_{14}	**7.52e–14**	4.54e–02	2.40e–02	8.6e–02	+	+
f_{17}	**3.32e–12**	–	6.00e–06	1.76e+00	-N/A-	+
f_{18}	**1.74e–04**	–	9.80e–04	1.04e+00	-N/A-	+
f_{25}	**0.28**	3.69	0.61	3.29	+	+
f_{26}	**0.05**	3.30	0.10	2.13	+	+
f_{27}	0.04	2.75	**0.01**	1.45	+	–

Table 4.3: Comparison between DGEP and CEP [102] on 30 standard benchmark functions. Results are averaged over 50 independent runs. Best results are marked with boldface font. The '+' indicates that DGEP is significantly better than CEP with at least 95% certainty, while '≈' means that the difference is not statistically significant.

No	f_{min}	DGEP		CEP		*t*-Test (DGEP vs. CEP)
		Mean	Std Dev	Mean	Std Dev	
f_1	0	**5.42e–08**	**1.48e–08**	2.4e–04	5.9e–05	+
f_2	0	**6.64e–12**	**3.61e–12**	2.6e–03	1.7e–04	+
f_3	0	**1.06**	**0.32**	3.2	1.2	+
f_4	0	**4.13e–08**	**9.25e–09**	5.0e–02	6.6e–02	+
f_5	0	**2.48e–04**	**1.33e–04**	7.3e–02	3.1e–02	+
f_6	0	**4.12e–02**	**7.97e–03**	5.0e–01	1.8e–01	+
f_7	0	**1.28**	**0.76**	6.17	13.61	+
f_8	0	**0**	**0**	577.76	1125.76	+
f_9	0	**1.94e–12**	**8.23e–13**	1.8e–02	6.4e–03	+
f_{10}	0	**1.56e–12**	**5.12e–13**	89.0	23.1	+
f_{11}	0	**2.85e–05**	**4.12e–06**	5.83	1.60	+
f_{12}	–12569.5	**2.10**	**1.38**	4.70e+03	1.3e+03	+
f_{13}	0	**2.47e–16**	**6.83e–17**	9.2	2.8	+
f_{14}	0	**7.52e–14**	**2.54e–14**	8.6e–02	1.2e–02	+
f_{15}	0	**9.53e–13**	**2.50e–13**	5.0e–02	7.3e–03	+
f_{16}	0	**5.50e–08**	**8.21e–09**	2.56	0.84	+
f_{17}	0	**3.32e–12**	**1.94e–13**	1.76	2.4	+
f_{18}	0	**1.74e–04**	**2.65e–05**	1.4	3.7	+
f_{19}	1	**0.02**	**0.01**	1.66	1.19	+
f_{20}	3.07e–04	**1.17e–04**	**8.2e–05**	4.7e–04	1.3e–04	+
f_{21}	–1.0316	0.00	0.00	0.00	4.9e–07	≈
f_{22}	0.398	0.00	0.00	0.00	1.5e–07	≈
f_{23}	–3.86	0.00	0.00	0.00	1.4e–05	≈
f_{24}	–3.32	**0.00**	**0.00**	0.04	5.8e–04	+
f_{25}	–10.15	**0.28**	**0.11**	3.29	2.67	+
f_{26}	–10.40	**0.05**	**1.1e–02**	2.13	0.95	+
f_{27}	–10.54	**0.04**	**8.8e–03**	1.45	0.72	+
f_{28}	0	**0.38**	**9.3e–02**	2.42	0.82	+
f_{29}	–9.66	**0**	**0**	0.53	0.18	+
f_{30}	–1.4	**0.18**	**5.5e–02**	0.93	0.47	+

Next, DGEP is compared with a number of representative evolutionary and swarm intelligence algorithms — GA [11], DE [195], PSO [196] and ABC [11]. All these algorithms have two parameters in common — the population size M and the total number of function evaluations (FE), which are set as $M = 50$ and $FE = 500{,}000$ for all the functions in Table 4.4. The other algorithm specific parameters and settings are as follows. For GA [11], binary coded standard GA is used with fitness scaling, seeded selection, random selection, crossover, mutation and elitism. The stochastic uniform sampling technique has been used as the selection operator. The single point crossover operation with crossover rate = 0.8 and bit flip mutation with mutation rate = 0.01 is used. Generation gap is set to 0.9 (i.e., 90% of the population is replaced in each generation). For the next algorithm — DE [195], the scaling factor F and the crossover rate CR are set to 0.5 and 0.9, respectively. For PSO [196], the cognitive component $¤_1$, social component $¤_2$ and the inertia weight w are set to 1.8, 1.8 and 0.6, respectively, as suggested in [196]. For the next algorithm ABC [11], the control parameter *limit* is set as $SN * D$, as recommended in [11]. Here, SN is the number of food sources or employed bees, which is set to 25 (i.e., $M/2$) and D is the dimensionality of the problem. Table 4.4 compares DGEP with GA [11], DE [195], PSO [196] and ABC [11] on a total of 17 high dimensional standard benchmark functions from f_1–f_{30}. The following points summarize the results.

GA vs. DGEP: DGEP performs significantly better than GA on all (i.e., 17 out of 17) the functions in Table 4.4.

PSO vs. DGEP: On majority of the functions (13 out of 17), DGEP performs better than PSO. For the remaining four functions, both the algorithms show similar performance.

DE vs. DGEP: The performance of DGEP is either better (i.e., 11 out of 17 functions) or at least equally good (i.e., remaining six functions) on all the functions.

ABC vs. DGEP: For most of the functions (12 out of 17), ABC and DGEP show similar performance. For the remaining five functions, DGEP performs better than ABC. Thus, the overall performance of DGEP is better than the basic ABC algorithm.

An overall evaluation of the algorithms can be made by using their mean absolute error over all the functions. To calculate the mean absolute errors, the total of the absolute errors over all the functions is divided by the number of functions (i.e., 17 for the results in Table 4.4). Fig. 4.6 shows that DGEP has the lowest mean absolute error, followed by the basic ABC algorithm, but the remaining algorithms (i.e., GA, PSO and DE) exhibit much larger error values.

Table 4.4: Comparison of DGEP with GA [11], DE [195], PSO [196] and ABC [11] on the standard benchmark functions. The best results are marked with boldface font.

No.	D	f_{min}	Mean Error				
			GA	PSO	DE	ABC	DGEP
f_1	30	0	1.1e+03	**0**	**0**	**0**	**0**
f_2	30	0	11.02	**0**	**0**	**0**	**0**
f_4	30	0	7.4e+03	**0**	**0**	**0**	**0**
f_5	24	0	9.70	1.1e–04	2.2e–07	3.1e–03	**8.4e–16**
f_6	30	0	1.2e+03	0.6666	0.6666	**0**	**0**
f_7	30	0	1.9e+05	15.08	18.20	0.088	**4.0e–02**
f_8	30	0	1.2e+03	**0**	**0**	**0**	**0**
f_9	30	0	1.8e–01	1.2e–03	1.4e–03	3.00e–03	**0**
f_{10}	30	0	52.92	43.97	11.72	**0**	**0**
f_{12}	30	–12569.48	8.8e+02	5.7e+03	2.3e+03	**0**	**0**
f_{13}	30	0	14.67	0.16	**0**	**0**	**0**
f_{14}	30	0	10.63	0.017	0.0015	**0**	**0**
f_{17}	30	0	13.38	0.021	**0**	**0**	**0**
f_{18}	30	0	125.06	7.7e–03	2.2e–03	**0**	**0**
f_{28}	10	0	29.57	1364.45	781.55	8.23	**0.06**
f_{29}	10	–9.66015	0.16	5.65	0.069	**0**	**0**
f_{30}	10	–1.4	0.76	1.39	0.35	0.97	**0.11**

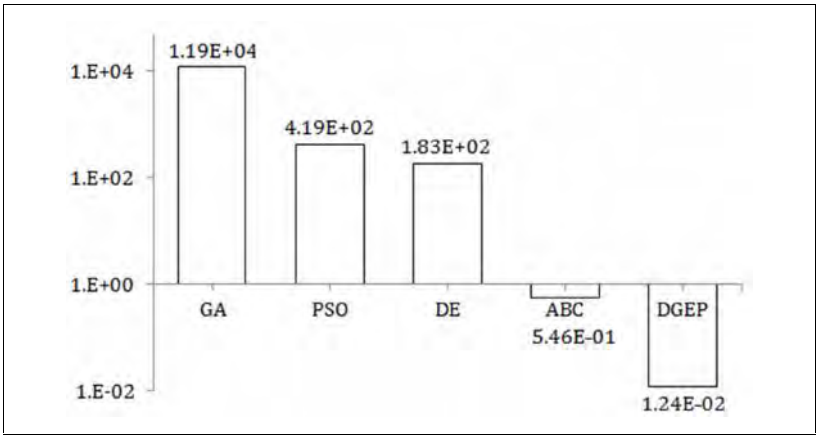


Figure 4.6: Comparison among DGEP, GA [11], DE [195], PSO [196] and ABC [11] based on their mean absolute errors on the standard benchmark functions. DGEP exhibits the best performance, i.e., lowest mean absolute error value.

The proposed system, DGEP is also compared with IMGA [142], [143], RTS [158] and DPGA [146] on the multimodal functions, f_{10}–f_{30} (Tables 4.5–4.7). The first nine functions, f_{10}–f_{18} are high dimensional functions for which DGEP outperforms IMGA [142], [143] and RTS [158] on five and four (out of six) functions respectively while they perform better only on the remaining one or two functions. The remaining algorithm, DPGA [146] performs better than DGEP on three functions, while DGEP outperforms it on the remaining three. For the low dimensional multimodal functions f_{19}-f_{30}, DGEP always either outperforms all of IMGA [142], [143], RTS [158] and DPGA [146] (on six functions) or shows similar performance (on the remaining three), as shown by Tables 4.6–4.7. The summary (Table 4.7) shows that DGEP outperforms all of IMGA [142], RTS [158] and DPGA [146] on the majority of these functions.

Table 4.5: Comparison among IMGA [142], [143], RTS [158], DPGA [146] and DGEP on the high dimensional multimodal functions, f_{10}–f_{18}. Best results are marked with boldface font.

Function	IMGA		RTS		DPGA		DGEP	
	Mean	Std Dev	Mean	Std Dev	Mean	Std Dev	Mean	Std Dev
f_{10}	0.358	0.746	**0**	**0**	**0**	**0**	$1.56\text{x}10^{-12}$	$5.12 \text{ x } 10^{-9}$
f_{12}	–12008.1	284.9	–12443.9	142.4	–12550.5	43.9	**–12567.4**	**0.18**
f_{13}	$4.69\text{x}10^{-15}$	$1.67\text{x}10^{-15}$	$5.26\text{x}10^{-15}$	$1.79\text{x}10^{-15}$	$3.55\text{x}10^{-15}$	$1.39\text{x}10^{-15}$	$\mathbf{2.47\text{x}10^{-16}}$	$\mathbf{6.83\text{x}10^{-17}}$
f_{14}	$3.54\text{x}10^{-3}$	$7.73\text{x}10^{-3}$	$2.07\text{x}10^{-3}$	$5.31\text{x}10^{-3}$	$1.28\text{x}10^{-3}$	$3.31\text{x}10^{-3}$	$\mathbf{7.52\text{x}10^{-14}}$	$\mathbf{2.54\text{x}10^{-14}}$
f_{17}	$2.48\text{x}10^{-7}$	$1.14\text{x}10^{-6}$	$1.79\text{x}10^{-32}$	$8.29\text{x}10^{-48}$	$\mathbf{1.57\text{x}10^{-32}}$	$\mathbf{8.29\text{x}10^{-48}}$	$3.32\text{x}10^{-12}$	$1.94 \text{ x } 10^{-13}$
f_{18}	$5.97\text{x}10^{-29}$	$3.48\text{x}10^{-28}$	$4.20\text{x}10^{-4}$	$1.55\text{x}10^{-3}$	$\mathbf{1.39\text{x}10^{-32}}$	$\mathbf{2.96\text{x}10^{-33}}$	$1.24\text{x}10^{-4}$	$2.65 \text{ x } 10^{-5}$

Table 4.6: Comparison among IMGA [142], [143], DPGA [146], RTS [158], and DGEP on the low dimensional multimodal functions, f_{19}–f_{27}. The best results by DGEP are marked with boldface font, if not identical with results from other algorithms.

Function	Gen	IMGA		RTS		DPGA		DGEP	
		Mean	Std Dev	Mean	Std Dev	Mean	Std Dev	Mean	Std Dev
f_{19}	100	1.410	0.676	1.018	0.141	1.355	0.687	**1.02**	**0.04**
f_{20}	4000	$6.53\text{x}10^{-4}$	$1.27\text{x}10^{-4}$	$5.81\text{x}10^{-4}$	$2.04\text{x}10^{-4}$	$5.86\text{x}10^{-4}$	$1.08\text{x}10^{-4}$	$\mathbf{2.17\text{x}10^{-4}}$	$\mathbf{8.2 \text{ x } 10^{-5}}$
f_{21}	100	–1.032	0	–1.032	0	–1.031	0	–1.032	0.00
f_{22}	100	0.398	0	0.398	0	0.398	0	0.398	0.00
f_{23}	100	–3.86	0	–3.86	0	–3.86	0	–3.86	0.00
f_{24}	200	–3.32	0.033	–3.28	0.057	–3.28	0.053	**–3.32**	$\mathbf{2.5\text{x}10^{-4}}$
f_{25}	100	–7.11	3.587	–9.14	2.543	–6.86	3.421	**–9.87**	**0.12**
f_{26}	100	–9.35	2.278	–10.04	1.480	–8.27	2.2597	**–10.27**	**0.08**
f_{27}	100	–9.27	2.760	–10.24	1.439	–9.10	2.052	**–10.51**	$\mathbf{2.2 \text{ x } 10^{-2}}$

Table 4.7: Comparison of DGEP with IMGA [142], RTS [158] and DPGA [146] based on statistical *t*-Test on the results from the previous Tables 4.5–4.6. The '+' symbol indicates that DGEP is significantly better than its competitor with at least 95% certainty, while '≈' means that the difference is not statistically significant.

Function	*t*-Test		
	DGEP vs. IMGA	DGEP vs. RTS	DGEP vs. DPGA
f_{10}	+	–	–
f_{12}	+	+	+
f_{13}	+	+	+
f_{14}	+	+	+
f_{17}	+	–	–
f_{18}	–	+	–
f_{19}	+	≈	+
f_{20}	+	+	+
f_{21}	≈	≈	≈
f_{22}	≈	≈	≈
f_{23}	≈	≈	≈
f_{24}	≈	+	+
f_{25}	+	+	+
f_{26}	+	+	+
f_{27}	+	+	+

Table 4.8 compares DGEP against another algorithm — RCMA with XHC [160]. Both DGEP and RCMA [160] are applied to six benchmark functions for 100,000 FEs and with dimensionality = 25, as suggested in [160]. Table 4.8 shows that RCMA with XHC [160] outperforms DGEP on only one unimodal function f_1, while DGEP outperforms RCMA [160] on the unimodal functions f_4, f_7 and the multimodal functions f_{10}, f_{13} and f_{14}. Although we can't perform *t*-test for the lack of availability of the standard deviation of the results, it is almost obvious from the magnitude of the results that the performance difference is statistically significant for all the functions.

It would be interesting to investigate the performance of DGEP against a hybrid algorithm that hybridizes machine learning techniques with techniques from EAs and SIAs. One such approach is the comprehensive learning particle swarm optimizer (CLPSO) [69], a variant of the particle swarm optimizer (PSO) [227]. CLPSO [69] uses a novel learning strategy in which all other particles' historical best information is used to update a particle's velocity to move the search process forward. It has demonstrated better performance than other variants of PSO for wide range of complex functions. Since the number of FEs used by CLPSO [69] is 200,000, DGEP is re-implemented for the same FEs. Table 4.9 presents results for DGEP and CLPSO [69] on eight functions over 50 independent runs. DGEP performs better than CLPSO [69] on five (out of

eight) functions, while CLPSO [69] outperforms DGEP on the remaining three only. The performance differences are clearly significant for all the functions. Thus, the overall performance of DGEP is better than its counterpart, CLPSO [69].

Table 4.8: Comparison of DGEP with RCMA-XHC [160] on six benchmark functions with dimensions = 25. Best results are marked with boldface font. The '+', '≈' and '–' indicate that DGEP is significantly better, similar and worse, respectively than RCMA-XHC [160].

Function	Mean Error		*t*-Test
	DGEP	RCMA	DGEP vs. RCMA
f_1	1.26e–11	**6.5e–101**	–
f_4	**1.05e–15**	3.81e–07	+
f_7	**1.83e–01**	2.2e+00	+
f_{10}	**1.17e–14**	1.4e+00	+
f_{13}	**7.89e–12**	7.9e–01	+
f_{14}	**9.23e–15**	1.3e–02	+

Table 4.9: Comparison between DGEP and CLPSO [69] on eight standard benchmark functions. Best results are marked with boldface font. The '+', '≈' and '–' indicate that DGEP is significantly better, similar and worse, respectively than its counterpart, CLPSO [69].

Function	Mean Error ± Std Dev		*t*-Test
	DGEP	CLPSO	DGEP vs. CLPSO
f_1	2.86e–10 ± 6.81e–11	**4.46e–14 ± 1.73e–14**	–
f_7	**0.45 ± 0.063**	2.10e+01 ± 2.98e+00	+
f_{10}	**5.73e–14 ± 2.55e–14**	4.85e–10 ± 3.63e–10	+
f_{11}	**9.37e–12 ± 3.50e–12**	4.36e–10 ± 2.44e–10	+
f_{12}	6.56e–13 ± 3.08e–13	**0.00e+00**	–
f_{13}	**2.56e–15 ± 8.44e–16**	4.32e–14 ± 2.55e–14	+
f_{14}	**9.14e–16 ± 5.27e–16**	4.56e–03 ± 4.81e–03	+
f_{16}	9.02e–10 ± 6.78e–10	**0.00e+00**	–

4.5.2 DGEP on CEC2005 Benchmark Functions

DGEP is also evaluated on a new set of benchmark functions introduced in the special session on real-parameter optimization at the CEC2005 [76]. A brief overview of the CEC2005 benchmark functions is presented in the previous section 2.17 (Table 2.4). More details on each function can be found in the Appendix A. An interested reader can find the complete procedure to compute each of the CEC2005 functions along with the necessary constants, matrices, data files, characteristics and sample plots in the technical paper [76].

To evaluate and compare the performance of DGEP, the dimension of all these functions is set to 30 and the FEs are set to be 3.0e+05. This setup is for a fair comparison with some other existing algorithms, e.g., LSRCMA [162], NSDE [43], [163] and CMAES [165]. The mean error values of 50 independent runs for DGEP, LSRCMA [162], NSDE [43], [163] and CMAES [165] are presented in Tables 4.10–4.11. Results indicate that the performance of DGEP is quite comparable to and often better than the other three algorithms. We have summarized our observations on their performance comparison in the following few points.

- The first five functions (F1–F5) are the only unimodal ones in the CEC2005 suite. Over these five functions, DGEP outperforms its competitors (i.e., LSRCMA, NSDE and CMAES) on three, three and two functions, while they perform better on the remaining two, two and three functions, respectively. Also, the mean absolute errors of the results from DGEP on both the non-composite functions F1–F14 and the hybrid composite functions F15–F25 are significantly smaller than all of LSRCMA, NSDE and CMAES, as has been shown in Fig. 4.8. This indicates that the overall performance of DGEP is better than all its competitors in the comparison, i.e., the LSRCMA [162], NSDE [163], [164] and CMAES [165].

Table 4.10: Comparison of DGEP, LSRCMA [162], NSDE [163], [164] and CMAES [165] on the benchmark functions F_1–F_{14} of the CEC2005 suite [76]. Results have been averaged over 50 independent runs. The best results are marked with boldface font.

Function	Mean Error			
	DGEP	**LSRCMA**	**NSDE**	**CMAES**
F_1	7.56e–08	9.36e–09	**0.00e+00**	5.28e–09
F_2	**3.98e–10**	8.71e–06	5.62e–08	6.93e–09
F_3	2.24e+01	8.77e+05	6.40e+05	**5.18e–09**
F_4	2.76e+03	3.96e+01	**9.02e+00**	9.26e+07
F_5	5.11e+00	2.18e+03	1.56e+03	**8.30e–09**
F_6	2.19e+01	4.95e+01	2.45e+01	**6.31e–09**
F_7	7.23e–06	1.32e–02	1.18e–02	**6.48e–09**
F_8	**2.00e+01**	2.07e+01	2.09e+01	**2.00e+01**
F_9	**1.87e–04**	6.80e–01	7.96e–02	2.91e+02
F_{10}	**2.08e+00**	9.05e+01	4.29e+01	5.63e+02
F_{11}	**5.15e+00**	3.11e+01	1.41e+01	1.52e+01
F_{12}	**3.09e+02**	4.39e+03	6.59e+03	1.32e+04
F_{13}	**1.03e–01**	3.96e+00	1.62e+00	2.32e+00
F_{14}	**5.13e+00**	1.25e+01	1.31e+01	1.40e+01

Table 4.11: Comparison of DGEP, LSRCMA [162], NSDE [163], [164] and CMAES [165] on the hybrid composite functions F15–F25 of the CEC2005 suite [76]. Results have been averaged over 50 independent runs. The best results are marked with boldface font.

Function	**DGEP**	**LSRCMA**	**NSDE**	**CMAES**
	Mean Error	Mean Error	Mean Error	Mean Error
F_{15}	4.88e+02	3.56e+02	3.64e+02	**2.16e+02**
F_{16}	**3.72e+01**	3.26e+02	6.90e+01	5.84e+01
F_{17}	1.62e+02	2.79e+02	**1.01e+02**	1.07e+03
F_{18}	**5.49e+02**	8.77e+02	9.04e+02	8.90e+02
F_{19}	**8.25e+02**	8.80e+02	9.04e+02	9.03e+02
F_{20}	**7.91e+02**	8.79e+02	9.04e+02	8.89e+02
F_{21}	5.00e+02	5.00e+02	5.00e+02	**4.85e+02**
F_{22}	**8.41e+02**	9.08e+02	8.89e+02	8.71e+02
F_{23}	**5.22e+02**	5.59e+02	5.34e+02	5.35e+02
F_{24}	**2.00e+02**	**2.00e+02**	**2.00e+02**	1.41e+03
F_{25}	2.05e+02	2.11e+02	**2.00e+02**	6.91e+02

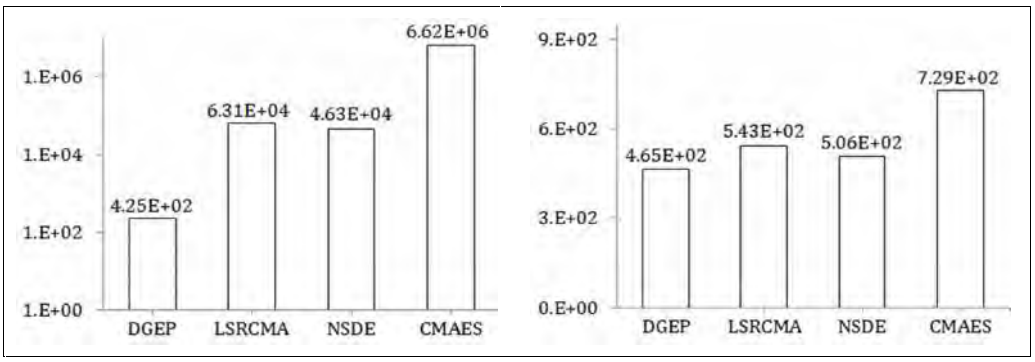


Figure 4.8: Comparison among DGEP, LSRCMA [162], NSDE [163] and CMAES [165], based on the mean absolute errors on the CEC2005 benchmark functions F1-F14 (on the left) and the hybrid composition functions F15-F25 (on the right).

- For the relatively more complex multimodal functions F6–F14, the superiority of DGEP is clearly visible. Not only does DGEP show the smallest error on F6–F15, but also it outperforms the other algorithms on most (six out of nine) of these functions.
- Functions F15–F25 are the most complex group of functions (hybrid composite functions) in the CEC2005 suite. The performance of all four algorithms is somewhat compromised on F15–F25, as shown by Table 4.11. However, DGEP still shows the best performance (i.e., outperforms the other three algorithms) on as many as seven (out of 11) hybrid composite functions, while its competitors (LSRCMA, NSDE and

CMAES) perform best only on one, three and two of these functions, respectively. Besides, DGEP show the lowest mean absolute error on F15–F25 (Fig. 4.8), which clearly indicates the superiority of DGEP over the other three algorithms.

4.6 Discussion on Results of DGEP

As observed in Tables 4.1–4.3, only three functions — f_3, f_7 and f_{12} seem to present some difficulty to all the EP-based algorithms (e.g., CEP [102], ALEP [56] and IFEP [57]) in our study. These functions are the Schwefel 2.21, Generalized Rosenbrock and the Generalized Schwefel's 2.26 functions. Although the generalized Rosenbrock function has been regarded as a unimodal function, there is evidence [228] suggesting that it contains several minima in high dimensional instances. The global optimum resides inside a long, narrow, parabolic shaped flat valley. Finding the valley is not difficult, but pinpointing the global optimum in an almost flat region is extremely difficult. For both of the Schwefel's functions (f_3 and f_{12}), the fitness landscapes have predominant flat and semi-flat search regions with narrow global basin. Searching for a narrow global basin in a flat search space with no useful gradient direction essentially turns into searching for a needle in a haystack. This is why all the algorithms (Tables 4.2, 4.3) face some difficulty, more or less, to locate the global minimum and may prematurely converge to some local minima for these functions. However, with sufficiently large number of function evaluations (e.g., 500,000 for the results in Table 4.4), DGEP successfully locates the global minimum and reaches very close proximity of the minimum value (e.g., mean error ≈ 0 for f_3, f_7 and f_{12} in Table 4.4 with FE = 500,000).

In Tables 4.6–4.7, DGEP frequently outperforms all of IMGA [142], [143], RTS [158] and DPGA [146] on the low dimensional multimodal functions f_{19}–f_{27}. However, DGEP receives relatively stronger competition from DPGA [146] on the high dimensional multimodal functions, f_{10}–f_{18}. The key difference between these two families of functions provides us an important insight. Functions f_{10}–f_{18} have hundreds of local optima, even with just two or three dimensions, as illustrated in Fig. 4.9 with the 3D surface plot of the 2D generalized Rastrigin function f_{10}. The number of local optima increases exponentially with the number of dimensions. With the high dimensionality = 30, the fitness landscape is occupied with exponentially many (i.e., very large number of) local minima. As a result, the distances among neighboring local peaks are usually quite small and escaping from a locally optimal point does not demand too high explorative capacity from the genetic search operators. On the other hand, functions f_{19}–f_{27} dimensionality ≤ 10, so their fitness landscapes have relatively fewer number of local optima. This often makes the peaks and valleys of their fitness landscapes to be well separated. Once the entire population converges to one or more isolated, strong locally optimal peaks, it requires relatively

stronger explorative capacity to break free from the locally optimum regions. Both DPGA [146] and IMGA [142], [143] maintain only two sub-populations, so it becomes difficult for them to represent all the distant local or global optimal points. RTS [158] performs relatively better on these functions, compared to DPGA [146] and IMGA [142], because its selection mechanism avoids crowding of chromosomes within the same optima and fosters well separated diverse chromosomes that evolve to the different optimal peaks in parallel. Since the number of optima for these functions is usually smaller than the population size, RTS [158] can keep track of all the optima and thus shows overall better performance than both of IMGA [142], [143] and DPGA [146]. However, DGEP performs still better on f_{19}–f_{27} outperforming all of IMGA [142], [143], RTS [158] and DPGA [146] because both its mutation and selection operators are well suited for such optimization scenario. After random initialization, the population of DGEP usually spans across the well separated local optima, which ensures a sufficiently large step size for the DGM mutation scheme, while its selection operator of GDEP discards similar chromosomes and fosters diverse ones that evolve towards different local peaks until the global optimum is found.

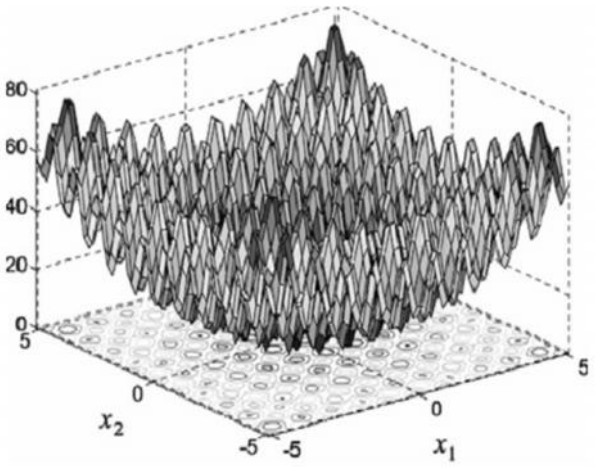


Figure 4.9: 3D surface plot of the 2-Dimensional Generalized Rastrigin Function

For the high dimensional multimodal functions f_{10}, f_{17} and f_{18}, DPGA [146] shows excellent strength in fine-tuning by minimizing to as low as 1e-32 (e.g., results for f_{17}, f_{18}, Table 4.5), which is better than the proposed DGEP scheme. Although DGEP locates the global optima for all these functions, it cannot perform such exploitations and fine tuning of the solutions. Once DGEP locates the global optima, the entire population reaches its close proximity and all the chromosomes become somewhat similar to each other. As a result, the selection operator of DGEP eliminates the nearly duplicate chromosomes and replaces them with randomly placed chromosomes to ensure more diversity and better exploration capacity, which is the key for locating the global optimum. This is the reason that makes it possible for DGEP to locate the

global optima for all the multimodal functions, f_{10}–f_{30}, while all three of IMGA [142], [143], RTS [158] and DPGA [146] miss the global optima on several occasions, especially for the functions f_{25}–f_{27}. However, this very strength of locating global optima by more explorations and more diversity does not allow DGEP to be completely exploitative, which is the only reason why its performance becomes worse than the extensive fine tuning of DPGA [146] on the three multimodal functions: f_{10}, f_{17} and f_{18}. However, a simple fine tuning scheme after the execution of DGEP may further improve its results that may outperform both of DPGA [146] and RTS [158] on all of these three functions.

Along the results of the Tables 4.1–4.11, DGEP has outperformed many other algorithms, including CEP [102], ALEP [56], IFEP [57], GA [11], DE [195], PSO [196], CLPSO [69], ABC [105], IMGA [142], DPGA [146], RTS [158], RCMA-XHC [160], NSDE [163], [164], LSRCMA [162] and CMAES [165]. But why the performance of DGEP is often better than these other algorithms? There reasons might be as follows. Firstly, DGEP emphasizes both global explorations and local exploitations using the population diversity information. The utilization of the distance of dissimilar or similar chromosomes in mutation clearly reflects such emphasis. CEP [102], IFEP [57] and ALEP [56] do not separate exploration and exploitation operations; rather, IFEP [57] and ALEP [56] primarily emphasize producing good offspring. The emphasis on only good solutions may reduce the population diversity resulting in poor overall performance. IMGA [142], [143] and RTS [158] alter the selection mechanism in such a way that it promotes diversity and allows better explorations, but at the cost of reduced exploitations and slower convergence speed. RCMA with XHC [160] performs exploration and exploitation separately by using PBX crossover [160] with BGA mutation [81] for exploration and a specialized crossover, XHC [160] for exploitation. The problem of using different operators lies in ensuring their synergistic effect [212]. DPGA [146] adapts the distance between its two populations in such a way that the reserve population mostly exerts exploitative pressure on the main population until the solutions get trapped into the local optima for several generations. A more balanced approach between exploitations and explorations might further improve its results. CLPSO [69] is a learning approach that does not employ exploration and exploitation operations separately. Although it utilizes the best information of all particles to update the velocity of any one particle, it may still trap into local optima due to many inherent problems of a learning scheme, e.g., difficulty in picking appropriate weight values to make a weighted combination of the old experiences and the new observations. CMAES [165] maintains and continuously adapts a covariance matrix in order to maximize the likelihood of producing better offspring which makes the algorithm more exploitative rather than explorative in nature. The remaining two algorithms, NSDE [163], [164] and LSRCMA [162] make attempts to combine the benefits of

neighborhood search and local search techniques with differential evolution and RCMA respectively. However, the depth of local or neighborhood search on a chromosome is decided based on the fitness of the chromosome, which makes both these algorithms more exploitative than explorative. Lack of balance between exploitations and explorations tends to decrease population diversity and leads to premature convergence.

Secondly, mutation in DGEP does not produce offspring blindly, but rather utilizes the distance to other chromosomes in order to produce an offspring using an appropriate step size. This mutation produces an offspring in such a way that it either facilitates the exploration of wider regions of the search space or performs the exploitation of existing solutions for immediate fitness improvement. The mutation in CEP [102], ALEP [56], IFEP [57], IMGA [142], [143] and RTS [158] does not use any information from other chromosomes and produces offspring blindly. The consequence of blind mutation is that the offspring produced may be dominated by chromosomes in the current population. RCMA with XHC [160] also uses blind mutation and crossover for exploration. All of the algorithms — DPGA [146], NSDE [163], LSRCMA [162] and CMAES [165] make use of population information either to guide the depth of genetic operations on each chromosome or to adapt step size values for increasing the likelihood of successful mutations which help them achieve better results than CEP [102], ALEP [56] and IFEP [57], and sometimes competitive results to DGEP.

Thirdly, DGEP uses selection strategies at two different levels in order to select offspring for the next generation. During each mutation, it selects an offspring to choose between exploration and exploitation. After mutating all the chromosomes, it performs a fitness based selection to pick the best chromosomes for the next generation. This two-level selection strategy is more likely to perform a better balance between exploitative and explorative features of the evolutionary search than the more exploitative fitness based schemes of IMGA [142], [143], RTS [158], DPGA [146], NSDE [163], [164], LSRCMA [162] and CMAES [165]. Also, the diversity preservation schemes of DGEP ensure adequate amount of population diversity throughout the evolution to assist its diversity guided mutation scheme. In case of CEP [102], ALEP [56] and IFEP [57], a tournament based selection scheme is adopted where for each parent or offspring, q opponents are chosen uniformly at random for pair wise fitness comparison from all the parents and offspring. The value of q affects the population diversity. A large value of q corresponds to high selection pressure, so the probability of the fittest chromosome being selected multiple times becomes high, resulting in loss of population diversity. RCMA with XHC [160] allows only better offspring for both exploration and exploitation, thus it tends to reduce population diversity, which is the main reason for premature convergence.

The incorporation of few simple ideas may further improve the results of DGEP, especially on the more complex rotated and hybrid composite functions. First, a simple random value of K is used for the diversity guided mutation (Fig. 4.1). An adaptive approach that can dynamically change K for explorations or exploitations during the course of evolution may be more appropriate than a simple random strategy. Second, DGEP showed remarkable explorative performance to locate the global optimum, but its exploitative capacity is not as good as some other existing approaches, such as DPGA [146]. It would be interesting to observe if DGEP could be hybridized with other algorithms, e.g., DPGA [146] or similar algorithms having better fine tuning capacity. Third, the mutation step size is guided by considering the distance information of only two neighboring or distant chromosomes. It could be better if more sophisticated distance information that involves more chromosomes, e.g., the relative density of chromosomes across the search space or the characteristics of the fitness landscape around the current point could be considered to extract better guidance for the mutation step size. Fourth, DGEP has been applied on continuous optimization problems only. It would be interesting to study how well DGEP performs for other problems, especially the real world ones. The incorporation of these ideas could be a topic for our future study.

4.7 Conclusion and Future Research Directions

This chapter introduces the novel diversity based algorithm — DGEP with the diversity guided DGM mutation scheme as its central component. The performance of DGEP and DGM mutations are evaluated in sections 4.5, 4.6, and later in sections 8.4, 8.8, where the proposed schemes are compared with several other existing research works on a wide range of low and high dimensional, unimodal and multimodal, regular, rotated and hybrid composite benchmark functions. Empirical results demonstrate the effectiveness of DGEP and the DGM mutation scheme by frequently outperforming the most other algorithms on most of the problems. However, the incorporation of a few simple ideas may further improve the results of DGEP, especially on the rotated and hybrid composite functions. First, a simple random value of K is used for the diversity guided mutation (Fig. 4.1). An adaptive approach that can dynamically change K for explorations or exploitations during the course of evolution may be more appropriate than a simple random strategy. Second, DGEP showed remarkable explorative performance to locate the global optimum, but its exploitative capacity is not as good as some other existing approaches, such as DPGA [146]. It would be interesting to observe if DGEP could be hybridized with other algorithms, e.g., DPGA [146] or similar algorithms having better fine

tuning capacity. Third, the mutation step size is guided by considering the distance information of only two neighboring or distant chromosomes. It could be better if more sophisticated distance information that involves more chromosomes, e.g., the relative density of chromosomes across the search space or the characteristics of the fitness landscape around the current point could be considered to extract better guidance for the mutation step size. Fourth, DGEP has been applied on continuous optimization problems only. It would be interesting to study how well DGEP performs for other problems, especially the real world ones. The incorporation of these ideas could be a topic for our future study.

Chapter 5

Artificial Bee Colony Algorithm with Self-Adaptive Mutation

5.1 Introduction

This chapter introduces a novel and improved ABC-variant — the ABC with Self-Adaptive Mutation (ABC-SAM) that makes attempts to dynamically adapt the mutation step size with which the artificial bees (i.e., candidate solutions) explore the search space. Mutation with small step size produces small variations of the existing candidate solutions which is better for exploitations, while large mutation steps are likely to produce large variations that facilitate more explorations of the search space. ABC-SAM fosters both large and small mutation steps as well as adaptively controls the mutation step size based on their effectiveness to produce better candidate solutions from the existing ones. We have evaluated and compared ABC-SAM on as many as 55 benchmark problems on continuous optimization, chosen from two different benchmark suites. The evaluation results are compared with the basic ABC algorithm and several other recent evolutionary and swarm intelligence algorithms, which show that the proposed self-adaptation scheme of ABC-SAM can facilitate more effective mutations and perform better optimizations than most other relevant EAs and SIAs.

5.2 Organization of the chapter

The rest of this chapter is organized as follows. Section 5.3 presents the improved ABC-variant — ABC-SAM with its pseudocode, explains how it significantly differs from the basic ABC algorithm and makes a brief analysis on how it improves the performance over the basic ABC algorithm. Section 5.4 presents the performance evaluation of ABC-SAM on two different benchmark suites, presents the parameter settings of ABC-SAM and some other algorithms in comparison, compares their results on the benchmark functions and makes comments on the results and comparisons. Finally, section 5.5 draws conclusions with a summary of this chapter and leaves a few suggestions for further research with ABC-SAM.

5.3 The Proposed Algorithm — ABC-SAM

The basic ABC algorithm drives its search using two operators — the selection and the mutation operators. The mutation operator is the only source of variations and search space explorations in ABC, while the greedy fitness based selection operator chooses between offspring and parent candidate solutions based on which one has the higher fitness value. The mutation operation of the basic ABC algorithm is described by formula (2.6), which is quite plain and simple — it picks a random parameter x_{ij} of the current candidate solution $\boldsymbol{x_i}$ and perturbs it using the information of another randomly picked candidate solution (i.e., $\boldsymbol{x_k}$ in (2.6)). This basic mutation operation treats every candidate solution $\boldsymbol{x_i}$ across the population equally, without considering its individual explorative/exploitative requirements. Besides, the explorative/exploitative requirements of a candidate solution usually does not remain the same along the course of the optimization process; rather it changes dynamically and often in an unpredicted way. To address this issue, this chapter introduces ABC-SAM — an improved variant of the basic ABC algorithm that tries to customize the degree of explorations and exploitations, separately for each candidate solution of the population, by introducing a separate scaling factor SF_i for every candidate solution $\boldsymbol{x_i}$ of the population. As the explorative/exploitative requirements of $\boldsymbol{x_i}$ evolve over time, ABC-SAM periodically adjusts the value of SF_i, separately for each $\boldsymbol{x_i}$, by examining whether $\boldsymbol{x_i}$ needs more explorations or more exploitations at the current stage of optimization.

ABC-SAM is different from the basic ABC algorithm in two important ways. First, ABC-SAM alters the basic perturbation formula (2.6) which is used to produce a new trial solution $\boldsymbol{v_i}$ from an existing candidate solution $\boldsymbol{x_i}$ in the steps 4 and 8 (the pseudocode in Fig. 2.9) of the basic ABC algorithm. Second, ABC-SAM executes an 'adaptation cycle', periodically after every K iterations (cycles), in order to automatically adapt the scaling factors, i.e., the SF_i values that ABC-SAM maintains for every candidate solution $\boldsymbol{x_i}$. Both these improvements work together to induce more effective mutations for producing better trial solutions from the existing ones. The scaling factor (i.e., the SF_i value) that ABC-SAM maintains and adapts for every candidate solution $\boldsymbol{x_i}$ of the population helps the mutation process to perform better exploitations and explorations around $\boldsymbol{x_i}$. ABC-SAM alters the basic mutation formula (2.6) by including the SF_i term as a magnifier of the difference term, i.e., the $\left(x_{kj} - x_{ij}\right)$ term in (2.6). Thus the SF_i value now tries either to enlarge or to shrink the magnitude of the variation on $\boldsymbol{x_i}$ in order to facilitate either more explorations or more exploitations. The mutation operation (i.e., eq. (2.6), in steps 4 and 8 of the pseudocode of ABC in Fig. 2.9) now gets replaced by the following eq. (5.1).

$$v_{ij} = x_{ij} + SF_i * \varphi_{ij} * \left(x_{kj} - x_{ij}\right) \tag{5.1}$$

All SF_i values are initiated to 1.0 during the beginning of the search process. As the search progresses and the candidate solutions proceed across several peaks, valleys and plateaus of the fitness landscape, the SF_i values are automatically adjusted by the periodically invoked adaptation cycles in order to take care of the current explorative/exploitative requirements of each candidate solution $\boldsymbol{x_i}$. Small values (less than unity) for SF_i would shrink the product term $SF_i * \Phi_{ij}$ in (5.1) to facilitate small mutation steps, thus ensuring more exploitations around the existing solutions. On the contrary, large enough values for SF_i would expand the product term $SF_i * \Phi_{ij}$ which is more likely to induce large variations on $\boldsymbol{x_i}$ that might be helpful for the search process to quickly get rid of any local optimum around $\boldsymbol{x_i}$, thus to perform more explorations of the search space without being trapped around any of the locally optimal points. Whether exploitation or exploration is better for $\boldsymbol{x_i}$ at the current search stage might not be apparent or could not be predicted beforehand. This is why the adaptation cycle executes periodically after each K cycles, performs mutations with different range of SF_i values and promotes only those SF_i values that produce more successful mutations. This process dynamically adapts each scaling factor value SF_i that is more suitable for the current optimization scenario around each candidate solution $\boldsymbol{x_i}$.

Fig. 5.1 presents the pseudocode of the mutation step size adaptation cycle. In this cycle, ABC-SAM adapts the scaling factor values of the 'elite' bees. The elite group consists of some top portion, say p%, of the bee population that exhibit maximum fitness values. In our implementation and in all the evaluations of ABC-SAM (i.e., Tables 5.1–5.10, Figs. 5.5–5.8) we have used $p = 10$. The adaptation cycle generates two uniform random values u_1 and u_2 from the ranges [-α, 0) and (0, α] respectively, for each scaling factor parameter SF_i maintained for each elite bee $\boldsymbol{x_i}$. Now, for each $\boldsymbol{x_i}$, three different offspring solutions are generated by using (5.1) — one by employing the existing value of SF_i, and the other two by multiplying SF_i with 2^{u_1} and 2^{u_2}. Since $u_1 < 0$ and $u_2 > 0$, the scaling factor $2^{u1} * SF_i$ would generate small steps using (5.1) for better exploitations, while the scaling factor $2^{u2} * SF_i$ would produce large step sizes for better search space explorations. ABC-SAM evaluates the fitness of all three offspring solutions produced by the three different scaling factors (addressed as $SF_{i,1}$, $SF_{i,2}$ and $SF_{i,3}$ in Fig. 5.1) and then accepts the best one of them as $\boldsymbol{x_i}$ in the next cycle. Also, it updates the scaling factor value SF_i to the weighted average of its current value (SF_i) and the scaling factor value, say, $SF_{i,k}$, that has produced the best offspring by using the following eq. (5.2).

$$SF_i = \beta * SF_i + (1 - \beta) * SF_{i,k} \tag{5.2}$$

Here, the constants β and $(1-\beta)$, with $0 < \beta < 1$, control the relative weight between the past and present knowledge of ABC-SAM on the appropriate scaling factor value SF_i for each candidate solution $\boldsymbol{x_i}$. A small value of β (i.e., $\beta \to 0$) ensures fast learning (and fast forgetting, too), which

makes ABC-SAM to mostly rely on the newly found scaling factor values (i.e., $SF_{i,k}$ in Fig. 5.1). However, this may lead to oscillations and instability in the learning behaviour of ABC-SAM. A large value of β (i.e., $\beta \rightarrow 1.0$) ensures stable, but possibly slow learning behavior. After some initial experiments, we have used $\beta = 0.9$ in all the experiments, which has led to stable learning behaviour and sufficiently good final solution quality for almost all the benchmark functions, as demonstrated by the results (Tables 5.1–5.10) in the evaluations section.

As we have observed during the experimentations, the SF_i values mostly decrease with the on-going iterations (cycles), because exploitative small mutation steps usually have better chance to succeed than explorative large steps. This gradual decrease in SF_i values shrinks the mutation steps for each candidate solution $\boldsymbol{x_i}$, which eventually might make the search process get trapped around the locally optimal points. To avoid this, ABC-SAM adopts two simple, yet effective techniques. First, SF_i is never allowed to drop below a minimum acceptable value (i.e., SF_{min}). Second, if a particular SF_i drops to SF_{min} and is stuck there for the last τ_1 cycles without any fitness improvement of the solution $\boldsymbol{x_i}$, then ABC-SAM manually resets SF_i to 1.0 and keeps it constant at this value for the next τ_2 generations. This resetting of the scaling factors to relatively higher values for τ_2 consecutive generations promotes more explorations with larger mutation steps and helps get rid of the locally optimal point whenever any solution gets entrapped around the local optima with prolonged fitness stagnation. Fig. 5.2 details this procedure of maintaining the scaling factor values with necessary resets.

Algorithm 5.1: Procedure *AdaptationCycle*( )
begin
  for each $i \in ELITE\_POP$ **do**
  {
    $u_1 = Uniform\_Random \sim [-\alpha, 0)$
    $u_2 = Uniform\_Random \sim (0, \alpha]$
    $SF_{i,1} = SF_i$
    $SF_{i,2} = 2^{u1} * SF_i$
    $SF_{i,3} = 2^{u2} * SF_i$
    for k = 1 to 3 **do**
    {
      $\boldsymbol{v_{i,k}} = \boldsymbol{v_i}$ computed using (5.1) with $SF_i = SF_{i,k}$
      $fitness[k] = Compute\_Fitness\ (\boldsymbol{v_{i,k}})$
    }
    Find k such that $fitness[k]$ is the maximum over the array $fitness[1]...[3]$
    $SF_i = \beta * SF_i + (1-\beta) * SF_{i,k}$
  }
end

Figure 5.1: Pseudocode for the mutation step size adaptation cycle of ABC-SAM

Algorithm 5.2: Procedure *UpdateSF*$_i$( )
SF_{min}: smallest allowed value for the scaling factors
PrevUpdated: the last iteration (cycle) when SF_i was updated
t: current iteration (cycle)

```
begin
    if t < PrevUpdated + τ1 then
            Update SFi using the pseudocode in Fig. 5.1
            SFi = max (SFmin, SFi)
            if SFi > SFmin then
                PrevUpdated = t
            endif
     elseif t < PrevUpdated + τ1 + τ2
            SFi = 1.0
     else
            PrevUpdated = t
     endif
end
```

Figure 5.2: Pseudocode for maintaining the scaling factor values with necessary resets

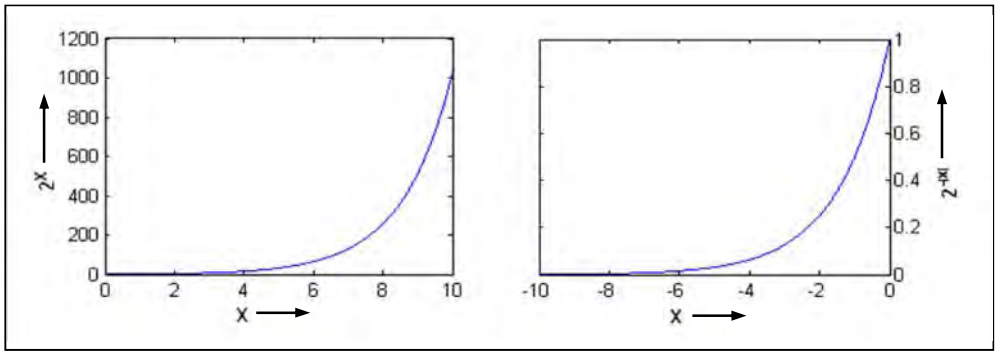


Figure 5.3: Distributions that facilitate explorations (on the left) and exploitations (right)

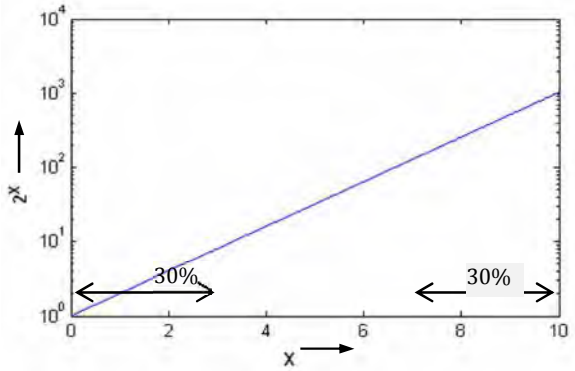


Figure 5.4: The exponential 2^x vs. x plot of previous Fig. 5.3, but now with logarithmic Y-axis. More than 30% times the random values it produces are $\leq 10^1$, while more than another 30% times the random values are $\geq 10^2$. Such a wide range of values, produced with significant probability, ensure better search space explorations.

Fig. 5.3 shows x vs. 2^x and x vs. $2^{-|x|}$ graphs. These two distributions produce the scaling factor values for explorations and exploitations respectively. Exponential distributions can produce wide range of values as scaling factors facilitating from very small to very large steps ensuring wide range of both exploitations and explorations. Fig. 5.4 reproduces the same graph of x vs. 2^x, but now with logarithmic Y-axis. It shows that, with $x \in [0, 10]$, more than 30% times the scaling factor value produced for explorations by this distribution is ≤ 10, while the value is 100 or even higher for more than another 30% of times. This ensures wide range of SF_i values to be produced with significant probability, which is necessary for excellent explorations of the search space. Similar analysis of the x vs. $2^{-|x|}$ graph indicates that this distribution can also facilitate wide range of exploitative step sizes for the existing candidate solutions.

5.4 Evaluation of ABC-SAM on Benchmark Functions

In this section, ABC-SAM is evaluated using two different suites of benchmark problems on numerical function optimization and compared with some other existing EAs and ABC-variants. First, ABC-SAM is evaluated on the standard benchmark suite (Table 2.3, section 2.17), consisting of 30 benchmark functions. Later in this section (i.e., subsection 5.4.4), ABC-SAM is also evaluated and compared on the recently introduced CEC2005 benchmark suite (briefly presented in the Table 2.4, section 2.17), which includes 25 more complex and challenging functions. Both the benchmark suites are further explained in the Appendix A and in [76].

5.4.1 Results of ABC-SAM on Standard Benchmark Functions

The standard benchmark suite functions are divided into three categories — the unimodal functions (f_1–f_9), the high dimensional multimodal functions (f_{10}–f_{18}) and the remaining multimodal functions with low dimensionality (f_{19}–f_{30}). Tables 5.1–5.2 present the results of ABC-SAM on all 30 benchmark functions and compare the results with the basic ABC algorithm. ABC-SAM is tested with four different settings of its parameter K (i.e., the adaptation period) — K=2, 5, 10 and 50. All the variants have made 50 independent runs on each function and the mean and standard deviation of the best found solutions from the different runs are presented in Tables 5.1–5.2. The parameters of ABC-SAM that are inherited from the basic ABC algorithm are the population size *SN*, maximum cycle number *MCN* and *limit*. The other parameters that are specific to ABC-SAM are τ_1, τ_2, α, β and SF_{min}. For the functions f_1–f_{18} with D=30, we used SN=100, MCN=1000 and $limit$=100 for both ABC and ABC-SAM. For the larger variants with

$D = 60$ (Table 5.2), we used $SN = 100$, $MCN = 2000$ and $limit = 200$. For the low dimensional functions f_{19}–f_{30}, we used $SN = 100$, $MCN = 100$ and $limit = 10 * D$ for both ABC and ABC-SAM. After some preliminary testing, the other parameters of ABC-SAM are set as: $\tau_1 = 40$, $\tau_2 = 20$, $SF_{min} = 10^{-8}$, $\alpha = 10$ and $\beta = 0.9$ for all the evaluations (i.e., Tables 5.1–5.10) in this chapter. These values are chosen after some initial evaluations and not meant for optimum. The results in Tables 5.1–5.2 can be summarized in the following few points.

- **Results of ABC-SAM variants (Table 5.1):** For most (25 out of 30) of the functions, the ABC-SAM variants reach very close proximity of the global minimum. The few functions for which the ABC-SAM variants face some difficulty are — f_3, f_7, f_{12}, f_{28} and f_{30}. However, the performance of the basic ABC algorithm is mostly (four out of five functions) worse than the ABC-SAM variants on these five functions (except f_7). This indicates that ABC-SAM has actually improved the performance of ABC on these functions. Besides, for almost all the functions, the magnitude of standard deviations of the results of ABC-SAM are quite low compared to the mean values, which indicates a high degree of consistency and robustness of all the ABC-SAM variants for most of these benchmark functions.
- **ABC vs. ABC-SAM variants (Table 5.2):** Out of the 18 high dimensional functions f_1–f_{18}, the ABC-SAM variants perform better that ABC on 14 functions, show similar performance on one (f_8), while ABC performs better on the remaining three functions only (f_7, f_{17}, f_{18}). These improvements are always statistically significant, as measured by t-test with 95% confidence interval. For the 12 low dimensional functions f_{19}–f_{20}, ABC and ABC-SAM often perform equally well (seven out of the 12 functions), though the ABC-SAM variants perform better on the remaining five functions. Thus, the overall performance of the ABC-SAM variants is far better than the basic ABC algorithm.
- The '+', '–' and '≈' symbols at the bottom rows (Table 5.2) count the number of functions where ABC-SAM performs significantly better, worse or similar, respectively compared to the basic ABC algorithm. Out of the 30 standard benchmark functions, ABC-SAM performs significantly better than ABC on 19 functions, shows similar performance on eight, while ABC has managed to perform better only on three functions.
- Fig. 5.5 compares the overall performance of ABC and the ABC-SAM variants by comparing their mean absolute error values over the 30 standard benchmark functions. ABC-SAM with $K = 10$ shows the best performance (i.e., minimum mean absolute error).

Table 5.1: Performance of four different variants of ABC-SAM on the 30 standard benchmark functions, each variant using a different value of *K*. Results have been averaged over 50 independent runs. The best result for each function is marked with boldface font, if not identical with the results from the other variants.

No	f_{min}	ABC-SAM (*K*=2)		ABC-SAM (*K*=5)		ABC-SAM (*K*=10)		ABC-SAM (*K*=50)	
		Mean	Std Dev	Mean	Std Dev	Mean	Std Dev	Mean	Std Dev
f_1	0	6.15e–12	2.14e–12	**1.61e–14**	**3.80e–14**	4.18e–14	5.37e–15	7.57e–12	8.44e–13
f_2	0	7.19e–07	4.42e–07	9.76e–08	7.58e–08	**2.47e–08**	**2.35e–09**	8.54e–07	5.66e–07
f_3	0	2.15e+01	3.53	2.13e+01	2.04	**1.69e+01**	**1.43**	3.40e+01	5.59
f_4	0	6.20e–09	3.56e–09	**8.16e–13**	**1.78e–13**	3.95e–12	1.05e–12	8.32e–11	5.90e–11
f_5	0	8.37e–01	9.56e–02	**2.50e–01**	**4.63e–02**	9.24e–01	2.08e–01	4.09e+00	1.64e+00
f_6	0	**1.85e–03**	**5.12e–04**	6.41e–02	9.91e–03	2.16e–03	6.37e–04	3.26e–03	9.58e–04
f_7	0	2.87e+01	4.11	2.52e+01	4.24	**2.28e+01**	**3.75**	3.14e+01	6.23
f_8	0	0	0	0	0	0	0	0	0
f_9	0	3.35e–12	7.07e–13	6.30e–15	2.55e–15	**3.66e–16**	**1.44e–17**	1.23e–13	7.63e–14
f_{10}	0	8.45e–15	3.04e–15	4.14e–15	7.47e–16	**1.26e–16**	**2.11e–17**	8.85e–15	3.40e–15
f_{11}	0	9.29e–09	4.12e–09	6.86e–10	2.35e–10	**4.60e–10**	**8.85e–11**	8.85e–09	4.92e–09
f_{12}	–12569.5	–11905.20	1.42e+02	–12332.78	7.50e+01	**–12416.19**	**4.02e+01**	–12084.23	1.21e+02
f_{13}	0	8.28e–07	3.27e–07	**1.58e–08**	**7.89e–09**	9.26e–08	1.89e–08	9.55e–06	4.82e–06
f_{14}	0	8.66e–08	3.47e–08	9.32e–10	3.36e–10	**8.36e–10**	**5.08e–11**	1.84e–09	7.11e–10
f_{15}	0	2.63e–06	6.80e–07	**1.46e–09**	**8.80e–10**	2.22e–08	3.93e–09	4.40e–05	8.93e–06
f_{16}	0	6.02e–01	1.95e–01	6.82e–03	2.24e–04	**5.78e–04**	**6.31e–05**	4.12e+00	1.88e+00
f_{17}	0	2.68e–10	8.39e–11	1.37e–10	6.30e–11	**9.78e–12**	**3.89e–12**	8.74e–10	3.56e–10
f_{18}	0	1.91e–03	7.53e–04	6.35e–04	3.28e–04	3.06e–02	8.59e–03	**1.08e–04**	**4.52e–05**
f_{19}	1	1.04	0.04	**1.03**	**0.01**	**1.03**	**0.02**	1.04	0.02
f_{20}	3.07e–04	**3.45e–04**	**2.79e–06**	3.67e–04	8.22e–06	4.32e–04	1.09e–05	5.20e–04	6.89e–5
f_{21}	–1.0316	–1.0316	0.00	–1.0316	0.00	–1.0316	0.00	–1.0316	0.00
f_{22}	0.398	0.398	0.00	0.398	0.00	0.398	0.00	0.398	0.00
f_{23}	–3.86	–3.86	0.00	–3.86	0.00	–3.86	0.00	–3.86	0.00
f_{24}	–3.32	–3.32	0.00	–3.32	0.00	–3.32	0.00	–3.32	0.00
f_{25}	–10.15	–9.85	0.11	**–10.15**	**1.87e–08**	–10.14	3.68e–07	–10.01	0.06
f_{26}	–10.40	–10.40	5.15e–03	–10.40	6.98e–03	**–10.40**	**3.94e–03**	–10.40	8.10e–03
f_{27}	–10.54	–10.54	3.36e–07	–10.54	8.48e–06	**–10.54**	**6.77e–07**	–10.54	9.63e–06
f_{28}	0	1.06e+01	3.50	7.36	2.48	**4.02**	**0.39**	8.56	2.12
f_{29}	–9.66	–9.66015	0	–9.66015	0	–9.66015	0	–9.66015	0
f_{30}	–1.4	–0.86	0.08	–0.93	0.05	**–1.04**	**0.06**	–0.80	0.11

Table 5.2: Performance comparison of ABC and ABC-SAM on the 30 standard benchmark functions. Results have been averaged over 50 independent runs. The best result for each function is marked with boldface font. A '+' or '–' in the *t*-test of ABC-SAM versus ABC indicate that ABC-SAM is significantly better or worse, respectively than ABC with 95% certainty, while a '≈' means that the difference is not statistically significant.

No	f_{min}	D	G	ABC		ABC-SAM (K=10)		t-Test (ABC-SAM vs. ABC)
				Mean	Std. Dev.	Mean	Std. Dev.	
f_1	0	30	1000	2.45e–11	7.72e–12	**4.18e–14**	**5.37e–15**	+
		60	2000	3.75e–10	2.01e–11	**6.09e–13**	**7.24e–14**	
f_2	0	30	1000	5.05e–07	1.74e–07	**2.47e–08**	**2.35e–09**	+
		60	2000	5.58e–06	1.17e–06	**5.06e–07**	**2.97e–07**	
f_3	0	30	1000	4.18e+01	5.90	**1.69e+01**	**1.43**	+
		60	2000	7.31e+01	6.88	**3.10e+01**	**5.12**	
f_4	0	30	1000	8.32e–10	9.75e–11	**3.95e–12**	**5.77e–13**	+
		60	2000	4.50e–09	5.64e–10	**7.54e–11**	**2.14e–11**	
f_5	0	24	1000	6.61e+00	1.07e+00	**9.24e–01**	**2.08e–01**	+
f_6	0	30	1000	6.67e–01	1.21e–08	**2.16e–03**	**6.37e–04**	+
		60	2000	6.66e–01	1.05e–07	**7.76e–02**	**1.63e–02**	
f_7	0	30	1000	**4.25e–01**	**1.18e–01**	2.28e+01	3.75	–
		60	2000	**2.02e–01**	**6.92e–02**	4.96e+01	7.80	
f_8	0	30	1000	0	0	0	0	≈
		60	2000	0	0	0	0	
f_9	0	30	1000	8.60e–13	8.32e–13	**3.66e–16**	**1.44e–17**	+
		60	2000	9.31e–12	7.17e–12	**4.76e–15**	**5.32e–16**	
f_{10}	0	30	1000	1.72e–14	1.56e–14	**1.26e–16**	**2.11e–17**	+
		60	2000	2.84e–13	8.01e–14	**8.55e–15**	**3.15e–16**	
f_{11}	0	30	1000	2.33e–08	7.49e–09	**4.60e–10**	**8.85e–11**	+
		60	2000	6.64e–07	1.51e–07	**6.80e–09**	**8.77e–10**	
f_{12}	–12569.5	30	1000	–11346.79	2.77e+02	**–12416.19**	**4.02e+01**	+
	–25138.9	60	2000	–22530.82	4.08e+02	**–23805.93**	**2.84e+02**	
f_{13}	0	30	1000	2.93e–06	3.38e–07	**9.26e–08**	**1.89e–08**	+
		60	2000	4.65e–06	1.07e–06	**2.07e–08**	**3.55e–08**	
f_{14}	0	30	1000	4.55e–08	6.54e–09	**8.36e–10**	**5.08e–11**	+
		60	2000	8.01e–07	2.64e–07	**1.56e–10**	**6.90e–11**	
f_{15}	0	30	1000	3.34e–04	3.76e–05	**2.22e–08**	**3.93e–09**	+
		60	2000	7.49e–03	9.58e–04	**1.17e–08**	**2.35e–09**	

Table 5.2 (continued): Performance of ABC and ABC-SAM on 30 standard benchmark functions

No	f_{min}	D	G	ABC		ABC-SAM (K=10)		t-Test (ABC-SAM vs. ABC)
				Mean	Std. Dev.	Mean	Std. Dev.	
f_{16}	0	30	1000	3.36e–01	9.58e–02	**5.78e–04**	**6.31e–05**	+
		60	2000	8.99e–01	3.09e–01	**9.20e–03**	**4.03e–03**	
f_{17}	0	30	1000	**5.47e–12**	**2.09e–13**	9.78e–12	3.89e–12	–
		60	2000	**7.47e–12**	**2.74e–12**	1.32e–11	5.15e–11	
f_{18}	0	30	1000	**2.63e–03**	**1.89e–04**	3.06e–02	8.59e–03	–
		60	2000	**2.66e–03**	**7.90e–04**	5.11e–02	7.39e–03	
f_{19}	1	2	100	1.04	0.04	1.03	0.03	≈
f_{20}	3.07e–04	4	100	5.98e–04	7.22e–05	**4.32e–04**	**1.09e–05**	+
f_{21}	–1.0316	2	100	–1.0316	0	–1.0316	0	≈
f_{22}	0.398	2	100	0.398	7.12e–08	0.398	2.75e–07	≈
f_{23}	–3.86	3	100	–3.86	7.09e–07	–3.86	1.54e–08	≈
f_{24}	–3.32	6	100	–3.32	4.74e–13	–3.32	6.26e–14	≈
f_{25}	–10.15	4	100	–9.61	0.14	**–10.14**	**3.68e–07**	+
f_{26}	–10.40	4	100	–10.40	8.61e–03	–10.40	7.94e–03	≈
f_{27}	–10.54	4	100	–10.52	0.01	**–10.54**	**6.77e–07**	+
f_{28}	0	10	100	13.77	3.80	**4.02**	**0.39**	+
f_{29}	-9.66015	10	100	-9.66015	0	-9.66015	0	≈
f_{30}	-1.4	10	100	–0.78	0.09	**–1.04**	**0.06**	+
Summary (t-Test)		+		19				
		–		3				
		≈		8				

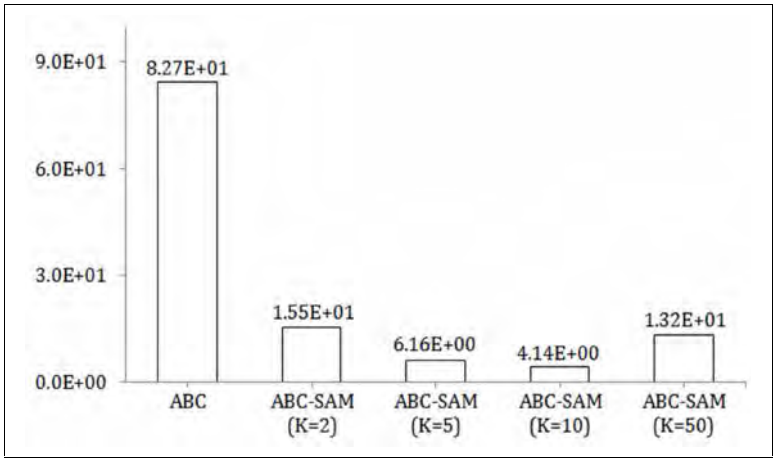


Figure 5.5: The mean absolute error values of ABC and ABC-SAM variants on the standard benchmark functions. All four ABC-SAM variants show smaller error values than the basic ABC algorithm. ABC-SAM with K=10 shows the best performance (minimum mean absolute error). This indicates that the performance of ABC-SAM may deteriorate with too frequent (e.g., K=2) or too delayed (K=50) occurrence of the adaptation cycles, while the moderate values of K (e.g., K=5 or K=10) usually yield better performance (i.e., smaller error values).

Fig. 5.6 presents the convergence graphs of ABC-SAM (with $K=10$) and the basic ABC algorithm for six 30-D functions — three of them are unimodal (f_1, f_4, f_8) and the remaining three are multimodal (f_{12}, f_{13}, f_{16}). For all six functions, ABC-SAM shows better convergence characteristics than the basic ABC algorithm. For example, consider the functions f_{12}, f_{13} and f_{16}. For f_{12}, the basic ABC algorithm prematurely converges to some intermediate locally minimal point with fitness stagnation, while ABC-SAM reaches very close vicinity of the global minimum. Also, ABC shows unacceptably slow convergence speed for f_{13} and f_{16} during the final cycles of its execution, while ABC-SAM shows no sign of fitness stagnation and maintains sufficient convergence speed, even during its final cycles. The plots for ABC for the functions f_1, f_4 and f_{16} show several flat and semi-flat segments, indicating short-length fitness stagnations of the basic ABC algorithm, while ABC-SAM does not show any noticeable flat segments due to fitness stagnations for these functions. Previously (Table 5.2) we have considered the performance of ABC and ABC-SAM to be equal on f_8, but Fig. 5.6 now reveals that ABC-SAM actually reaches the global minimum of f_8 much earlier than the basic ABC algorithm. Thus, the overall convergence characteristics of ABC-SAM are better than the basic ABC algorithm.

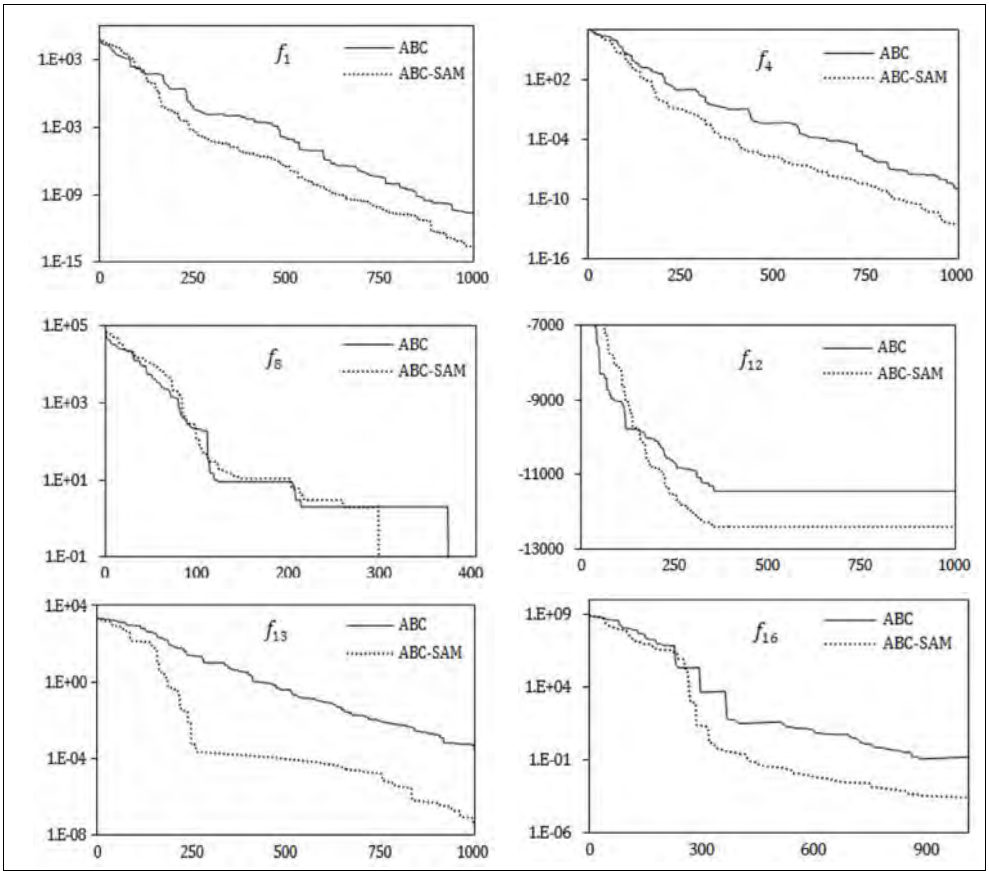


Figure 5.6: Convergence characteristics of ABC and ABC-SAM on three unimodal (f_1, f_4, f_8) and three multimodal (f_{12}, f_{13}, f_{16}) functions. The vertical axis is the function value, while the horizontal axis is the number of cycles elapsed.

5.4.2 Comparison of ABC-SAM with Other EAs and SIAs

In this section, ABC-SAM is compared with some basic and representative evolutionary and swarm intelligence algorithms, such as GA [11], DE [195], PSO [196] and the basic ABC algorithm [11]. Similar to ABC and ABC-SAM, mutation is the only perturbation operator in evolutionary programming (EP) based algorithms; so we also include a number of EP-based algorithms in the comparison, such as Classical EP [3], Improved Fast EP (IFEP) [57] and EP with Adaptive Lévy Mutation (ALEP) [56]. All these algorithms have two parameters in common — the population size *SN* and the total number of function evaluations (*FE*). For the next comparison (Table 5.3), they are set as *SN*=50 and *FE*=500,000. The other parameters specific to GA [11], DE [195], PSO [196], ABC [11] and ABC-SAM are specified below.

GA, DE and PSO Settings: The particular GA scheme we employed uses binary coded standard GA [11], single point uniform crossover (with crossover rate = 0.80), bit flip mutation (with mutation rate = 0.01), fitness scaling, selection by stochastic uniform sampling technique and elitism with generation gap = 0.9, as suggested in [11]. The particular version of PSO [196] that we used has three more parameters — the cognitive component ¤1, social component ¤2, and the inertia weight *w*, which are set to 1.8, 1.8 and 0.6, respectively, as suggested in [196]. The standard DE algorithm has two specific parameters — the scaling factor *F* and the crossover rate *CR*, which we have set as *F*=0.5 and *CR*=0.9, as recommended in [195].

ABC and ABC-SAM Settings: The common parameters of ABC and ABC-SAM are set as — population size *SN*=50, number of function evaluations *FE*=500,000 and *limit*=*SN***D*, as recommended in [11]. Both employed and onlooker bees are set to 50% of the colony size *SN*. The number of scout bees is set to 1. Number of elite bees is set to eight, and with this setting, the number of fitness evaluations in an adaptation cycle is not large (i.e., only three evaluations per elite bee), which is close to (actually, smaller than) half of the fitness evaluations in a regular cycle. The adaptation cycle is invoked in every *K*=10 cycles. After some initial evaluations, other parameters of ABC-SAM are set as — τ_1=40, τ_2=20, $SF_{min}=10^{-8}$, α=10 and β=0.9.

Table 5.3 compares ABC-SAM with GA [11], DE [195], PSO [196] and the basic ABC algorithm [11] on a total of 17 standard benchmark functions, most of which have dimensionality *D*=30, (the only exceptions are f_{28}–f_{30}, each one having *D*=10). The results are summarized in the following few points.

GA vs. ABC-SAM: On all (i.e., 17 out of 17) of the functions, ABC-SAM performs significantly better than GA.

PSO vs. ABC-SAM: On most functions (i.e., 13 out of 17), the performance of ABC-SAM is better than PSO. On the remaining four functions, both of them perform equally well.

DE vs. ABC-SAM: ABC-SAM outperforms DE on as many as nine out of the 17 functions, shows similar performance on six, while DE performs better only on the remaining two functions (f_5 and f_{30}).

ABC vs. ABC-SAM: Since both the algorithms are allowed very large number of function evaluations (i.e., 500,000), both of them precisely locate and reach the global optimum for most of the functions (i.e., 12 out of 17 functions). However, for the remaining five functions, ABC-SAM outperform ABC on four, while ABC performs better only on the remaining one function (i.e., f_7). Thus, the overall performance of ABC-SAM is better than basic ABC algorithm.

Fig. 5.7 makes an overall evaluation of the algorithms by comparing their mean absolute error values on these 17 standard benchmark functions. It shows that both ABC and ABC-SAM have much smaller error values in comparison to all of their counterparts — GA [11], DE [195] and PSO [196]. The best overall performance (i.e., minimum mean absolute error value) is demonstrated by ABC-SAM (error value = 2.48e–01), followed by the basic ABC algorithm (error value = 5.46e–01), while the rest of the algorithms show much larger error values.

Table 5.3: Comparison of ABC-SAM with GA [11], DE [195], PSO [196] and ABC [11] on the standard benchmark functions. The best result for each function is marked with boldface font.

No.	D	f_{min}	Mean Error				
			GA	PSO	DE	ABC	ABC-SAM
f_1	30	0	1.1e+03	**0**	**0**	**0**	**0**
f_2	30	0	11.02	**0**	**0**	**0**	**0**
f_4	30	0	7.4e+03	**0**	**0**	**0**	**0**
f_5	24	0	9.70	1.1e–04	**2.2e–07**	3.1e–03	6.83e–05
f_6	30	0	1.2e+03	0.6666	0.6666	**0**	**0**
f_7	30	0	1.9e+05	15.08	18.20	**0.088**	3.10e+00
f_8	30	0	1.2e+03	**0**	**0**	**0**	**0**
f_9	30	0	1.8e–01	1.2e–03	1.4e–03	3.00e–03	**9.65e–21**
f_{10}	30	0	52.92	43.97	11.72	**0**	**0**
f_{12}	30	–12569.48	8.8e+02	5.7e+03	2.3e+03	**0**	**0**
f_{13}	30	0	14.67	0.16	**0**	**0**	**0**
f_{14}	30	0	10.63	0.017	0.0015	**0**	**0**
f_{17}	30	0	13.38	0.021	**0**	**0**	**0**
f_{18}	30	0	125.06	7.7e–03	2.2e–03	**0**	**0**
f_{28}	10	0	29.57	1364.45	781.55	8.23	**0.48**
f_{29}	10	–9.66015	0.16	5.65	0.069	**0**	**0**
f_{30}	10	–1.4	0.76	1.39	**0.35**	0.97	0.65

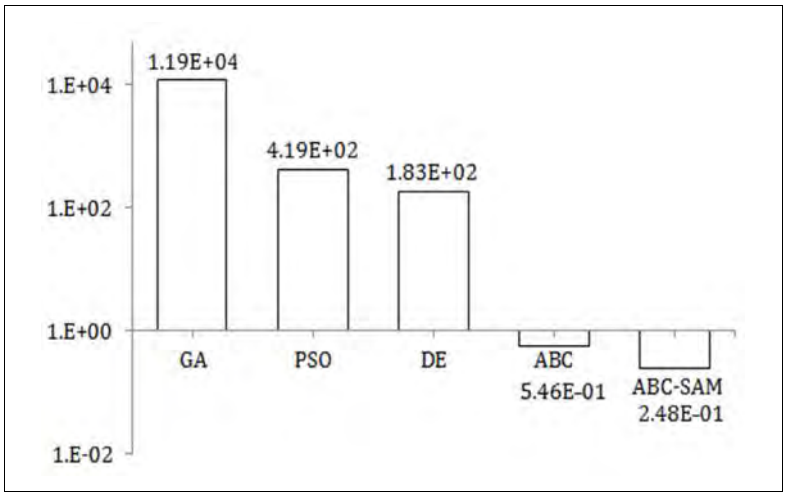


Figure 5.7: Comparison of ABC-SAM with GA [11], DE [195], PSO [196] and the basic ABC [11] algorithm based on their mean absolute error values on the standard benchmark functions. ABC-SAM shows the overall best performance, i.e., the lowest mean absolute error value.

Next, ABC-SAM is compared with some EP-based algorithms, including CEP [102], ALEP [56] and IFEP [57]. Like ABC-SAM, each of these EP-based algorithms is entirely based on the mutation operator for perturbing the candidate solutions. For example, CEP [102] employs the Gaussian distribution for producing the step sizes for mutations, IFEP [57] tries to increase the degree of explorations by mixing a more explorative distribution (i.e., the Cauchy distribution) with the Gaussian distribution, ALEP [56] tries to adaptively control the degree of explorations and exploitations by adapting the parameters of the Lévy distribution. During mutations, IFEP [57] tries to balance explorations with exploitations by generating two candidate solutions from each parent solution — one by using the Gaussian distribution, another by Cauchy distribution, then chooses the better candidate solution to enter into the population. ALEP [56], on the other hand, generates four offspring solutions from each parent solution by using Lévy distribution with four different parameter settings. It has been shown that ALEP [56] and IFEP [57] performed better than either their non-adaptive versions or the CEP [102]. For comparison (Table 5.4), ABC-SAM and CEP [102] have been run for the same population size and the same number of *FE*s as of ALEP [56] and IFEP [57]. The results of ALEP [56] and IFEP [57] and are directly taken from the corresponding papers. Since their results are not available for all the functions of the standard benchmark suite, Table 5.4 compares them on the available 12 functions only. Results show that ABC-SAM outperforms all its competitors on most of the available functions.

Table 5.4: Comparison of ABC-SAM with ALEP [56], IFEP [57] and CEP [102] on the standard benchmark functions. The best result for each function is marked with boldface font. The '+' indicates that ABC-SAM is significantly better than the compared algorithm with at least 95% level of certainty, while '≈' means that the difference is not statistically significant.

Function	Mean Error				*t*-Test (ABC-SAM vs.)	
	ABC-SAM	IFEP	ALEP	CEP	IFEP	ALEP
f_1	**6.12e–20**	4.16e–05	6.32e–04	9.1e–04	+	+
f_3	8.58e–01	–	**4.28e–02**	2.1e+02	-N/A-	–
f_7	**2.65**	–	4.34e+01	8.6e+01	-N/A-	+
f_{10}	**4.15e–18**	–	5.85e+00	4.3e+01	-N/A-	+
f_{12}	3.06e+01	**8.87e–02**	–	4.0e+01	–	-N/A-
f_{13}	**4.34e–09**	4.83e–03	1.90e–02	1.5e+00	+	+
f_{14}	**9.50e–10**	4.53e–02	2.40e–02	8.7e–02	+	+
f_{17}	**6.67e–12**	–	6.00e–06	4.8e–01	-N/A-	+
f_{18}	**3.75e–02**	–	9.80e–05	8.9e–02	-N/A-	–
f_{25}	**0.01**	4.08	1.03	2.85	+	+
f_{26}	**0.00**	3.41	0.26	0.78	+	+
f_{27}	**0.01**	1.71	0.62	0.85	+	+

5.4.3 Comparison of ABC-SAM with Existing ABC-Variants

In the Tables 5.5–5.8, ABC-SAM is compared with a few other recent and state-of-the-art improved ABC-variants, including cooperative ABC (CABC) [60], Gbest-guided ABC (GABC) [64], ABC with diversity strategy (DABC) [61] and Chaotic ABC (ChABC) [62]. At first, ABC-SAM is compared with CABC [60], which is an explorative and cooperative ABC variant. CABC [60] is introduced in two different variants — CABC_S and CABC_H. The first variant — CABC_S enforces more search space explorations by decomposing the search space and employing different bee colonies for the different subspaces. The other variant — CABC_H follows an execution model that repeatedly alternates between explorative CABC_S and exploitative ABC. For a fair comparison, ABC-SAM is implemented with the same settings, as in [60]. Table 5.5 shows that ABC-SAM outperforms CABC_S on three out of the six functions, shows similar performance on one (f_7), while CABC_S performs better on the remaining two. The other variant CABC_H outperforms ABC-SAM on three (out of six) functions, while ABC-SAM performs better on the remaining three. Thus the overall performance of ABC-SAM is at least comparable to the cooperative CABC counterparts. The next algorithm GABC [64] alters the basic perturbation formula (2.6) of ABC by incorporating the information about the global best solution found so far in order to increase the convergence speed of the algorithm. For a fair comparison with GABC [64], ABC-SAM is run with the same settings of GABC [64], i.e., with SN=80 and MCN=5000, as suggested in [64]. Table 5.6 shows that ABC-SAM outperforms GABC [64] on

most (four out of five) of the functions. The reason might be that GABC [64] becomes too much exploitative by using the best-so-far solution during its perturbation operations. More exploitations may improve the convergence speed and the final solution quality for some unimodal and low dimensional functions, such as f_7 in Table 5.6, but fails for most of the high dimensional functions, e.g., four out of the five functions with $D = 30$ in Table 5.6.

Table 5.5: Comparison of ABC-SAM with CABC_S [60] and CABC_H [60]. Boldface font marks the best performance for each function.

Function	CABC_S		CABC_H		ABC-SAM	
	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
f_1	3.30e–19	2.00e-19	5.92e–18	3.56e–18	**8.17e–21**	**2.68e–21**
f_7	6.33e+00	7.68e+00	**4.80e–01**	**8.55e–01**	6.75e+00	2.24e+00
f_{10}	**0**	**0**	**0**	**0**	4.83e–21	6.09e–22
f_{12}	1.30e–04	5.21e–06	**1.27e–04**	**0**	5.58e+00	1.08e+00
f_{13}	1.83e–14	9.86e–15	8.35e–15	4.13e–15	**2.59–16**	**7.98e–17**
f_{14}	4.42e–02	2.99e–02	7.96e–03	9.06e–03	**9.12e–12**	**4.16e-12**
+	3		3			
–	2		3			
≈	1		0			

Table 5.6: Comparison between ABC-SAM and GABC [64] based on the final solution quality. The best result for each function is marked with boldface font.

Function	D	GABC (C=1.0)		GABC (C=1.5)		ABC-SAM	
		Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
f_1	30	4.31e–16	7.49e–17	4.17e–16	7.36e–17	**6.02e–93**	**2.92e–93**
	60	1.43e–15	1.43e–16	1.43e–15	1.37e–16	**3.28e–28**	**6.50e–29**
f_7	2	3.93e–04	4.45e–04	**1.68e–04**	**1.45e–04**	9.47e–03	4.31e–03
	3	**2.63e–03**	**2.11e–03**	2.65e–03	2.22e–03	1.05e–01	8.19e–02
f_{10}	30	9.47e–15	2.15e–14	1.32e–14	2.44e–14	**4.33e–105**	**7.67e–106**
	60	4.16e–13	1.77e–13	3.52e–13	1.24e–13	**2.74e–36**	**8.73e–37**
f_{13}	30	3.31e–14	2.90e–15	3.21e–14	3.25e–15	**1.36e–24**	**6.03e–25**
	60	1.04e–13	1.07e–14	1.00e–13	6.08e–15	**5.63e–27**	**1.08e–27**
f_{14}	30	8.88e–17	8.45e–17	2.96e–17	4.99e–17	**9.51e–22**	**4.60e–23**
	60	9.47e–16	7.84e–16	7.54e–17	4.12e–16	**7.52e–20**	**3.62e–20**
+	4		4				
–	1		1				
≈	0		0				

Table 5.7: Comparison between ABC-SAM and DABC [61]. The best result for each function is marked with boldface font; if not both the algorithms produce identical results.

Function	*D*	DABC		ABC-SAM	
		Mean	Std. Dev.	Mean	Std. Dev.
f_1	10	2.01e–17	5.63e–17	**0**	**0**
	30	2.01e–16	2.85e–17	**2.98e–46**	**5.78e–47**
f_7	10	**2.73e–03**	**7.04e–03**	6.26e–02	2.40e–02
	30	**1.42e–02**	**2.53e–02**	2.34e–01	9.63e–02
f_{10}	10	0	0	0	0
	30	0	0	0	0
f_{14}	10	0	0	0	0
	30	2.59e–16	1.22e–16	**7.85e–20**	**3.42e-20**
+		3			
–		2			
≈		3			

Table 5.8: Comparison between ABC-SAM and ChABC [62] based on the final solution quality. The best result for each function is marked with boldface font.

Function	*D*	ChABC		ABC-SAM	
		Mean	Std. Dev.	Mean	Std. Dev.
f_1	30	2.99e–16	3.54e–17	**8.28e–74**	**3.82e–74**
f_7	30	**6.33e–02**	**8.96e–02**	6.72e–02	5.40e–02
f_{10}	30	**0**	**0**	5.90e–89	2.06e–89
f_{12}	30	**3.81e–04**	**2.07e–04**	2.30e–02	6.48e–03
f_{13}	30	2.93e–14	2.99e–15	**5.76e–21**	**1.41e–22**
f_{14}	30	2.70e–16	6.20e–17	**3.96e–18**	**1.51e–18**
+		3			
–		2			
≈		1			

Next, ABC-SAM is compared with two more explorative and improved ABC-variants — the DABC [61] (Table 5.7) and the ChABC [62] (Table 5.8). DABC [61] replaces the basic perturbation equation (2.6) using two different equations — one explorative and the other one exploitative. In every cycle, DABC [61] measures the existing diversity d of the population of candidate solutions. If d falls below some predefined threshold div_{low}, then the explorative perturbation variant is employed. Otherwise, the exploitative variant is used for perturbations.

In Table 5.7, ABC-SAM is compared with DABC [61] on four functions, with parameter values of SN=20, MCN=5000 and $limit$=100, as suggested in [61]. Results indicate that ABC-SAM performs better than DABC on two out of the four functions (f_1 and f_{14}), shows similar performance on one (i.e., f_{10}), while DABC [61] performs better only on one function (i.e., f_7). The reason why DABC [61] can't perform as well as ABC-SAM may be that DABC [61] is entirely based on measuring the population diversity, but there is no accurate and universally accepted metric of diversity that can correctly measure the maturity of the optimization process. Besides, the simple strategy of using a fixed value of div_{low} to decide on explorations/exploitations may cause repeated oscillations between explorations and exploitations, reducing the convergence speed and compromising the final solution quality of DABC [61]. The other explorative ABC-variant — ChABC [62] tries to perform more search space explorations by using chaotic (instead of random) search behavior during perturbations. Chaotic dynamics are reported to provide a simple mechanism to escape from local minima [62]. Chaotic dynamics can be produced in many ways, e.g., using the logistic equations (4)–(7) in [62]. For a fair comparison, both ABC-SAM and ChABC [62] are executed for 5000 cycles with population size of 70 and $limit$=200, as suggested in [62]. Results (Table 5.8) show that ABC-SAM outperforms ChABC [62] on three out of the six functions, shows similar performance on one (f_7), while ChABC [62] performs better on the remaining two functions. Thus the overall performance of ABC-SAM is better than ChABC [62]. The reason may be that, unlike ABC-SAM, ChABC [62] does not consider the individual exploitative/explorative requirements of each candidate solution separately; rather ChABC [62] employs the same chaotic strategy uniformly for all the candidate solutions across the population, completely ignoring their individual explorative/exploitative needs, which may not be a good strategy to deal with complex optimization tasks.

5.4.4 ABC-SAM on CEC2005 Benchmark Functions

ABC-SAM is also evaluated using the recently introduced CEC2005 benchmark suite [76], which is introduced in the special session on real parameter optimization at the IEEE 2005 Congress on Evolutionary Computation (CEC-2005), held on 2-4 September 2005 at Edinburgh, UK. The CEC2005 benchmark suite consists of 25 more complex and challenging functions, briefly introduced previously in the Table 2.4, section 2.17. A more detailed description can be found in the Appendix A and in [76].

Using the CEC2005 benchmark suite, ABC-SAM is tested and compared with a few other evolutionary and swarm intelligence algorithms, such as the basic ABC algorithm [11], the self-adaptive differential evolution (SADE) [230], the dynamic multi-swarm PSO with local search (DMS-PSO) [231] and the PSO with recombination by dynamic linkage discovery (PSO-RDL) [232]. SADE [230] is an improved DE variant that uses a learning strategy over its past successes and failures to produce better trial solutions from the existing ones by gradually

self-adapting the values of some control parameters of the standard DE algorithm. DMS-PSO [231] is an improved and more explorative PSO variant that tries to improve both the degree of explorations (by employing multiple dynamic swarms, instead of a single swarm) and the intensity of exploitations (by increasing local search operations using a quasi-Newton method). PSO-RDL [232] puts efforts to discover the linkages among the variables and to identify important building blocks present in the good quality solutions, then tries to produce better trial solutions using the good building blocks and the linkage information. All these algorithms have been evaluated on the CEC2005 functions with dimensionality $D=10$ and FE = 100,000. The other common parameters of ABC and ABC-SAM are the colony size SN and *limit*, which are set as $SN=20$ and $limit=200$. The remaining parameters of ABC-SAM are set as — $\tau_1=40$, $\tau_2=20$, $SF_{min}=10^{-8}$, $\alpha=10$ and $\beta=0.9$. The results of SADE [230], DMS-PSO [231] and PSO-RDL [232] are obtained directly from the corresponding papers. Tables 5.9–5.10 present their mean errors over 25 independent runs on each function. The results can be summarized in following few points.

- Out of the 14 functions in Table 5.9 (i.e., non-composite functions F1–F14), ABC-SAM becomes the best performer on four functions (F2, F4, F10 and F14), outperforming all other algorithms on these functions. SADE performs slightly better, by becoming the best performer on five functions, while the other competitors show similar or worse performance — ABC, PSO-RDL and DMS-PSO become the best performer only on one, one and four functions, respectively.
- For the more complex hybrid composition functions F15–F25 (Table 5.10), ABC-SAM show the best performance on as many as five (out of 11) functions, while DMS-PSO, PSO-RDL, SADE and ABC show the best performance only on two, zero, two and two functions, respectively.
- To compare ABC-SAM with the basic ABC algorithm, it is remarkable that ABC-SAM has improved the results of ABC for as many as 18 out of the 25 functions. This clearly indicates the effectiveness of the more explorative and self-adaptive design of ABC-SAM, especially for these more complex optimization tasks.
- As Fig. 5.8 demonstrates, ABC-SAM shows the minimum mean absolute error on the hybrid composite functions F15–F25, outperforming all other algorithms in comparison. For the non-hybrid functions F1–F14, the error value of ABC-SAM is much smaller than the errors of both of ABC and PSO-RDL, but worse than DMS-PSO and SADE.
- Summarizing all the points above, we can conclude that the overall performance of ABC-SAM is at least comparable to and often better that all of its counterparts in this comparison on the CEC2005 benchmark functions.

Table 5.9: Comparison of ABC-SAM with DMS-PSO [231], PSO-RDL [232], SADE [230] and ABC [11] on the non-composite functions F1–F14 of the CEC2005 benchmark suite [76]. The best result for each function is marked with boldface font.

Function	DMS-PSO	PSO-RDL	SADE	ABC	ABC-SAM
F1	**0.00e+00**	2.50e–14	**0.00e+00**	4.89e–17	3.73e–21
F2	1.30e–13	1.77e–13	1.05e–13	4.81e–14	**7.03e–16**
F3	**7.01e–09**	9.6e–02	1.67e–05	2.50e+03	7.80e+01
F4	1.8e–03	2.57e–07	1.42e–05	1.50e–16	**6.55e–18**
F5	1.16e–06	**2.09e–07**	1.23e–02	5.82e+01	4.72e+00
F6	6.89e–08	9.57e–01	**1.20e–08**	3.31e+00	5.61e–02
F7	4.52e–02	5.73e–02	**1.99e–02**	2.52e–01	3.13e+00
F8	2.00e+01	2.00e+01	2.00e+01	2.03e+01	2.03e+01
F9	**0.00e+00**	1.25e+01	**0.00e+00**	4.87e–17	8.58e–22
F10	3.62e+00	3.86e+01	4.97e+00	2.22e+01	**3.11e+00**
F11	**4.62e+00**	5.58e+00	4.89e+00	5.46e+00	9.27e+00
F12	2.40e+00	1.31e+02	**4.50e–07**	9.85e+01	4.43e–01
F13	3.69e–01	8.87e–01	2.20e–01	**2.96e–02**	7.09e–01
F14	2.36e+00	3.78e+00	2.92e+00	3.41e+00	**2.23e+00**

Table 5.10: Comparison of ABC-SAM with DMS-PSO [231], PSO-RDL [232], SADE [230] and ABC [11] on the hybrid composition functions F15–F25 of the CEC2005 benchmark suite [76]. The best result for each function is marked with boldface font.

Function	DMS-PSO	PSO-RDL	SADE	ABC	ABC-SAM
F15	4.85e+00	2.71e+02	3.20e+01	1.53e+01	**4.46e+00**
F16	9.48e+01	2.22e+02	1.01e+02	1.75e+02	**8.20e+01**
F17	**1.10e+02**	2.22e+02	1.14e+02	1.96e+02	2.13e+02
F18	7.61e+02	1.02e+03	7.19e+02	4.46e+02	**4.32e+02**
F19	7.14e+02	9.85e+02	7.05e+02	4.51e+02	**4.18e+02**
F20	8.22e+02	9.59e+02	7.13e+02	**4.38e+02**	5.11e+02
F21	5.36e+02	9.94e+02	**4.64e+02**	4.87e+02	4.69e+02
F22	**6.92e+02**	8.87e+02	7.32e+02	8.59e+02	7.65e+02
F23	7.30e+02	1.08e+03	6.64e+02	5.98e+02	**5.63+02**
F24	2.24e+02	7.20e+02	**2.00e+02**	2.02e+02	2.01e+02
F25	3.66e+02	1.76e+03	3.76e+02	**3.38e+02**	3.48e+02

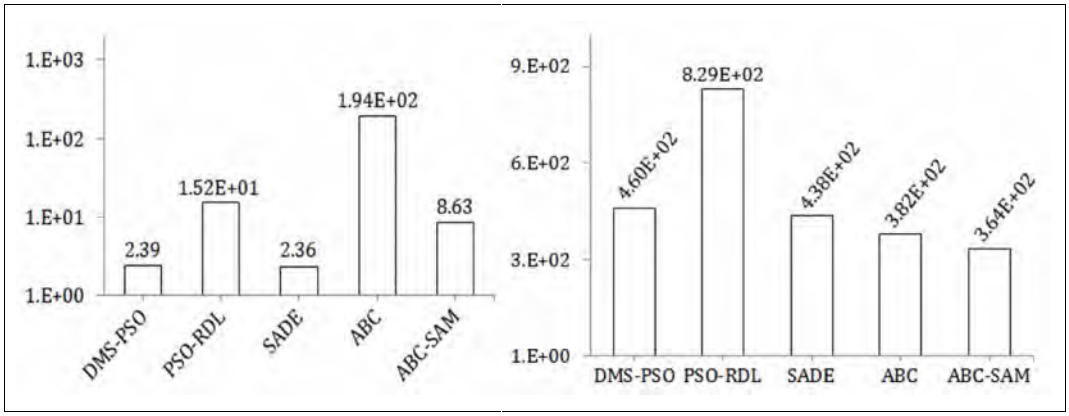


Figure 5.8: Comparison of ABC-SAM with DMS-PSO [231], PSO-RDL [232], SADE [230] and the basic ABC algorithm [11], based on their mean absolute error values on the CEC2005 benchmark functions F1-F14 (on the left) and the more complex hybrid composition functions F15-F25 (on the right).

5.5 Conclusion and Future Research Directions

This chapter introduces ABC-SAM — a novel variant of the standard ABC algorithm that incorporates a self-adaptive mutation scheme which automatically adapts the mutation step size, suitably either for better exploitations or for more explorations, separately for every candidate solution of the population. ABC-SAM significantly outperforms the basic ABC algorithm and several other evolutionary and swarm intelligence algorithms, including some recently introduced improved ABC-variants on two different suites of benchmark functions. The empirical results and comparisons prove the better effectiveness of the proposed step size adaptation scheme of ABC-SAM.

There might be several possible future research works based on ABC-SAM. For example, we have adapted only the scaling factors, i.e., the SF_i values, maintained separately for each candidate solution $\boldsymbol{x}_i$. For the other control parameters of ABC-SAM (i.e., *SN*, *limit*, τ_1, τ_2, SF_{min}, α, β), we have simply used fixed, predefined values all through the optimization process. An interesting research topic would be how to make some (or, all) of these control parameters adaptive and/or self-adaptive along the course of the optimization process. Secondly, we have used simple forms of exponential and negative-exponential distributions for explorations and exploitations, respectively. Some other distributions, such as the Lévy [56] and Cauchy [57] distributions, possibly with adaptive distribution parameters, may also be tested to evaluate how well they can achieve the dynamic explorative/exploitative objectives during optimization.

Thirdly, the self-adaptive techniques introduced in ABC-SAM are generic enough; so similar techniques may be incorporated and hybridized with many other existing and recently introduced evolutionary and swarm intelligence algorithms to alter and (possibly) improve their explorative/exploitative characteristics. Fourthly, ABC-SAM is evaluated and tested only on the continuous optimization problems. It would be interesting to evaluate how well ABC-SAM can perform on many other existing optimization problems, especially the discrete and the real world problems.

Chapter 6

Artificial Bee Colony Algorithm with Improved Explorations

6.1 Introduction

As explained in the previous sections 2.14–2.15, the premature convergence is a major problem for both evolutionary and swarm intelligence algorithms, which usually originates from the insufficient degree of explorative search capability during the optimization process. In addition to premature convergence, another problem with ABC is the fitness stagnation, where the population of solutions stops improving towards the global optimum, even without converging to any of the locally optimal points [54]. The risk of fitness stagnation and premature convergence usually rises with reduced explorations and increased exploitations. This chapter introduces ABC with Improved Explorations (ABC-IX) — a novel algorithm that modifies both the selection and perturbation operations of the basic ABC algorithm in an explorative and self-adaptive way. First, an explorative selection scheme based on simulated annealing allows ABC-IX to probabilistically accept both better and worse candidate solutions, while the standard ABC algorithm can accept better solutions only. Secondly, a self-adaptive strategy enables ABC-IX to automatically adapt the perturbation rate, separately for each candidate solution, during producing new candidate solutions by perturbing the existing ones. Both these new improvements are intended to increase the explorative search capacity of the basic ABC algorithm during the optimization process. We have evaluated ABC-IX using several benchmark problems on numerical optimization and compared the results with a number of existing state-of-the-art evolutionary and swarm intelligence algorithms.

6.2 Organization of the Chapter

The rest of this chapter is organized as follows. Section 6.3 briefly presents a number of recently proposed improved ABC-variants and explains how the proposed algorithm — ABC-IX is significantly different from most of them. Section 6.4 introduces ABC-IX and explains its improvements on the plain selection and perturbation operations of the basic ABC algorithm along with the necessary pseudocode. Section 6.5 evaluates ABC-IX using a number of benchmark problems, specifies the parameter settings of ABC-IX and some other algorithms in comparison, compares their results and makes a few comments and analysis on their results and comparisons. Finally, section 6.6 concludes with a summary of this chapter and provides a few directions for further research with ABC-IX.

6.3 Differences of ABC-IX with Other Existing Works

There exist several recent works (e.g., [59]–[68]) that try to tweak the explorative and/or exploitative properties of the standard ABC algorithm. For example, the ABC with self-adaptive mutation (ABC-SAM), as described in chapter 5, introduces an adaptive perturbation scaling factor SF_i for every bee agent and tries to perform sufficient degree of both explorations and exploitations by regularly adapting the SF_i values by using two different distributions — one explorative and the other exploitative. The cooperative ABC (CABC) [60] algorithm decomposes the search space into a number of smaller subspaces and enforces more explorations by employing several bee colonies to explore the different subspaces. Another explorative ABC-variant — ABC with diversity strategy (DABC) [61] tries to preserve sufficient amount of diversity within the bee population by switching between two different perturbation schemes — one explorative and the other exploitative. Chaotic ABC (ChABC) [62] is another explorative ABC-variant that uses dynamic chaotic sequence generators, instead of random number generators, to improve the explorative characteristics of the basic ABC algorithm. The explorative search capacity of ABC may also be improved by intelligent organization of the locally optimal points, as demonstrated by Fenglei et al. [63]. Another variant — Gbest-guided ABC (GABC) [64] tries to improve the degree of exploitations and the rate of convergence of ABC by using the information of the global best solution found so far during the perturbation operations. Hooke Jeeves ABC (HJABC) [65] is another improved hybrid ABC-variant that hybridizes an extensively exploitative local search technique, i.e., the Hooke Jeeves pattern search, with the ABC algorithm. Another hybrid ABC variant is the elitist ABC (EABC) [68] which hybridizes the standard ABC algorithm with two different local search operators to intensify the degree of exploitations around the best candidate solution found so far. Quan and Shi [67]

reported significant improvement of the convergence speed of ABC by introducing and carefully employing an exploitative search iteration operator that is based on the fixed point theorem of contractive mapping. Qingxian and Haijun [66] employed the Boltzmann selection scheme to replace the traditional roulette wheel selection of the basic ABC algorithm and introduced an improved initialization scheme to improve the degree of exploitations and convergence speed of the basic ABC algorithm.

A major weakness of most of the existing ABC-variants, as presented in the previous paragraph, is that none of these algorithms (i.e., [60]–[68]) considers the individual explorative/exploitative requirements of every candidate solution separately; rather they treat all the candidate solutions equally, employing some population-wide uniform strategy, identically on all the solutions across the population, completely ignoring their different explorative/exploitative needs. Another limitation of most existing algorithms is that they try to improve either the explorative (e.g., [60]–[63]) or the exploitative (e.g., [64]–[68]) properties of the standard ABC algorithm. The explorative enhancements are usually based on more explorative perturbation, selection and/or initialization ([59], [62], [63]) or employing some technique to maintain more population diversity (e.g., [60], [61]), while the exploitative developments are usually based on increasing the local search operations around the best candidate solutions of the population (e.g., [64], [65], [68]). However, only a few (i.e., [59], [61], [64]) of these algorithms make some efforts, more or less, to balance between the explorative and exploitative improvements. But most of them use some fixed, rather than adaptive, strategy to balance the explorations with exploitations. For example, DABC [61] uses a fixed threshold value d_{low} and tries to maintain the population diversity always above d_{low}, GABC [64] controls the degree of exploitations with some fixed value of its control parameter C, and ABC-SAM (chapter 5) uses a fixed, exploitative perturbation rate and tries to induce an equal (rather than adaptive) proportion of explorative and exploitative perturbations.

ABC-IX differs from most other existing ABC-variants (e.g., [60]–[68]) in several important ways. Firstly, ABC-IX tries to customize the degree of explorations and exploitations, separately for every candidate solution $\boldsymbol{x_i}$ of the population by introducing and separately maintaining a control parameter $\boldsymbol{x_i}.q$ (i.e., perturbation rate) for each $\boldsymbol{x_i}$. Secondly, ABC-IX adopts a self-adaptive (rather than fixed, as in [59], [61], [64]) technique to adaptively control and customize the perturbation rate for each candidate solution. Thirdly, ABC-IX tweaks both the selection and perturbation operations of the basic ABC algorithm, because both these operations of ABC are biased towards more exploitations (rather than explorations). ABC-IX pushes the selection operation of ABC towards more explorations by using a simulated

annealing based probabilistic selection scheme. Also, the basic perturbation operation of ABC is modified by using a self-adaptive perturbation rate that can customize the degree of explorations and exploitations, separately for every candidate solution of the population. Customization of the degree of explorations and exploitations at the individual solution level is more rational and usually more effective than some population-wide global strategy, because each candidate solution is usually at a different region of the search space with different and dynamically changing explorative/exploitative requirements. Hence, adaptation of the control parameters, separately for each candidate solution, is more suitable for automatically dealing with its dynamically evolving exploitative/explorative requirements.

6.4 The Proposed Algorithm — ABC-IX

The proposed algorithm — ABC-IX differs from the standard ABC algorithm in two important ways. Firstly, ABC accepts a newly produced candidate solution $\boldsymbol{v_i}$ only if $\boldsymbol{v_i}$ has higher fitness value than the original solution $\boldsymbol{x_i}$ (Fig. 2.9, steps 5, 9). This exploitative selection scheme denies any possible downhill movement and allows only uphill steps in the fitness landscape, which may turn the entire population into parallel hill-climbers and may lead to premature convergence to the locally optimal points, failing to locate the global optimum. In contrast, ABC-IX promotes more search space explorations by probabilistically allowing some downhill steps using a simulated annealing-based selection scheme (Fig. 6.2). Secondly, ABC perturbs only a single parameter of an existing candidate solution $\boldsymbol{x_i}$ by using (2.6) to produce the new solution $\boldsymbol{v_i}$ (Fig. 2.9, steps 4, 8). This means ABC has a perturbation rate of $1/D$, which is kept constant all through the execution. Such a fixed and small perturbation rate is likely to produce the offspring solution $\boldsymbol{v_i}$ in close vicinity of the parent $\boldsymbol{x_i}$, which is exploitative. In contrast, ABC-IX introduces a self-adaptive scheme to automatically control and adapt the perturbation rate at the individual solution level. ABC-IX includes a perturbation probability within each solution $\boldsymbol{x_i}$, which is addressed as $\boldsymbol{x_i}.q$ (Fig. 6.2) and the value of $\boldsymbol{x_i}.q$ is self-adapted gradually, cycle by cycle, separately for each candidate solution $\boldsymbol{x_i}$.

Fig. 6.1 presents the pseudocode for ABC-IX, which has the same algorithmic framework of the basic ABC algorithm (Fig. 2.9). However, ABC-IX differs from ABC in how the existing solutions are perturbed (i.e., perturbation steps 4 and 8 in Fig. 2.9) and how the new solutions are re-inserted into the population (i.e., selection steps 5 and 9). Both these modifications, as presented in Fig. 6.2, are to increase the explorative capacity of the basic ABC algorithm. They are further elaborated in the following paragraphs.

Algorithm 6.1: Artificial Bee Colony (ABC) Algorithm with Improved Exploration

1: Initialize a population of SN food source positions (candidate solutions) $\boldsymbol{x_i}$, for i = 1, 2, …, SN. Each $\boldsymbol{x_i}$ is a vector of D parameters: $\boldsymbol{x_i} = [x_{i1}, x_{i2}, \ldots, x_{iD}]^T$

2: Evaluate the fitness of each food source position using (2.3)

3: **repeat**

4*: For each employed bee, perturb its food source position $\boldsymbol{x_i}$ to produce a new food position $\boldsymbol{v_i}$ by using the pseudocode for perturbation in Fig. 6.2

5*: Evaluate each new solution $\boldsymbol{v_i}$ by (2.3). Select either $\boldsymbol{x_i}$ or $\boldsymbol{v_i}$ using the simulated annealing based probabilistic selection scheme (Fig. 6.2)

6. Calculate the probability p_i associated with each food source position $\boldsymbol{x_i}$ using (2.7)

7: For each onlooker bee, assign it to a food source $\boldsymbol{x_i}$, proportionally based on the probability p_i

8*: For each onlooker bee, perturb its food source position $\boldsymbol{x_i}$ to produce a new food position $\boldsymbol{v_i}$ by using the pseudocode for perturbation in Fig. 6.2

9*: Evaluate each new solution $\boldsymbol{v_i}$ using (2.3). Select either $\boldsymbol{x_i}$ or $\boldsymbol{v_i}$ by using the simulated annealing based probabilistic selection scheme (Fig. 6.2)

10: If a food source has not improved during the last *limit* cycles, then abandon it and replace it with a new randomly placed scout bee with its food source $\boldsymbol{x_i}$ produced by (2.8)

11: Memorize the best food source position found so far

12*: Set cycle counter C=C + 1 and system temperature $T = \alpha * T$

13: **until** C = Maximum cycle number (MCN)

14: **return** the best food source position (i.e., candidate solution) found so far

Figure 6.1: Pseudocode of ABC-IX. Steps that differ from basic ABC (Fig. 2.9) are marked with '*'

Algorithm 6.2: Simulated Annealing (SA) based Probabilistic Selection Scheme for ABC-IX

input: Two candidate solutions $\boldsymbol{x_i}$ and $\boldsymbol{v_i}$

output: Either of $\boldsymbol{x_i}$ or $\boldsymbol{v_i}$, selected to be included into the population

begin
  if $fitness(\boldsymbol{v_i}) \geq fitness(\boldsymbol{x_i})$ **then**
    return $\boldsymbol{v_i}$
  else
    $\Delta E = fitness(\boldsymbol{x_i}) - fitness(\boldsymbol{v_i})$
    if $rand(0,1) \leq \exp(-\Delta E/T)$ **then**
      return $\boldsymbol{v_i}$
    endif
  endif
  return $\boldsymbol{x_i}$
end

Algorithm 6.3: Perturbation by ABC-IX
input: An existing candidate solution $\boldsymbol{x_i}$
output: Perturbed candidate solution $\boldsymbol{v_i}$

begin
  $\boldsymbol{v_i} = \boldsymbol{x_i}$
  if $rand(0,1) \leq t$ **then**
    $\boldsymbol{v_i}.q = q_{min} + (1.0 - q_{min}) * rand(0,1)$
  else
    $\boldsymbol{v_i}.q = \boldsymbol{x_i}.q$
  endif
  for j = 1 to D **do**
    if $rand(0,1) \leq \boldsymbol{v_i}.q$ **then**
      $k = rand\{1, 2, \ldots, SN\}$
      $\varphi_{ij} = rand(-1,1)$
      $v_{ij} = x_{ij} + \varphi_{ij} * (x_{kj} - x_{ij})$
    endif
  enddo
  return $\boldsymbol{v_i}$
end

Figure 6.2: Pseudocode of simulated annealing based probabilistic selection scheme (on the left) and perturbation with self-adaptive perturbation rate (on the right) for ABC-IX

6.4.1 Simulated Annealing Based Probabilistic Selection Scheme

Simulated Annealing (SA) is a heuristic for finding the global optimum in the presence of multiple local optima. Starting from an initial solution, SA produces new, randomly perturbed solutions. SA accepts both better and worse new solutions, but the probability of accepting a

worse new solution is reduced over time, based on a gradually decreasing control parameter T (i.e. system temperature, from the analogy to the real annealing procedure in metallurgy). More specifically, given an initial candidate solution $\boldsymbol{x_i}$, SA randomly perturbs $\boldsymbol{x_i}$ to produce $\boldsymbol{y_i}$ in the neighborhood of $\boldsymbol{x_i}$. Then the change ΔE of the objective function value is computed as $\Delta E = obj(\boldsymbol{x}_i) - obj(\boldsymbol{y}_i)$. If $\boldsymbol{y_i}$ is better than $\boldsymbol{x_i}$ (i.e., $\Delta E < 0$), SA readily accepts $\boldsymbol{y_i}$ as its current state and discards the old solution $\boldsymbol{x_i}$. However, in case the new solution $\boldsymbol{y_i}$ is worse than $\boldsymbol{x_i}$ (i.e., $\Delta E > 0$), SA may still accept $\boldsymbol{y_i}$, but only with probability = $exp(-\Delta E/T)$, where T is the current system temperature that is gradually decreased during the entire procedure. SA usually starts with a high initial temperature T_0 to ensure high degree of initial explorations by frequently accepting worse solutions (i.e., large value of T makes $\Delta E/T \to 0$, thus the probability $exp(-\Delta E/T) \to 1$). As T gradually decreases with the increasing iterations, SA becomes increasingly exploitative, accepting better solutions only. Fig. 6.2 presents the pseudocode for the SA-based probabilistic selection scheme, which probabilistically allows both better and worse new solutions and thus ensures both uphill and downhill moves in the fitness landscape to make ABC-IX more resilient against local optima and premature convergence. Temperature T is gradually decreased by the exponential cooling schedule [233], which relates temperatures $T(t)$ and $T(t+1)$ over two successive cycles by $T(t+1) = \alpha * T(t)$. For most of the evaluations of ABC-IX (e.g., Tables 6.2–6.10), we used $\alpha = 0.99$, and the initial temperature T_0 is set to 50 times of the difference of fitness values of the best and worst candidate solutions of the first generation.

6.4.2 Self-adaptive Perturbation Rate

The standard ABC algorithm perturbs only a single, randomly picked parameter of the existing candidate solutions using (2.6). This performs search along only one dimension at a time, which is usually suitable for separable problems where the parameters are independent of each other, but may be inappropriate for more complex non-separable problems. In contrast, ABC-IX can perturb any number of parameters allowing search along any possible direction. Fig. 6.3 illustrates this using an example of a 2D search space. Allowing perturbation of both the parameters (i.e., both x_{i1} and x_{i2}) can produce $\boldsymbol{v_i}$ along any possible direction from $\boldsymbol{x_i}$. This can be more effective, especially for non-separable problems, than perturbing either x_{i1} or x_{i2}, one at a time as is done by the basic ABC algorithm. This is why ABC-IX tries to perform search along any possible direction from $\boldsymbol{x_i}$ by maintaining and automatically adapting a control parameter $\boldsymbol{x_i}.q$, separately for every candidate solution $\boldsymbol{x_i}$, that controls the perturbation rate during producing the trial solution $\boldsymbol{v_i}$ from $\boldsymbol{x_i}$. To perform an automatic self-adaptation of the value of $\boldsymbol{x_i}.q$, ABC-IX

does the following — before perturbing any other parameter of $\boldsymbol{x_i}$ during producing $\boldsymbol{v_i}$, the value of $\boldsymbol{x_i}.q$ is perturbed first, with probability t, using (6.1). This perturbed value of $\boldsymbol{x_i}.q$ is inherited by $\boldsymbol{v_i}$, which is now referred as $\boldsymbol{v_i}.q$ and is used as the probability of perturbing the parameters of $\boldsymbol{x_i}$ to produce $\boldsymbol{v_i}$ from $\boldsymbol{x_i}$. A more effective value of $\boldsymbol{v_i}.q$ is likely to produce fitter new solutions, which are supposed to survive better than $\boldsymbol{x_i}$ and produce better, newer solutions which will propagate the better values of the perturbation probability. Thus a gradual self-adaptation towards better, more effective perturbation rates will take place, allowing a self-adaptive and appropriate perturbation rate for all the candidate solutions across the population.

$$v_i.q = \begin{cases} q_{min} + rand\,(0,1) * (q_{max} - q_{min}) & if\ rand\,(0,1) < t \\ x_i.q & otherwise \end{cases} \tag{6.1}$$

Here, t is the probability that the perturbation probability $\boldsymbol{x_i}.q$ itself is perturbed first before perturbing any other parameter of $\boldsymbol{x_i}$. For all our experiments, we have set $q_{max} = 1.0$, $q_{min} = 1/D$ and t to either of 0.10, 0.20 or 0.30 (for ABC-IX[1], ABC-IX[2] and ABC-IX[3] in Table 6.1, respectively). For all other Tables in this chapter (i.e., Tables 6.2–6.10), t is set to 0.10.

But how the degree of explorations and exploitations around each candidate solution $\boldsymbol{x_i}$ is affected by its self-adaptive perturbation rate $\boldsymbol{x_i}.q$? A small perturbation rate is likely to perturb only a few parameters of the candidate solution $\boldsymbol{x_i}$ which usually produces the new solution $\boldsymbol{v_i}$ in the vicinity of $\boldsymbol{x_i}$ and hence is exploitative. Conversely, a high perturbation rate can induce large variation on a candidate solution by perturbing many of its parameters at once, which is suitable for explorations. The gradual adjustment of the perturbation rate control parameter $\boldsymbol{x_i}.q$ through the successive applications of (6.1) makes it possible to gradually adapt and customize the degree of explorations and exploitations around every candidate solution $\boldsymbol{x_i}$.

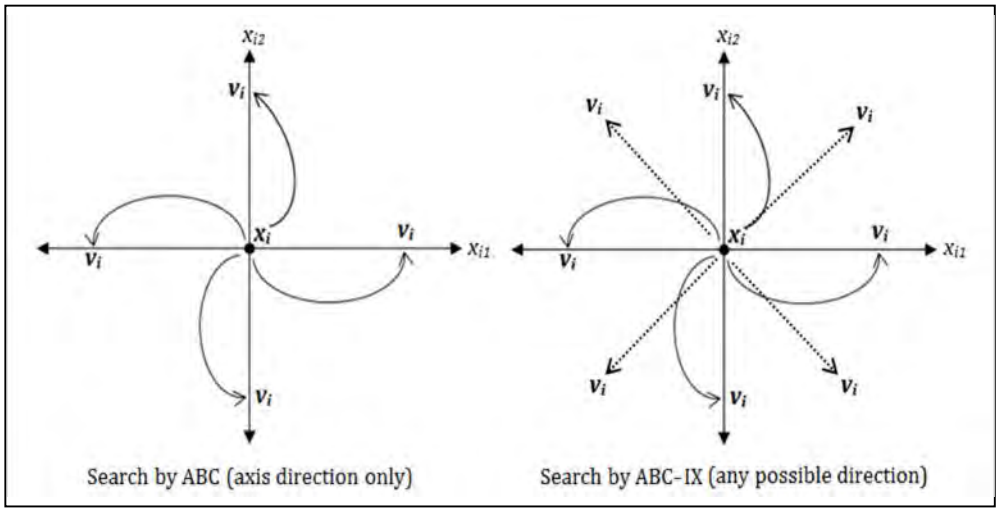


Figure 6.3: Search direction by ABC (on the left) and ABC-IX (on the right) in 2D search space

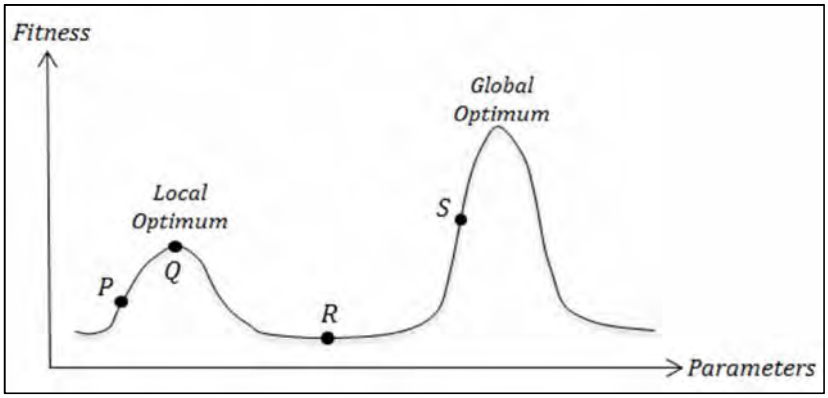


Figure 6.4: Candidate solutions ***P***, ***Q***, ***R*** and ***S*** at different regions of the fitness landscape with different exploitative/explorative requirements.

This is more rational than some population-wide global parameter settings, because the scenario around each candidate solution can be unique and very different from other solutions of the population. A candidate solution may need to explore through several peaks, valleys and plateaus of the fitness landscape. As a result, the explorative/exploitative requirement around each candidate solution can change suddenly and often in an unpredicted way. When an individual candidate solution tries to reach some local or global optimal point of the fitness landscape, such as the solution ***P*** or ***S*** in Fig. 6.4, it may need a sequence of increasingly small and exploitative perturbations to pinpoint the optimal point. After ***P*** has reached the locally optimal peak (such as the solution ***Q***) or has landed into a plain, flat plateau of the fitness landscape (like the solution ***R***), it may need large, explorative perturbations to break free from the imminent fitness stagnation and to reach the global optimum. This is why attaching each solution $\boldsymbol{x_i}$ with some control parameter (e.g., $\boldsymbol{x_i.q}$) that can dynamically adapt and customize the degree of explorations and exploitations around it can become very useful throughout the optimization process. Since $\boldsymbol{x_i}$ may have to go through several phases of explorations and exploitations, the automatic adaptation of its control parameters is often necessary to achieve an adaptively appropriate degree of explorations and exploitations around $\boldsymbol{x_i}$, based on its current explorative/exploitative requirements.

6.5 Evaluation of ABC-IX on Benchmark Functions

In this section, ABC-IX is evaluated using two different suites of benchmark problems on numerical function optimization. At first, we have used a standard benchmark suite consisting of 30 different functions, all of which have been well-studied and widely used in many recent studies on EAs and SIAs (e.g., [10], [11], [55]–[69], [146], [160]–[165], [196]). Then ABC-IX is evaluated on a more recent benchmark suite consisting of 25 more challenging and complex functions introduced in the special session on real parameter optimization at CEC2005 [76].

A brief overview of both the benchmark suites is presented in the previous section 2.17 (Tables 2.3–2.4). The reader can get more details on each of these benchmark functions in the Appendix A and in [76].

6.5.1 ABC-IX on the Standard Benchmark Functions

The standard benchmark suite (Table 2.3, section 2.17) consists of 30 benchmark functions, including both unimodal (f_1–f_9) and multimodal (f_{10}–f_{30}), separable (e.g., f_1, f_3, f_{15}, f_{16}) and non-separable (e.g., f_2, f_4, f_{14}, f_{18}), high (f_1–f_{18}) and low (f_{19}–f_{30}) dimensional functions. The results of ABC-IX and the basic ABC algorithm on all these functions are presented in Table 6.1. ABC-IX is executed with three different parameter settings — ABC-IX^1, ABC-IX^2 and ABC-IX^3 use the exponential cooling schedule parameter $\alpha = 0.97$, 0.99 and 0.998, respectively, while the value of t in (6.1) is set as 0.10, 0.20 and 0.30, respectively. Since slow (fast) cooling and high (low) perturbation rate indicate more explorations (exploitations), ABC-IX^3 is the most explorative variant, while ABC-IX^1 is the most exploitative one among these three ABC-IX variants. All these algorithms have three parameters in common — the population size *SN*, maximum number of function evaluations *FE* and *limit*. For high dimensional functions f_1–f_{18}, the parameters are set as $SN = 50$, $FE = 100000$ and $limit = 100$. For the low dimensional functions f_{19}–f_{30}, we have used $SN = 100$, $FE = 10000$ and $limit = 10 * D$. Each algorithm has made 50 independent runs on each of the functions and the mean and standard deviation of the best found solutions from the different runs are reported in Table 6.1. The following points summarize our observations on the results in Table 6.1.

- Out of the 18 high dimensional functions f_1–f_{18}, ABC-IX^1 performs better than ABC on as many as 15 functions, shows similar performance on one (f_8), while ABC manages to perform better on the remaining two functions only (f_7 and f_{18}). However, the performance of the more explorative variants — ABC-IX^2 and ABC-IX^3 are still better. Both of them outperform ABC on as many as 17 (out of 18) high dimensional functions and show similar results on the remaining one (f_8).
- Out of the 12 low dimensional functions f_{19}–f_{30}, the basic ABC algorithm performs slightly better only on one function (f_{26}), while the ABC-IX variants have always performed either equally well (five functions— f_{21}–f_{24}, f_{29}) or better (six functions) than ABC on the remaining functions.
- Among the three ABC-IX variants, the best performance is demonstrated by ABC-IX^2, which outperforms ABC on as many as 23 out of the 30 benchmark functions, shows similar performance on six and slightly worse performance on one function (f_{26}) only.

Table 6.1: Performance of the ABC-IX variants, compared to the basic ABC algorithm [11] on the standard benchmark functions. Results are averaged over 50 independent runs. The best results are marked with boldface font, if not identical with the results from other algorithms.

No	f_{min}	**ABC**		**ABC-IX**[1]		**ABC-IX**[2]		**ABC-IX**[3]	
		Mean Error	Std Dev	Mean Error	Std Dev	Mean Error	Std Dev	Mean Error	Std Dev
f_1	0	3.58e–11	8.14e–12	**8.25e–40**	**2.83e–40**	2.86e–38	6.58e–39	7.39e–36	8.84e–37
f_2	0	1.04e–14	5.33e–14	**6.77e–19**	**4.68e–19**	6.52e–18	2.65e–18	2.88e–16	1.46e–16
f_3	0	9.37e+00	3.22e+00	4.93e+00	1.55e+00	**1.17e–02**	**6.02e–03**	4.68e+00	1.59e+00
f_4	0	2.75e–10	2.49e–10	9.45e–33	3.41e–33	**1.86e–36**	**8.03e–37**	4.89e–35	5.90e–11
f_5	0	2.50e+00	9.25e–01	5.10e–01	9.27e–02	**7.64e–02**	**2.70e–02**	4.09e–01	1.06e–01
f_6	0	6.67e–01	5.74e–09	6.41e–04	9.91e–03	**9.22e–05**	**3.93e–05**	7.54e–04	3.05e–04
f_7	0	2.75e+00	8.08e–01	5.75e+00	2.47e+00	1.95e–01	8.55e–02	**1.09e–01**	**6.23e–02**
f_8	0	0	0	0	0	0	0	0	0
f_9	0	8.61e–13	7.07e–13	3.84e–60	9.74e–61	**1.64e–63**	**4.14e–64**	6.11e–62	2.53e–62
f_{10}	0	5.79e–15	2.48e–15	7.66e–36	1.02e–36	6.14e–41	8.54e–42	**1.78e–42**	**5.82e–43**
f_{11}	0	8.82e–09	2.33e–09	5.83e–11	1.16e–11	**8.51e–12**	**2.93e–12**	5.77e–11	1.79e–11
f_{12}	–12569.5	3.49e+02	1.18e+02	2.95e+02	1.46e+02	**1.56e+02**	**5.69e+01**	1.95e+02	6.21e+01
f_{13}	0	3.08e–06	3.96e–07	5.23e–12	8.68e–13	**3.82e–15**	**4.28e–16**	9.98e–13	2.87e–13
f_{14}	0	4.35e–08	8.47e–09	**2.39e–42**	**7.48e–43**	9.70e–40	8.30e–41	6.02e–38	2.06e–38
f_{15}	0	6.90e–06	2.15e–06	7.72e–08	3.48e–08	**8.21e–10**	**9.34e–11**	6.75e–09	9.12e–10
f_{16}	0	3.03e–02	8.69e–03	5.61e–05	2.69e–05	**6.74e–07**	**2.16e–07**	6.05e–04	8.88e–05
f_{17}	0	5.82e–08	9.42e–09	5.79e–12	9.43e–13	7.40e–11	2.06e–11	**1.66e–15**	**7.85e–16**
f_{18}	0	2.64e–03	8.53e–04	7.14e–02	3.42e–02	2.61e–03	1.96e–04	**4.98e–04**	**8.04e–05**
f_{19}	1	0.03	0.013	0.02	0.012	**0.01**	**0.003**	0.02	0.01
f_{20}	3.07e–04	7.60e–05	6.69e–06	4.01e–05	4.22e–06	**3.32e–05**	**2.05e–06**	4.25e–05	6.25e–06
f_{21}	–1.0316	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f_{22}	0.398	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f_{23}	–3.86	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f_{24}	–3.32	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f_{25}	–10.15	0.30	0.11	0.17	0.10	**0.08**	**0.024**	0.15	0.07
f_{26}	–10.40	**0.02**	**0.0025**	0.03	0.013	0.03	0.010	0.04	0.016
f_{27}	–10.54	0.12	0.045	0.06	0.023	0.04	0.016	**0.03**	**0.013**
f_{28}	0	8.05	2.89	1.44	0.62	0.97	0.28	**0.86**	**0.23**
f_{29}	–9.66	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f_{30}	–1.4	0.54	0.18	0.32	0.14	**0.23**	**0.09**	0.51	0.18

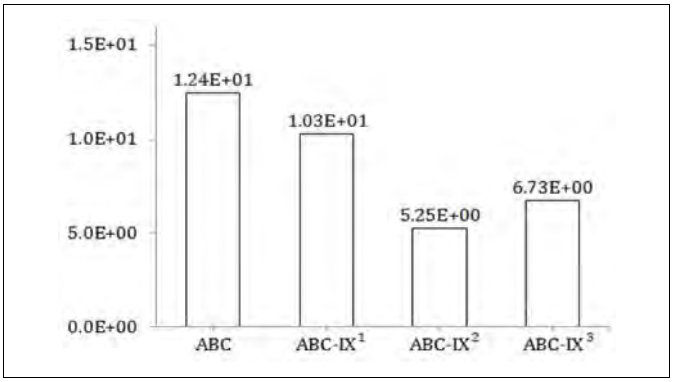


Figure 6.5: The mean absolute error values of ABC and the ABC-IX variants on the standard benchmark functions. The ABC-IX variants show smaller error values in comparison to ABC.

Table 6.2: Performance comparison of ABC-IX and ABC-SAM (chapter 5) on f_1–f_{18}. Results are averaged over 50 independent runs. Best result for each function is marked with boldface font. A '+' or '–' in the *t*-test indicates that ABC-IX is significantly better or worse, respectively than ABC with 95% level of confidence, while a '≈' means the difference is not statistically significant.

No	f_{min}	D	G	ABC-IX		ABC-SAM		*t*-Test (ABC-IX vs. ABC-SAM)
				Mean	Std. Dev.	Mean	Std. Dev.	
f_1	0	30	1000	**2.84e–38**	**1.03e–38**	4.18e–14	5.37e–15	+
		60	2000	**6.07e–30**	**8.55e–31**	6.09e–13	7.24e–14	
f_2	0	30	1000	**8.30e–15**	**3.14e–15**	2.47e–08	2.35e–09	+
		60	2000	**3.01e–12**	**4.74e–13**	5.06e–07	2.97e–07	
f_3	0	30	1000	**8.37e+00**	**2.64**	1.69e+01	1.43	+
		60	2000	**1.41e+01**	**5.15**	3.10e+01	5.12	
f_4	0	30	1000	**6.43e–31**	**7.37e–32**	3.95e–12	5.77e–13	+
		60	2000	**1.18e–26**	**4.95e–27**	7.54e–11	2.14e–11	
f_5	0	24	1000	**3.63e–01**	**6.18e–02**	9.24e–01	2.08e–01	+
f_6	0	30	1000	**5.86e–04**	**7.31e–05**	2.16e–03	6.37e–04	+
		60	2000	**9.88e–03**	**8.59e–03**	7.76e–02	1.63e–02	
f_7	0	30	1000	**8.96e+00**	**3.63e+00**	2.28e+01	3.75	+
		60	2000	**1.35e+00**	**8.93e–01**	4.96e+01	7.80	
f_8	0	30	1000	0	0	0	0	≈
		60	2000	0	0	0	0	
f_9	0	30	1000	**8.31e–52**	**2.19e–52**	3.66e–16	1.44e–17	+
		60	2000	**6.09e–41**	**2.43e–42**	4.76e–15	5.32e–16	
f_{10}	0	30	1000	**2.21e–35**	**1.60e–36**	1.26e–16	2.11e–17	+
		60	2000	**2.91e–32**	**6.44e–33**	8.55e–15	3.15e–16	
f_{11}	0	30	1000	**5.10e–11**	**9.78e–12**	4.60e–10	8.85e–11	+
		60	2000	**3.63e–10**	**1.35e–10**	6.80e–09	8.77e–10	
f_{12}	–12569.5	30	1000	**9.64e+01**	**1.52e+01**	1.53e+02	4.02e+01	+
	–25138.9	60	2000	**8.57e+02**	**1.35e+02**	1.33e+03	2.84e+02	
f_{13}	0	30	1000	**4.20e–15**	**5.23e–16**	9.26e–08	1.89e–08	+
		60	2000	**8.84e–14**	**1.40e–14**	2.07e–08	3.55e–08	
f_{14}	0	30	1000	**5.12e–33**	**1.55e–33**	8.36e–10	5.08e–11	+
		60	2000	**4.94e–32**	**9.98e–33**	1.56e–10	6.90e–11	

Table 6.2 (continued): Comparison of ABC-IX and ABC-SAM (chapter 5) on f_1–f_{18}.

No	f_{min}	D	G	ABC-IX		ABC-SAM		t-Test (ABC-IX vs. ABC-SAM)
				Mean	Std. Dev.	Mean	Std. Dev.	
f_{15}	0	30	1000	8.75e–08	2.38e–08	**2.22e–08**	**3.93e–09**	–
		60	2000	7.08e–08	1.33e–08	**1.17e–08**	**2.35e–09**	
f_{16}	0	30	1000	**5.24e–06**	**2.04e–06**	5.78e–04	6.31e–05	+
		60	2000	**2.72e–05**	**5.12e–06**	9.20e–03	4.03e–03	
f_{17}	0	30	1000	3.22e–10	7.88e–11	**9.78e–12**	**3.89e–12**	–
		60	2000	6.46e–10	1.59e–10	**1.32e–11**	**5.15e–11**	
f_{18}	0	30	1000	7.95e–02	2.14e–02	**3.06e–02**	**8.59e–03**	–
		60	2000	9.70e–02	2.05e–02	**5.11e–02**	**7.39e–03**	
Summary (t-Test)		+		14				
		–		3				
		≈		1				

- An overall evaluation of the algorithms can be made by comparing their mean absolute error values over all the functions. Fig. 6.5 shows that all the ABC-IX variants have smaller mean absolute error values compared to the basic ABC algorithm, with ABC-IX^2 having the smallest mean absolute error value.
- For almost all the functions, the ABC-IX variants have not only reached sufficiently close to the global minimum (i.e., mean error ≈ 0), but also accomplished this with very low standard deviations of their results. This indicates the high degree of accuracy, consistency and robustness of the ABC-IX variants for all these benchmark functions.

Table 6.2 compares the performance of ABC-IX with an improved ABC-variant — ABC-SAM (chapter 5). ABC-SAM tries to balance the degree of explorations and exploitations by maintaining a scaling factor SF_i for every bee agent $\boldsymbol{x_i}$ and by regularly adapting the value of SF_i using both explorative and exploitative distributions. The regular tuning of the SF_i values performs an automatic self-adaptation of the step size for perturbations, separately for every individual bee $\boldsymbol{x_i}$. Both ABC-SAM and ABC-IX are tested on the high dimensional functions f_1–f_{18}, with D=30 and 60. The common parameters are set as — SN=100, $limit$=100 (for 30D functions) or 200 (for 60D) and FE=100000 (for 30D) or 200000 (60D). The other parameters of ABC-IX are the same as ABC-IX^2, since it shows smaller mean absolute error value (Fig. 6.5) in comparison to ABC-IX^1 and ABC-IX^3. Table 6.2 demonstrates that ABC-IX often performs better than ABC-SAM (14 out of 18 functions), while ABC-SAM shows equal (f_8) or better performance only on a few occasions (i.e., f_{15}, f_{17}, f_{18}). Most of these performance differences are statistically significant, with at least 95% level of confidence, as shown by the t-test.

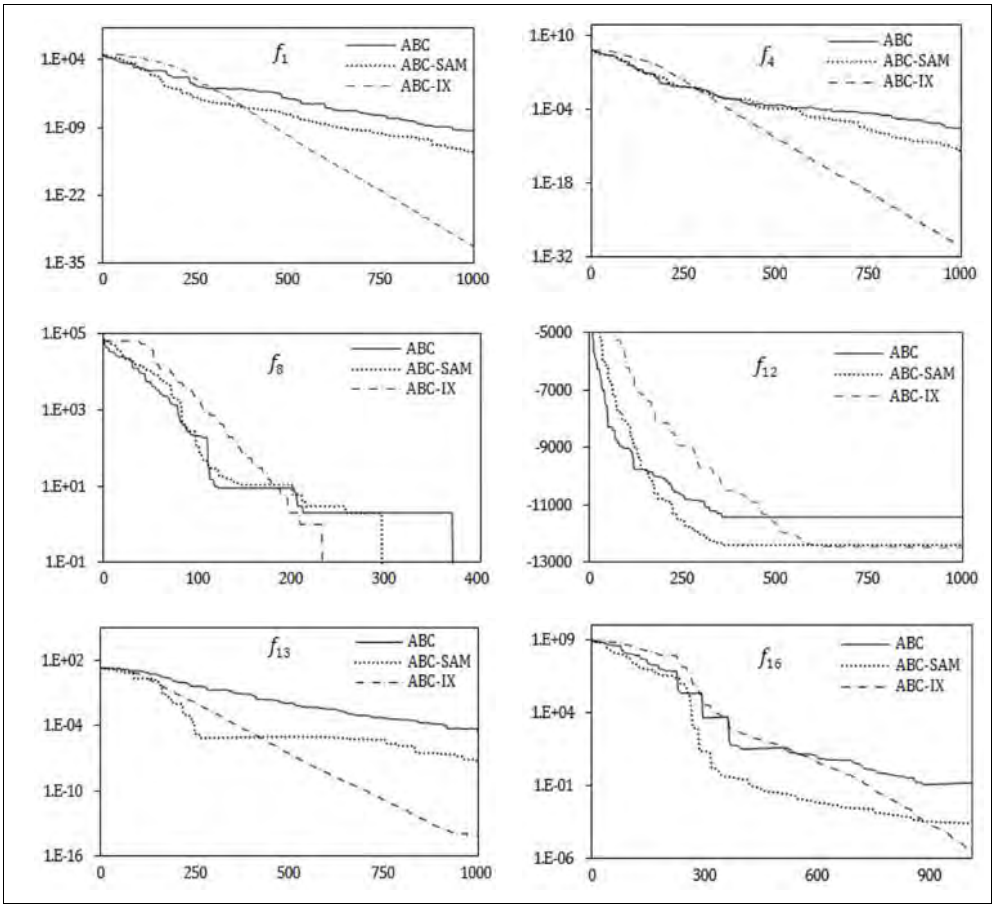


Figure 6.6: Convergence characteristics of ABC [11], ABC-SAM (chapter 5) and ABC-IX on three unimodal (f_1, f_4, f_8) and three multimodal (f_{12}, f_{13}, f_{16}) functions. The vertical axis is the function value, while horizontal axis is number of cycles elapsed.

Fig. 6.6 presents the convergence graphs of ABC, ABC-SAM and ABC-IX for six benchmark functions — including three unimodal (f_1, f_4, f_8) and three multimodal (f_{12}, f_{13}, f_{18}) functions with dimensionality D=30. Here, all three algorithms use the following parameter settings– SN=100, maximum cycle number MCN=1000 and $limit$=100. ABC-IX shows better convergence characteristics than both ABC and ABC-SAM for all the functions in Fig. 6.6. For example, consider the functions f_{13} and f_{16}, where both ABC and ABC-SAM converges to some local minima with fitness stagnation during the end stage of their execution. However, ABC-IX does not show any sign of fitness stagnation for both these functions, even after reaching very close proximity of the global minimum. For most of these functions (e.g., f_1, f_4, f_8, f_{12}), ABC-IX initially shows relatively slower convergence speed because of the high system temperature T that ensures high degree of explorations during the early stage of the run. As the algorithm progresses, the system temperature drops and the perturbation rate control parameters, separately for every candidate solution, gradually self-adapt towards more effective values, which gradually help ABC-IX to achieve better convergence speed (e.g., f_1, f_4, f_{13}, f_{16}) than both

ABC and ABC-SAM. Previously in Tables 6.1–6.2, we had considered the performance of ABC, ABC-SAM and ABC-IX to be similar for the function f_8. But Fig. 6.6 now reveals that ABC-IX actually reaches the global minimum of f_8 much earlier (i.e., fewer than 250 cycles) than both ABC (nearly 400 cycles) and ABC-SAM (around 300 cycles), which again indicates the effectiveness of the ABC-IX algorithm.

6.5.2 Comparison of ABC-IX with GA, DE, PSO and ABC

Now ABC-IX is compared with some basic and representative evolutionary and swarm intelligence algorithms, such as genetic algorithm (GA) [11], differential evolution (DE) [195], particle swarm optimization (PSO) [196] and the standard ABC [11] algorithm. The common parameters are set as — SN = 50 and FE = 500,000, while the other algorithm specific parameters are as follows.

GA Settings: We have used binary coded standard GA with fitness scaling, seeded selection, random selection, single point crossover, bit flip mutation and elitism. The crossover and mutation rates are set to 0.8 and 0.01, respectively. The stochastic uniform sampling technique is used as the selection operator. The generation gap (i.e., the portion of the population to be replaced in each generation) parameter is set to 0.9.

DE Settings: The standard DE has two parameters — the scaling factor F and the crossover rate CR, which are set as: F = 0.5 and CR = 0.9, as recommended in [195].

PSO Settings: The PSO [196] has three more control parameters — the cognitive component $¤_1$, social component $¤_2$, and the inertia weight w, which are set to 1.8, 1.8 and 0.6, respectively, as suggested in [196].

ABC Settings: In addition to the common parameters (SN and FE), the standard ABC algorithm has one more control parameter — *limit*, which is set to $SN * D$, as suggested in [11].

Table 6.3 compares the performance of ABC-IX with GA, PSO, DE and ABC on a total of 17 standard benchmark functions from f_1–f_{30} consisting of eight unimodal and nine multimodal functions, 13 (out of 17) of which have dimensionality D = 30, one has D = 24 and the remaining three (f_{28}–f_{30}) has D = 10. In the following brief points, we have summarized our observations on the results of Table 6.3.

GA vs. ABC-IX: ABC-IX performs better than GA on all (i.e., 17 out of 17) of the functions.

PSO vs. ABC-IX: The performance of ABC-IX is always either better (i.e., 13 out of 17 functions) or at least similar (on the remaining four functions) to PSO.

DE vs. ABC-IX: ABC-IX outperforms DE on 11 out of the 17 functions. On the remaining six functions, they show similar performance by successfully reaching the global minimum.

ABC vs. ABC-IX: On most of the functions (12 out of 17), ABC and ABC-IX show similar performance. However, for the remaining five functions, ABC-IX performs better than ABC.

Table 6.3: Comparison of ABC-IX with GA [11], DE [125], PSO [126] and the basic ABC [11] algorithm on 17 standard benchmark functions. The best results are marked with boldface font.

No.	D	f_{min}	Mean Error				
			GA	PSO	DE	ABC	ABC-IX
f_1	30	0	1.1e+03	**0**	**0**	**0**	**0**
f_2	30	0	11.02	**0**	**0**	**0**	**0**
f_4	30	0	7.4e+03	**0**	**0**	**0**	**0**
f_5	24	0	9.70	1.1e–04	**2.2e–07**	3.1e–03	7.54e–07
f_6	30	0	1.2e+03	0.6666	0.6666	**0**	**0**
f_7	30	0	1.9e+05	15.08	18.20	0.088	**6.24e–03**
f_8	30	0	1.2e+03	**0**	**0**	**0**	**0**
f_9	30	0	1.8e–01	1.2e–03	1.4e–03	3.00e–03	**6.78–52**
f_{10}	30	0	52.92	43.97	11.72	**0**	**0**
f_{12}	30	–12569.48	8.8e+02	5.7e+03	2.3e+03	**0**	**0**
f_{13}	30	0	14.67	0.16	**0**	**0**	**0**
f_{14}	30	0	10.63	0.017	0.0015	**0**	**0**
f_{17}	30	0	13.38	0.021	**0**	**0**	**0**
f_{18}	30	0	125.06	7.7e–03	2.2e–03	**0**	**0**
f_{28}	10	0	29.57	1364.45	781.55	8.23	**0.37**
f_{29}	10	–9.66015	0.16	5.65	0.069	**0**	**0**
f_{30}	10	–1.4	0.76	1.39	0.35	0.97	**0.23**

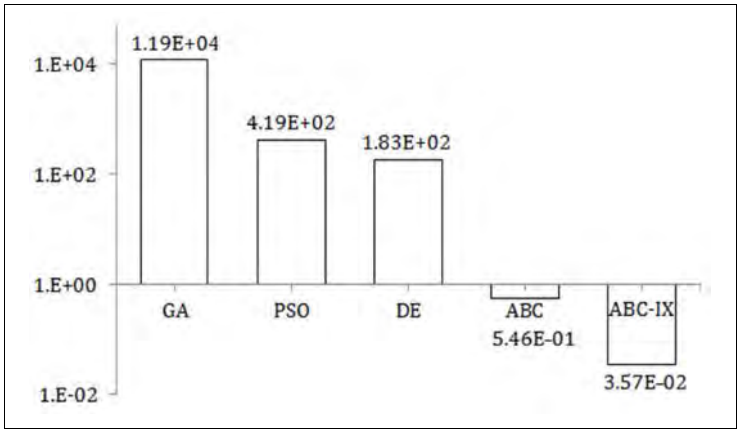


Figure 6.7: Comparison of ABC-IX with GA [11], DE [195], PSO [196] and the basic ABC [11] algorithm based on their mean absolute error values on the standard benchmark functions. ABC-IX shows the best performance, i.e., lowest mean absolute error value.

The overall performance of the algorithms can be compared by their mean absolute error values on these 17 benchmark functions. Fig. 6.7 shows that ABC-IX has the minimum mean absolute error, followed by the standard ABC algorithm. The other three algorithms (i.e., GA, PSO and DE) show much larger error values in comparison to ABC and ABC-IX.

6.5.3 Comparison of ABC-IX with Other ABC-Variants

In this section we compare ABC-IX with a number of recent improved variants of the ABC algorithm, each one of which alters the explorative and/or exploitative properties of the basic ABC algorithm, such as the cooperative ABC (CABC) [60], ABC with diversity strategy (DABC) [61], chaotic ABC (ChABC) [62], gbest-guided ABC (GABC) [64] and Hooke Jeeves ABC (HJABC) [65] algorithms. The first three variants (e.g., [60]–[62]) try to increase the degree of explorations to achieve better strength against local optima and premature convergence, while the last two variants (e.g., [64]–[65]) increase the degree of exploitations in order to achieve better convergence speed than the basic ABC algorithm.

First, ABC-IX is compared with CABC [60], which is a cooperative variant of the basic ABC algorithm. CABC has been introduced in two different versions — CABC_S and CABC_H. In order to perform more explorations, CABC_S decomposes the search space into multiple sub-spaces and employs different bee colonies to search and explore the different sub-spaces. The other variant, CABC_H tries to perform more exploitations than CABC_S by repeatedly alternating between the explorative CABC_S and exploitative ABC. For comparison, ABC-IX is re-implemented with the same settings — $SN = 40$, no. of function evaluations $FE = 100{,}000$ and $limit = SN * D$. Table 6.4 shows that ABC-IX significantly outperforms both the CABC variants on four out of the six benchmark functions, while the CABC variants perform better on the remaining two functions only. Thus, the overall performance of ABC-IX is better than the CABC variants. The reason may lie in the difficulties that the CABC variants have to face to properly decompose the search space into multiple sub-spaces and then, to effectively combine the partial solutions into a complete solution to the whole problem.

The next comparison is made between ABC-IX and DABC [61]. DABC tries to maintain sufficient amount of diversity among the candidate solutions in order to allow more search space explorations. DABC regularly measures the existing population diversity d and employs either an explorative or exploitative perturbation based on the value of d. As suggested in [61], ABC-IX is re-implemented with $SN = 20$, $MCN = 5000$ and $limit = 100$ to compare with DABC. Results presented in Table 6.5 show that ABC-IX performs better than DABC on two out of four functions (f_1 and f_{14}) and shows mixed performance on the remaining two (f_7 and f_{10}). For the 10D variants of f_7 and f_{10}, ABC-IX performs either equally well or better than DABC, while DABC performs better only on the larger 30D variants of f_7 and f_{10}. Thus, the overall performance of

ABC-IX is better than DABC. The reason may be that DABC completely relies on its estimated value of population diversity to choose between explorations and exploitations, while there is no accurate and universally accepted metric for diversity that can correctly estimate the maturity of an ongoing optimization process. Besides, DABC uses a fixed threshold diversity value (i.e., d_{low} in [61]) to switch between explorations and exploitations, which may cause repeated oscillations between conflicting explorative and exploitative perturbations and thus may have reduced its convergence speed, compromising the final solution quality.

Table 6.4: Comparison of ABC-IX with CABC_S and CABC_H [60]. The best performance for each function is marked with boldface font.

Function	**CABC_S**		**CABC_H**		**ABC-IX**	
	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
f_1	3.30e–19	2.00e-19	5.92e–18	3.56e–18	**9.41e–48**	**2.04e–48**
f_7	6.33e+00	7.68e+00	4.80e–01	8.55e–01	**3.21e–07**	**4.27e–08**
f_{10}	**0**	**0**	**0**	**0**	3.86e–52	8.72e–53
f_{12}	1.30e–04	5.21e–06	**1.27e–04**	**0**	1.86e–01	4.58e–02
f_{13}	1.83e–14	9.86e–15	8.35e–15	4.13e–15	**1.14e–16**	**3.03e–17**
f_{14}	4.42e–02	2.99e–02	7.96e–03	9.06e–03	**3.85e–51**	**1.66e–51**
+	4		4			
–	2		2			
≈	0		0			

Table 6.5: Performance comparison between ABC-IX and DABC [61]. Best results are marked with boldface font; if not both the algorithms produce identical results.

Function	***D***	**DABC**		**ABC-IX**	
		Mean	Std. Dev.	Mean	Std. Dev.
f_1	10	2.01e–17	5.63e–17	**0**	**0**
	30	2.01e–16	2.85e–17	**2.82e–63**	**6.45e–64**
f_7	10	2.73e–03	7.04e–03	**5.33e–04**	**1.91e–04**
	30	**1.55e–05**	**2.53e–06**	1.59e–03	4.05e–04
f_{10}	10	0	0	0	0
	30	**0**	**0**	4.29e–73	8.34e–74
f_{14}	10	0	0	0	0
	30	2.59e–16	1.22e–16	**7.60e–67**	**8.16e-68**
+		4			
–		2			
≈		2			

Next, ABC-IX is compared with the Chaotic ABC (ChABC) [62] algorithm. ChABC employs chaotic search behavior during perturbations to produce new food positions from the existing ones. Chaotic dynamics are produced by the logistic equations (eq. (4)–(7) in [62]) which provide a simple mechanism to escape from local minima and avoid premature convergence. For comparison, ABC-IX is executed for 5000 cycles with population size of 70 and *limit* = 200, as suggested in [62]. Results (Table 6.6) show that ABC-IX outperforms ChABC on as many as five out of the six functions, while ChABC performs better on the remaining one function (f_{10}) only. The reason may be that ChABC employs the same chaotic strategy uniformly for all the candidate solutions across the population, without considering their individual exploitative/explorative requirements, while ABC-IX considers and customizes the degree of explorations and exploitations separately for every candidate solution, which should be more effective for complex optimization tasks.

Next, ABC-IX is compared with GABC [64], which is an exploitative ABC-variant that tries to improve the convergence speed of ABC by using the information of the global best solution found so far during the perturbation operations. For a fair comparison, ABC-IX is executed with the same settings [64] and results are presented in Table 6.7. In [64], GABC is tested with several values of its parameter C, but the best results are always observed with C = 1.0 or 1.5, so Table 6.7 includes both the results. Results show that ABC-IX outperforms GABC on four out of the five functions, while GABC performs better on the remaining one (f_7) only. The reason may be that the perturbation operation of GABC becomes too exploitative because of pushing the candidate solutions explicitly towards the best solution. Increased exploitations, at the cost of reduced explorations, may improve the solution quality for unimodal and low dimensional

Table 6.6: Comparison between ABC-IX and ChABC [62] based on the final solution quality. The best result for each function is marked with boldface font.

Function	***D***	**ChABC**		**ABC-IX**	
		Mean	Std. Dev.	Mean	Std. Dev.
f_1	30	2.99e–16	3.54e–17	**3.28e–97**	**7.82e–98**
f_7	30	6.33e–02	8.96e–02	**8.15e–03**	**2.34e–03**
f_{10}	30	**0**	**0**	4.66e–105	1.40e–105
f_{12}	30	**3.81e–04**	**2.07e–04**	2.76e–02	7.35e–03
f_{13}	30	2.93e–14	2.99e–15	**7.08e–26**	**2.23e–26**
f_{14}	30	2.70e–16	6.20e–17	**3.60e–49**	**7.84e–50**
+		4			
–		2			
≈		0			

functions, such as the f_7 in Table 6.7, which has dimensionality D = 2 or 3, but is likely to fail for more complex multimodal and high dimensional functions, as observed in the other four high dimensional functions in Table 6.7 with D = 30 and 60.

Table 6.7: Comparison between ABC-IX and GABC [64] based on the final solution quality. The best result for each function is marked with boldface font.

Function	*D*	GABC (*C*=1.0)		GABC (*C*=1.5)		ABC-IX	
		Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
f_1	30	4.31e–16	7.49e–17	4.17e–16	7.36e–17	**2.75e–107**	**8.54e–108**
	60	1.43e–15	1.43e–16	1.43e–15	1.37e–16	**3.26e–42**	**7.25e–43**
f_7	2	3.93e–04	4.45e–04	**1.68e–04**	**1.45e–04**	2.93e–03	9.31e–04
	3	**2.63e–03**	**2.11e–03**	2.65e–03	2.22e–03	6.23e–03	1.08e–03
f_{10}	30	9.47e–15	2.15e–14	1.32e–14	2.44e–14	**9.20e–107**	**3.45e–107**
	60	4.16e–13	1.77e–13	3.52e–13	1.24e–13	**2.11e–48**	**5.66e–49**
f_{13}	30	3.31e–14	2.90e–15	3.21e–14	3.25e–15	**4.14e–15**	**8.12e–16**
	60	1.04e–13	1.07e–14	1.00e–13	6.08e–15	**5.19e–31**	**2.20e–31**
f_{14}	30	8.88e–17	8.45e–17	2.96e–17	4.99e–17	**5.68e–105**	**2.00e–105**
	60	9.47e–16	7.84e–16	7.54e–17	4.12e–16	**1.25e–52**	**4.16e–53**
+		4		4			
–		1		1			
≈		0		0			

Table 6.8: Comparison between ABC-IX and HJABC [65] based on their convergence speed. The best result for each function is marked with boldface font.

Function	*D*	Number of function evaluations	
		HJABC	ABC-IX
f_1	30	18322	**14919**
f_2	30	**12509**	18656
f_3	30	**120315**	–
f_4	30	43939	**37810**
f_7	30	**102718**	–
f_8	30	17755	**15205**
f_9	30	–	**11415**
f_{10}	30	**15376**	22187
f_{13}	30	54497	**45337**
f_{14}	30	56855	**35640**
f_{15}	30	99686	**86636**
+		7	
–		4	
≈		0	

Next, ABC-IX is compared with HJABC [65], which is a hybrid ABC-variant that intensifies the degree of exploitations by hybridizing ABC with an efficient local search technique (i.e., Hooke Jeeves pattern search). Table 6.8 compares ABC-IX with HJABC based on the number of function evaluations (*NFE*) required to achieve a predefined level of accuracy. Both ABC-IX and HJABC are run with $SN = 25$ and $limit = SN * D$, until either *NFE* reaches a predefined maximum value (NFE_{max}) or the condition that $\left| f^* - \tilde{f} \right| \leq \varepsilon_1$ is satisfied, where f^* is the global minimum, $\tilde{f}$ is the best function value found so far by the algorithm, $\varepsilon_1 = 10^{-8}$ and $NFE_{max} = 300000$, as suggested in [65]. For seven out of the eleven functions in Table 6.8, ABC-IX performs better than HJABC, by showing a faster convergence speed, while HJABC performs better on the remaining four. However, ABC-IX can't achieve the predefined level of accuracy within NFE_{max} function evaluations for two functions (f_3 and f_7), while HJABC fails to do so only for one function (f_9). In short, the overall performance of ABC-IX is quite comparable to HJABC. The reason that HJABC often requires larger number of function evaluations, even after using the efficient Hooke Jeeves local searcher [65], may be that — HJABC regularly tries to find an appropriate search direction by exploring along the axis directions only, exploring just one variable at a time, which is usually not suitable for the non-separable problems. Hence, this may produce improper search directions that might have reduced the convergence speed of HJABC.

6.5.4 ABC-IX on CEC2005 Benchmark Functions

In addition to the standard benchmark functions, ABC-IX is also evaluated using the recently introduced CEC2005 benchmark suite [76]. This new suite was introduced in the special session on real parameter optimization at the 2005 IEEE Congress on Evolutionary Computation (CEC'05) at Edinburgh, UK. We presented a brief overview on the CEC2005 suite functions in the previous section 2.17 (Table 2.4). Further details on each of these functions are available in the Appendix A and also in [76].

Using the CEC2005 benchmark suite, we now evaluate and compare ABC-IX with a few more evolutionary and swarm intelligence algorithms, such as the self-adaptive differential evolution (SADE) [230], dynamic multi-swarm PSO with local search (DMS-PSO) [231], PSO with recombination by dynamic linkage discovery (PSO-RDL) [232] and the basic ABC algorithm [11]. DMS-PSO [231] is an improved PSO variant that divides the candidate solutions into several small, dynamic swarms. The swarms are regrouped frequently by using various regrouping schedules and information is exchanged among the swarms. Also, a quasi-Newton method is used to improve its local search ability. The next algorithm — PSO-RDL [232] employs a special recombination operator that dynamically discovers the linkages among the variables and tries to identify the important building blocks that are present within good quality solutions,

Table 6.9: Comparison of ABC-IX with DMS-PSO [231], PSO-RDL [232], SADE [230] and standard ABC [11] algorithm on the non-composite functions F1–F14 of the CEC2005 benchmark suite [76]. The best result for each function is marked with boldface font.

Function	DMS-PSO	PSO-RDL	SADE	ABC	ABC-IX
	Mean Error	Mean Error	Mean Error	Mean Error	Mean Error
F1	**0.00e+00**	2.50e–14	**0.00e+00**	4.89e–17	5.92e–26
F2	1.30e–13	1.77e–13	1.05e–13	4.81e–14	**8.73e–18**
F3	**7.01e–09**	9.6e–02	1.67e–05	2.50e+03	6.22e+00
F4	1.8e–03	2.57e–07	1.42e–05	1.50e–16	**2.82e–18**
F5	1.16e–06	**2.09e–07**	1.23e–02	5.82e+01	1.04e–03
F6	6.89e–08	9.57e–01	**1.20e–08**	3.31e+00	9.28e–03
F7	4.52e–02	5.73e–02	**1.99e–02**	2.52e–01	5.25e–01
F8	2.00e+01	2.00e+01	2.00e+01	2.03e+01	2.00e+01
F9	**0.00e+00**	1.25e+01	**0.00e+00**	4.87e–17	3.82e–20
F10	3.62e+00	3.86e+01	4.97e+00	2.22e+01	**2.11e+00**
F11	4.62e+00	5.58e+00	4.89e+00	5.46e+00	**3.17e+00**
F12	2.40e+00	1.31e+02	**4.50e–07**	9.85e+01	6.99e–02
F13	3.69e–01	8.87e–01	2.20e–01	**2.96e–02**	4.72e–02
F14	2.36e+00	3.78e+00	2.92e+00	3.41e+00	**2.05e+00**

Table 6.10: Comparison of ABC-IX with DMS-PSO [231], PSO-RDL [232], SADE [230] and standard ABC [11] algorithm on the hybrid composition functions F15–F25 of the CEC2005 benchmark suite [76]. The best result for each function is marked with boldface font.

Function	DMS-PSO	PSO-RDL	SADE	ABC	ABC-IX
	Mean Error	Mean Error	Mean Error	Mean Error	Mean Error
F15	4.85e+00	2.71e+02	3.20e+01	1.53e+01	**3.17e+00**
F16	9.48e+01	2.22e+02	1.01e+02	1.75e+02	**7.37e+01**
F17	**1.10e+02**	2.22e+02	1.14e+02	1.96e+02	2.02e+02
F18	7.61e+02	1.02e+03	7.19e+02	4.46e+02	**4.12e+02**
F19	7.14e+02	9.85e+02	7.05e+02	4.51e+02	**3.92e+02**
F20	8.22e+02	9.59e+02	7.13e+02	**4.38e+02**	4.49e+02
F21	5.36e+02	9.94e+02	**4.64e+02**	4.87e+02	4.74e+02
F22	**6.92e+02**	8.87e+02	7.32e+02	8.59e+02	6.98e+02
F23	7.30e+02	1.08e+03	6.64e+02	5.98e+02	**5.57e+02**
F24	2.24e+02	7.20e+02	**2.00e+02**	2.02e+02	**2.00e+02**
F25	3.66e+02	1.76e+03	3.76e+02	**3.38e+02**	3.53e+02

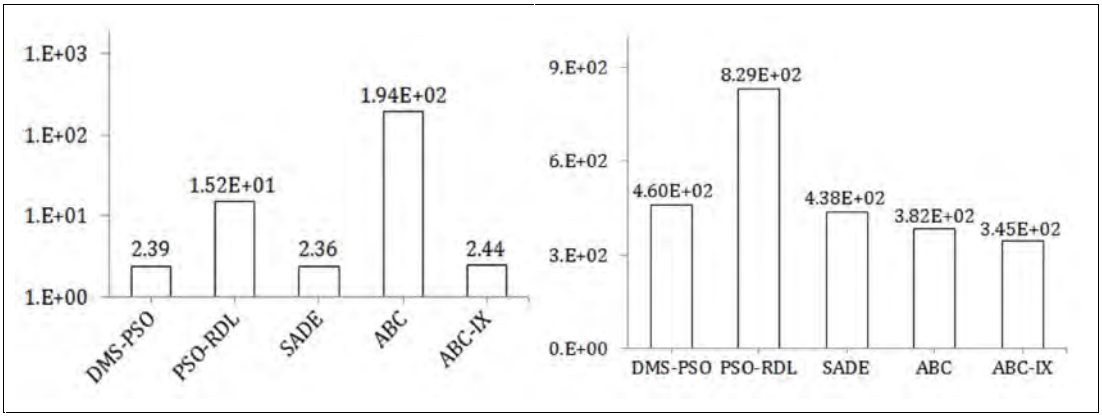


Figure 6.8: Comparison of ABC-IX with DMS-PSO [231], PSO-RDL [232], SADE [230] and standard ABC [11] algorithm, based on their mean absolute errors values on the CEC2005 benchmark functions F1–F14 (on the left) and hybrid composite functions F15–F25 (right).

which are then effectively used to produce better trial solutions from the existing ones. SADE [230] is an improved DE variant that employs a learning strategy to gradually self-adapt the values of some parameters of the standard DE algorithm. All these algorithms have been compared on the CEC2005 suite functions with dimensionality $D=10$ and the number of function evaluation set to 100,000. Both ABC and ABC-IX have colony size = 20 and *limit* = 200. The remaining parameters of ABC-IX are α and *t*, which are set to 0.99 and 0.10, respectively. The results of SADE [230], DMS-PSO [231] and PSO-RDL [232] are obtained directly from the corresponding papers. Tables 6.9–6.10 present the mean error over 25 independent runs on each function by all the algorithms. The results can be summarized by the following few points.

- Out of the 14 functions in Table 6.9 (i.e., non-composite functions F1–F14), ABC-IX becomes the best performer on five functions (F2, F4, F10, F11 and F14), outperforming all other algorithms on these functions. Another algorithm — SADE also becomes the best performer on five functions, while DMS-PSO and PSO-RDL show the best performance only on three and one function, respectively.
- For the more complex hybrid composition functions F15–F25 (Table 6.10), ABC-IX show the best performance on as many as six (out of 11) functions, while DMS-PSO, SADE and ABC show the best performance only on one, two and two functions, respectively.
- To compare ABC-IX with the basic ABC algorithm, it is remarkable that ABC-IX has improved the results of ABC for as many as 20 out of the 25 functions. This clearly indicates the effectiveness of the more explorative and self-adaptive design of ABC-IX, especially for these more complex optimization tasks.
- As Fig. 6.8 demonstrates, ABC-IX show the minimum mean absolute error on the hybrid composite functions F15–F25, outperforming all other algorithms in comparison. For

the non-hybrid functions F1–F14, the error value of ABC-IX is much smaller than both of ABC and PSO-RDL, and very similar to (actually, slightly worse than) the error of DMS-PSO and SADE.

- Summarizing all the points above, we can conclude that the overall performance of ABC-IX is at least comparable to and often better that all of its counterparts in the comparison on the CEC2005 functions.

6.6 Conclusion and Suggestion for Further Study

This chapter introduces ABC-IX — an improved variant of the standard ABC algorithm [11], [75] that attempts to improve its explorative capacity by incorporating two novel techniques — the simulated annealing based probabilistic selection scheme and the self-adaptive perturbation rate for every candidate solution of the population. While the more explorative selection scheme contributes towards improving the explorative characteristics of ABC, the self-adaptive perturbation scheme tries to customize and balance the degree of explorations and exploitation around each candidate solution of the population. Empirical results show that ABC-IX performs better than standard ABC and several other improved variants of the ABC algorithm.

There are several future research directions suggested by this study. Firstly, currently ABC-IX uses a simple exponential cooling schedule for the system temperature T. A more sophisticated cooling strategy, e.g., a strategy parameterized by the population diversity or current explorative/exploitative requirements of the population or some other metric that can estimate the current maturity of the ongoing optimization process, may be more effective to balance between exploitations and explorations. Secondly, ABC-IX concentrates more on improving the explorative capacity of the algorithm. Putting some more emphasis to control the exploitations, especially around the best candidate solutions found so far, may further improve the results. Thirdly, the quality of the final solution might be improved further by using an efficient and exploitative local searcher after the execution of ABC-IX is over. Fourthly, the techniques incorporated by ABC-IX are generic enough; so they can be easily incorporated or hybridized with other suitable evolutionary, swarm intelligence and/or machine learning techniques. This may lead to developing new algorithms with improved convergence speed, final solution quality and better resilience against fitness stagnation and premature convergence. Finally, ABC-IX has been applied mainly on continuous optimization problems. It would be interesting to study how well ABC-IX performs on many other existing problems, especially the discrete and real world ones.

Chapter 7

Artificial Bee Colony Algorithm with Adaptive Explorations and Exploitations

7.1 Introduction

This chapter introduces an improved ABC variant — the ABC with Adaptive eXplorations and eXploitations (ABC-AX^2). ABC-AX^2 augments each bee (i.e., candidate solution) of the bee colony with three additional control parameters that control the perturbation rate, magnitude of perturbations and proportion of explorative and exploitative perturbations. Together, all the control parameters try to adapt the degree of global explorations and local exploitations around each candidate solution by affecting how new trial solutions are produced from the existing ones. The control parameters are automatically adapted at the individual solution level, separately for each candidate solution, based on their successes and failures to produce better trial solutions from the existing ones. We have evaluated ABC-AX^2 on a number of benchmark problems on continuous optimization and compared the results with the basic ABC algorithm and several other recently introduced improved variants of the basic ABC algorithm.

7.2 Organization of the Chapter

The rest of this chapter is organized as follows. Section 7.3 explains how ABC-AX^2 is significantly different from most other existing ABC-variants. Section 7.4 introduces the proposed algorithm — ABC-AX^2, explains all its components, control parameters, its perturbation procedure and the adaptive and self-adaptive schemes that guide the control parameters, each topic with sufficient details and necessary pseudocode. The next section 7.5 presents the performance evaluation of ABC-AX^2 on two different benchmark suites, specifies the parameter

settings of ABC-AX^2 and the other algorithms in comparison and compares their results with a few comments and analysis on the results. Finally, section 7.6 presents a short discussion on the results and concludes the chapter by leaving a few suggestions on future research directions.

7.3 Differences of ABC-AX^2 with Other Existing Works

There exist a number of recent works (e.g., [59]–[68]) that try to alter the explorative and/or exploitative properties of the basic ABC algorithm, as explained and briefly reviewed in the previous section 6.3. However, most of them try to improve either the exploitative (e.g., [64]–[68]) or the explorative (e.g., [59]–[63]) characteristics of the basic ABC algorithm. The exploitative improvements are usually based on intensifying the search around the best solution(s) found so far (e.g., [64], [65], [66]) and/or hybridizing efficient local search operators with the basic ABC algorithm (e.g., [65], [66], [67]), while the explorative improvements can be based on more population diversity (e.g., [60], [61]) and/or more explorative initialization, selection and/or perturbation operations ([59], [62], [63]). But only a few (e.g., [59], [61], [64]) of these algorithms have actually made efforts, to some extent, to balance between their explorative and exploitative improvements. But most of them employ some fixed, rather than adaptive, strategy to balance between explorations and exploitations. For example, DABC [61] uses a fixed diversity threshold value d_{low} and tries to maintain the population diversity above d_{low} by picking either explorative or exploitative perturbation. MABC [55] sets its control parameter $P=0.7$ for all the test problems, while GABC [64] experiments with several fixed values of its control parameter C within the range [0.5, 4.0]. Although ABC-SAM (chapter 5, also Ref. [59]) employs an adaptive technique to adjust the step size for perturbations, it still uses a fixed, exploitative perturbation rate and tries to impose an equal, rather than adaptive, proportion of explorations and exploitations. Besides, none of these algorithms (e.g., [60]–[68]), except ABC-SAM (chapter 5), considers the explorative/exploitative requirements of the individual candidate solutions separately; rather they employ some population-wide global strategy, identically across all the solutions, irrespective of their (possibly) different explorative/exploitative requirements. However, the proposed ABC-variant — ABC-AX^2 carefully tries to consider all these issues.

ABC-AX^2 is significantly different from most other existing ABC-variants (e.g., [59]–[68]) in two important ways. Firstly, ABC-AX^2 considers explorations and exploitations as complementary, rather than conflicting, operations and emphasizes an adaptive balance between explorations and exploitations, while most existing works are biased either towards more explorations (e.g., [59]–[63]) or towards more exploitations (e.g., [64]–[68]). ABC-AX^2 introduces three control parameters within each candidate solution that affect the degree and

proportion of explorations and exploitations during its perturbations. The control parameters are adapted gradually, cycle by cycle, using adaptive and self-adaptive techniques that emphasize both explorations and exploitations in order to reach an adaptive balance between them. Secondly, ABC-AX^2 customizes the degree of explorations and exploitations at the individual solution level, separately for each candidate solution of the population, while most other existing ABC-variants employ some explorative and/or exploitative strategy at the entire population level, identically for all the candidate solutions across the population. Customization of explorations and exploitations at the individual solution level is more rational that some population-wide global strategy (e.g., [60]–[68]), because each candidate solution is usually at a different region of the search space with dynamically changing explorative/exploitative requirements. This is why the adaptation of the control parameters, separately for each candidate solution, as adopted by ABC-AX^2, is more suitable to automatically cope with the dynamically evolving exploitative/explorative requirements of the problem.

7.4 The Proposed Algorithms — ABC-AX^2

ABC-AX^2 tries to improve over the basic ABC algorithm by adapting and customizing the degree of explorations and exploitations at the individual solution level, i.e., separately for every candidate solution. ABC-AX^2 includes three control parameters — p_i, q_i and $\boldsymbol{\eta_i}$ within each solution $\boldsymbol{x_i}$. The control parameter p_i controls the proportion of explorative and exploitative perturbations; q_i controls the perturbation rate to produce $\boldsymbol{v_i}$ from $\boldsymbol{x_i}$; and, $\boldsymbol{\eta_i}=[\eta_{i1}, \eta_{i2}, \ldots, \eta_{iD}]^T$ is a vector with D components, each one (say, η_{ij}) of which controls the distribution of the scaling factor values (i.e., φ_{ij} values in (2.6)) during perturbations along the corresponding (i.e., j-th) dimension. Each control parameter is gradually adapted to achieve higher rate of 'successful' perturbations. A perturbation is considered 'successful' only if the new trial solution $\boldsymbol{v_i}$ has higher fitness value than the original solution $\boldsymbol{x_i}$. In the following paragraphs, we explain the role of each control parameter, how it affects explorations and exploitations in perturbations and how it is gradually adapted by ABC-AX^2.

A. Control parameter $\boldsymbol{p_i}$ for adaptive proportion of explorations and exploitations: The basic ABC algorithm uses the single perturbation scheme (2.6), with no attempt to differentiate between explorative and exploitative perturbations. In contrast, ABC-AX^2 employs two different perturbation schemes — one for explorations, the other for exploitations. Both the perturbation schemes are based on the same expression (2.6), but they differ in how $\boldsymbol{x_i}$ selects

Algorithm 7.1: The ABC Algorithmwith Adaptive Explorations and Exploitations (ABC-AX²)

1: Initialize a population of *SN* food source positions (candidate solutions) $\boldsymbol{x_i}$, for i = 1, 2, …, *SN*. Each $\boldsymbol{x_i}$ is a vector of *D* parameters: $\boldsymbol{x_i} = [x_{i1}, x_{i2}, \ldots, x_{iD}]^T$

2: Evaluate the fitness of each food source position using (2.3).

3: **repeat**

4*: For each employed bee, perturb its current food source position $\boldsymbol{x_i}$ to produce a new food source position $\boldsymbol{v_i}$ by using the perturbation operator of ABC-AX² (i.e., the pseudocode in Fig. 7.5).

5: Evaluate each new solution $\boldsymbol{v_i}$ by using (2.3). If $\boldsymbol{v_i}$ has higher fitness than $\boldsymbol{x_i}$, then accept $\boldsymbol{v_i}$ to replace $\boldsymbol{x_i}$. Otherwise, discard $\boldsymbol{v_i}$.

6. Calculate the probability value w_i associated with each food source position $\boldsymbol{x_i}$ following the same way the probability value p_i was computed in (2.7).

7: For each onlooker bee, assign it to a food source position $\boldsymbol{x_i}$, proportionally based on the probability w_i.

8*: For each onlooker bee, perturb the food source position of its employed bee $\boldsymbol{x_i}$ to produce a new position $\boldsymbol{v_i}$ by using the perturbation operator of ABC-AX² (i.e., the pseudocode in Fig. 7.5).

9: Evaluate each new solution $\boldsymbol{v_i}$ using (2.3). If $\boldsymbol{v_i}$ has higher fitness value than $\boldsymbol{x_i}$, then accept $\boldsymbol{v_i}$ to replace $\boldsymbol{x_i}$. Otherwise, discard $\boldsymbol{v_i}$.

10: If a food source has not improved during the last *limit* cycles, then abandon it and replace it with a new randomly placed scout bee with its food source $\boldsymbol{x_i}$ produced by (2.8).

11: Memorize the best food source position found so far.

12*: Set cycle counter $C = C + 1$. Also, for each food position $\boldsymbol{x_i}$, update the values of the control parameters p_i, q_i and $\boldsymbol{\eta}_i$ by using (7.3)–(7.6).

13: **until** *C* = Maximum cycle number (*MCN*).

14: **return** the best food source position (i.e., candidate solution) found so far.

Figure 7.1: Pseudocode for the ABC-AX² algorithm. Here, the '*' symbols mark the steps where ABC-AX² differ from the basic ABC algorithm in Fig. 2.9.

its supporting candidate solution $\boldsymbol{x_k}$ in (2.6). For explorative perturbations, $\boldsymbol{x_k}$ is picked by three-tier explorative tournament selection (3T-ETS), while the exploitative perturbations use two-tier exploitative tournament selection (2T-ETS) procedure. These selection procedures are introduced in Fig. 7.2 and Fig. 7.3.

Explorative perturbation:

$$v_{ij} = x_{ij} + \varphi_{ij}\left(x_{kj} - x_{ij}\right), \text{ where } \boldsymbol{x_k} \sim \text{Three\_Tier\_Explorative\_Tournament\_Selection}(\boldsymbol{x_i}) \quad (7.1)$$

Exploitative perturbation:

$$v_{ij} = x_{ij} + \varphi_{ij}\left(x_{kj} - x_{ij}\right), \text{ where } \boldsymbol{x_k} \sim \text{Two\_Tier\_Exploitative\_Tournament\_Selection}(\boldsymbol{x_i}) \quad (7.2)$$

Here, $j \in \{1, 2, \ldots, D\}$ and $k \in \{1, 2, \ldots, SN\}$ are randomly picked indices, φ_{ij} is uniform random value produced from [-1, 1]. The explorative 3T-ETS scheme tries to pick a candidate solution $\boldsymbol{x_k}$ that is not only fit, but also dissimilar (from the current solution $\boldsymbol{x_i}$) and diverse (from the other solutions of the population). Dissimilarity of $\boldsymbol{x_k}$ from $\boldsymbol{x_i}$ is measured as their Euclidean distance (ED), while diversity of $\boldsymbol{x_k}$ is estimated as its ED from the centroid of population of solutions. High dissimilarity of $\boldsymbol{x_k}$ from $\boldsymbol{x_i}$ ensures a large $|x_{kj} - x_{ij}|$ in (7.1) to make a large, explorative perturbation on $\boldsymbol{x_i}$, while the high diversity of $\boldsymbol{x_k}$ tries to pull $\boldsymbol{x_i}$ away from the population centroid to promote more diversity and to avoid being trapped around local optima. In contrast,

the exploitative 2T-ETS scheme tries to pick an $\boldsymbol{x_k}$ that is both fit and has high degree of similarity to $\boldsymbol{x_i}$. This tries to ensure a small $|x_{kj} - x_{ij}|$ in (7.2) to make small, exploitative steps towards the better regions of the search space.

But how does ABC-AX^2 decide on whether to perform explorative or exploitative perturbation on $\boldsymbol{x_i}$? This is done probabilistically — the current values of p_i and $1–p_i$ denote the probability of exploitative and explorative perturbations on $\boldsymbol{x_i}$, respectively. The value of p_i is automatically adapted using the incremental learning experience of $\boldsymbol{x_i}$, which includes the number of successes and failures by explorative and exploitative perturbations on $\boldsymbol{x_i}$ during the last τ_1 cycles (learning period). Initially, p_i is set to 0.5 for every solution $\boldsymbol{x_i}$, which makes exploitative and explorative perturbations equally desired. After the initial learning period of τ_1 cycles, ABC-AX^2 starts adjusting the p_i value for each $\boldsymbol{x_i}$. To do this, ABC-AX^2 keeps record of the number of successes and failures by exploitative and explorative perturbations on $\boldsymbol{x_i}$ over the last τ_1 cycles. Suppose ns_{ET} and nf_{ET} (ns_{ER} and nf_{ER}) are the number of successes and failures, respectively by the exploitative (explorative) perturbations on $\boldsymbol{x_i}$ during the last τ_1 cycles. Then, success ratios of exploitative perturbation (SR_{ET}) and explorative perturbation (SR_{ER}) on $\boldsymbol{x_i}$ are computed as: $SR_{ET} = \frac{ns_{ET}}{ns_{ET} + nf_{ET}}$ and $SR_{ER} = \frac{ns_{ER}}{ns_{ER} + nf_{ER}}$. Now, the adjusted probability of exploitative perturbation on $\boldsymbol{x_i}$ (i.e., the adjusted value of p_i) is computed using (7.3), which also ensures $0.1 \le p_i \le 0.9$ to avoid the complete domination by either mode of perturbations.

$$p_i = min\left(0.9,\ max\left(0.1, \frac{SR_{ET}}{SR_{ER} + SR_{ET}}\right)\right) \tag{7.3}$$

Once the value of p_i for each candidate solution $\boldsymbol{x_i}$ is computed by ABC-AX^2 using (7.3), it is kept unchanged for the next τ_2 cycles ($\tau_2 < \tau_1$), which allows some time for the adjusted value of p_i to produce both successes and failures by each type of perturbation. ABC-AX^2 regularly adjusts the value of p_i for each candidate solution $\boldsymbol{x_i}$ using (7.3), periodically after each τ_2 cycles, using the recorded values of number of successes and failures by each type of perturbation on $\boldsymbol{x_i}$ over the last τ_1 cycles. After some initial experiments, we have set τ_1=50 and τ_2=10.

B. Control parameter q_i for self-adaptive perturbation rate: The basic ABC algorithm perturbs only a single, random parameter of $\boldsymbol{x_i}$ using (2.6). This usually produces the trial solution $\boldsymbol{v_i}$ in the neighbourhood of the original solution $\boldsymbol{x_i}$, which is exploitative. Perturbing a single parameter allows search along a single dimension at a time. This may work well for separable problems, but not suitable for non-separable problems where the parameters are not independent. Fig. 7.4 shows an example using a 2D search space. Allowing perturbation of both the parameters (i.e., x_{i1} and x_{i2}) can produce $\boldsymbol{v_i}$ along any possible direction from $\boldsymbol{x_i}$. This is more efficient than perturbing either x_{i1} or x_{i2}, one at a time, as is done by the basic ABC algorithm that

```
Algorithm 7.2: Three_Tier_Explorative_Tournament_Selection(x_i)
  global P: Population of candidate solutions
  global t1, t2, t3: Tournament sizes for the dissimilarity, diversity
    and fitness based tournaments, respectively

begin
    return Tier3_Dissimilarity_Tournament(x_i)
end

procedure Tier3_Dissimilarity_Tournament(x_i)
    best ← Tier2_Diversity_Tournament()
    for i from 2 to t3 do
        next ← Tier2_Diversity_Tournament()
        if distance(next, x_i) > distance(best, x_i) then
            best ← next
    return best

procedure Tier2_Diversity_Tournament()
    best ← Tier1_Fitness_Tournament()
    for i from 2 to t2 do
        next ← Tier1_Fitness_Tournament()
        if diversity(next) > diversity(best) then
            best ← next
    return best

procedure Tier1_Fitness_Tournament()
   best ← a solution picked at random from P
   for i from 2 to t1 do
      next ← a solution picked at random from P
      if fitness(next) > fitness(best) then
         best ← next
   return best
```

Figure 7.2: Pseudocode for the three-tier explorative tournament selection (3T-ETS) for the proposed ABC-AX^2 algorithm

allows search along axis directions only. In contrast, ABC-AX^2 tries to perform search along any possible direction from $\boldsymbol{x_i}$ by maintaining and automatically adapting a control parameter q_i, separately for every candidate solution $\boldsymbol{x_i}$, that controls the perturbation rate during producing the trial solution $\boldsymbol{v_i}$ from $\boldsymbol{x_i}$.

When ABC-AX^2 wants to perturb a solution $\boldsymbol{x_i}$ to produce $\boldsymbol{v_i}$, the value of q_i is perturbed first, with probability = u_1 using (7.4), before perturbing any other parameter of $\boldsymbol{x_i}$. This perturbed value of q_i is inherited by $\boldsymbol{v_i}$, which is henceforth referred as $\boldsymbol{v_i}.q$ and is used as the probability of perturbing the parameters of $\boldsymbol{x_i}$ during producing $\boldsymbol{v_i}$ from $\boldsymbol{x_i}$. A more appropriate value of $\boldsymbol{v_i}.q$ is likely to produce fitter new solutions, which are supposed to survive better than $\boldsymbol{x_i}$ and produce better, newer solutions and hence, propagate the better value of the perturbation probability. Thus a gradual self-adaptation towards better, more effective q_i values takes place, allowing a self-adaptive and appropriate perturbation rate for the candidate solutions across the population.

$$v_i.q = \begin{cases} q_{min} + rand(0,1) * (q_{max} - q_{min}) & if\ rand(0,1) < u_1 \\ q_i & otherwise \end{cases} \quad (7.4)$$

Here, u_1 is the probability that the perturbation probability q_i itself is perturbed before perturbing the parameters of $\boldsymbol{x_i}$. In our implementation, we have set u_1=0.10, q_{max}=1.0 and q_{min}=1/D after some initial experiments.

```
Algorithm 7.3: Two_Tier_Exploitative_Tournament_Selection(x_i)
  global P: Population of candidate solutions
  global s1, s2: Tournament sizes for the similarity and fitness
                 based tournaments, respectively

  begin
     return Tier2_Similarity_Tournament(x_i)
  end

procedure Tier2_Similarity_Tournament(x_i)
    best ← Tier1_Fitness_Tournament( )
    for i from 2 to s2 do
        next ← Tier1_Fitness_Tournament( )
        if distance(next, x_i) < distance(best, x_i) then
            best ← next
    return best

procedure Tier1_Fitness_Tournament( )
    best ← a solution picked at random from P
    for i from 2 to s1 do
        next ← a solution picked at random from P
        if fitness(next) > fitness(best) then
            best ← next
    return best
```

Figure 7.3: Pseudocode for the two-tier exploitative tournament selection (2T-ETS) for the ABC-AX^2 algorithm

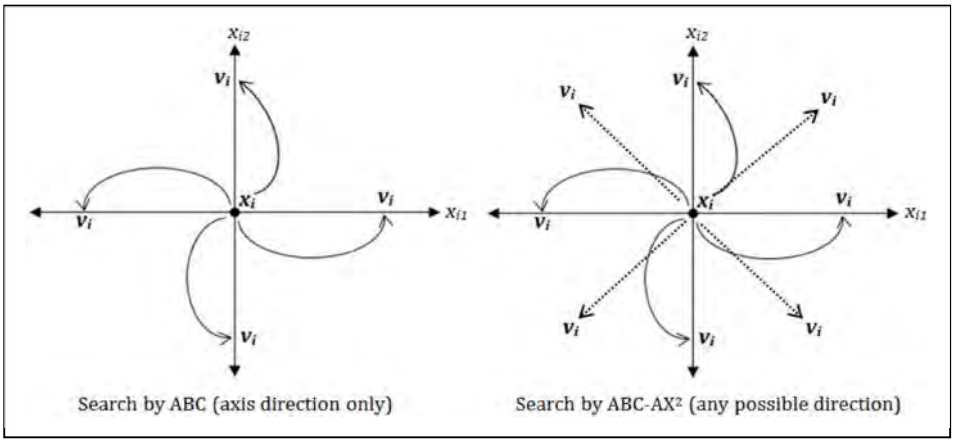


Figure 7.4: Search direction by ABC and ABC-AX^2 in 2D search space

C. Control parameter $\boldsymbol{\eta_i}$ for self-adaptive perturbation scaling factors: The basic ABC algorithm draws the φ_{ij} values in (2.6) uniformly at random from [-1,1], without any attempt to perform adaptation of the φ_{ij} values for more effective perturbations on $\boldsymbol{x_i}$. In contrast, ABC-AX^2 produces the φ_{ij} values from a Gaussian distribution with mean=0 and standard deviation=η_{ij}, where $\boldsymbol{\eta_i}=[\eta_{i1}, \eta_{i2}, \ldots, \eta_{iD}]^T$ is a control parameter vector that is maintained separately for each candidate solution $\boldsymbol{x_i}$ and is gradually self-adapted using (7.5) and (7.6). Although this scheme is similar to the self-adaptation strategy adopted in some other evolutionary algorithms [58], it has not yet been employed and tested with the ABC algorithm.

$$\eta'_{ij} = \eta_{ij} exp\left(\tau' N(0,1) + \tau N_j(0,1)\right) \quad for\ j = 1, \ldots, D \tag{7.5}$$

$$\boldsymbol{v_i.\eta} = \begin{cases} \boldsymbol{\eta'_i} & if\ rand(0,1) < u_2 \\ \boldsymbol{\eta_i} & otherwise \end{cases} \tag{7.6}$$

Here u_2 is the probability that the new trial solution $\boldsymbol{v_i}$ gets a control parameter $\boldsymbol{v_i.\eta}$ that is different from $\boldsymbol{\eta_i}$ of the original solution $\boldsymbol{x_i}$. We used u_2=0.5. The $N(0,1)$ and $N_j(0,1)$ are random numbers produced from the Normal distribution with mean=0 and standard deviation=1. The subscript j in $N_j(0,1)$ indicates that the random number is generated anew for each value of j. The τ and τ' are called learning rates and are set as $\left(\sqrt{2\sqrt{D}}\right)^{-1}$ and $\left(\sqrt{2D}\right)^{-1}$ respectively, as suggested in [58]. ABC-AX^2 maintains a separate $\boldsymbol{\eta_i}$ for every solution $\boldsymbol{x_i}$, which enables each $\boldsymbol{x_i}$ to customize its own degree of explorations and exploitations, separately along the D different axis directions of the search space, using the corresponding components of $\boldsymbol{\eta_i}=[\eta_{i1}, \eta_{i2}, \ldots, \eta_{iD}]^T$. An effective value for $\boldsymbol{v_i.\eta}$ is likely to produce better, fitter new solutions that should survive better than $\boldsymbol{x_i}$ and thus a gradual self-adaptation towards better, more effective $\boldsymbol{\eta_i}$ values can take place, cycle by cycle, across the population.

The key difference of ABC-AX^2 from the basic ABC algorithm is how ABC-AX^2 perturbs an existing candidate solution $\boldsymbol{x_i}$ using its control parameters p_i, q_i and $\boldsymbol{\eta_i}$ and how ABC-AX^2 gradually adapts the values of these control parameters for $\boldsymbol{x_i}$. Fig. 7.5 presents the pseudocode of the perturbation operation of both ABC and ABC-AX^2, one after another, for an easy comparison. The synergy and interaction of all three parameters — p_i, q_i and $\boldsymbol{\eta_i}$ make it possible to gradually adapt and customize the degree of explorations and exploitations at the individual solution level, separately for each candidate solution $\boldsymbol{x_i}$. This is more rational than a population wide global parameter setting, because the scenario around each candidate solution can be unique and very different from other solutions of the population. A candidate solution may need to explore through several peaks, valleys and plateaus of the fitness landscape, which makes the explorative/exploitative requirements around each candidate solution to change suddenly and often in an unpredicted way. Therefore, attaching each solution $\boldsymbol{x_i}$ with a number of

self-adaptive control parameters can be very useful to dynamically adapt and customize the degree of explorations and exploitations at the individual solution level.

Algorithm 7.4: Perturbation_by_ABC
input: An existing candidate solution $\boldsymbol{x_i}$
output: Perturbed candidate solution $\boldsymbol{v_i}$

begin
  $\boldsymbol{v_i} = \boldsymbol{x_i}$
  k = *Uniform_Random* ~ {1, 2, ..., *SN*} **comment:** *SN* is the number of employed or onlooker bees
  j = *Uniform_Random* ~ {1, 2, ..., *D*} **comment:** *D* is the dimensionality of the problem
  φ_{ij} = *Uniform_Random* ~ [-1, 1]
  $v_{ij} = x_{ij} + \varphi_{ij} * (x_{kj} - x_{ij})$
  return $\boldsymbol{v_i}$
end

Algorithm 7.5: Perturbation_by_ABC-AX²
input: An existing candidate solution $\boldsymbol{x_i}$
output: Perturbed candidate solution $\boldsymbol{v_i}$

begin
  $\boldsymbol{v_i} = \boldsymbol{x_i}$
  r_1= *Uniform_Random* ~ [0, 1]
  if $r_1 \leq u_1$ **then** **comment:** u_1 is the probability that the trial solution $\boldsymbol{v_i}$ gets a different perturbation probability, $\boldsymbol{v_i}.q$ than that of the original solution $\boldsymbol{x_i}$. We used u_1=0.1, q_{max}=1.0, $q_{min} = 1/D$
    r_2 = *Uniform_Random* ~ [0, 1]
    $\boldsymbol{v_i}.q = q_{min} + (q_{max} - q_{min}) * r_2$
  else
    $\boldsymbol{v_i}.q = q_i$
  endif

  r_3 = *Uniform_Random* ~ [0, 1]
  if $r_3 \leq u_2$ **then** **comment:** u_2 is the probability that the scaling factor vector $\boldsymbol{\eta_i}$ is perturbed first, before producing the trial solution $\boldsymbol{v_i}$ from $\boldsymbol{x_i}$. We used u_2 =0.5
    for j = 1 to D **do**
      $\boldsymbol{v_i}.\eta_j = \eta_{ij} \exp(\tau' * N(0,1) + \tau * N_j(0,1))$
    enddo
  else
    $\boldsymbol{v_i}.\boldsymbol{\eta} = \boldsymbol{\eta_i}$
  endif

  r_4= *Uniform_Random* ~ [0, 1]
  if $r_4 \leq p_i$ **then** **comment:** Probability of exploitative perturbation on $\boldsymbol{x_i}$ is p_i
    Select $\boldsymbol{x_k}$ using the two-tier exploitative tournament selection scheme
  else
    Select $\boldsymbol{x_k}$ using the three-tier explorative tournament selection scheme
  endif

  for j = 1 to D **do**
    r_5 = *Uniform_Random* ~ [0, 1]
    if $r_5 \leq \boldsymbol{v_i}.q$ **then** **comment:** Perturb each parameter of $\boldsymbol{x_i}$ with probability=$\boldsymbol{v_i}.q$
      φ_{ij} = Gaussian_Random(0, $\boldsymbol{v_i}.\eta_j$)
      $v_{ij} = x_{ij} + \varphi_{ij} * (x_{kj} - x_{ij})$
    endif
  enddo
return $\boldsymbol{v_i}$
end

Figure 7.5: The perturbation operation by ABC and ABC-AX²

7.5 Evaluation of ABC-AX^2 on Benchmark Functions

To evaluate the performance of ABC-AX^2 and to compare it with the basic ABC [11] and some other recent ABC-variants (e.g., [59]–[68]), we have used two different suites of benchmark problems. First, ABC-AX^2 is tested on a suite of 30 standard benchmark functions, including both unimodal and multimodal, separable and non-separable, low and high dimensional functions. Later, ABC-AX^2 is also tested on a recent set of benchmark functions introduced in the special session on real parameter optimization at CEC2005 [76]. The new suite consists of 25 benchmark functions of more complexity, including many shifted, rotated, scaled, expanded and hybrid composite functions. Both the benchmark suites are briefly introduced in the previous section 2.17 (Tables 2.3, 2.4). Further details on each benchmark function can be found in the Appendix A and also in [76].

7.5.1 ABC-AX^2 on Standard Benchmark Functions

Table 7.1 presents the results of ABC-AX^2 on the 30 standard benchmark functions (Table 2.3) and compares the results with the basic ABC [11] and ABC with self-adaptive mutation (ABC-SAM) (chapter 5). All the algorithms made 50 independent runs on each function and the mean and standard deviation of the best found solutions are presented in Table 7.1. The algorithms have three parameters in common — population size *SN*, maximum cycle number *MCN* and *limit*. For functions f_1–f_{18} with D=30, we used SN=100, MCN=1000 and *limit*=100. For the larger variants with D=60, the value of SN is kept the same (i.e., 100), but *limit* and *MCN* are set to 200 and 2000, respectively. For the low dimensional f_{19}–f_{30}, we used SN=100, MCN=100, $limit = 10 * D$. Other parameters of ABC-AX^2 are set as: τ_1=50, τ_2=10, u_1=0.1, u_2=0.5, $q_{min} = 1/D$, q_{max}=1.0. Tournament sizes for 3T-ETS and 2T-ETS selection schemes (Fig. 7.2, 7.3) are set as: t_1=t_2=6, t_3=4 and s_1=6, s_2=4. During initializations, the control parameter p_i of each solution $\boldsymbol{x_i}$ is set to 0.5, and the q_i and η_{ij} values are initialized to random values from [q_{min}, q_{max}] and [-1, 1]. These values are chosen with some initial evaluations and not meant for optimum.

- **ABC vs. ABC-AX^2:** Out of the 18 high dimensional functions f_1–f_{18}, ABC-AX^2 outperforms ABC on as many as 16 functions, shows similar performance on one (f_8), while ABC manages to perform better only on one function (f_7). Each time, the difference is statistically significant, as measured by *t*-test with 95% confidence interval. For the low dimensional functions f_{19}–f_{30}, both ABC and ABC-AX^2 perform equally well on eight functions, while ABC-AX^2 performs better on the remaining four.
- **ABC-SAM vs. ABC-AX^2:** On all of the 30 functions, ABC-AX^2 performs either better than or as well as ABC-SAM. For the high dimensional functions f_1–f_{18}, ABC-AX^2 significantly

outperforms ABC-SAM on as many as 16 functions and shows similar performance on two (f_7 and f_8). On most (nine out of twelve) of the low dimensional functions f_{19}–f_{30}, both the algorithms perform equally well (i.e., *t*-test does not find any significant difference between the results), though ABC-AX^2 performs better on the remaining three.

Table 7.1: Comparison of ABC-AX^2 with basic ABC [11] and ABC-SAM (chapter 5) on the 30 standard benchmark suite functions. The best results are marked with boldface font.

No	f_{min}	D	G	ABC		ABC-SAM (K=10)		ABC-AX^2	
				Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
f_1	0	30	1000	2.45e–11	7.72e–12	4.18e–14	5.37e–15	**5.51e–24**	**3.73e–25**
		60	2000	3.75e–10	2.01e–11	6.09e–13	7.24e–14	**9.43e–28**	**7.26e–29**
f_2	0	30	1000	5.05e–07	1.74e–07	2.47e–08	2.35e–09	**4.23e–15**	**3.54e–16**
		60	2000	5.58e–06	1.17e–06	5.06e–07	2.97e–07	**2.98e–17**	**1.07e–17**
f_3	0	30	1000	4.18e+01	5.90	1.69e+01	1.43	**6.60e–02**	**5.21e–03**
		60	2000	7.31e+01	6.88	3.10e+01	5.12	**2.78**	**0.77**
f_4	0	30	1000	8.32e–10	9.75e–11	3.95e–12	5.77e–13	**3.42e–16**	**8.83e–18**
		60	2000	4.50e–09	5.64e–10	7.54e–11	2.14e–11	**8.84e–20**	**5.45e–21**
f_5	0	24	1000	6.61e+00	1.07e+00	9.24e–01	2.08e–01	**2.23e–02**	**3.75e–03**
f_6	0	30	1000	6.67e–01	1.21e–08	2.16e–03	6.37e–04	**5.91e–05**	**5.67e–06**
		60	2000	6.66e–01	1.05e–07	7.76e–02	1.63e–02	**8.33e–05**	**1.71e–05**
f_7	0	30	1000	**4.25e–01**	**1.18e–01**	2.28e+01	3.75	2.39e+01	3.66
		60	2000	**2.02e–01**	**6.92e–02**	4.96e+01	7.80	5.15e+01	7.69
f_8	0	30	1000	0	0	0	0	0	0
		60	2000	0	0	0	0	0	0
f_9	0	30	1000	8.60e–13	8.32e–13	3.66e–16	1.44e–17	**8.87e–34**	**6.78e–35**
		60	2000	9.31e–12	7.17e–12	4.76e–15	5.32e–16	**6.31e–32**	**2.16e–33**
f_{10}	0	30	1000	1.72e–14	1.56e–14	1.26e–16	2.11e–17	**4.68e–24**	**9.03e–26**
		60	2000	2.84e–13	8.01e–14	8.55e–15	3.15e–16	**6.12e–31**	**8.67e–33**
f_{11}	0	30	1000	2.33e–08	7.49e–09	4.60e–10	8.85e–11	**1.04e–13**	**3.16e–14**
		60	2000	6.64e–07	1.51e–07	6.80e–09	8.77e–10	**4.25e–13**	**7.32e–14**
f_{12}	–12569.5	30	1000	–11346.79	2.77e+02	–12416.19	4.02e+01	**–12569.48**	**1.50e–02**
	–25138.9	60	2000	–22530.82	4.08e+02	–23805.93	2.84e+02	**–25016.6**	**1.89e+01**
f_{13}	0	30	1000	2.93e–06	3.38e–07	9.26e–08	1.89e–08	**8.13e–13**	**6.71e–14**
		60	2000	4.65e–06	1.07e–06	2.07e–08	3.55e–08	**3.62e–14**	**1.15e–15**
f_{14}	0	30	1000	4.55e–08	6.54e–09	8.36e–10	5.08e–11	**5.63e–23**	**7.35e–25**
		60	2000	8.01e–07	2.64e–07	1.56e–10	6.90e–11	**7.04e–31**	**5.77e–32**

Table 7.1 (continued): Comparison of ABC-AX² with basic ABC [11] and ABC-SAM (chapter 5) on the standard benchmark suite functions.

No	f_{min}	D	G	ABC		ABC-SAM (K=10)		ABC-AX²	
				Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
f_{15}	0	30	1000	3.34e–04	3.76e–05	2.22e–08	3.93e–09	**8.56e–13**	**1.56e–13**
		60	2000	7.49e–03	9.58e–04	1.17e–08	2.35e–09	**5.37e–13**	**1.25e–13**
f_{16}	0	30	1000	3.36e–01	9.58e–02	5.78e–04	6.31e–05	**6.46e–09**	**8.32e–10**
		60	2000	8.99e–01	3.09e–01	9.20e–03	4.03e–03	**5.38e–08**	**9.19e–10**
f_{17}	0	30	1000	**5.47e–12**	**2.09e–13**	9.78e–12	3.89e–12	**3.85e–14**	**4.93e–15**
		60	2000	**7.47e–12**	**2.74e–12**	1.32e–11	5.15e–11	**3.50e–14**	**2.60e–15**
f_{18}	0	30	1000	2.63e–03	1.89e–04	3.06e–02	8.59e–03	**2.33e–21**	**7.55e–22**
		60	2000	2.66e–03	7.90e–04	5.11e–02	7.39e–03	**7.52e–26**	**1.29e–26**
f_{19}	1	2	100	1.04	0.04	1.03	0.03	**1.01**	**0.01**
f_{20}	3.07e–04	4	100	5.98e–04	7.22e–05	4.32e–04	1.09e–05	**3.10e–04**	**8.73e–06**
f_{21}	–1.0316	2	100	–1.0316	0	–1.0316	0	–1.0316	0
f_{22}	0.398	2	100	0.398	7.12e–08	0.398	2.75e–07	0.398	1.83e–07
f_{23}	–3.86	3	100	–3.86	7.09e–07	–3.86	1.54e–08	–3.86	6.77e–10
f_{24}	–3.32	6	100	–3.32	4.74e–13	–3.32	6.26e–14	–3.32	2.61e–15
f_{25}	–10.15	4	100	–9.61	0.14	–10.14	3.68e–07	**–10.15**	**9.15e–08**
f_{26}	–10.40	4	100	–10.40	8.61e–03	–10.40	7.94e–03	–10.40	2.56e–03
f_{27}	–10.54	4	100	–10.52	0.01	–10.54	6.77e–07	**–10.55**	**7.84e–08**
f_{28}	0	10	100	13.77	3.80	4.02	0.39	**4.19e–01**	**6.54e–02**
f_{29}	-9.66015	10	100	-9.66015	0	-9.66015	0	-9.66015	0
f_{30}	-1.4	10	100	–0.78	0.09	–1.04	0.06	**–1.28**	**0.03**
Summary (*t*-Test)		+		20		19			
		–		1		0			
		≈		9		11			

- For almost all the functions, ABC-AX² shows very low standard deviation of its results. On average, the magnitude of the standard deviation of the results by ABC-AX² of is only around 9% of their mean, while the standard deviation of the results of ABC and ABC-SAM is 18% and 17%, respectively, of their mean results (Fig. 7.7). ABC-AX² also shows the smallest mean absolute error in comparison to ABC and ABC-SAM, as shown by Fig. 7.7. This indicates the high degree of both robustness and consistency of ABC-AX² over all these standard benchmark functions.
- The '+', '–' and '≈' symbols at the bottom rows count the number of functions where ABC-AX² produces significantly better, worse and similar results, respectively compared to ABC or ABC-SAM. Out of the 30 functions, ABC-AX² performs significantly better than

ABC and ABC-SAM on 20 and 19 functions, shows similar performance on 9 and 11 functions, respectively, while ABC performs better on one function (f_7) only. Thus the overall performance of ABC-AX^2 is much better than both ABC and ABC-SAM.

Fig. 7.6 shows the convergence graphs of ABC, ABC-SAM and ABC-AX^2 for three unimodal (f_1, f_4, f_8) and three multimodal (f_{12}, f_{13}, f_{18}) functions with D=30. ABC-AX^2 shows far better convergence characteristics than its counterparts for all these functions. For example, consider the functions f_{12} and f_{18}, where both ABC and ABC-SAM converges to a local minimum and gets stuck there till the end of their execution. In contrast, ABC-AX^2 easily reaches the global minimum for f_{12} and shows no sign of fitness stagnation for f_{18}, even after reaching the vicinity of the global minimum. For some functions, e.g., f_1, f_4, f_{13} and f_{18}, ABC-SAM initially shows somewhat higher convergence speed than ABC-AX^2, but eventually it either gets stuck at local optima (f_{13}, f_{18}) or gradually slows down (f_1, f_4) and at the end, ABC-AX^2 shows significantly higher convergence speed than both ABC and ABC-SAM. Fig. 7.6 shows that ABC-AX^2 has always reached very close proximity to the global minimum, while ABC and ABC-SAM can get stuck at

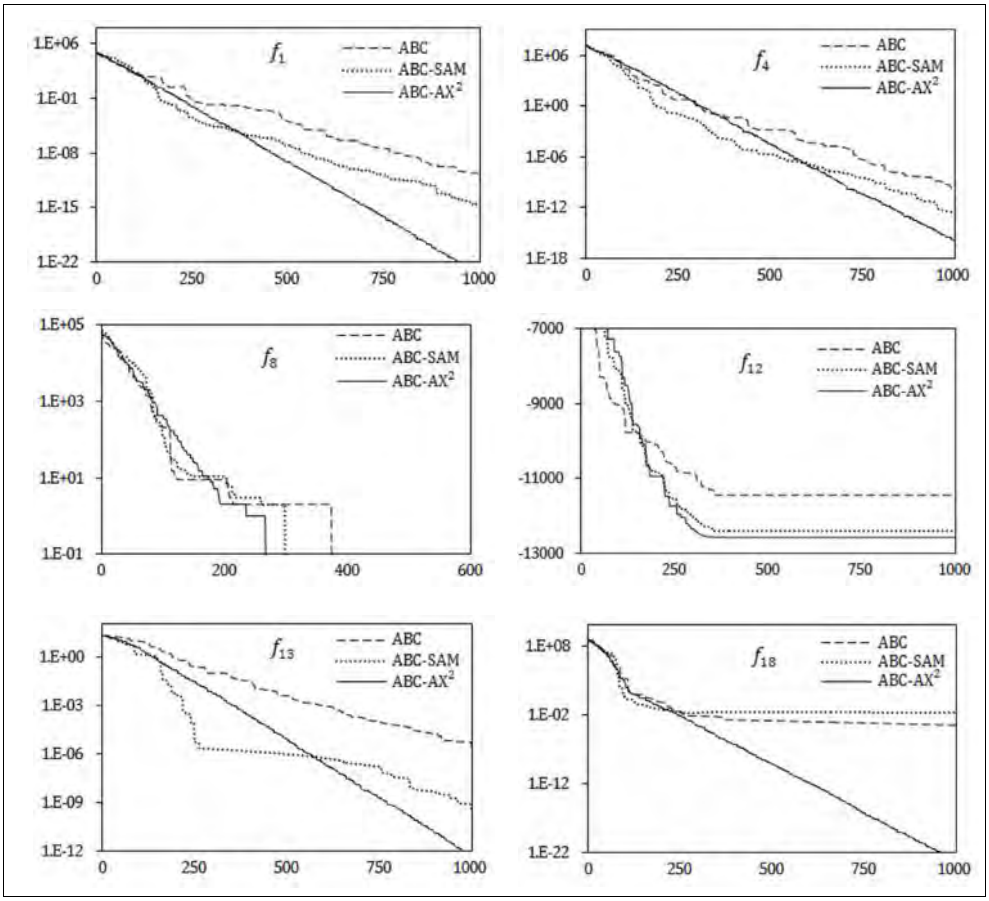


Figure 7.6: Convergence characteristics of ABC, ABC-SAM and ABC-AX^2 on three unimodal (f_1, f_4, f_8) and three multimodal (f_{12}, f_{13}, f_{18}) functions. The vertical axis is the function value, while the horizontal axis is the number of cycles elapsed.

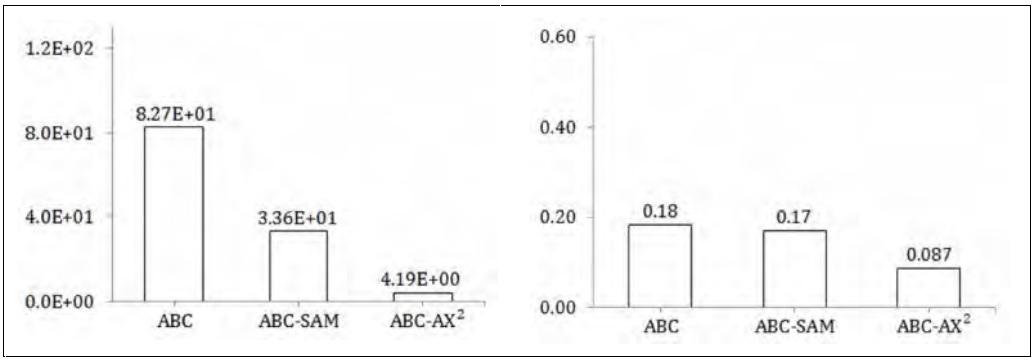


Figure 7.7: Comparison among ABC, ABC-SAM and ABC-AX^2, based on their mean absolute errors (on the left) and the ratio of the standard deviation to the mean of errors (on the right) over the standard benchmark functions f_1–f_{30}.

several intermediate local optima (note the semi-flat and flat regions of the plot of ABC in f_4 and ABC-SAM in f_{13}). Previously Table 7.1 considered the performance of all three algorithms to be similar on f_8, but Fig. 7.6 now reveals that ABC-AX^2 actually reaches the global minimum of f_8 much earlier (i.e., fewer than 300 cycles) than both ABC-SAM and ABC.

Results in Table 7.1 and Fig. 7.6 indicate that both ABC-SAM and ABC-AX^2 show improved results and better convergence characteristics than the basic ABC algorithm. Both of them outperform ABC on most of the high dimensional functions f_1–f_{18}. ABC-SAM maintains a scaling factor value (SF_i) for every individual bee $\boldsymbol{x_i}$ and periodically adjusts the value of SF_i using two different distributions — one explorative and the other exploitative. The gradual adjustment of the SF_i values performs an automatic self-adaptation of perturbation step size, separately for every individual bee $\boldsymbol{x_i}$. However, the results from ABC-AX^2 are further improved and it outperforms both ABC and ABC-SAM on most (16 out of the 18) of the high dimensional functions f_1–f_{18}, including all the multimodal functions f_{10}–f_{18}. The reason may be that ABC-AX^2 employs the idea of self-adaptation not only to perturbation scaling factors (using $\boldsymbol{\eta_i}$), as is done by ABC-SAM (by using SF_i), but also to perturbation rate (using q_i) and proportion of explorative and exploitative perturbations (using p_i). The synergy and interaction among p_i, q_i and $\boldsymbol{\eta_i}$ improves the results further for ABC-AX^2, as has been shown later (Table 8.24, chapter 8). The only function that seems to present some difficulty to both ABC-AX^2 and ABC-SAM is the Rosenbrock function f_7. Although it is often regarded as a unimodal function, as in [11], [57], there is some evidence (e.g., [228], [229]) that it contains several minima in high dimensional instances. The global minimum is situated inside a long, narrow valley which is almost flat. Finding the valley is not difficult, but locating the global minimum in the almost flat valley is extremely difficult. This is like searching for a needle in a haystack, because the flat region does not provide any useful gradient direction pointing towards the global minimum.

7.5.2 Comparison of ABC-AX^2 with Other ABC-variants

In this section we compare ABC-AX^2 with some other recent variants of ABC, each of which alters the explorative and/or exploitative properties of the basic ABC algorithm, such as cooperative ABC (CABC) [60], ABC with diversity strategy [61], ABC with improved explorations (ABC-IX) (chapter 6), chaotic ABC (ChABC) [62], Gbest-guided ABC (GABC) [64] and Hooke Jeeves ABC (HJABC) [65]. The first three variants (e.g., [60]–[62]) primarily increase the degree of explorations to achieve better strength against local optima and premature convergence, while the last two variants (e.g., [64], [65]) mainly increase the intensity of exploitations to attain better convergence speed than the basic ABC algorithm.

First, ABC-AX^2 is compared with CABC [60], which is a cooperative variant of the basic ABC algorithm. CABC has been introduced in two different versions — CABC_S and CABC_H. In order to perform more explorations, CABC_S decomposes the search space into multiple sub-spaces and employs different bee colonies to search and explore the different sub-spaces. The other variant, CABC_H tries to perform more exploitations than CABC_S by repeatedly alternating between explorative CABC_S and exploitative ABC. For comparison, ABC-AX^2 is re-implemented with the same settings [60] — SN = 40, no. of function evaluations FE = 100,000 and $limit$=SN ∗ D. Table 7.2 shows that ABC-AX^2 significantly outperforms both the CABC variants on four out of the six benchmark functions, while CABC_S and CABC_H perform better on one or two functions only. Thus the overall performance of ABC-AX^2 is better than the CABC variants. The reason may lie in the difficulties that the CABC variants have to face to properly decompose the search space into multiple sub-spaces and then, to effectively combine the partial solutions into a complete solution to the whole problem.

The next comparison is made between ABC-AX^2 and DABC [61]. DABC tries to maintain sufficient amount of diversity among the candidate solutions to allow more search space explorations. DABC regularly measures the existing population diversity d and employs either its explorative or exploitative perturbation based on the value of d. As suggested in [61], ABC-AX^2 is re-implemented with SN=20, MCN=5000 and $limit$=100 to compare with DABC. Results presented in Table 7.3 show that ABC-AX^2 performs better than DABC on two out of four functions (f_1 and f_{14}), shows similar performance on one (f_{10}), while DABC performs better on the remaining one function (f_7) only. The reason may be that DABC completely relies on its estimated value of population diversity to choose between explorations and exploitations, while there is no accurate and universally accepted metric for diversity that can correctly measure the maturity of the optimization process. Besides, DABC uses a naïve strategy of using a fixed threshold diversity value (i.e., d_{low} in [61]) to switch between explorations and exploitations,

which may cause repeated oscillations between conflicting explorative and exploitative perturbations and thus may have reduced its convergence speed.

Table 7.2: Comparison of ABC-AX² with CABC_S [60] and CABC_H [60]. Boldface font marks the best performance for each function.

Function	CABC_S		CABC_H		ABC-AX²	
	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
f_1	3.30e–19	2.00e-19	5.92e–18	3.56e–18	**7.22e–51**	**4.08e–52**
f_7	6.33e+00	7.68e+00	**4.80e–01**	**8.55e–01**	5.94e+00	4.35e+00
f_{10}	**0**	**0**	**0**	**0**	8.56e–54	3.83e–55
f_{12}	1.30e–04	5.21e–06	1.27e–04	0	**1.04e–06**	**7.98e–08**
f_{13}	1.83e–14	9.86e–15	8.35e–15	4.13e–15	**7.14e–18**	**7.83e–19**
f_{14}	4.42e–02	2.99e–02	7.96e–03	9.06e–03	**3.84e–54**	**2.04e–55**
+	4		4			
–	1		2			
≈	1		0			

Table 7.3: Comparison between ABC-AX² and DABC [61]. Best results are marked with bold font; if not both the algorithms produce identical results.

Function	*D*	DABC		ABC-AX²	
		Mean	Std. Dev.	Mean	Std. Dev.
f_1	10	2.01e–17	5.63e–17	**0**	**0**
	30	2.01e–16	2.85e–17	**7.26e–51**	**6.47e–52**
f_7	10	**2.73e–03**	**7.04e–03**	9.46e–02	8.22e–03
	30	**1.42e–02**	**2.53e–02**	5.88e–01	8.32e–02
f_{10}	10	0	0	0	0
	30	0	0	0	0
f_{14}	10	0	0	0	0
	30	2.59e–16	1.22e–16	**1.78e–50**	**5.28e–51**
+		3			
–		2			
≈		3			

Next, we compare ABC-AX^2 to ABC with improved explorations (ABC-IX), which is another improved variant of the ABC algorithm. ABC-IX alters both the selection and perturbation operators of ABC in a more explorative manner. The selection operator is modified to accept not only better but also worse solutions, using a probabilistic simulated annealing-like procedure, while the perturbation operator is modified by incorporating a self-adaptive perturbation rate. Because of its improved explorative characteristics, ABC-IX performs better than many other existing ABC-variants (chapter 6). Table 7.4 compares ABC-AX^2 with ABC-IX on the high dimensional multimodal functions f_{10}–f_{18} with the following parameter settings — SN=50, MCN=1000 and *limit*=100. Results show that ABC-AX^2 often (6 out of 9 functions) performs significantly better than ABC-IX, while ABC-IX performs better on three functions only. This clearly indicates the better effectiveness of the adaptive and self-adaptive schemes of ABC-AX^2 over the more explorative scheme of ABC-IX.

Next, ABC-AX^2 is compared with the Chaotic ABC (ChABC) [62] algorithm. ChABC employs chaotic search behavior during perturbations to produce new food positions from the existing ones. Chaotic dynamics are produced by the logistic equations (eq. (4)–(7) in [62]) which provide a simple mechanism to escape from local minima and avoid premature convergence. For comparison, ABC-AX^2 is executed for 5000 cycles with population size of 70 and *limit*=200, as suggested in [62]. Results (Table 7.5) show that ABC-AX^2 outperforms ChABC on as many as five out of the six functions, while ChABC performs better on the remaining one function (f_{10}) only. The reason may be that ChABC employs same chaotic strategy uniformly for all the candidate solutions across the population, without considering their individual exploitative/explorative requirement, while ABC-AX^2 considers and customizes the degree of explorations and exploitations separately for every candidate solution, which should be more effective for complex optimization tasks.

Next, ABC-AX^2 is compared with GABC [64]. GABC is an exploitative ABC-variant that tries to improve the convergence speed of ABC by using the information of the global best solution found so far into the perturbation scheme (2.6) of basic ABC. For fair comparison, ABC-AX^2 is executed with the same settings [64] and results are presented in Table 7.6. In [64], GABC is tested with several values of its parameter C, but the best results are always observed with C = 1.0 or 1.5, so Table 7.6 includes both the results. Results show that ABC-AX^2 outperforms GABC on four out of the five functions, while GABC performs better on the remaining one (f_7) only. The reason may be that the perturbation operation of GABC becomes too exploitative by pushing its candidate solutions towards the best solution found so far. Increased exploitations, at the cost of

reduced explorations, may improve the final solution quality for the unimodal and low dimensional function f_7, but is likely to fail for the other four high dimensional functions in Table 7.6 with D=30 and 60.

Table 7.4: Comparison between ABC-AX² and ABC-IX (chapter 6) based on the final solution quality. Best results are marked with boldface font.

Function	*D*	ABC-IX		ABC-AX²	
		Mean	Std. Dev.	Mean	Std. Dev.
f_{10}	30	**6.14e–41**	**8.11e–42**	8.60e–19	1.14e–19
f_{11}	30	8.51e–12	2.93e–12	**4.42e–15**	**7.47e–16**
f_{12}	30	1.56e+02	5.69e+01	**5.06e–01**	**1.77e–01**
f_{13}	30	**3.82e–15**	**4.28e–16**	3.21e–11	8.09e–12
f_{14}	30	**9.70e–40**	**8.30e–41**	7.56e–18	2.70e–18
f_{15}	30	8.21e–10	9.34e–11	**2.48e–17**	**4.09e–18**
f_{16}	30	6.74e–07	2.16e–07	**3.85e–08**	**8.12e–09**
f_{17}	30	7.40e–11	2.06e–11	**2.48e–12**	**7.09e–13**
f_{18}	30	2.61e–03	1.96e–04	**1.85e–16**	**3.12e–17**
+		6			
–		3			
≈		0			

Table 7.5: Comparison between ABC-AX² and ChABC [62] based on the final solution quality. Best results are marked with boldface font.

Function	*D*	ChABC		ABC-AX²	
		Mean	Std. Dev.	Mean	Std. Dev.
f_1	30	2.99e–16	3.54e–17	**9.76e–119**	**7.17e–120**
f_7	30	6.33e–02	8.96e–02	**8.48e–07**	**4.99e–08**
f_{10}	30	**0**	**0**	8.60e–129	5.14e–130
f_{12}	30	3.81e–04	2.07e–04	**5.32e–06**	**6.04e–07**
f_{13}	30	2.93e–14	2.99e–15	**2.48e–17**	**4.09e–18**
f_{14}	30	2.70e–16	6.20e–17	**1.85e–129**	**5.75e–130**
+		5			
–		1			
≈		0			

Table 7.6: Comparison between ABC-AX² and GABC [64] based on the final solution quality. Best results are marked with boldface font.

Function	*D*	GABC (*C*=1.0)		GABC (*C*=1.5)		ABC-AX²	
		Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
f_1	30	4.31e–16	7.49e–17	4.17e–16	7.36e–17	**9.66e–105**	**8.05e–106**
	60	1.43e–15	1.43e–16	1.43e–15	1.37e–16	**1.98e–36**	**3.58e–37**
f_7	2	3.93e–04	4.45e–04	**1.68e–04**	**1.45e–04**	7.88e–03	5.35e–04
	3	**2.63e–03**	**2.11e–03**	2.65e–03	2.22e–03	2.13e–01	6.12e–02
f_{10}	30	9.47e–15	2.15e–14	1.32e–14	2.44e–14	**6.30e–108**	**1.52e–109**
	60	4.16e–13	1.77e–13	3.52e–13	1.24e–13	**7.42e–40**	**3.75e–41**
f_{13}	30	3.31e–14	2.90e–15	3.21e–14	3.25e–15	**4.04e–19**	**8.28e–20**
	60	1.04e–13	1.07e–14	1.00e–13	6.08e–15	**1.63e–17**	**2.66e–18**
f_{14}	30	8.88e–17	8.45e–17	2.96e–17	4.99e–17	**9.50e–101**	**2.00e–102**
	60	9.47e–16	7.84e–16	7.54e–17	4.12e–16	**1.81e–33**	**3.25e–34**
+		4		4			
–		1		1			
≈		0		0			

Next, ABC-AX² is compared with HJABC [65], which is a hybrid ABC-variant that intensifies the degree of exploitations by hybridizing basic ABC with an efficient local search technique (i.e., Hooke Jeeves pattern search). Table 7.7 compares ABC-AX² and HJABC based on the number of function evaluations (*NFE* in [65]) required to achieve a predefined level of accuracy. Both ABC-AX² and HJABC are run with *SN*=25 and *limit*=$SN * D$, until either *NFE* reaches a predefined maximum value (NFE_{max}) or the condition that $\left| f^* - \tilde{f} \right| \leq \varepsilon_1$ is satisfied, where f^* is the global minimum, $\tilde{f}$ is the best function value found so far by the algorithm, $\varepsilon_1 = 10^{-8}$ with NFE_{max} = 300000, as suggested in [65]. For seven out of the eleven functions in Table 7.7, ABC-AX² performs better than HJABC, by showing a faster convergence speed, while HJABC performs better on the remaining four. However, ABC-AX² can't achieve the predefined level of accuracy within NFE_{max} function evaluations for two functions (f_3 and f_7), while HJABC fails to do so only for one function (f_9). In short, the overall performance of ABC-AX² is quite comparable to HJABC. The reason that HJABC often requires larger number of function evaluations, even after using the efficient Hooke Jeeves local searcher [65], may be that – HJABC regularly tries to find an appropriate search direction by exploring along the axis directions only, exploring just one variable at a time, which is not suitable for the non-separable problems. Hence, this may produce improper search directions that can reduce the convergence speed of HJABC.

Table 7.7: Comparison between ABC-AX² and HJABC [65] based on convergence speed. Best results are marked with boldface font.

Function	*D*	Number of function evaluations	
		HJABC	ABC-AX²
f_1	30	18322	**13805**
f_2	30	**12509**	17987
f_3	30	**120315**	–
f_4	30	43939	**35502**
f_7	30	**102718**	–
f_8	30	17755	**13986**
f_9	30	–	**12230**
f_{10}	30	**15376**	20713
f_{13}	30	54497	**42609**
f_{14}	30	56855	**31582**
f_{15}	30	99686	**81678**
+		7	
–		4	
≈		0	

7.5.3 ABC-AX² on CEC2005 Benchmark Functions

In addition to the standard benchmark functions, as described in the previous subsections, ABC-AX² is also evaluated using a recent benchmark suite introduced in the special session on real parameter optimization at CEC2005 [76]. The functions in the CEC2005 suite are more complex and challenging to optimize, because they include many shifted, rotated, scaled, expanded and hybrid composition of other benchmark functions. An interested reader can find further details on each of the CEC2005 functions in the previous section 2.17 (Table 2.4), the Appendix A and in [76].

Using the CEC2005 suite, ABC-AX² is evaluated and compared with some other existing evolutionary and swarm intelligence algorithms, such as dynamic multi-swarm PSO with local search (DMS-PSO) [231], PSO with recombination by dynamic linkage discovery (PSO-RDL) [232], self-adaptive differential evolution (SADE) [230], covariance matrix adaptation evolution strategy with restarts (R-CMAES) [234] and the basic ABC algorithm [11]. DMS-PSO [231] is an improved PSO variant that divides the candidate solutions into several small, dynamic swarms. The swarms are regrouped frequently by using various regrouping

schedules and information is exchanged among the swarms. Also, the quasi-Newton method is employed to improve its local search ability. The next algorithm — PSO-RDL [232] introduces a special recombination operator that dynamically discovers the linkages among the variables and tries to identify the important building blocks that are present among the good quality solutions, which are then effectively used to produce better trial solutions from the existing ones. SADE [230] is an improved DE variant that employs a learning strategy to gradually self-adapt some parameter values of the standard DE algorithm. CMAES [234] is an evolution strategy that maintains a full covariance matrix of Normal mutation distributions and continuously adjusts the matrix to adapt the mutation step size for more successful mutations. The R-CMAES improves the basic CMAES algorithm by incorporating a restart strategy with successively increasing population size whenever the basic CMAES prematurely converges to local optima. A larger population is likely to make the search more explorative and robust against local optima. All the algorithms have been compared on the CEC2005 suite functions with D=10 and the number of function evaluations set to 100,000. Both ABC and ABC-AX^2 have colony size=20 and *limit*=200. The values of the remaining parameters of ABC-AX^2 are the same as before. The results of DMS-PSO [231], PSO-RDL [232], SADE [230] and R-CMAES [234] are obtained directly from the corresponding papers. Tables 7.8–7.9 present the mean error over 25 independent runs on each function by all the algorithms. The results have been summarized by following points.

- Out of the first 14 functions (i.e., non-composite functions F1–F14), ABC-AX^2 becomes the best performer on five functions (F2, F4, F11, F13 and F14), outperforming all other algorithms on these functions. In contrast, DMS-PSO and SADE show best performance only on two and three functions, respectively. However, R-CMAES also becomes best performer on five functions, i.e., the same number of functions as of ABC-AX^2.
- For the hybrid composition functions F15–F25, R-CMAES and ABC-AX^2 show the best performance on six and five functions, respectively, while DMS-PSO and SADE have managed to perform best only on one function each.
- To summarize, the overall performance of ABC-AX^2 is clearly better that DMS-PSO, PSO-RDL and SADE, and quite comparable to R-CMAES.
- To compare ABC-AX^2 with the basic ABC algorithm, it is remarkable that ABC-AX^2 has improved the results of ABC for as many as 24 out of the 25 functions. This clearly indicates the better effectiveness of ABC-AX^2 for more complex optimization tasks.

Table 7.8: Comparison of ABC-AX² with DMS-PSO [231], PSO-RDL [232], SADE [230], R-CMAES [234], and basic ABC [11] on the non-composite functions F1–F14 of the CEC2005 benchmark suite [76]. Best results are marked with boldface font.

Function	DMS-PSO	PSO-RDL	SADE	R-CMAES	ABC	ABC-AX²
F1	**0.00e+00**	2.50e–14	**0.00e+00**	5.20e–09	4.89e–17	5.63e–16
F2	1.30e–13	1.77e–13	1.05e–13	4.70e–09	4.81e–14	**1.12e–15**
F3	7.01e–09	9.6e–02	1.67e–05	**5.60e–09**	2.50e+03	4.42e–06
F4	1.8e–03	2.57e–07	1.42e–05	5.02e–09	1.50e–16	**3.49e–19**
F5	1.16e–06	2.09e–07	1.23e–02	**6.58e–09**	5.82e+01	3.62e–03
F6	6.89e–08	9.57e–01	1.20e–08	**4.87e–09**	3.31e+00	6.89e–02
F7	4.52e–02	5.73e–02	1.99e–02	**3.31e–09**	2.52e–01	2.02e–04
F8	2.00e+01	2.00e+01	2.00e+01	2.00e+01	2.03e+01	2.00e+01
F9	**0.00e+00**	1.25e+01	**0.00e+00**	2.39e–01	4.87e–17	1.47e–17
F10	3.62e+00	3.86e+01	4.97e+00	**7.96e–02**	2.22e+01	2.10e+00
F11	4.62e+00	5.58e+00	4.89e+00	9.34e–01	5.46e+00	**7.08e–02**
F12	2.40e+00	1.31e+02	**4.50e–07**	2.93e+01	9.85e+01	1.19e+01
F13	3.69e–01	8.87e–01	2.20e–01	6.96e–01	2.96e–02	**6.56e–03**
F14	2.36e+00	3.78e+00	2.92e+00	3.01e+00	3.41e+00	**2.24e+00**

Table 7.9: Comparison of ABC-AX² with DMS-PSO [231], PSO-RDL [232], SADE [230], R-CMAES [234] and the basic ABC [11] on the hybrid composition functions F15–F25 of the CEC2005 benchmark suite [76]. Best results are marked with boldface font.

Function	DMS-PSO	PSO-RDL	SADE	R-CMAES	ABC	ABC-AX²
F15	4.85e+00	2.71e+02	3.20e+01	2.28e+02	1.53e+01	**3.78e+00**
F16	9.48e+01	2.22e+02	1.01e+02	**9.13e+01**	1.75e+02	1.69e+02
F17	**1.10e+02**	2.22e+02	1.14e+02	1.23e+02	1.96e+02	1.51e+02
F18	7.61e+02	1.02e+03	7.19e+02	**3.32e+02**	4.46e+02	3.59e+02
F19	7.14e+02	9.85e+02	7.05e+02	**2.26e+02**	4.51e+02	3.07e+02
F20	8.22e+02	9.59e+02	7.13e+02	**3.00e+02**	4.38e+02	3.92e+02
F21	5.36e+02	9.94e+02	4.64e+02	5.00e+02	4.87e+02	**4.58e+02**
F22	6.92e+02	8.87e+02	7.32e+02	**7.29e+02**	8.59e+02	7.62e+02
F23	7.30e+02	1.08e+03	6.64e+02	5.59e+02	5.98e+02	**5.56e+02**
F24	2.24e+02	7.20e+02	**2.00e+02**	**2.00e+02**	2.02e+02	**2.00e+02**
F25	3.66e+02	1.76e+03	3.76e+02	3.74e+02	3.38e+02	**3.27e+02**

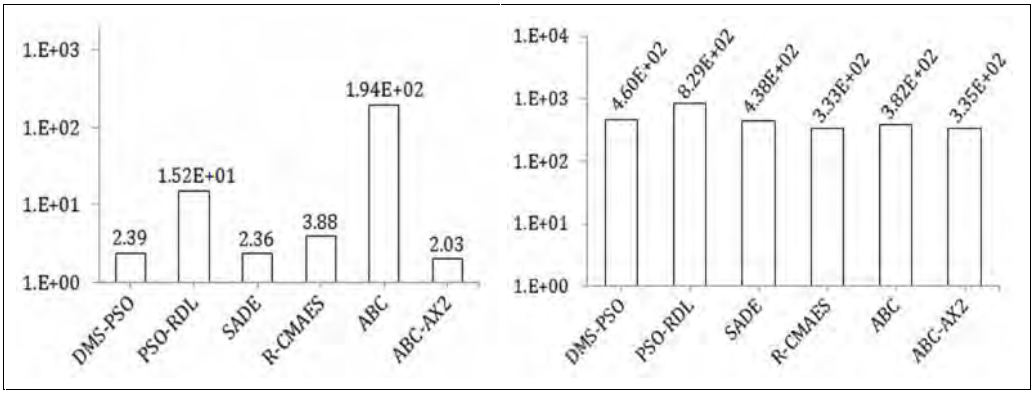


Figure 7.8: Comparison of ABC-AX^2 with DMS-PSO [231], PSO-RDL [232], SADE [230], R-CMAES [234] and the basic ABC [11] algorithms, based on their mean absolute errors on the CEC2005 benchmark functions F1-F14 (on the left) and the hybrid composition functions F15-F25 (on the right).

- The mean absolute error of the results from ABC-AX^2 on the non-composite functions F1–F14 is the smallest among all of its counterparts — DMS-PSO, PSO-RDL, SADE, R-CMAES and the basic ABC algorithm. On the hybrid composite functions F15–F25, ABC-AX^2 also shows smaller mean absolute errors in comparison to all its competitors, except the R-CMAES, which shows slightly smaller error than ABC-AX^2. Summarizing all these observations, we can conclude that the performance of ABC-AX^2 is at least comparable to, and often better than its all its competitors in comparison, i.e., DMS-PSO, PSO-RDL, SADE, R-CMAES and the basic ABC algorithm.

7.6 Conclusion and Suggestions for Further Study

This chapter introduces ABC-AX^2 — an improvement of the basic ABC algorithm [11], [75] that tries to adaptively control the degree of explorations and exploitations, separately for each candidate solution. ABC-AX^2 includes three control parameters — p_i, q_i and $\boldsymbol{\eta_i}$ within each candidate solution $\boldsymbol{x_i}$ and employs adaptive and self-adaptive techniques to adapt their values gradually. The control parameter p_i controls the proportion of exploitative and explorative perturbations on $\boldsymbol{x_i}$ and is gradually adapted by ABC-AX^2 based on the previous successes and failures of the exploitative and explorative perturbations on $\boldsymbol{x_i}$. The other two control parameters — q_i and $\boldsymbol{\eta_i}$ control the perturbation rate and perturbation scaling factors for $\boldsymbol{x_i}$ and they have to go through gradual self-adaptation, using (7.4)–(7.6).

ABC-AX^2 significantly differs from most other existing variants of ABC algorithm. Most ABC-variants view exploitations and explorations as conflicting operations, so they try to improve either the local exploitations (e.g., GABC [64], HJABC [65]) or the global explorations (e.g., CABC_S [60], CABC_H [60], DABC [61], ChABC [62]) of the basic ABC algorithm, without trying to establish a proper balance between exploitations and explorations. In contrast,

ABC-AX^2 considers exploitations and explorations to be complementary, rather than conflicting, operations and try to achieve some degree of both exploitations and explorations throughout the entire optimization procedure. For example, ABC-AX^2 keeps the value of p_i always within [0.1, 0.9] to avoid the complete domination by either exploitative or explorative perturbations. Also, ABC-AX^2 uses fixed values of u_1 and u_2 (e.g., 0.1 and 0.5, respectively, as in our implementation), so there is always significant possibility that the values of q_i and $\boldsymbol{\eta_i}$ will be randomized using (7.4) and (7.5), respectively, which can induce both explorations and exploitations on $\boldsymbol{x_i}$ throughout the entire optimization procedure. Such a design is inspired from the idea that explorations and exploitations can be complementary, rather than conflicting, to each other. For example, some exploitation is always necessary after exploring a new, unvisited search region in order to realize the potentials of the newly explored solutions. Also, long exploitations can lead to locally optimal points, so some successive explorative operations may be helpful to break free from the local optima. Evaluation results (Tables 7.1–7.9) clearly show that ABC-AX^2 has significantly improved its results over the basic ABC algorithm [11] as well as several other recent variants of ABC (e.g., [59]–[65]). Another important difference of ABC-AX^2 from many other existing algorithms is that ABC-AX^2 customizes the degree of explorations and exploitations at the individual solution level, using three control parameters separately for every candidate solution. Differentiating and customizing the degree of explorations and exploitations for every candidate solution makes it possible to allocate different and dynamic search intensity around the neighborhoods of each of the candidate solutions. Such an approach is more suitable to handle the dynamically evolving optimization scenario around each candidate solution of the population.

There may be several possible future research directions based on this study. Firstly, ABC-AX^2 uses a simple strategy to adjust the values of its control parameters — p_i, q_i and $\boldsymbol{\eta_i}$ for each candidate solution $\boldsymbol{x_i}$. A more sophisticated strategy, such as considering the properties of fitness landscape around $\boldsymbol{x_i}$, or using a strategy parameterized by the existing population diversity, maturity of the ongoing optimization procedure, density of the solutions around $\boldsymbol{x_i}$ or some other estimate of current explorative/exploitative requirements of $\boldsymbol{x_i}$ may be more effective to balance between exploitations and explorations around $\boldsymbol{x_i}$. Secondly, the quality of the final solution could be improved further by using an exploitative and efficient local searcher. This may help to pinpoint the global minimum more precisely after the execution of ABC-AX^2 is over. Thirdly, ABC-AX^2 can be hybridized with many other existing evolutionary, swarm intelligence and machine learning techniques to further improve its results. Finally, ABC-AX^2 has been applied mainly on continuous optimization problems. It would be interesting to know how well ABC-AX^2 performs on many other existing problems, especially the discrete and real world ones.

Chapter 8

Experiments

8.1 Introduction

Along the course of this thesis, we have developed a number of improved evolutionary and swarm intelligence algorithms, such as the RTEP (chapter 3), DGEP (chapter 4), ABC-SAM (chapter 5), ABC-IX (chapter 6) and ABC-AX^2 (chapter 7). The first two algorithms belong to the evolutionary algorithm family, as they are improved variants of the standard evolutionary programming (EP), while the last three algorithms belong to the swarm intelligence family, as they are based on the Artificial Bee Colony (ABC) algorithm. In this chapter we conduct a number of experiments on each of these algorithms in order to gain a better understanding of how they work, how they improve their results over the basic ABC or EP-based algorithms, their specific strengths or weaknesses (if any) and whether and how we can extend and improve them further to develop new, more effective algorithm variants.

8.2 Organization of the chapter

The rest of this chapter is organized as follows. Section 8.3 and 8.4 carries out a number of experiments on our two improved EP-variants — the RTEP and DGEP, respectively. The next three sections, i.e., sections 8.5 to 8.7 perform several experimentations on each of the three improved ABC-variants — the ABC-SAM, ABC-IX and ABC-AX^2, respectively. The next section 8.8 makes a number of comparisons among all these algorithms and tries to find out their specific strengths and weaknesses. In section 8.9, we demonstrate how the performance of these algorithms could be improved further by thoughtfully incorporating a few simple techniques within them. Finally, section 8.10 draws conclusions with a summary of this chapter.

8.3 Experiments on RTEP

In this section, we conduct a number of experiments on RTEP to examine its various aspects, such as the role and effect of its control parameters, how they influence the performance of RTEP for the three different families of benchmark functions (i.e., the unimodal and high and low dimensional multimodal functions), the role and effect of the recurring explorations and exploitations of RTEP, the effect of using neighbors and strangers during its mutations and so on. We have also tried to find an optimal proportion of explorative and exploitative operations for RTEP, separately for each of the standard benchmark functions f_1–f_{30}, as well as for each of the three different families of benchmark functions.

The main two control parameters of RTEP are K_1 and K_2, where K_1 is the length of the exploration stage and K_2 is the length of the exploitation stage. Therefore, the ratio K_2/K_1 controls the ratio of exploitations to explorations, while the individual values of K_1 and K_2 control how frequently RTEP alternates between explorations and exploitations. For example, consider these two settings of RTEP — (K_1=4, K_2=8) and (K_1=40, K_2=80). Both these settings offer the same ratio of exploitations to explorations (i.e., 2:1), but the former setting ensures very frequent alternations between explorations and exploitations, while the latter one perform slow, infrequent alternations.

Tables 8.1–8.3 and Fig. 8.1 compare the performance of RTEP on the unimodal function family f_1–f_9 with several different values of K_1 and K_2. For each function, four different values of K_1 is tested — K_1=4, 10, 20 and 50. For each particular value of K_1, eight different values of K_2 are selected such that the following eight different values of K_2/K_1 ratio (i.e., exploitation-to-exploration ratio) is tested — K_2/K_1=0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 4.0 and 5.0. For example, when K_1=10, we need to pick the values of K_2 from the following set — {5, 10, 15, 20, 25, 30, 40, 50}. In Table 8.1, the best result (i.e., minimum error) for each particular value of K_1 is marked with boldface font, while the best result for a particular function (i.e., the best result along an entire column, across 4x8=32 different values of (K_1,K_2)) is marked with underlined font. Table 8.2 tries to summarize the results of Table 8.1 by showing the best value of K_2 and K_2/K_1 ratio for each particular value of K_1. It is interesting to notice that, although we have tested eight different values K_2/K_1 ratio, only four of them are present as the optimal K_2/K_1 ratio in Table 8.2, with K_2/K_1=2.5 being the most common (14 instances), followed by K_2/K_1=3.0 (11 instances), K_2/K_1=2.0 (4 instances) and K_2/K_1=4.0 (3 instances). This means that the overall desired exploitation-to-exploration ratio for the unimodal functions f_1–f_9 is typically within [2.0, 4.0]. Table 8.2 computes the overall desired exploitation-to-exploration ratio (i.e., K_2/K_1) for the unimodal function family by taking the simple arithmetic mean of the optimal K_2/K_1 ratio values (i.e., the rightmost column of Table 8.2) over all the unimodal functions f_1–f_9, which is computed to be 2.75. This result nicely agrees with the next Table (Table 8.3) and Fig. 8.1. Table 8.3 shows

the mean absolute error values of RTEP over the unimodal functions f_1–f_9 for different K_2/K_1 ratios (i.e., exploitation-to-exploration ratios), which reveals that the best performance (minimum mean absolute error value) is achieved by RTEP with K_2/K_1=2.5 for the unimodal functions f_1–f_9. Similarly, Fig. 8.1 shows that the minimum mean absolute error value is reached by RTEP with K_2/K_1 = 2.5, and also, the error values remain satisfactorily small when the K_2/K_1 ratio is kept within [2.0, 3.0]. However, beyond this range, the error gradually starts to escalate for both increased and decreased values of K_2/K_1 ratio. This means that there should be some balance between the degree of exploitations and explorations, and increasing either of them, ignoring the other one disproportionally, can easily worsen the final solution quality and deteriorate the performance of the algorithm.

Table 8.1: Performance of RTEP(K_1, K_2) with different values of the control parameters K_1 and K_2 for the unimodal functions f_1–f_9, where K_1: length of the exploration stage, and K_2: length of the exploitation stage. The best result for each particular value of K_1 is marked with boldface font, and the best result for each function is marked with underlined boldface font.

(K_1, K_2)	**Unimodal functions f_1–f_9**								
	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9
(4,2)	7.94e–16	5.07e–10	8.63e+01	3.89e–14	7.52e+00	1.80e+00	8.76e+00	0	8.49e–18
(4,4)	7.48e–17	2.09e–11	1.94e+01	5.13e–14	6.19e–03	8.95e–01	4.78e+00	0	2.73e–25
(4,6)	7.57e–17	3.44e–11	1.28e+01	1.39e–15	3.94e–03	3.26e–01	2.18e+00	0	9.51e–36
(4,8)	2.40e–20	2.91e–12	1.93e+00	2.14e–15	2.02e–03	1.18e–01	1.65e+00	0	2.23e–37
(4,10)	1.50e–22	4.77e–13	**1.39e+00**	8.79e–15	2.39e–03	**7.70e–02**	8.74e–01	0	5.12e–41
(4,12)	5.70e–22	**9.26e–15**	9.90e+00	**6.97e–16**	**2.41e–05**	7.80e–01	**6.84e–01**	0	**1.12e–46**
(4,16)	**6.37e–24**	4.70e–13	8.57e–01	8.33e–15	9.93e–05	8.30e–01	1.57e+00	0	2.17e–35
(4,20)	4.38e–20	6.38e–12	7.07e–01	1.14e–13	6.02e–04	5.77e+00	3.20e+00	0	7.71e–28
(10,5)	2.59e–15	4.46e–09	4.65e+01	2.17e–14	1.61e–01	8.10e+00	1.70e+00	0	3.60e–38
(10,10)	7.62e–15	8.49e–10	2.03e+00	7.39e–13	1.27e–02	1.40e+00	1.42e+00	0	4.18e–37
(10,15)	2.17e–17	3.54e–10	7.04e+00	3.94e–13	8.79e–04	9.69e–01	6.05e–01	0	7.84e–35
(10,20)	8.35e–17	5.17e–12	9.41e–01	5.58e–15	4.45e–04	6.55e–01	**4.34e–02**	0	6.13e–38
(10,25)	1.29e–18	1.69e–12	**4.10e–01**	**1.80e–15**	**8.10e–05**	8.04e–01	7.91e–02	0	**6.72e–44**
(10,30)	5.27e–20	**8.01e–13**	8.66e–01	2.52e–15	7.40e–04	**5.12e–01**	5.29e–01	0	3.70e–35
(10,40)	**1.76e–21**	9.02e–12	6.00e+00	2.66e–13	8.94e–04	1.02e+00	2.46e–01	0	7.15e–31
(10,50)	4.14e–20	2.26e–10	1.67e+02	5.36e–13	8.21e–04	8.63e+00	1.12e+00	0	3.74e–25
(20,10)	7.31e–12	5.59e–08	1.21e+02	2.19e–10	2.52e–01	8.69e+00	3.61e+00	0	1.36e–16
(20,20)	8.65e–14	2.46e–09	2.81e+01	5.22e–12	8.09e–02	1.18e+01	1.30e–01	0	6.21e–21
(20,30)	4.55e–15	3.12e–10	1.27e+01	9.54e–13	2.54e–04	3.07e+00	1.10e–01	0	3.23e–29
(20,40)	8.34e–16	7.21e–09	5.50e+00	**2.60e–14**	2.33e–04	5.25e+00	3.53e–01	0	9.73e–32
(20,50)	9.30e–16	4.48e–09	**2.42e+00**	4.21e–14	**1.76e–04**	1.27e+00	**9.38e–02**	0	**8.49e–35**

Table 8.1 (continued)

(K_1, K_2)	**Unimodal functions f_1–f_9**								
	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9
(20,60)	4.90e–18	**2.00e–10**	6.54e+00	3.89e–11	9.65e–03	**7.78e–01**	7.40e–01	0	5.62e–31
(20,80)	**4.11e–19**	9.14e–10	1.59e+01	7.77e–10	1.31e–01	4.27e+00	3.22e+00	0	4.87e–30
(20,100)	1.81e–17	6.32e–08	5.20e+01	4.81e–08	9.12e–02	6.37e+00	3.32e+00	0	2.75e–27
(50,25)	6.69e–09	9.64e–07	2.19e+02	6.35e–08	5.33e–01	2.45e+01	1.27e+01	0	3.15e–21
(50,50)	9.61e–11	1.79e–08	4.95e+01	6.86e–10	4.73e–03	1.26e+01	3.76e+00	0	7.62e–25
(50,75)	7.75e–12	4.02e–09	1.58e+01	7.76e–11	4.58e–03	3.92e+00	9.58e–01	0	1.32e–27
(50,100)	9.87e–12	9.61e–10	8.10e+00	7.44e–12	**4.82e–04**	4.75e+00	**5.72e–01**	0	5.51e–30
(50,125)	2.04e–14	**8.35e–10**	**6.42e+00**	**3.96e–12**	9.39e–03	**1.52e+00**	8.68e–01	0	1.41e–35
(50,150)	**8.47e–15**	4.67e–08	9.81e+00	4.69e–10	9.24e–02	6.24e+00	3.43e+00	0	**6.61e–36**
(50,200)	9.23e–15	1.74e–09	1.41e+01	3.18e–08	4.62e–02	1.77e+01	8.46e+00	0	6.54e–32
(50,250)	9.26e–12	5.39e–08	8.80e+01	3.84e–07	3.35e+00	1.67e+01	1.54e+01	0	9.46e–27

Table 8.2: For each particular value of K_1 in the previous Table 8.1, the optimal value of K_2 and the corresponding K_2/K_1 ratio, where K_1 and K_2 are the lengths of the exploration and exploitation stages, respectively.

Functions (Unimodal)	**For each K_1, optimal K_2 and K_2/K_1 ratio**								Mean optimal K_2/K_1 ratio (for each function)
	K_1=4		K_1=10		K_1=20		K_1=50		
	Optimal K_2	Optimal K_2/K_1	Optimal K_2	Optimal K_2/K_1	Optimal K_2	Optimal K_2/K_1	Optimal K_2	Optimal K_2/K_1	
f_1	16	4.0	40	4.0	80	4.0	150	3.0	3.75
f_2	12	3.0	30	3.0	60	3.0	125	2.5	2.88
f_3	10	2.5	25	2.5	50	2.5	125	2.5	2.50
f_4	12	3.0	25	2.5	40	2.0	125	2.5	2.50
f_5	12	3.0	25	2.5	50	2.5	100	2.0	2.50
f_6	10	2.5	30	3.0	60	3.0	125	2.5	2.75
f_7	12	3.0	20	2.0	50	2.5	100	2.0	2.38
f_9	12	3.0	25	2.5	50	2.5	150	3.0	2.75
Overall optimal K_2/K_1 ratio for the unimodal functions f_1–f_9 ==>									**2.75**

Table 8.3: The mean absolute error values of RTEP(K_1, K_2) on the unimodal functions f_1–f_9 for different values of K_2/K_1 ratio (i.e., exploitation-to-exploration ratio) in the previous Table 8.1. The best performance (i.e., minimum error value) is achieved by RTEP with $K_2/K_1 = 2.5$.

Function family	K_2/K_1 ratio	Mean absolute error
Unimodal functions f_1–f_9	0.5	15.31
	1.0	3.78
	1.5	1.68
	2.0	0.83
	2.5	0.45
	3.0	1.14
	4.0	2.07
	5.0	10.32

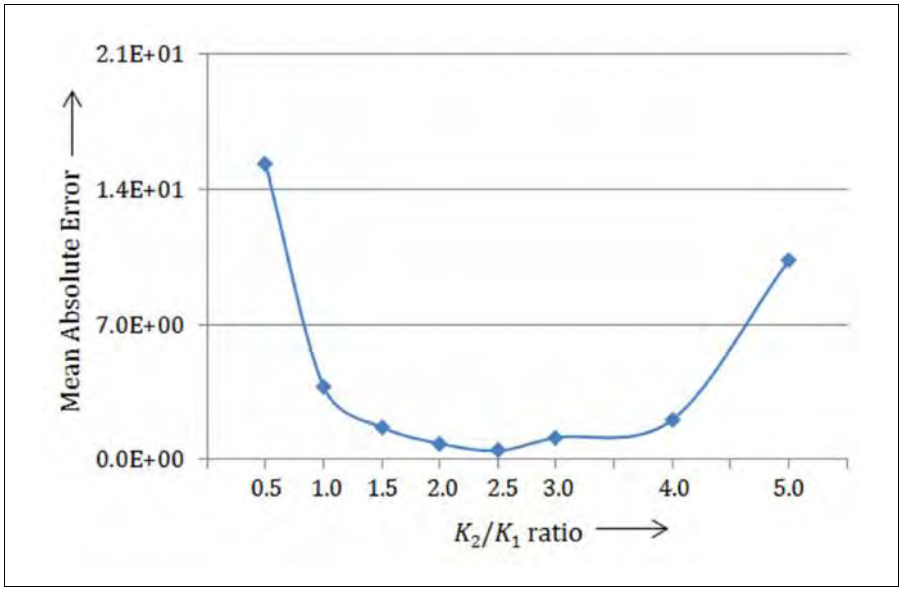


Figure 8.1: The mean absolute error values of the final results from RTEP(K_1,K_2) with eight different values of the K_2/K_1 ratio (i.e., exploitation-to-exploration ratio) for the unimodal functions f_1–f_9. The best result (i.e., lowest error) is reached by RTEP(K_1,K_2) with $K_2/K_1 = 2.5$.

Tables 8.4–8.6 and Fig. 8.2 compare the performance of RTEP for different values of K_1 and K_2 on the high dimensional multimodal functions f_{10}–f_{18}. These experiments are just identical to the previous ones (Tables 8.1–8.3, Fig. 8.1), but now the multimodal functions f_{10}–f_{18} are tested, instead of the unimodal functions f_1–f_9. Similar to the previous experiment (Table 8.1), eight different values of K_2/K_1 ratio is tested — $K_2/K_1 = 0.5$, 1.0, 1.5, 2.0, 2.5, 3.0, 4.0 and 5.0. Table 8.5 summarizes the results of Table 8.4 by showing the best value of the K_2/K_1 ratio for each particular value of K_1 for each function. As Table 8.5 reveals, for these high dimensional multimodal functions f_{10}–f_{18}, the most effective exploitation-to-exploration ratio is 2.0, as $K_2/K_1 = 2.0$ in most (i.e., 22 out of 36) instances in Table 8.5. The overall optimal K_2/K_1 ratio for these multimodal functions is also computed to be 1.86 (Table 8.5), which is close to 2.0. Table 8.6 and Fig. 8.2 also show that, over these multimodal functions f_{10}–f_{18}, the mean absolute error value of RTEP reaches its minimum when K_2/K_1 is in [1.5, 2.0], which nicely agrees with the computed overall optimal value of K_2/K_1 (i.e., 1.86). This value indicates that the high dimensional multimodal functions (f_{10}–f_{18}) generally require relatively more explorations than the previous unimodal functions f_1–f_9, for which the overall optimal exploitation-to-exploration ratio (i.e., K_2/K_1 ratio) was computed to be 2.75 (Table 8.2).

Table 8.4: Performance of RTEP(K_1,K_2) with different values of the control parameters K_1 and K_2 for the high dimensional multimodal functions f_{10}–f_{18}, where K_1: length of the exploration stage, and K_2: length of the exploitation stage. The best result for each particular value of K_1 is marked with boldface font, and the best result for each function is marked with underlined boldface font.

(K_1,K_2)	**Multimodal functions (high dimensional) f_{10}–f_{18}**								
	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}	f_{17}	f_{18}
(4,2)	7.36e–12	6.27e–04	7.07e+02	1.78e–08	7.73e–18	9.25e–09	5.95e–05	1.87e–09	1.07e–03
(4,4)	4.48e–13	5.55e–04	7.57e+02	2.56e–08	1.46e–19	2.27e–10	7.92e–07	6.58e–10	4.84e–03
(4,6)	**9.67e–16**	3.87e–05	**2.74e+02**	7.88e–09	4.77e–19	**1.75e–12**	**5.58e–08**	1.22e–12	1.24e–04
(4,8)	1.88e–14	**1.14e–06**	3.56e+02	**2.43e–09**	**8.41e–20**	2.89e–12	6.10e–08	**1.75e–13**	**7.23e–05**
(4,10)	7.19e–14	5.40e–03	6.95e+02	1.51e–08	9.14e–20	3.85e–12	1.58e–07	4.31e–12	6.40e–04
(4,12)	1.91e–11	6.55e–03	8.96e+02	1.44e–07	8.85e–18	9.73e–11	5.38e–07	9.08e–12	7.73e–03
(4,16)	1.31e–10	5.21e–03	1.28e+03	1.69e–07	2.71e–17	1.47e–10	7.76e–05	6.66e–10	8.40e–03
(4,20)	4.06e–10	1.16e–02	1.89e+03	2.27e–07	9.50e–16	7.25e–10	4.60e–04	1.98e–10	5.52e–03
(10,5)	5.77e–12	1.32e–03	1.57e+03	3.55e–06	3.88e–11	6.82e–08	8.17e–04	3.15e–09	4.96e–03
(10,10)	5.48e–12	5.68e–03	9.36e+02	1.21e–08	7.20e–15	7.70e–10	5.26e–05	1.61e–10	5.90e–04
(10,15)	2.95e–13	**2.85e–05**	6.82e+02	2.86e–08	1.63e–15	6.33e–09	5.02e–05	4.81e–11	7.26e–04
(10,20)	**9.50e–14**	6.66e–05	**5.64e+02**	**1.23e–09**	2.91e–16	8.94e–10	**1.60e–07**	**9.59e–13**	**1.09e–04**
(10,25)	2.49e–12	1.51e–04	7.11e+02	8.07e–09	**1.07e–16**	**6.00e–10**	9.99e–06	4.88e–12	6.30e–04
(10,30)	6.04e–10	8.62e–04	1.08e+03	5.56e–08	1.73e–15	7.37e–09	6.83e–05	1.28e–11	6.32e–04
(10,40)	8.87e–10	1.97e–02	1.39e+03	5.83e–07	7.79e–16	1.54e–09	8.87e–05	1.34e–09	7.74e–04
(10,50)	3.19e–09	5.14e–01	1.78e+03	1.55e–07	7.24e–15	8.13e–08	8.05e–05	9.38e–09	7.76e–04
(20,10)	9.80e–11	7.45e–02	1.71e+03	9.42e–06	9.12e–05	3.35e–08	5.50e–03	7.02e–08	2.03e–03
(20,20)	8.90e–11	7.37e–03	9.94e+02	**2.73e–08**	1.99e–08	1.46e–09	2.91e–04	3.94e–10	4.67e–03
(20,30)	1.76e–12	4.35e–04	**6.69e+02**	7.27e–08	9.74e–09	9.76e–09	9.72e–06	5.34e–10	6.17e–03
(20,40)	**9.55e–14**	**2.10e–04**	9.72e+02	8.61e–08	4.95e–11	**6.94e–10**	**9.85e–07**	**5.57e–11**	**7.56e–04**
(20,50)	3.32e–13	1.58e–03	1.28e+03	6.61e–06	**5.70e–13**	8.06e–10	8.36e–05	8.84e–10	1.86e–03
(20,60)	8.14e–10	5.61e–03	1.60e+03	5.87e–06	9.07e–12	3.49e–09	9.80e–04	9.51e–09	2.70e–03
(20,80)	1.45e–10	6.13e–02	2.31e+03	1.10e–06	1.97e–08	9.74e–08	1.49e–04	6.28e–07	4.95e–03
(20,100)	8.58e–09	1.72e–01	2.64e+03	4.69e–06	1.87e–10	1.29e–08	3.74e–01	2.78e–06	9.66e–03
(50,25)	1.76e–08	1.70e–01	2.29e+03	1.84e–01	5.45e–06	7.49e–05	1.42e–02	1.39e–06	2.56e–02
(50,50)	3.28e–09	7.99e–02	1.55e+03	4.02e–02	1.07e–07	3.14e–06	2.71e–03	1.98e–08	2.90e–03
(50,75)	7.86e–13	**7.74e–04**	9.67e+02	**5.18e–05**	3.58e–08	**4.12e–09**	**9.30e–04**	8.38e–10	2.55e–03
(50,100)	**4.66e–13**	8.79e–04	**8.40e+02**	8.95e–05	**1.65e–10**	9.30e–09	5.05e–03	**4.37e–10**	1.66e–03
(50,125)	1.60e–12	4.25e–02	1.23e+03	5.54e–04	4.62e–08	9.08e–09	1.07e–01	9.06e–09	**1.36e–03**
(50,150)	2.33e–10	1.81e–01	1.98e+03	1.83e–03	1.34e–07	9.94e–08	1.04e+00	4.08e–07	1.50e–02
(50,200)	1.72e–08	6.61e+00	2.45e+03	1.52e+00	9.28e–06	6.54e–06	4.47e+00	4.60e–04	5.86e–02
(50,250)	1.13e–07	6.46e+00	3.24e+03	2.13e+00	7.67e–06	1.89e–06	1.58e+01	7.79e–02	5.73e–02

Table 8.5: For each particular value of K_1 in the previous Table 8.4, the optimal value of K_2 and the corresponding K_2/K_1 ratio.

Functions (Multimodal, high dimensional)	**For each K_1, optimal K_2 and K_2/K_1 ratio**								Mean optimal K_2/K_1 ratio (for each function)
	K_1=4		K_1=10		K_1=20		K_1=50		
	Optimal K_2	Optimal K_2/K_1	Optimal K_2	Optimal K_2/K_1	Optimal K_2	Optimal K_2/K_1	Optimal K_2	Optimal K_2/K_1	
f_{10}	6	1.5	20	2.0	40	2.0	100	2.0	1.88
f_{11}	8	2.0	15	1.5	40	2.0	75	1.5	1.75
f_{12}	6	1.5	20	2.0	30	1.5	100	2.0	1.75
f_{13}	8	2.0	20	2.0	20	1.0	75	1.5	1.63
f_{14}	8	2.0	25	2.5	50	2.5	100	2.0	2.25
f_{15}	6	1.5	25	2.5	40	2.0	75	1.5	1.88
f_{16}	6	1.5	20	2.0	40	2.0	75	1.5	1.75
f_{17}	8	2.0	20	2.0	40	2.0	100	2.0	2.00
f_{18}	8	2.0	20	2.0	40	2.0	125	2.5	1.88
Overall Optimal K_2/K_1 ratio for Multimodal (high dimensional) functions f_{10}–f_{18} ==>									**1.86**

Table 8.6: The mean absolute errors of RTEP(K_1,K_2) on high dimensional multimodal functions f_{10}–f_{18} for different values of K_2/K_1 ratio (i.e., exploitation-to-exploration ratio) in the previous Table 8.4. The best performance (i.e., lowest error) is achieved by RTEP with K_2/K_1 = 1.5.

Function family	K_2/K_1 ratio	Mean absolute error
Multimodal functions (high dimensional) f_{10}–f_{18}	0.5	1.74e+02
	1.0	1.17e+02
	1.5	7.20e+01
	2.0	7.59e+01
	2.5	1.09e+02
	3.0	1.54e+02
	4.0	2.07e+02
	5.0	2.66e+02

Tables 8.7–8.9 and Fig. 8.3 perform the same experiments, as of Tables 8.4–8.6 and Fig. 8.2, but now on the low dimensional multimodal functions f_{19}–f_{30}. Table 8.7 presents the mean error of RTEP with several different values of K_1 and K_2, each time taking eight different values of the exploitation-to-exploration ratio (i.e., K_2/K_1) — K_2/K_1 = 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 4.0 and 5.0. The next Table (Table 8.8) summarizes the results in Table 8.7 by showing the best found value of the K_2/K_1 ratio for each function. For these functions, the most suitable value of the exploitation-to-exploration ratio is found to be 2.0, as the value K_2/K_1 is mostly (i.e., 20 out of 28 instances) 2.0 in Table 8.8. The overall optimal K_2/K_1 ratio for these low dimensional multimodal functions f_{19}–f_{30} is computed to be 1.96. Table 8.9 and Fig. 8.3 show the mean absolute error values of RTEP for different values of the K_2/K_1 ratio, where the best performance (i.e., minimum mean absolute error) of RTEP is observed for K_2/K_1=2.0, which closely follows the computed overall optimal value of K_2/K_1 (i.e., 1.96) in Table 8.8.

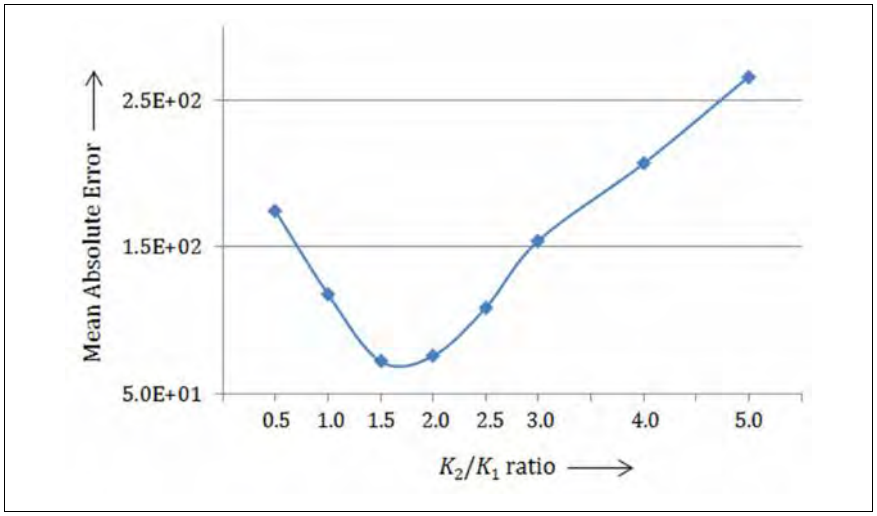


Figure 8.2: The mean absolute error values from RTEP(K_1,K_2) with eight different values of the K_2/K_1 ratio (exploitation-to-exploration ratio) for the high dimensional multimodal functions f_{10}–f_{18}. The best result (i.e., lowest error) is reached by RTEP(K_1,K_2) when K_2/K_1in [1.5, 2.0].

Table 8.7: Performance of RTEP(K_1,K_2) with different values of the control parameters K_1 and K_2 for the low dimensional multimodal functions f_{19}–f_{30}, where K_1: length of the exploration stage, and K_2: length of the exploitation stage. The best result for each particular value of K_1 is marked with boldface font, and the best result for each function is marked with underlined boldface font.

(K_1,K_2)	**Multimodal functions (low dimensional) f_{19}–f_{30}**								
	f_{19}	f_{20}	f_{21}–f_{24}	f_{25}	f_{26}	f_{27}	f_{28}	f_{29}	f_{30}
(4,2)	0.005	0.0013	0	0.49	0.41	0.37	0.53	0	0.41
(4,4)	0.004	0.0012	0	0.39	0.43	0.31	0.50	0	0.32
(4,6)	**0.002**	0.0010	0	**0.33**	**0.38**	0.25	0.45	0	0.29
(4,8)	**0.002**	**0.0008**	0	0.37	0.40	**0.22**	0.42	0	**0.26**
(4,10)	**0.002**	**0.0008**	0	0.40	0.48	0.23	**0.40**	0	0.28
(4,12)	0.004	0.0001	0	0.45	0.59	0.28	0.42	0	0.40
(4,16)	0.004	0.0011	0	0.53	0.62	0.29	0.46	0	0.45
(4,20)	0.005	0.0012	0	0.58	0.60	0.32	0.55	0	0.42
(10,5)	0.005	0.0013	0	0.62	0.52	0.27	0.59	0	0.41
(10,10)	0.006	0.0012	0	0.52	0.50	0.26	0.55	0	0.32
(10,15)	0.006	0.0012	0	0.49	0.49	0.22	0.48	0	0.29
(10,20)	**0.005**	0.0013	0	**0.44**	0.43	**0.20**	**0.40**	0	**0.26**
(10,25)	**0.005**	**0.0009**	0	0.51	**0.42**	0.22	0.47	0	0.28
(10,30)	0.006	**0.0009**	0	0.55	0.47	0.25	0.52	0	0.40
(10,40)	0.009	0.0011	0	0.63	0.52	0.33	0.58	0	0.45
(10,50)	0.013	0.0012	0	0.70	0.58	0.33	0.65	0	0.42
(20,10)	0.010	0.0033	0	0.55	0.64	0.37	0.56	0	0.41
(20,20)	0.008	0.0032	0	0.49	0.55	0.31	0.45	0	0.32

Table 8.7 (continued)

(K_1, K_2)	**Multimodal functions (low dimensional) f_{19}–f_{30}**								
	f_{19}	f_{20}	f_{21}–f_{24}	f_{25}	f_{26}	f_{27}	f_{28}	f_{29}	f_{30}
(20,30)	0.007	0.0030	0	**0.42**	0.54	0.26	0.46	0	0.29
(20,40)	**0.006**	**0.0027**	0	0.44	**0.50**	**0.25**	**0.46**	0	**0.28**
(20,50)	0.005	0.0028	0	0.50	0.54	0.25	0.47	0	0.29
(20,60)	0.008	0.0032	0	0.53	0.57	0.29	0.50	0	0.40
(20,80)	0.009	0.0033	0	0.60	0.58	0.36	0.58	0	0.45
(20,100)	0.013	0.0042	0	0.66	0.65	0.44	0.63	0	0.42
(50,25)	0.008	0.0033	0	0.70	0.61	0.45	0.66	0	0.58
(50,50)	0.010	0.0030	0	0.63	0.59	0.38	0.59	0	0.53
(50,75)	0.012	**0.0026**	0	0.56	0.58	**0.32**	0.55	0	**0.46**
(50,100)	**0.012**	0.0029	0	**0.52**	**0.54**	0.36	**0.50**	0	0.48
(50,125)	0.015	0.0032	0	0.55	0.57	0.38	0.53	0	0.50
(50,150)	0.023	0.0035	0	0.63	0.59	0.44	0.60	0	0.55
(50,200)	0.023	0.0037	0	0.66	0.65	0.49	0.67	0	0.63
(50,250)	0.028	0.0040	0	0.76	0.69	0.54	0.73	0	0.66

Table 8.8: For each particular value of K_1 in the previous Table 8.7, the optimal value of K_2 and the corresponding K_2/K_1 ratio, where K_1 and K_2 are the lengths of the exploration and exploitation stages, respectively.

Function (Multimodal, low dimensional)	**For each K_1, optimal K_2 and K_2/K_1 ratio**								Mean optimal K_2/K_1 ratio (for each function)
	K_1=4		K_1=10		K_1=20		K_1=50		
	Optimal K_2	Optimal K_2/K_1	Optimal K_2	Optimal K_2/K_1	Optimal K_2	Optimal K_2/K_1	Optimal K_2	Optimal K_2/K_1	
f_{19}	8	2.0	20	2.0	40	2.0	100	2.0	2.00
f_{20}	8	2.0	25	2.5	40	2.0	75	1.5	2.00
f_{25}	6	1.5	20	2.0	30	1.5	100	2.0	1.75
f_{26}	6	1.5	25	2.5	40	2.0	100	2.0	2.00
f_{27}	8	2.0	20	2.0	40	2.0	75	1.5	1.88
f_{28}	10	2.5	20	2.0	40	2.0	100	2.0	2.13
f_{30}	8	2.0	20	2.0	40	2.0	100	2.0	2.00
Overall Optimal K_2/K_1 ratio for Multimodal (low dimensional) functions f_{19}–f_{30} ==>									**1.96**

Table 8.9: The mean absolute error of RTEP(K_1, K_2) on low dimensional multimodal functions f_{19}–f_{30} for different values of the K_2/K_1 ratio (exploitation-to-exploration ratio) in the previous Table 8.7. The best performance (i.e., lowest error) is achieved by RTEP(K_1, K_2) with $K_2/K_1 = 2.0$.

Function family	K_2/K_1 ratio	Mean absolute error
Multimodal functions (low dimensional) f_{19}–f_{30}	0.5	0.261
	1.0	0.230
	1.5	0.209
	2.0	0.199
	2.5	0.213
	3.0	0.243
	4.0	0.271
	5.0	0.292

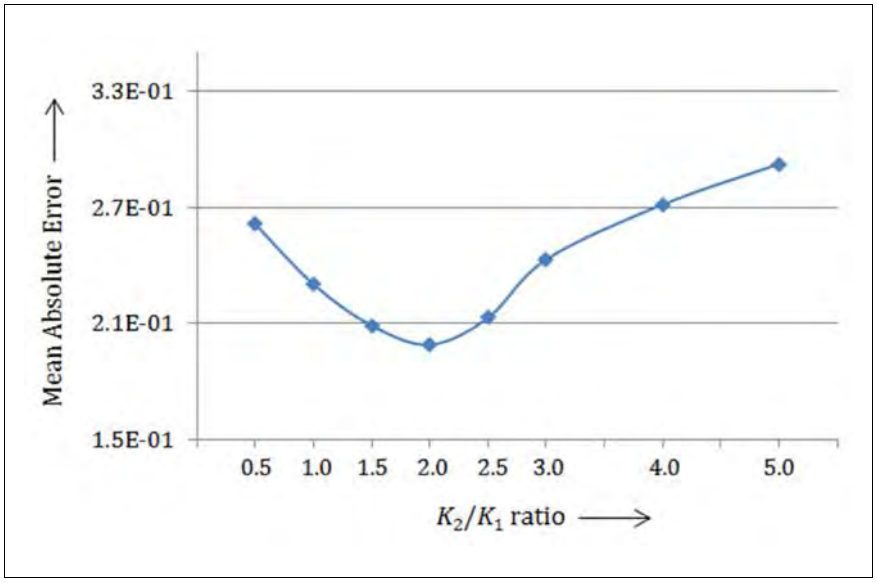


Figure 8.3: The mean absolute error values of the final results from RTEP(K_1,K_2) using eight different values of the K_2/K_1 ratio (exploitation-to-exploration ratio) for the multimodal functions (low dimensional) f_{19}–f_{30}. The best result (lowest error) is achieved with $K_2/K_1 \approx 2.0$.

Table 8.10: The optimal exploitation-to-exploration ratio (i.e., the K_2/K_1 ratio) with the mean absolute error values of RTEP for the three different function families — the unimodal (f_1–f_9), high dimensional multimodal (f_{10}–f_{18}) and low dimensional multimodal (f_{19}–f_{30}) functions.

Function family	Optimal K_2/K_1 ratio	Mean Absolute Error
Unimodal functions (f_1–f_9)	2.5	0.45
Multimodal (high dimensional) functions (f_{10}–f_{18})	1.5	7.20e+01
Multimodal (low dimensional) functions (f_{19}–f_{30})	2.0	0.199

In the previous Tables 8.3, 8.6 and 8.9, we have presented the mean absolute error values of RTEP for different exploitation-to-exploration ratio (i.e., the K_2/K_1 ratio) for the three different families of the standard benchmark functions — the unimodal functions f_1–f_9, the high dimensional multimodal functions f_{10}–f_{18} and the low dimensional multimodal functions f_{19}–f_{30}. The next Table 8.10 summarizes Tables 8.3, 8.6 and 8.9 by presenting the overall optimal K_2/K_1 ratio and the corresponding mean absolute error values of RTEP for these three different function families.

All of the previous Tables 8.1–8.10 try to find an optimal K_2/K_1 ratio (i.e., exploitation-to-exploration ratio) for the different functions. But what should be the individually optimal values of K_1 and K_2 (rather than K_2/K_1)? In the next Table 8.11, we try to find out the optimal values of K_1 and K_2, instead of the optimal K_2/K_1 ratio. In Table 8.11, we present the results of several

RTEP variants, each one using a different value of (K_1, K_2). It is interesting to discover that the best results are always achieved for small (e.g., $K_1 = K_2 = 5$) or moderate ($K_1 = K_2 = 10 \sim 20$) values of the K_1 and K_2. Such small stage lengths of RTEP have always performed better than the larger stage lengths, as also observed in previous Tables 8.1, 8.4 and 8.7. This is reasonable because when the stage lengths are increased (e.g., $K_1 = K_2 = 100$), the recurring nature of RTEP starts to diminish, forcing RTEP to operate in somewhat sequential (i.e., non-recurring) manner. Actually, what seems to be important to RTEP is not the length of the stages but their recurring nature. A frequent alternation to and from explorations and exploitations is more effective for RTEP, as demonstrated by the much improved results with (K_1, K_2) = (5,5) or (10,10), in comparison to (K_1, K_2) = (50,50) or (100,100). To examine the effect of delayed, infrequent alternations, we now present another experiment on RTEP (Table 8.12) by introducing a new variant of RTEP — the Sequential Two-Stage Evolutionary Programming (STEP). STEP is equivalent to RTEP(K_1, K_2) with $K_1 = K_2 = TN/2$, where TN is the total number of generations to run. This means STEP executes the explorative and exploitative stages sequentially, one after another, each one for the half of the total length of the execution. For the results in Table 8.12, STEP is run with population size = 100 for a total of 1000 generations, with the first half (i.e., first 500 generations) as the exploration stage of RTEP, followed by its exploitation stage for the remaining 500 generations. Table 8.12 shows the results of STEP, averaged over 50 independent runs. A simple bare eyed comparison of the results presented in Table 8.12 with those presented in Table 8.11, as well as the highest mean absolute error value of STEP in Fig. 8.4, indicates that the sequential STEP variant performs much worse than its recurring counterpart — the RTEP. The difference is often by several orders of magnitude and obviously statistically significant. This proves the necessity of interleaving and mixing the explorative and exploitative operations, instead of trying to make a perfect switching from one stage to another.

Table 8.11: Performance of RTEP with different values of the control parameters K_1 and K_2. Here, K_1: length of the exploration stage, and K_2: length of the exploitation stage. The best result for each function is marked with boldface font.

Function	Mean Error				
	RTEP (5,5)	**RTEP (10,10)**	**RTEP (20,20)**	**RTEP (50,50)**	**RTEP (100,100)**
f_1	**1.5e–14**	2.7e–13	7.2e–11	8.5e–10	6.1e–10
f_2	**2.3e–11**	3.3e–09	1.6e–08	6.0e–03	8.5e–01
f_3	**2.3e+00**	4.9e+00	6.6e+00	1.8e+01	3.8e+01
f_4	**7.5e–14**	8.2e–11	2.0e–10	5.0e–08	9.3e–05
f_5	7.7e–03	**2.5e–03**	8.4e–02	2.0e+00	5.6e+00
f_6	6.1e–01	**4.8e–01**	1.5e+00	6.0e+00	1.7e+01
f_7	**2.1e+00**	4.7e+00	3.0e+01	2.9e+01	1.0e+03
f_8	0	0	0	0	0
f_9	**6.1e–37**	2.6e–36	3.9e–31	3.8e–28	2.2e–25

Table 8.11 (continued)

Function	Mean Error				
	RTEP (5, 5)	**RTEP (10, 10)**	**RTEP (20, 20)**	**RTEP (50, 50)**	**RTEP (100, 100)**
f_{10}	**1.7e–11**	2.5e–09	7.4e–08	1.1e–06	9.6e–04
f_{11}	**9.6e–06**	2.0e–05	4.8e–02	2.6e–01	9.3e–01
f_{12}	4.3e–05	**3.7e–06**	2.2e–05	4.1e–02	2.1e–01
f_{13}	**6.8e–09**	2.3e–07	1.9e–07	4.6e–06	2.5e–04
f_{14}	8.4e–17	1.7e–16	**1.1e–17**	2.5e–12	7.0e–11
f_{15}	**9.7e–11**	6.7e–09	5.1e–08	4.4e–05	4.7e–04
f_{16}	**4.5e–08**	8.6e–07	2.6e–07	7.5e–05	2.3e–05
f_{17}	2.1e–12	**1.9e–13**	5.4e–13	2.5e–11	9.2e–08
f_{18}	2.9e–04	**1.8e–06**	1.4e–05	9.7e–04	5.7e–02
f_{19}	**0.004**	0.004	0.007	0.012	0.015
f_{20}	**0.0012**	0.0017	0.0028	0.0033	0.0045
f_{23}	**0.00**	**0.00**	0.07	0.18	0.22
f_{24}	**0.00**	**0.00**	0.10	0.15	0.28
f_{25}	**0.38**	0.41	0.52	0.55	0.56
f_{26}	**0.33**	0.40	0.56	0.48	0.51
f_{27}	0.31	**0.25**	0.39	0.82	1.27
f_{28}	**0.42**	0.48	0.88	0.92	1.04
f_{29}	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
f_{30}	**0.25**	0.28	0.45	0.44	0.52

Table 8.12: Performance of STEP, based on the final solution quality (mean error), on the standard benchmark functions f_1–f_{30}. Results have been averaged over 50 independent runs.

Function	STEP (Mean Error)	Function	STEP (Mean Error)
f_1	8.3e–04	f_{15}	2.6e–02
f_2	5.6e–03	f_{16}	1.0e–02
f_3	5.5e+01	f_{17}	9.8e–01
f_4	7.4e–02	f_{18}	5.2e–03
f_5	3.7e+02	f_{19}	0.025
f_6	5.3e+01	f_{20}	0.036
f_7	3.7e+02	f_{23}	1.21
f_8	0	f_{24}	0.87
f_9	6.0e–02	f_{25}	1.77
f_{10}	1.1e+01	f_{26}	1.61
f_{11}	8.5e+00	f_{27}	1.87
f_{12}	2.8e+01	f_{28}	1.26
f_{13}	1.3e+00	f_{29}	0.05
f_{14}	7.1e–01	f_{30}	0.88

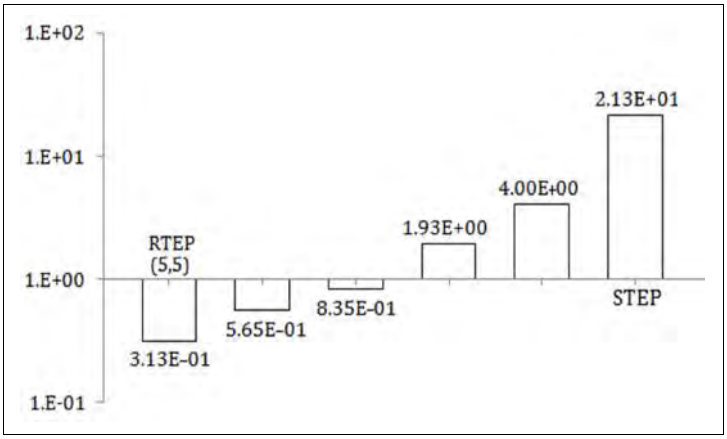


Figure 8.4: The mean absolute error values of the final results from RTEP(5,5), RTEP(10,10), RTEP(20,20), RTEP(50,50), RTEP(100,100) and STEP, shown in the same order, from the left to the right. RTEP(5,5) shows the best performance, i.e., the minimum mean absolute error value, while the sequential variant — STEP shows the worst performance with the highest error value.

Table 8.13: Comparison between RTEP(2,4) and Naïve-RTEP, based on their final solution quality, on the standard benchmark functions f_1–f_{30}. Results have been averaged over 50 independent runs. For each function, the best result is marked with boldface font.

Function	Mean Error		Function	Mean Error	
	Naïve-RTEP	RTEP(2,4)		Naïve-RTEP	RTEP(2,4)
f_1	6.2e–10	**7.5e–18**	f_{16}	1.9e–03	**2.2e–07**
f_2	3.2e–07	**1.7e–09**	f_{17}	8.5e–05	**3.2e–13**
f_3	6.6e+00	**1.7e+00**	f_{18}	4.2e–06	**7.1e–08**
f_4	1.8e–06	**2.4e–15**	f_{19}	**0.002**	**0.002**
f_5	6.2e–02	**2.6e–03**	f_{20}	4.2e–02	**0.0004**
f_6	5.1e–02	**1.3e–01**	f_{21}	**0.00**	**0.00**
f_7	1.9e+01	**1.1e+00**	f_{22}	0.73	**0.00**
f_8	**0.00**	**0.00**	f_{23}	0.73	**0.03**
f_9	8.3e–04	**8.0e–38**	f_{24}	**0.00**	**0.00**
f_{10}	8.2e–02	**2.5e–14**	f_{25}	0.90	**0.55**
f_{11}	2.0e–03	**2.9e–07**	f_{26}	0.75	**0.38**
f_{12}	3.9e+03	**7.1e+02**	f_{27}	0.90	**0.25**
f_{13}	6.5e–02	**2.0e–10**	f_{28}	1.62	**0.85**
f_{14}	3.2e–10	**2.7e–25**	f_{29}	**0.00**	**0.00**
f_{15}	8.6e–06	**7.8e–10**	f_{30}	0.82	**0.29**

Now we examine whether the usage of neighbors and strangers is necessary for the improved performance of RTEP. RTEP uses the distance between neighbors and strangers as the standard deviations of its Gaussian mutations. Therefore, RTEP has to keep track of the

neighbors and strangers for every individual of the population throughout the evolution. To examine whether the usage of neighbors and strangers have contributed significantly for the better performance of RTEP, we introduce here another variant of RTEP — the naïve-RTEP, which does not use neighbors or strangers; instead, it selects an individual randomly from the rest of the population. The distance of this random individual from the current individual is then used as the standard deviation for the Gaussian mutation of RTEP. The results of RTEP $(2,4)$ and naïve-RTEP with population size = 50 and number of function evaluations = 150,000 are presented and compared in Table 8.13. It is apparent from the results that naïve-RTEP performed much worse than RTEP on most (25 out of 30) of the functions. The performance gap is often by several orders of magnitude and obviously statistically significant. This indicates the necessity of using proper standard deviation and directional information for better performance and more effective explorations and exploitations during the optimization process.

8.4 Experiments on DGEP

In this section, we perform a few experiments on DGEP to examine some of its aspects, such as the effect of DGEP on the population diversity, how the population diversity evolves under its influence, whether the DGM mutation scheme can achieve a higher success rate and the correlation between the success rate of the DGM scheme and the performance of the DGEP algorithm on the standard benchmark functions f_1–f_{18}.

At first, we investigate how the population diversity is evolved by the DGEP algorithm compared to the basic CEP [102] algorithm. CEP [102] is the classical evolutionary programming approach and it shows no explicit concern for the degree of explorations, exploitations and diversity of the evolving population, while DGEP is founded on the idea of inter-neighborhood and intra-neighborhood diversities to guide its mutation operation. DGEP also alters the selection operator to promote more population diversity. Table 8.14 compares the last generation population diversity of CEP [102] and DGEP for the standard benchmark functions f_1–f_{18}. Both CEP [102] and DGEP use real valued representation for the chromosomes, i.e., each chromosome is a point in the D-dimensional parameter space. Therefore, the diversity is measured as the average Euclidean distance among the chromosomes across the population. More formally, the diversity $div(P)$ of the current population P is measured by using the eqs. (8.1)–(8.2), where D is the dimensionality of the problem and M is the population size.

$$dist\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = \sqrt{\sum_{k=1}^{D}\left(x_{ik} - x_{jk}\right)^2} \tag{8.1}$$

$$div\left(P\right) = \frac{1}{M}\frac{1}{M-1}\sum\nolimits_{\boldsymbol{x}_i \in P}\sum\nolimits_{\boldsymbol{x}_j \in P} dist\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) \tag{8.2}$$

Using such a simple, average distance based metric to quantify diversity not only has the benefit of plain interpretation, but also is consistent with the scheme of DGEP, because DGEP uses the Euclidean distance between neighbors and non-neighbors to guide the mutation step size, and also to find duplicate chromosomes for elimination. From the diversity present at last generations (Table 8.14), it is observed that DGEP fosters significantly higher amount of diversity in comparison to CEP [102] for most of the functions. In Table 8.14, DGEP shows higher amount of diversity on as many as 15 (out of 18) functions, while CEP shows better diversity only on the remaining three. This indicates that DGEP performs an overall higher amount of search space explorations than CEP. Fig. 8.5 shows the evolution of the population diversity with the ongoing generations for two multimodal functions — f_{10} and f_{18}. It is seen that CEP [102] drops the genetic diversity rapidly, within only a few early generations, and then never becomes able to restore the lost diversity. In contrast, DGEP maintains an overall higher amount of diversity all through the evolution and exhibits both ups and downs in the diversity values which might indicate the ability of DGEP to break free from some locally optimal point. The 'up's in Fig. 8.5 are indication of the strength of DGEP to restore some significant portion of the lost diversity when it breaks free from some locally optimal point (where it got trapped previously) by using its diversity guided mutation and diversity-promoting operations, such as elimination of duplicate individuals and replacement of the individuals stuck at the locally optimal points by randomly placed new, diverse individuals.

Table 8.14: Comparison of DGEP and CEP [102] based on the population diversity for the high dimensional functions f_1–f_{18}. Diversity of the population is measured using eqs. (8.1)–(8.2). For each function, the higher diversity value is marked with boldface font.

Function		Population Diversity		Function		Population Diversity	
		CEP	DGEP			CEP	DGEP
f_1		1.27e–06	**6.25e–03**	f_{10}		1.39e–08	**2.27e–03**
f_2		5.61e–05	**5.22e–04**	f_{11}		**4.52e–03**	6.82e–03
f_3		**4.81e–01**	5.27e–01	f_{12}		6.43e–03	**1.09e–02**
f_4		4.79e–04	**2.10e–03**	f_{13}		8.53e–08	**7.90e–05**
f_5		5.22e–03	**9.64e–02**	f_{14}		6.47e–06	**8.62e–03**
f_6		5.48e–03	**5.67e–01**	f_{15}		5.28e–07	**2.59e–03**
f_7		**6.35e–01**	3.73e–01	f_{16}		6.32e–03	**7.01e–02**
f_8		8.95e–03	**9.50e–02**	f_{17}		2.86e–09	**4.57e–04**
f_9		5.09e–07	**2.14e–04**	f_{18}		1.07e–07	**5.33e–04**
Summary	DGEP +	7		**Summary**	DGEP +	8	
	DGEP –	2			DGEP –	1	

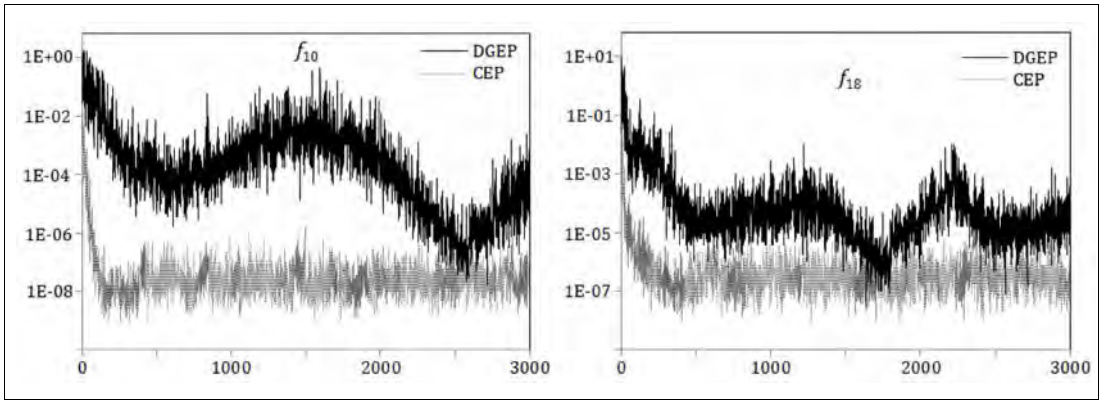


Figure 8.5: The evolution of the population diversity with the ongoing generations, under the influence of CEP [102] and DGEP for two multimodal functions — f_{10} (left) and f_{18} (right).

DGEP introduces the DGM (Diversity Guided Mutation) scheme for more effective mutations of the individuals. But does the DGM mutation scheme have a higher success rate than the traditional Gaussian mutation scheme of the classical CEP [102] algorithm? To find out, we have measured the average success rates of mutations of both DGEP and CEP and presented in Table 8.15. Here, a mutation operation is considered 'successful' if and only if the new, mutated candidate solution is better (i.e., has higher fitness value) than the original solution. Both DGEP and CEP are executed with population size = 100 and maximum number of function evaluations (FE) = 100,000. The other parameters of DGEP are set to the same values as of DGEP[1] in Table 4.1 (chapter 4). Table 8.15 demonstrates that, for all (18 out of 18) the functions f_1–f_{18}, the percentage of successful mutations is higher by DGEP than by the CEP algorithm. For some functions (e.g., f_2, f_{10} and f_{14}), the success rate of DGEP is noticeably much higher than CEP, which interestingly coincides with the much improved results of DGEP for the very same functions (f_2, f_{10} and f_{14}) in Table 8.15. Does there exist any direct positive correlation between the higher success rate of the DGM mutation scheme and the improved performance of the DGEP algorithm? To find out, we have plotted (Fig. 8.6) the quality of the final solution $\boldsymbol{x}^*$, measured as $-\log_{10} f(\boldsymbol{x}^*)$, along the y-axis and the corresponding success rate of the DGM mutation scheme along the x-axis. The correlation between these two series of data points is computed to be 0.71, which indicates a strong, positive correlation. The plot in Fig. 8.6 also indicates the existence of a strong positive correlation, which becomes clear from the general trend of those data points. This positive correlation between improved results by DGEP and the higher success rate of the DGM mutation scheme indicates that the proposed diversity guided mutation scheme of DGEP can effectively induce more successful mutations to produce better offspring solutions from the existing ones, which results in significantly improved performance of DGEP over the CEP [102] and many other existing evolutionary and swarm intelligence algorithms, as has already been demonstrated by Tables 4.1–4.11 (Chapter 4).

Table 8.15: Comparison between DGEP and CEP [102] based on the final solution quality and the successful mutation rate on the high dimensional standard benchmark functions f_1–f_{18}. The higher mutation success rate for each function is marked with boldface font.

Function	CEP		DGEP	
	Final Solution Quality, f^*	Successful Mutation (%)	Final Solution Quality, f^*	Successful Mutation (%)
f_1	4.28e–04	25.08	2.81e–07	**33.72**
f_2	2.90e–03	15.05	1.07e–11	**38.95**
f_3	5.15e+02	10.21	6.99e+00	**18.94**
f_4	3.36e–02	22.46	4.85e–06	**31.12**
f_5	3.78e+00	12.01	5.03e–02	**21.73**
f_6	8.58e+00	10.56	2.42e–01	**22.55**
f_7	1.75e+01	11.25	7.11e+00	**22.46**
f_8	2.80e+00	7.48	0	**18.28**
f_9	9.65e–01	17.19	5.40e–06	**26.65**
f_{10}	5.23e+01	14.64	2.74e–11	**36.64**
f_{11}	8.40e+00	13.76	9.33e–03	**27.37**
f_{12}	6.90e+01	5.19	1.65e+01	**20.35**
f_{13}	4.25e+00	22.94	8.58e–12	**29.31**
f_{14}	2.77e+00	32.18	1.71e–12	**41.37**
f_{15}	7.80e–02	17.42	3.54e–07	**28.05**
f_{16}	4.56e–02	14.50	6.22e–05	**31.44**
f_{17}	6.58e–01	14.59	3.73e–09	**29.19**
f_{18}	3.39e–02	16.36	5.26e–03	**28.05**

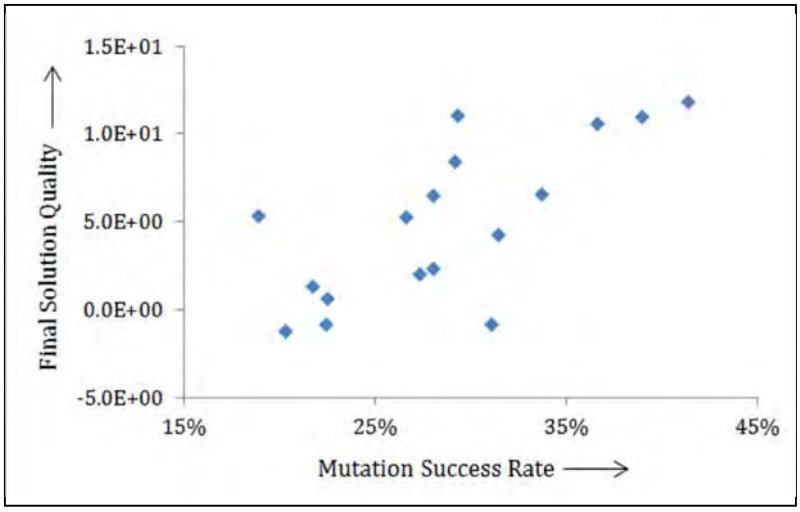


Figure 8.6: Scatter plot of the final solution quality vs. the mutation success rate for the DGEP algorithm. The vertical axis shows the final solution quality, measured as $-\log_{10}(f^*)$, where $f^*=f(\boldsymbol{x}^*)$ and $\boldsymbol{x}^*$ is the best-so-far solution found by DGEP), while the horizontal axis shows the mutation success of DGEP. The plot suggests a strong positive correlation between better solution quality of DGEP and higher mutation success rates by the DGM mutation scheme.

8.5 Experiments on ABC-SAM

In this section, we carry out a number of experiments on ABC-SAM to study the role and effect of its control parameters, examine how they influence its performance, evaluate some of the design choices of ABC-SAM, measure how the success rate of mutations gets affected by the self-adaptive mutation scheme and how this affects the final solution quality of ABC-SAM.

In addition to the common parameters of every ABC-based algorithm (i.e., *SN*, *MCN* and *limit*), ABC-SAM has a number of specific control parameters, such as K, τ_1, τ_2, SF_{min}, α and β, each one of which is introduced in chapter 5, and their default values are specified in section 5.4.1. However, among these parameters, K and α are the two most influential ones that significantly affect the performance of ABC-SAM. As a brief reminder for the reader, K denotes the adaptation period (i.e., number of cycles between two successive 'adaptation' cycles of ABC-SAM), while α controls the intensity of both explorations and exploitations (because, the explorative and exploitative perturbation scaling factors of ABC-SAM are produced randomly, within the range $[0, 2^{\alpha}]$ and $[0, 2^{-\alpha}]$).

Table 8.16 presents the performance (i.e., mean error of the final solutions) of ABC-SAM for several different values of K for the standard benchmark functions f_1–f_{30}. Results show that the best result for each function is always achieved with $K=5$ or $K=10$. For most of these functions, the results start to deteriorate for both increasing the value of K beyond $K=10$ or decreasing the value of K below $K=5$. Fig. 8.7 plots the mean absolute error values of ABC-SAM for different values of K, which also identifies the optimal value of K as $K=10$. Fig. 8.7 also shows that the error values start to escalate as K deviates from this optimal value (i.e., 10). This suggests that adapting the mutation step size too frequently (e.g., with $K<5$) or too scarcely (e.g., $K=100$ or 250) worsens the performance of ABC-SAM, while a moderate frequency of adaptation (e.g., $K=5\sim10$) produces the best possible results for most optimization problems.

Now, we examine the effect of the control parameter α on the performance of ABC-SAM. The value of α controls the intensity of both explorations and exploitations, because ABC-SAM produces the explorative and exploitative scaling factors for mutations from $[0, 2^{\alpha}]$ and $[0, 2^{-\alpha}]$, respectively. Table 8.17 presents the results of ABC-SAM for several different values of α, which shows that the best result for each function is always achieved with α in $[4, 12]$. For most (18 out of 25) functions, ABC-SAM produces the best results with $\alpha=8$ or 10. Results start to deteriorate for both increasing and decreasing values of α beyond the range of $[4, 12]$. Fig. 8.8 plots the mean absolute error values of ABC-SAM for different values of α, which also identifies the same optimal range of values for α (i.e., $[8, 10]$) and shows that the error value starts to rise quickly as α deviates beyond the range of $[4, 12]$. Since this range of values produces the best results from ABC-SAM for all the standard benchmark functions, we suggest this value range for a new user to start with ABC-SAM for any optimization problem at hand.

Table 8.16: Performance (mean error) of eight different variants of ABC-SAM on the standard benchmark functions f_1–f_{30}, each variant using a different value of the control parameter K (i.e., the adaptation period). The best result for each function is marked with boldface font.

Value of K	Function								
	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9
K=2	6.15e–12	7.19e–07	2.15e+01	6.20e–09	8.37e–01	6.85e–03	2.87e+01	0	3.35e–12
K=5	**1.61e–14**	9.76e–08	2.13e+01	**8.16e–13**	**2.50e–01**	**1.41e–03**	2.52e+01	0	6.30e–15
K=10	4.18e–14	**2.47e–08**	**1.69e+01**	3.95e–12	9.24e–01	2.16e–03	**2.28e+01**	0	**3.66e–16**
K=20	3.46e–12	5.19e–07	1.88e+01	8.58e–12	1.52e+00	2.79e–02	2.69e+00	0	2.65e–14
K=30	1.06e–11	6.33e–07	2.75e+01	1.32e–11	3.59e+00	7.22e–02	2.82e+00	0	7.05e–14
K=50	7.57e–12	8.54e–07	3.40e+01	8.32e–11	4.09e+00	3.26e–03	3.14e+01	0	1.23e–13
K=100	2.56e–10	6.82e–05	3.89e+01	8.89e–08	6.65e+00	1.94e–01	3.79e+01	0	9.11e–10
K=250	3.94e–05	8.77e–02	4.40e+01	5.52e–05	8.76e+00	1.90e+00	4.61e+01	0	7.82e–06

Value of K	Function								
	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}	f_{17}	f_{18}
K=2	8.45e–15	9.29e–09	–11905.2	8.28e–07	8.66e–08	2.63e–06	6.02e–01	2.68e–10	1.91e–03
K=5	4.14e–15	6.86e–10	–12332.8	**1.58e–08**	9.32e–10	**1.46e–09**	6.82e–03	1.37e–10	6.35e–04
K=10	**1.26e–16**	**4.60e–10**	**–12416.2**	9.26e–08	**8.36e–10**	2.22e–08	**5.78e–04**	**9.78e–12**	**3.06e–04**
K=20	1.92e–15	5.64e–09	–12226.2	5.76e–07	9.92e–10	8.27e–07	7.43e–02	4.76e–11	8.88e–02
K=30	5.25e–15	5.09e–09	–12109.2	5.69e–06	1.89e–09	2.56e–07	1.25e–01	3.09e–10	9.80e–03
K=50	8.85e–15	8.85e–09	–12084.2	9.55e–06	1.84e–09	4.40e–05	4.12e+00	8.74e–10	8.08e–03
K=100	7.65e–13	8.03e–07	–11830.2	4.83e–05	7.42e–08	4.46e–04	6.67e+00	6.12e–08	4.54e–03
K=250	9.92e–12	9.56e–06	–11201.2	4.65e–04	6.79e–07	1.79e–03	8.43e+00	6.28e–07	7.89e–03

Value of K	Function						
	f_{19}	f_{20}	f_{25}	f_{26}	f_{27}	f_{28}	f_{30}
K=2	1.04	3.65e–04	–9.85	**–10.40**	–10.53	1.13e+01	–0.86
K=5	**1.03**	**3.47e–04**	**–10.15**	**–10.40**	**–10.54**	7.36	–0.93
K=10	**1.03**	4.32e–04	–10.14	–10.38	–10.52	**4.02**	**–1.04**
K=20	1.04	4.74e–04	–10.07	–10.38	–10.51	6.04	–0.94
K=30	1.04	4.90e–04	–10.05	–10.38	–10.52	6.92	–0.95
K=50	1.04	5.20e–04	–10.01	**–10.40**	**–10.54**	8.56	–0.80
K=100	1.07	5.38e–04	–9.99	–10.33	–10.48	8.77	–0.78
K=250	1.06	5.88e–04	–9.86	–10.34	–10.35	9.54	–0.67

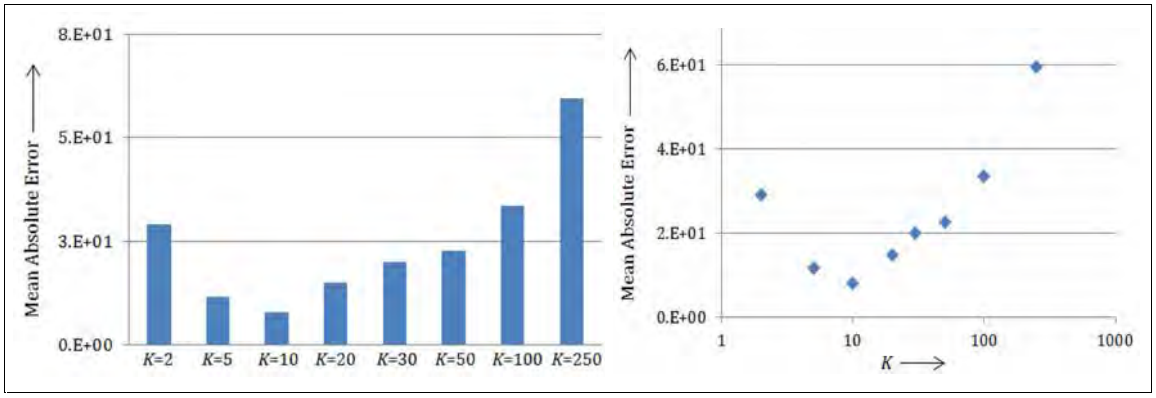


Figure 8.7: The mean absolute error values of the final results from ABC-SAM for eight different settings of the control parameter K (i.e., the adaptation period). ABC-SAM with $K=10$ shows the best performance, i.e., the minimum mean absolute error value.

Table 8.17: Performance of nine different variants of ABC-SAM on the standard benchmark functions f_1–f_{30}, each variant using a different value of the control parameter α. The best result for each function is marked with boldface font.

Value of α	Function								
	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9
$\alpha=0.5$	9.88e–12	1.62e–07	2.47e+01	3.95e–10	5.33e+00	1.27e–01	6.25e+01	0	8.05e–14
$\alpha=1.0$	7.28e–12	9.67e–08	2.23e+01	2.77e–10	4.88e+00	5.61e–01	2.59e+01	0	6.81e–14
$\alpha=2.0$	9.58e–13	5.44e–08	2.09e+01	7.96e–11	1.77e+00	6.01e–01	2.37e+01	0	3.86e–14
$\alpha=4.0$	5.96e–13	5.53e–08	1.84e+01	**9.60e–13**	1.75e+00	6.79e–02	**2.17e+01**	0	2.73e–15
$\alpha=8.0$	1.92e–13	**1.97e–08**	1.78e+01	1.70e–12	4.47e–01	5.22e–02	2.26e+01	0	8.84e–15
$\alpha=10$	**4.18e–14**	2.47e–08	**1.69e+01**	3.95e–12	9.24e–01	**2.16e–03**	2.28e+01	0	3.66e–15
$\alpha=12$	2.82e–13	7.06e–08	1.92e+01	4.29e–11	**6.55e–02**	3.96e–02	2.83e+01	0	**5.69e–16**
$\alpha=15$	1.36e–12	1.98e–07	3.33e+01	8.69e–11	4.67e+00	5.87e–01	3.19e+01	0	8.43e–15
$\alpha=20$	5.89e–12	2.35e–07	3.85e+01	6.56e–10	6.06e+00	6.09e–01	3.39e+01	0	6.12e–14

Value of α	Function								
	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}	f_{17}	f_{18}
$\alpha=0.5$	7.33e–15	5.62e–09	–11963.5	2.21e–06	3.03e–08	6.35e–06	2.50e–03	2.27e–11	8.76e–02
$\alpha=1.0$	4.78e–15	6.85e–09	–12106.6	8.15e–07	1.34e–08	1.72e–06	3.90e–03	1.63e–11	9.32e–02
$\alpha=2.0$	4.66e–15	7.60e–10	–12389.0	7.51e–07	1.71e–08	4.97e–07	6.02e–04	9.87e–12	6.76e–02
$\alpha=4.0$	**8.74e–17**	6.61e–10	**–12505.8**	7.58e–07	3.91e–09	2.35e–07	1.22e–04	8.25e–12	5.05e–02
$\alpha=8.0$	5.90e–16	6.42e–10	–12434.1	7.73e–08	6.89e–09	9.21e–08	**1.71e–05**	4.05e–13	3.47e–02
$\alpha=10$	1.26e–16	**4.60e–10**	–12416.2	9.26e–08	**8.36e–10**	**2.22e–08**	5.78e–04	**9.78e–12**	**3.06e–02**
$\alpha=12$	9.54e–16	3.59e–09	–12348.0	**5.91e–08**	4.26e–09	8.49e–08	7.49e–03	5.67e–12	6.91e–02
$\alpha=15$	1.59e–15	5.77e–09	–12108.7	1.04e–07	3.45e–08	6.98e–07	8.92e–03	6.02e–12	8.55e–02
$\alpha=20$	9.24e–15	6.34e–09	–11706.8	1.56e–06	3.90e–08	2.53e–06	8.18e–02	2.90e–11	8.49e–02

Table 8.17 (continued)

Value of α	Function						
	f_{19}	f_{20}	f_{25}	f_{26}	f_{27}	f_{28}	f_{30}
α=0.5	1.04	5.47e–04	–9.90	–10.40	–10.52	9.55	–0.80
α=1.0	1.04	5.15e–04	–9.96	–10.40	–10.52	6.83	–0.82
α=2.0	1.05	4.82e–04	–10.10	–10.40	–10.53	5.10	–0.90
α=4.0	**1.02**	4.85e–04	–10.05	–10.40	**–10.54**	5.66	**–1.04**
α=8.0	**1.02**	4.70e–04	–10.11	–10.40	**–10.54**	4.84	–0.99
α=10	1.03	**4.32e–04**	**–10.14**	–10.40	–10.53	**4.02**	**–1.04**
α=12	1.03	4.66e–04	–9.85	–10.40	–10.53	5.38	–0.95
α=15	1.04	5.31e–04	–9.78	–10.40	–10.52	7.97	–0.82
α=20	1.04	5.56e–04	–9.73	–10.40	–10.52	8.65	–0.79

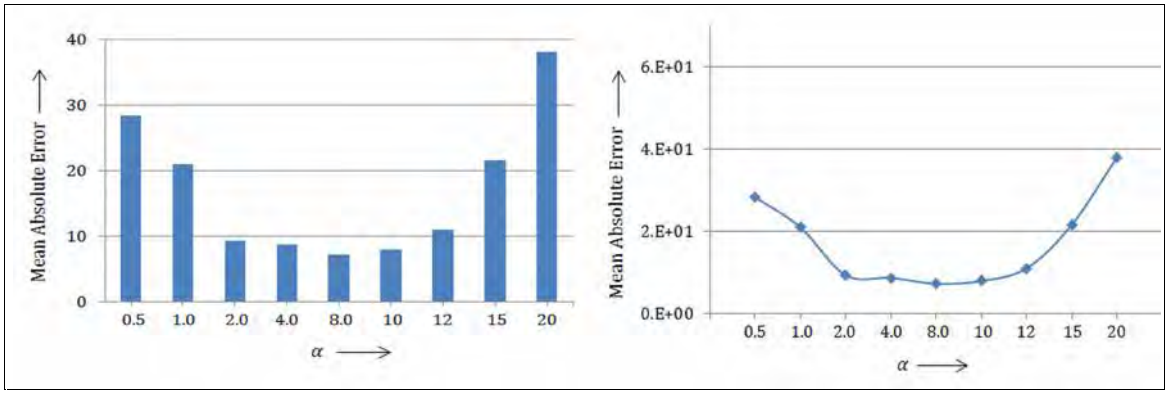


Figure 8.8: The mean absolute error values of ABC-SAM with nine different settings of the control parameter α. ABC-SAM with α=8.0 shows the best performance (i.e., minimum error value). Also, $\alpha \sim [4, 10]$ produces sufficiently good results, i.e., small mean absolute error values.

Our next experiment is on evaluating two important design choices of ABC-SAM. As explained in chapter 5, ABC-SAM maintains a scaling factor value SF_i for every candidate solution $\boldsymbol{x}_i$ and periodically adapts the value of SF_i, either for more exploitations (by decreasing the value of SF_i) or for more explorations (i.e., by increasing the value of SF_i). During this adaptation process, the value of SF_i is never allowed to fall below SF_{min}. However, if the value of a particular SF_i is get stuck at SF_{min} for a very long period (e.g., the previous τ_1 cycles), then SF_i is reset to its default value of 1.0. After each such reset, the value of SF_i is 'frozen' (i.e., kept constant at 1.0) for the next τ_2 cycles. But are these two design choices (i.e., *SF*-reset and *SF*-stagnation) necessary for the good performance of ABC-SAM? In order to find out, we have introduced two more variants of ABC-SAM — ABC-SAM-II and ABC-SAM-III. The first variant (ABC-SAM-II) is just like the standard ABC-SAM algorithm, but only one exception — it never resets any SF_i value to 1.0, even if that particular SF_i gets stuck at SF_{min} for the previous τ_1 (or, more) cycles. The other variant — ABC-SAM-III allows the necessary resetting of the SF_i values,

like ABC-SAM, if that particular SF_i value is stuck at its minimum value (i.e., SF_{min}) for the last τ_1 cycles. But unlike ABC-SAM, it does not 'freeze' that reset SF_i value at 1.0 for the next τ_2 cycles. All three algorithms — ABC-SAM, ABC-SAM-II and ABC-SAM-III are run with the following parameter values — K=10, SN=100, MCN=1000, $limit$=100, τ_1=40, τ_2=20, SF_{min}=10^{-8}, α=10 and β=0.9. Table 8.18 presents their results on the standard benchmark functions f_1–f_{30}. Results indicate that the standard ABC-SAM algorithm, which includes both the *SF*-reset and *SF*-stagnation schemes, significantly outperforms both the other two ABC-SAM variants on most (18 out of 30) of the functions. This empirically proves the necessity of both the *SF*-reset and *SF*-stagnation schemes. Actually, both these schemes increase the explorative capacity of the ABC-SAM algorithm. Resetting a particular scaling factor value, say SF_i, from its minimum possible of SF_{min} (i.e., 10^{-8}) to the default value (i.e., 1.0) makes an exponential leap of its value to promote the degree of explorations around the candidate solution $\boldsymbol{x}_i$. Freezing this scaling factor value at 1.0 for the next τ_2 cycles promotes the continuation of search space explorations around $\boldsymbol{x}_i$ for at least the next τ_2 iterations. Otherwise, the value of SF_i might rapidly revert back to SF_{min} because of the repeated application of eq. (5.2). The improved results of ABC-SAM (Table 8.18) in comparison to the other two variants (i.e., ABC-SAM-II without *SF*-reset and ABC-SAM-III without *SF*-stagnation) indicate that the increased degree of explorations by the *SF*-reset and *SF*-stagnation strategies is important and essential for the better performance of ABC-SAM on most of the standard benchmark functions f_1–f_{30}.

Table 8.18: Comparison among ABC-SAM, ABC-SAM-II and ABC-SAM-III based on the final solution quality on the standard benchmark functions f_1–f_{30}. For each function, the best unique performance is marked with boldface font.

No.	Mean Result ± Standard Deviation			*t*-Test	
	ABC-SAM (includes both *SF*-reset and *SF*-stagnation)	**ABC-SAM-II** (ABC-SAM without *SF*-reset)	**ABC-SAM-III** (ABC-SAM without *SF*-stagnation)	**ABC-SAM vs. ABC-SAM-II**	**ABC-SAM vs. ABC-SAM-III**
f_1	**4.18e–14 ± 5.37e–15**	6.25e–14 ± 2.95e–14	5.05e–14 ± 2.62e–14	≈	≈
f_2	**2.47e–08 ± 2.35e–09**	5.38e–05 ± 2.77e–15	2.28e–05 ± 9.67e–15	+	+
f_3	**16.94 ± 1.43**	25.96 ± 7.23	24.27 ± 6.72	+	+
f_4	**3.95e–12 ± 1.05e–12**	9.69e–10 ± 9.10e–09	4.76e–10 ± 5.53e–10	+	+
f_5	**9.24e–01 ± 2.08e–01**	8.78 ± 1.70	6.92 ± 2.15	+	+
f_6	**2.16e–03 ± 6.37e–04**	4.21e–01 ± 1.34e–02	1.85e–02 ± 6.28e–03	+	+
f_7	**22.83 ± 3.75**	31.66 ± 3.28	29.83 ± 2.93	+	+
f_8	0 ± 0	0 ± 0	0 ± 0	≈	≈
f_9	**3.56e–16 ± 1.44e–17**	3.64e–16 ± 1.09e–17	3.59e–16 ± 9.77e–17	≈	≈
f_{10}	**1.26e–16 ± 2.11e–17**	1.62e–10 ± 5.24e–11	1.35e–10 ± 8.16e–11	+	+
f_{11}	**4.60e–10 ± 8.85e–11**	4.61e–05 ± 2.31e–05	4.56e–05 ± 1.36e–05	+	+
f_{12}	**–12416.19 ± 40.22**	–11828.33 ± 163.16	–12008.33 ± 65.90	+	+

Table 8.18 (continued)

No.	Mean Result ± Standard Deviation			*t*-Test (ABC-SAM vs.)	
	ABC-SAM	**ABC-SAM-II**	**ABC-SAM-III**	**ABC-SAM-II**	**ABC-SAM-III**
f_{13}	**9.26e–08 ± 1.89e–08**	4.58e–05 ± 4.74e–06	9.17e–06 ± 3.55e–06	+	+
f_{14}	**8.36e–10 ± 5.08e–11**	6.51e–07 ± 9.79e–08	8.92e–07 ± 3.25e–07	+	+
f_{15}	**2.22e–08 ± 3.93e–09**	3.97e–06 ± 5.20e–07	3.18e–06 ± 9.46e–07	+	+
f_{16}	**5.78e–04 ± 6.31e–05**	3.84e–02 ± 6.94e–15	2.63e–02 ± 9.05e–05	+	+
f_{17}	**9.78e–12 ± 3.89e–12**	8.65e–07 ± 3.25e–07	9.66e–07 ± 3.46e–07	+	+
f_{18}	3.06e–02 ± 8.59e–03	2.98e–02 ± 9.18e–03	**2.94e–02 ± 1.17e–02**	≈	≈
f_{19}	1.03 ± 0.02	1.03 ± 0.002	**1.028 ± 0.02**	≈	≈
f_{20}	**4.32e–04 ± 1.09e–05**	4.35e–04 ± 9.56e–05	4.34e–04 ± 8.25e–05	≈	≈
f_{21}	–1.0316 ± 0.00	–1.0316 ± 0.00	–1.0316 ± 0.00	≈	≈
f_{22}	0.398 ± 0.00	0.398 ± 0.00	0.398 ± 0.00	≈	≈
f_{23}	–3.86 ± 0.00	–3.86 ± 0.00	–3.86 ± 0.00	≈	≈
f_{24}	–3.32 ± 0.00	–3.32 ± 0	–3.32 ± 0.00	≈	≈
f_{25}	**–10.14 ± 3.68e–07**	–9.90 ± 0.11	–9.94 ± 0.08	+	+
f_{26}	–10.40 ± 3.94e–03	–10.40 ± 0.00	–10.40 ± 0.00	≈	≈
f_{27}	**–10.54 ± 6.77e–07**	–10.38 ± 0.14	–10.34 ± 0.15	+	+
f_{28}	**4.02 ± 0.39**	6.96 ± 1.34	6.13 ± 0.45	+	+
f_{29}	–9.66015 ± 0	–9.66015 ± 0.00	–9.66015 ± 0.00	≈	≈
f_{30}	**–1.04 ± 0.06**	–0.89 ± 0.08	–0.90 ± 0.06	+	+

How does the proposed self-adaptive mutation strategy affect the mutation success rate of ABC-SAM? To find out, we have measured the percentage of successful mutations for both ABC and ABC-SAM, and compared them side-by-side in Table 8.19. Results show that the self-adaptive mutation strategy of ABC-SAM has higher success rate than the basic mutation scheme of the ABC algorithm on most (15 out of 18) of the functions. But does this higher success rate have any direct correlation with the improved results of ABC-SAM? To find out, we have plotted the final solution quality (*y*-axis) of ABC-SAM against its rate of successful mutations (*x*-axis) in Fig. 8.9. The general trend of the points in Fig. 8.9 clearly indicates strong positive correlation between better results (i.e., smaller function value) and higher mutation success rate. Strong correlation is also observed in Table 8.19. For example, ABC-SAM outperforms ABC on 14 (out of 18) functions in Table 8.19, and on each of them ABC-SAM shows higher mutation success rates than ABC. The only three (out of 18) functions where ABC-SAM performs worse than ABC are — f_7, f_{17} and f_{18}, and interestingly, these three functions are the only ones in Table 8.19 where ABC-SAM shows lower mutation success rates than ABC. This positive correlation between improved results and higher mutation success rate suggests that the proposed self-adaptive mutation scheme can effectively induce more successful mutations to produce better trial solutions and thus can improve the performance of the algorithm.

Table 8.19: Comparison of ABC and ABC-SAM based on their final solution quality and successful mutation rate on the high dimensional benchmark functions f_1–f_{18}. Instances with better solution quality due to higher mutation success rates are marked with boldface font.

Function	ABC		ABC-SAM	
	Final Solution Quality	Successful Mutation Rate (%)	Final Solution Quality, f^*	Successful Mutation Rate (%)
f_1	2.45e–11	22.08	**4.18e–14**	**25.33**
f_2	5.05e–07	14.45	**2.47e–08**	**15.74**
f_3	4.18e+01	10.62	**1.69e+01**	**12.38**
f_4	8.32e–10	18.96	**3.95e–12**	**22.78**
f_5	6.61e+00	13.01	**9.24e–01**	**15.10**
f_6	6.67e–01	14.73	**2.16e–03**	**18.03**
f_7	**4.25e–01**	**8.63**	2.28e+01	7.15
f_8	0	12.69	0	**15.44**
f_9	8.60e–13	19.48	**3.56e–16**	**28.18**
f_{10}	1.72e–14	12.62	**1.26e–16**	**22.09**
f_{11}	2.33e–08	12.66	**4.60e–10**	**19.80**
f_{12}	–11346.79	5.67	**–12416.19**	**10.81**
f_{13}	2.93e–06	23.39	**9.26e–08**	**25.43**
f_{14}	4.55e–08	19.53	**8.36e–10**	**22.63**
f_{15}	3.34e–04	15.32	**2.22e–08**	**18.96**
f_{16}	3.36e–01	12.28	**5.78e–04**	**18.82**
f_{17}	**5.47e–12**	**22.91**	9.78e–12	21.45
f_{18}	**2.63e–03**	**15.60**	3.06e–02	13.55

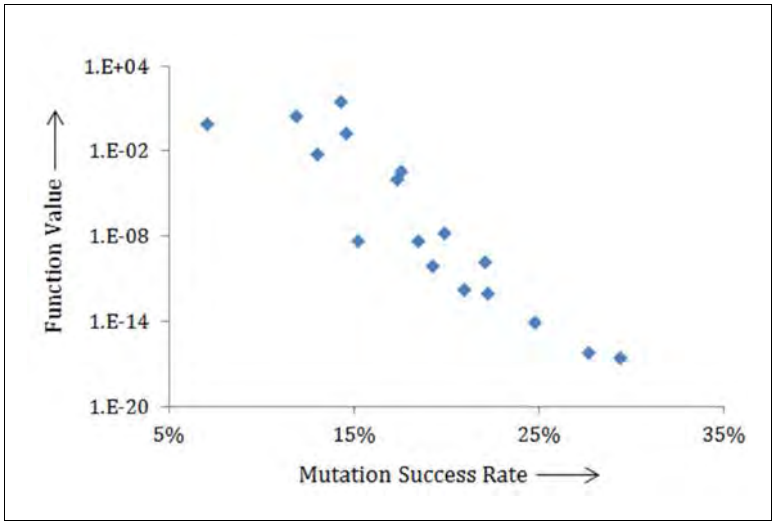


Figure 8.9: Scatter plot of the final solution quality vs. the mutation success rates of ABC-SAM on the standard benchmark functions f_1–f_{18}. The vertical axis shows the final solution quality (i.e., mean error) in the logarithmic scale, while the horizontal axis shows the percentage of successful mutations. The plot indicates that there is strong positive correlation between better solution quality (i.e., lower function value) and higher mutation success rates.

8.6 Experiments on ABC-IX

In this section, we conduct several experiments on ABC-IX to comprehend how it improves the results over the basic ABC algorithm. In the following paragraphs (Tables 8.20–8.23, Figs. 8.10–8.11), we present the experiments and their results, one by one, intending to find out the answers to some questions, such as — whether each component of ABC-IX is essential for its improved performance, how does the population diversity evolve by ABC-IX, what is the effect of ABC-IX on the mutation success rate, whether (and, how) any self-adaptation takes place on the perturbation rates of the individuals and does the effective perturbation rate depend on the characteristics of the function, such as its modality and separability.

Our first experiment on ABC-IX is to validate its design choices — Are both its components (i.e., the simulated annealing based probabilistic selection scheme and the separate self-adaptive perturbation rate for each candidate solution) necessary for its good performance? Do both these design choices contribute significantly to the improved results of ABC-IX? To find out, we have designed two more variants — ABC-SimAn and ABC-SAD. ABC-SimAn includes the simulated annealing based probabilistic selection scheme, but does not include the self-adaptive perturbation rate of ABC-IX. The other variant — ABC-SAD includes the scheme for self-adaptive perturbation rate, like ABC-IX, but does not employ the explorative simulated annealing based probabilistic selection scheme. Both ABC-SimAn and ABC-SAD are tested and compared with ABC and ABC-IX on the 18 high dimensional functions f_1–f_{18} with $SN = 100$, $MCN = 1000$ and $limit = 200$. Results (Table 8.20) show that both ABC-SimAn and ABC-SAD outperform the basic ABC algorithm on almost all (16 out of 18) of the functions (with the only exceptions of f_8 and f_{18}), which indicates the effectiveness of both the new selection and perturbation schemes to improve the results over the basic ABC algorithm. However, when both the schemes are combined, as is done by ABC-IX, the results are further improved, as shown for all (18 out of 18) of the functions in Table 8.20. These improvements are often (13 out of 18 functions) statistically significant, as shown by the *t*-test with at least 95% level of confidence between the results of ABC-IX and each of ABC-SimAn and ABC-SAD. This also indicates that the explorative simulated annealing based selection scheme and the self-adaptive perturbation rate of ABC-IX work synergistically and cooperatively to improve the results significantly over the basic ABC algorithm, as well as over both the partially improved variants — ABC-SimAn and ABC-SAD.

Table 8.20: Comparison of ABC-IX with ABC-SimAn, ABC-SAD and the basic ABC algorithm on high dimensional functions f_1–f_{18}. The best result for each function is marked with boldface font.

No	ABC		ABC-SimAn		ABC-SAD		ABC-IX		*t*-Test (ABC-IX vs.	
	Mean Error	Std. Dev.	Mean Error	Std. Dev.	Mean Error	Std. Dev.	Mean Error	Std. Dev.	ABC-SimAn	ABC-SAD
f_1	6.38e–10	8.30e–11	1.37e–10	2.12e–11	1.34e–15	5.29e–16	**1.22e–24**	**5.58e–25**	+	+
f_2	1.68e–12	2.61e–13	1.81e–13	4.66e–14	7.80e–15	2.80e–15	**7.63e–15**	**2.25e–15**	+	≈
f_3	9.15e+00	1.73e+00	5.50e–02	8.55e–03	3.50e–01	1.13e–01	**5.41e–02**	**8.16e–03**	≈	+
f_4	5.67e–08	1.22e–08	1.39e–11	3.00e–12	5.98e–19	8.43e–20	**1.70e–25**	**3.50e–26**	+	+
f_5	4.09e+00	7.63e–01	8.92e–01	1.76e–01	2.75e–01	3.81e–02	**1.50e–02**	**3.22e–03**	+	+
f_6	6.67e–01	2.77e–02	9.84e–02	3.17e–02	6.53e–02	8.77e–03	**8.10e–04**	**2.46e–04**	+	+
f_7	3.11e+00	1.30e+00	1.94e+00	9.43e–01	9.12e–01	1.19e–01	**6.13e–01**	**1.09e–01**	+	+
f_8	0	0	0	0	0	0	0	0	≈	≈
f_9	9.01e–11	2.66e–11	5.07e–22	9.62e–23	2.43e–31	5.68e–32	**9.68–38**	**9.96e–39**	+	+
f_{10}	5.02e–14	2.42e–14	2.53e–17	8.64e–18	6.85e–18	3.80e–18	**8.71e–20**	**9.10e–21**	+	+
f_{11}	2.13e–08	7.78e–09	9.71e–09	9.78e–10	5.23e–12	2.85e–12	**4.82e–13**	**1.98e–13**	+	+
f_{12}	7.52e+02	3.34e+02	5.29e+02	2.45e+02	8.90e+01	3.65e+01	**2.15e+01**	**1.08e+01**	+	+
f_{13}	3.04e–07	8.11e–08	8.50e–09	5.41e–09	8.37e–10	4.45e–10	**8.89e–11**	**4.02–11**	+	+
f_{14}	9.84e–10	4.57e–10	1.91e–19	7.92e–20	3.78e–19	1.62e–19	**2.07e–23**	**2.44e–24**	+	+
f_{15}	6.98e–06	2.86e–06	2.78e–08	1.49e–08	2.92e–08	1.33e–08	**2.58e–08**	**1.08e–08**	≈	≈
f_{16}	2.83e–02	7.30e–03	5.24e–06	3.05e–06	5.78e–06	2.31e–06	**4.96e–06**	**2.06e–06**	≈	≈
f_{17}	7.18e–10	5.19e–10	7.55e–14	2.79e–14	5.09e–14	1.68e–14	**1.91e–14**	**3.73e–15**	+	+
f_{18}	2.63e–03	1.07e–03	2.64e–03	8.60e–04	2.63e–03	6.02e–04	**2.61e–03**	**5.95e–04**	≈	≈
Summary (*t*-Test)		+	13		13					
		–	0		0					
		≈	5		5					

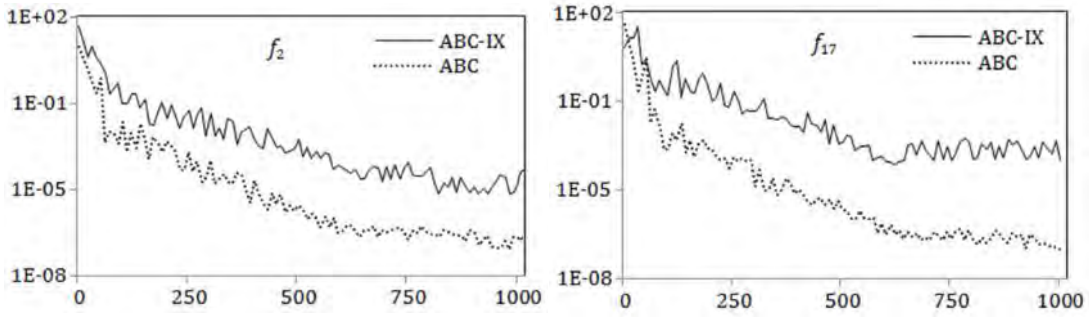


Figure 8.10: Diversity of the population of candidate solutions, evolving by ABC and ABC-IX, for the functions f_2 and f_{17}. The vertical axis shows the population diversity, while the horizontal axis is the number of cycles elapsed.

Table 8.21: Comparison of ABC-IX with ABC, based on their population diversity after the end of their run, for the high dimensional functions f_1–f_{18}. Diversity is measured as the average Euclidean distance of the individuals from the centroid of the population. The higher diversity value for each function is marked with boldface font.

Function	Population Diversity		Function	Population Diversity	
	ABC	ABC-IX		ABC	ABC-IX
f_1	5.83e–05	**7.92e–03**	f_{10}	6.35e–07	**6.91e–05**
f_2	8.59e–05	**3.57e–04**	f_{11}	2.52e–02	**6.02e–02**
f_3	3.01e–01	**7.27e–01**	f_{12}	7.54e–03	**8.13e–01**
f_4	2.61e–03	**8.77e–03**	f_{13}	5.15e–06	**5.32e–04**
f_5	3.37e–02	**8.07e–02**	f_{14}	5.63e–05	**3.84e–03**
f_6	9.40e–02	**9.27e–01**	f_{15}	4.16e–05	**1.44e–04**
f_7	9.69e–02	**6.80e–01**	f_{16}	7.40e–03	**2.46e–02**
f_8	1.88e–03	**1.74e–02**	f_{17}	3.82e–08	**7.07e–04**
f_9	2.58e–06	**6.03e–03**	f_{18}	9.77e–07	**3.67e–03**
Average f_1–f_9	5.89e–02	**2.73e–01**	**Average** f_{10}–f_{18}	4.47e–03	**1.01e–01**

The previous experiment (Table 8.20) demonstrates the presence of synergy among the components of ABC-IX. But how they affect the diversity of the population? The simulated annealing based explorative selection scheme of ABC-IX probabilistically accepts not only better solutions but also worse ones into the population. Does it increase the diversity and explorative capacity of the algorithm? To find out, Fig. 8.10 plots the evolution of the population diversity under the influence of both ABC and ABC-IX for two arbitrarily chosen functions — f_2 and f_{17}. Both the algorithms are run with SN=100, MCN=1000 and $limit$=100. The diversity value is measured as the mean distance of the candidate solutions to the centroid of the population. Fig. 8.10 shows that the population diversity drops more rapidly by ABC than by ABC-IX. Also, ABC-IX maintains higher level of population diversity than ABC all throughout its execution. Table 8.21 shows the diversity of the final generation population of both ABC and ABC-IX, where ABC-IX shows higher amount of diversity for all (18 out of 18) the functions. This is usually well desired for many complex optimization problems, because an increased level of population diversity is generally considered necessary for increased degree of search space explorations without being trapped around the locally optimal points of the search space.

How the success rate of perturbations gets affected by ABC-IX? We consider a perturbation as 'successful' if the new, perturbed solution is better (i.e., has higher fitness value) than the original candidate solution. In the next experiment (Table 8.22), we try to find out whether the self-adaptive perturbation rate of ABC-IX has actually improved the success rate of perturbations. To accomplish this, we have measured the average successful perturbation rate

achieved by ABC and ABC-IX for the high dimensional functions f_1–f_{18} with SN=100, MCN=1000 and *limit*=100. Table 8.22 demonstrates that, for almost all (15 out of 18) the functions of f_1–f_{18}, the percentage of successful perturbations is significantly higher by ABC-IX than by the standard ABC algorithm. For some functions (e.g., f_{10} and f_{14}) the success rate of ABC-IX is noticeably much higher than ABC, which interestingly coincides with the much improved results of ABC-IX for the very same functions (f_{10} and f_{14}) in Table 6.1 (chapter 6). This positive correlation between improved results by ABC-IX on a particular function and its higher success rate of perturbations for the very same function indicates that the proposed self-adaptive perturbation rate scheme of ABC-IX can effectively induce more successful perturbations to produce better trial solutions from the existing ones, which results in significantly improved performance of ABC-IX over the basic ABC algorithm.

Table 8.22: Comparison between ABC and ABC-IX based on their successful perturbation rates for the high dimensional functions f_1–f_{18}. The best result (i.e., higher success rate) for each function is marked with boldface font.

Function	***D***	**Successful Perturbation Rate (%)**	
		ABC	**ABC-IX**
f_1	30	17.23	**22.62**
f_2	30	14.35	**19.24**
f_3	30	10.96	**13.71**
f_4	30	18.82	**22.30**
f_5	24	11.31	**16.52**
f_6	30	12.92	**15.36**
f_7	30	**17.75**	14.84
f_8	30	**19.38**	19.62
f_9	30	18.46	**27.66**
f_{10}	30	18.30	**33.56**
f_{11}	30	15.42	**25.60**
f_{12}	30	12.64	**17.78**
f_{13}	30	13.38	**26.99**
f_{14}	30	12.75	**34.12**
f_{15}	30	13.26	**25.75**
f_{16}	30	10.42	**21.08**
f_{17}	30	16.01	**23.48**
f_{18}	30	**16.60**	16.64
+		15	
–		1	
≈		2	

The next experiment on ABC-IX tries to find out whether the value of the self-adaptive perturbation probability $\boldsymbol{x}_i.q$, maintained separately for each candidate solution $\boldsymbol{x}_i$, gradually adapts to more suitable values with the on-going optimization process. To find out, we have run ABC-IX with SN=100, MCN=1000 and $limit$=100, then measured the mean value of $\boldsymbol{x}_i.q$, averaged over all the candidate solutions of the population during each cycle, which is shown as $\overline{q}$ in Fig. 8.11. The functions in Fig. 8.11 include both unimodal (f_1, f_4, f_8) and multimodal (f_{12}, f_{13}, f_{18}), separable (f_1, f_8, f_{12}) and non-separable (f_4, f_{13}, f_{18}) functions. It is interesting to find out that the control parameter $\boldsymbol{x}_i.q$ gradually self-adapts either towards its maximum possible value, i.e., 1.0 (e.g., for f_4, f_{13} and f_{18}), or towards very small values (e.g., f_1, f_8 and f_{12}) depending on whether the function is separable or not. Table 8.23 shows the value of $\overline{q}$ during the final generation of

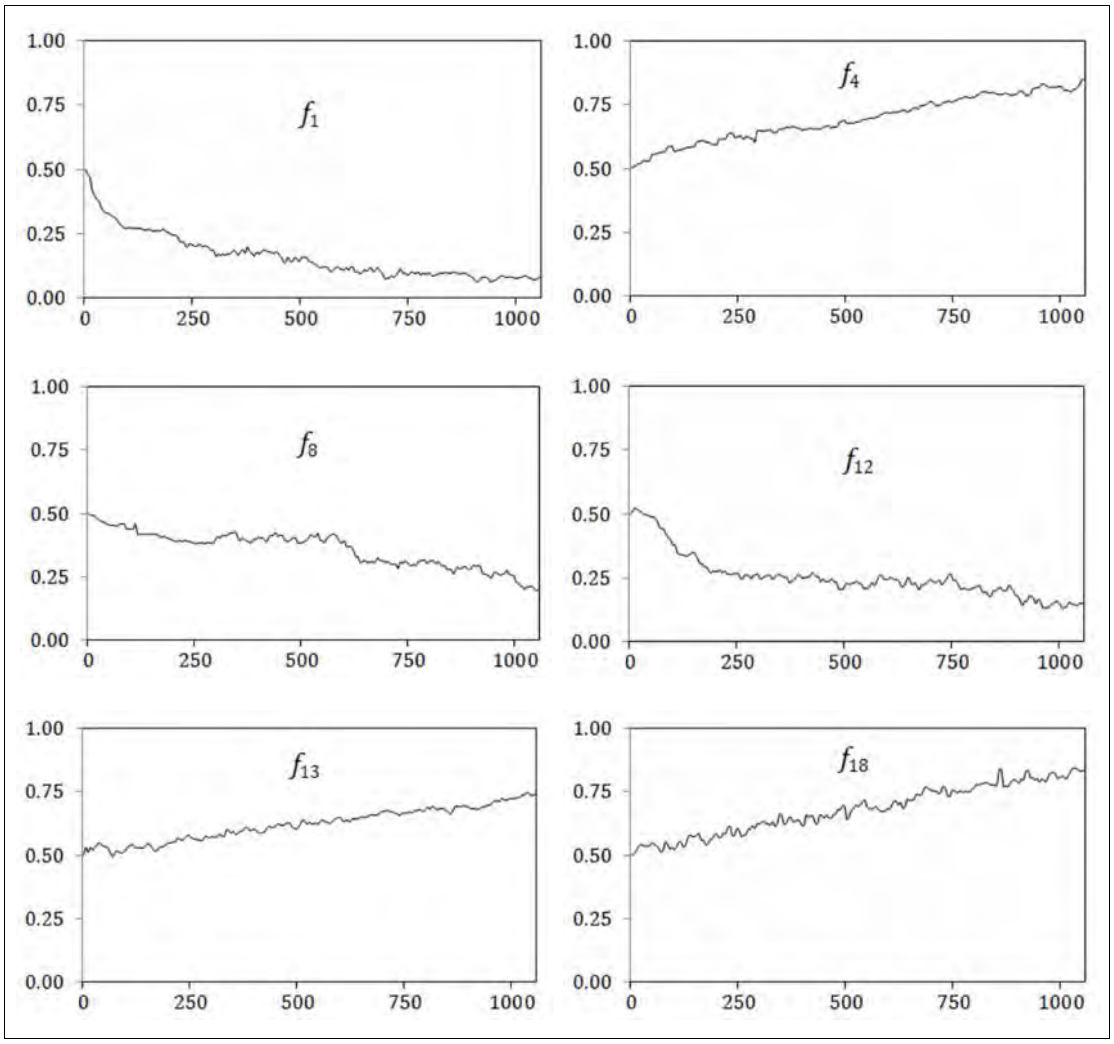


Figure 8.11: Evolution of the average perturbation probability $\overline{q}$ by ABC-IX for three unimodal (f_1, f_4, f_8) and three multimodal (f_{12}, f_{13}, f_{18}) functions. The vertical axis shows the value of $\overline{q}$, while the horizontal axis is the number of cycles elapsed.

Table 8.23: Mean value of the control parameter q (i.e., $\overline{q}$), measured during the final generation of ABC-IX, for each of the high dimensional functions f_1–f_{18}. Here, C is the function characteristics with values — U: Unimodal, M: Multimodal, S: Separable and N: Non-Separable.

Function	C	$\overline{q}$	**Function**	C	$\overline{q}$
f_1	US	0.08	f_{10}	MS	0.17
f_2	UN	0.86	f_{11}	MS	0.08
f_3	US	0.12	f_{12}	MS	0.15
f_4	UN	0.85	f_{13}	MN	0.74
f_5	UN	0.81	f_{14}	MN	0.86
f_6	UN	0.84	f_{15}	MS	0.12
f_7	UN	0.80	f_{16}	MS	0.17
f_8	US	0.20	f_{17}	MN	0.80
f_9	US	0.10	f_{18}	MN	0.83
Overall Mean $\overline{q}$	Separable Functions		**0.13**		
	Non-Separable Functions		**0.82**		

both ABC and ABC-IX for each of the high dimensional functions f_1–f_{18}. The overall mean $\overline{q}$ in Table 8.23 for the separable and non-separable functions are computed to be as 0.13 and 0.82, respectively, which reveals the same pattern of values in Fig. 8.11. For a non-separable function, the parameters are not independent, so perturbing a subset of the parameters is usually not useful. This is why $\overline{q}$ has self-adapted towards larger values, which is observed for all the non-separable functions in Fig. 8.11 and Table 8.23. This indicates the necessity to perturb most (or, all) of the parameters at once for successful perturbations for these non-separable functions. However, for the separable functions (e.g., f_1, f_8 and f_{12}), the value of $\overline{q}$ self-adapts to rather small values, which indicates that the parameters are now independent of each other and perturbing only one or just a few of them is often effective for the optimization.

8.7 Experiments on ABC-AX^2

In this section, we carry out a number of experiments on ABC-AX^2 to get a better understanding on how it achieves significantly improved performance than the basic ABC algorithm. These experiments, as presented in the following paragraphs, provide us with useful insights on several important aspects of ABC-AX^2, such as — whether each of the three adaptation and self-adaptation strategies of ABC-AX^2 is necessary for its improved performance, do these strategies have any synergistic effect on each other, whether any significant adaptation takes place on the control parameters p_i, q_i and $\boldsymbol{\eta_i}$ that are maintained separately for each candidate

solution $\boldsymbol{x_i}$ by ABC-IX, is there any important pattern on their direction of adaptation, how the success rate of perturbation gets affected by ABC-IX and is there any significant correlation between the improved results of ABC-IX and its rate of successful perturbations.

At first, we perform an experiment (Table 8.24) to find out whether all the three adaptation and self-adaptation schemes of ABC-AX^2 (i.e., Eqs. (7.3)–(7.6)), adjusting the values of its control parameters p_i, q_i and $\boldsymbol{\eta_i}$, can contribute significantly and synergistically to improve the results. To examine this, we have designed three more variants of the basic ABC algorithm — ABC-II, ABC-III and ABC-IV. ABC-II uses the control parameter p_i, just like ABC-AX^2, for each candidate solution $\boldsymbol{x}_i$ to control the proportion of exploitative and explorative perturbations on $\boldsymbol{x}_i$, but does not use the other two control parameters — q_i and $\boldsymbol{\eta_i}$. The next variant, ABC-III uses q_i for self-adaptive perturbation rate, but excludes p_i and $\boldsymbol{\eta}_i$. The third variant, ABC-IV makes use of scaling factor vector $\boldsymbol{\eta}_i$ for self-adaptation of the φ_{ij} values in (2.6), just like ABC-AX^2, but does not use p_i and q_i. Table 8.24 presents the results of all three ABC-variants, along with the results of basic ABC and ABC-AX^2. All of them are tested on the high dimensional functions f_1–f_{18} with D=30, SN=100, MCN=1000 and $limit$=100. Results (Table 8.24) show that ABC has been always outperformed by each of ABC-II, ABC-III and ABC-IV, which indicates the necessity of all three control parameter to improve the results over ABC. The performance rank (i.e., rightmost column) presents the order of the algorithms based on their results, which frequently (i.e., 10 out of the 18 functions) shows the following rank.

ABC-AX^2, ABC-III, II, IV, ABC

The performance rank column shows that ABC-AX^2 ranks 1st (i.e., best) on 16 (out of 18) functions. Ignoring the results on f_8, for which all the algorithms perform equally well, the overall average performance rank of ABC-AX^2 is the minimum (i.e., 1.24), followed by the average rank of ABC-III (i.e., 2.24), then ABC-II (rank=3.12), ABC-IV (rank=3.59) and the basic ABC algorithm (rank=4.82). This indicates that the self-adaptive perturbation rate (adopted by ABC-III, by using q_i) often contributes most for the improved results of ABC-AX^2, followed by the contributions of p_i (i.e., adaptive proportion of explorations and explorations, used by ABC-II) and $\boldsymbol{\eta_i}$ (i.e., self-adaptive scaling factors vector, used by ABC-IV). Furthermore, ABC-AX^2, which includes all three control parameters, outperforms all other ABC-variants on almost all (16 out of 18) of the functions, indicating that the roles of the three control parameters are synergistic and cumulative to improve the results for ABC-AX^2.

Table 8.24: Comparison of ABC-AX² with ABC-II, ABC-III, ABC-IV and the basic ABC algorithm on the high dimensional functions f_1–f_{18}. The best results are marked with boldface font.

Function	*D*	ABC	ABC-II	ABC-III	ABC-IV	ABC-AX²	Performance Rank (from best to worst)
f_1	30	2.45e–11	2.61e–14	5.98e–21	3.68e–12	**5.51e–24**	ABC-AX², ABC-III, II, IV, ABC
f_2	30	5.05e–07	4.40e–12	1.09e–13	3.86e–09	**4.23e–15**	ABC-AX², ABC-III, II, IV, ABC
f_3	30	4.18e+01	7.45e–01	5.66e+00	1.36e–01	**6.60e–02**	ABC-AX², ABC-IV, II, III, ABC
f_4	30	8.32e–10	2.72e–14	1.64e–13	7.02e–12	**3.42e–16**	ABC-AX², ABC-II, III, IV, ABC
f_5	24	6.61e+00	5.19e+00	7.30e–02	5.80e+00	**2.23e–02**	ABC-AX², ABC-III, II, IV, ABC
f_6	30	6.67e–01	3.11e–03	1.63e–03	1.67e–03	**5.91e–05**	ABC-AX², ABC-III, IV, II, ABC
f_7	30	4.25e–01	2.52e+00	**4.03e–01**	7.36e+00	2.39e+01	ABC-III, ABC, ABC-II, IV, ABC-AX²
f_8	30	0	0	0	0	0	- all identical results -
f_9	30	8.60e–13	7.78e–15	1.47e–15	6.97e–15	**8.87e–34**	ABC-AX², ABC-III, II, IV, ABC
f_{10}	30	1.72e–14	1.53e–18	8.02e–19	8.40e–16	**4.68e–24**	ABC-AX², ABC-III, II, IV, ABC
f_{11}	30	2.33e–08	4.20e–10	4.18e–12	8.77e–09	**1.04e–13**	ABC-AX², ABC-III, II, IV, ABC
f_{12}	30	–11346.79	–12018.08	–12568.94	–12287.40	**–12569.48**	ABC-AX², ABC-III, IV, II, ABC
f_{13}	30	2.93e–06	3.85e–08	6.39e–09	8.23e–07	**8.13e–13**	ABC-AX², ABC-III, II, IV, ABC
f_{14}	30	4.55e–08	9.54e–17	2.12e–18	5.71e–16	**5.63e–23**	ABC-AX², ABC-III, II, IV, ABC
f_{15}	30	3.34e–04	7.89e–10	6.95e–08	2.27e–11	**8.56e–13**	ABC-AX², ABC-IV, II, III, ABC
f_{16}	30	3.36e–01	1.84e–06	9.46e–08	3.89e–03	**6.46e–09**	ABC-AX², ABC-III, II, IV, ABC
f_{17}	30	5.47e–12	5.14e–14	4.28e–14	7.27e–13	**3.85e–14**	ABC-AX², ABC-III, II, IV, ABC
f_{18}	30	2.63e–03	5.32e–13	5.97e–17	4.98e–16	**2.33e–21**	ABC-AX², ABC-III, IV, II, ABC
Average Rank		4.82	3.12	2.24	3.59	**1.24**	ABC-AX², ABC-III, II, IV, ABC

In the next experiment, we try to find out whether and how the values of the control parameters p_i, q_i and $\boldsymbol{\eta_i}$ gradually evolve with the on-going optimization process. To discover the general trend of these control parameter values along the optimization process, we consider only their average values $\overline{p}$, $\overline{q}$ and $\overline{\eta}$, averaged across all the individuals of the population during each generation. More formally, the values of $\overline{p}$, $\overline{q}$ and $\overline{\eta}$ are computed as follows.

$$\overline{p} = \frac{1}{SN}\sum_{i=1}^{SN} \boldsymbol{x_i}.p \tag{8.3}$$

$$\overline{q} = \frac{1}{SN}\sum_{i=1}^{SN} \boldsymbol{x_i}.q \tag{8.4}$$

$$\overline{\boldsymbol{\eta}} = \frac{1}{SN}.\frac{1}{D}\sum_{i=1}^{SN}\sum_{j=1}^{D} \boldsymbol{x_i}.\eta_j \tag{8.5}$$

Thus $\overline{p}$, $\overline{q}$ and $\overline{\eta}$ are the population-wide average values of the control parameters p_i, q_i and $\boldsymbol{\eta_i}$ of every candidate solution $\boldsymbol{x_i}$ across the population. Figs. 8.12–8.14 show the gradual evolution of the values of $\overline{p}$, $\overline{q}$ and $\overline{\eta}$ along the course of the optimization process. The functions in these Figs. 8.12–8.14 include both unimodal (f_1, f_4, f_8) and multimodal (f_{12}, f_{13}, f_{18}), separable (f_1, f_8, f_{12}) and non-separable (f_4, f_{13}, f_{18}) functions. Our observations on these results (Figs. 8.12–8.14, Tables 8.25–8.27) are briefly summarized in the following few points.

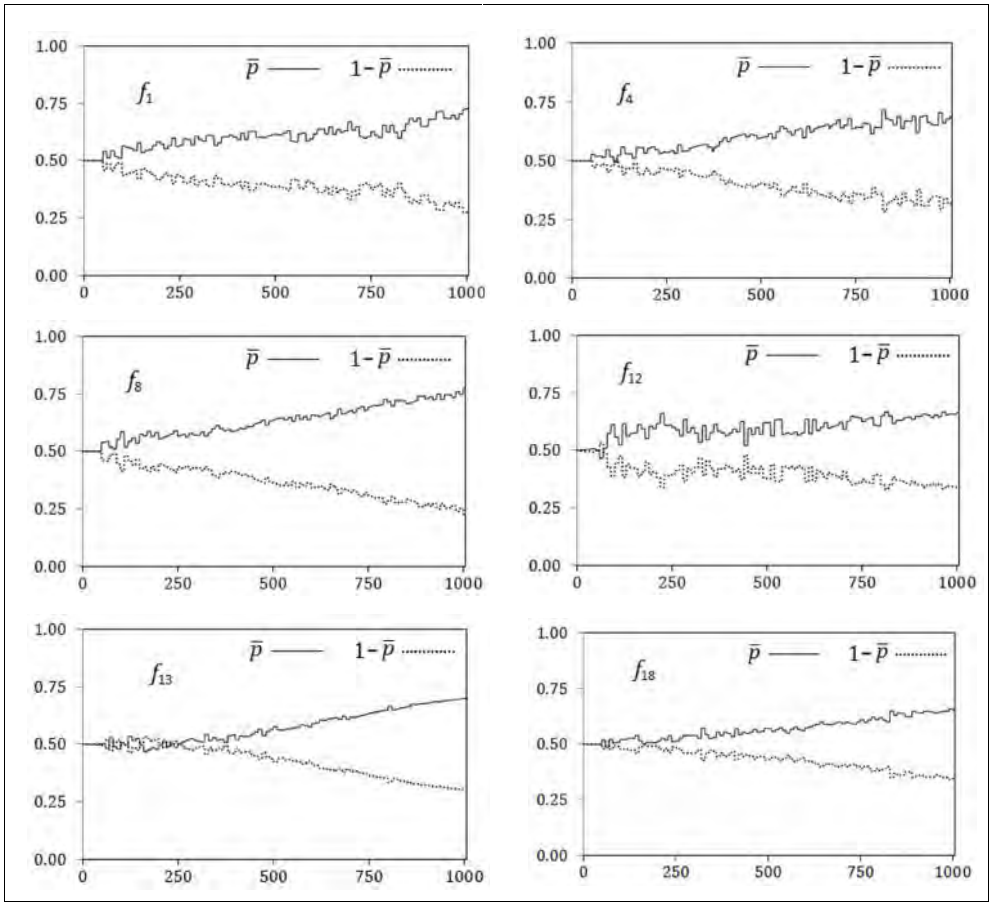


Figure 8.12: Evolution of the control parameter p_i (and, $1-\overline{p}_i$) by ABC-AX² along the ongoing cycles (iterations). The vertical axis shows the mean value of the control parameter (i.e., $\overline{p}_i$ and $1-\overline{p}_i$), while the horizontal axis shows the number of cycles elapsed.

Table 8.25: Mean values of the control parameter p_i (i.e., $\overline{p}$), averaged over all the candidate solutions of the population after the final generation of ABC-AX². Here, C is the function characteristics with values — *U*: Unimodal, *M*: Multimodal, *S*: Separable and *N*: Non-Separable.

Function	C	$\overline{p}$	**Function**	C	$\overline{p}$
f_1	*US*	0.78	f_{10}	*MS*	0.66
f_2	*UN*	0.87	f_{11}	*MS*	0.65
f_3	*US*	0.85	f_{12}	*MS*	0.66
f_4	*UN*	0.69	f_{13}	*MN*	0.69
f_5	*UN*	0.86	f_{14}	*MN*	0.62
f_6	*UN*	0.82	f_{15}	*MS*	0.64
f_7	*UN*	0.84	f_{16}	*MS*	0.60
f_8	*US*	0.79	f_{17}	*MN*	0.62
f_9	*US*	0.86	f_{18}	*MN*	0.65
Average		**0.82** (Unimodal)	**0.64** (Multimodal)		**0.73** (Overall)

Firstly, we consider the evolution of the control parameter p_i of the candidate solutions $\boldsymbol{x}_i$'-s across the population. The value of p_i and $1-\overline{p}_i$ indicate the probability of exploitative and explorative perturbations, respectively on the candidate solution $\boldsymbol{x}_i$. For all the functions in Fig. 8.12, $\overline{p}$ gradually increases towards larger values, indicating that exploitative perturbations become gradually more and more attractive to the candidate solutions, irrespective of whether the function is unimodal or multimodal, separable or non-separable. However, the intensity of 'more exploitations' is not the same for the unimodal and multimodal functions. Table 8.25 shows that the overall average $\overline{p}$ during the last cycle is 0.82 for the unimodal functions f_1–f_9, while it is 0.64 for the multimodal functions f_{10}–f_{18}. This indicates that the optimization process of the unimodal functions become more exploitative than the multimodal functions, which is rationally expected. Besides, during their final cycles, both the unimodal and the multimodal functions require more exploitations than explorations (i.e., the average $\overline{p}$ is 0.82 (unimodal) and 0.64 (multimodal), both of which are greater than 0.50). This is very rational, because more exploitations during the final cycles, especially after reaching the neighbourhood of the global minimum, are always more effective than explorations to reach the globally optimal peak of the fitness landscape.

Secondly, the evolution of the control parameter q_i is considered in Fig. 8.13 and Table 8.26. Each individual $\boldsymbol{x}_i$ has its control parameter q_i, which is self-adapted automatically along the optimization process. To observe the general trend and the direction of self-adaptation, we consider only the average value of q_i (denoted as $\overline{q}$), averaged over all the candidate solutions across the population using eq. (8.4). Fig. 8.13 shows that $\overline{q}$ is gradually self-adapted either towards larger values (e.g., for f_4, f_{13} and f_{18}) or towards smaller values (e.g., f_1, f_8 and f_{12}) depending on whether the function is separable or not. For a non-separable function, the parameters are not independent, so perturbing a subset of the parameters may not be useful. This is why $\overline{q}_i$ self-adapts towards larger values, which is observed for all three non-separable functions f_4, f_{13} and f_{18}, showing the necessity to perturb most (or, all) of the parameters at once for these functions. However, for the separable functions, f_1, f_8 and f_{12}, the value of $\overline{q}_i$ self-adapts to rather smaller values, which indicates that perturbing only one or just a few parameters is more effective for these separable problems, as the parameters are independent of each other. Table 8.26 shows the value of $\overline{q}$, measured after the final generation of ABC-AX^2, for all of the high dimensional functions f_1–f_{18}, which indicates the same general trend by the separable and non-separable functions — the average value of $\overline{q}$ over the non-separable functions is 0.84, which is quite high, while the same measure for separable functions is only 0.17.

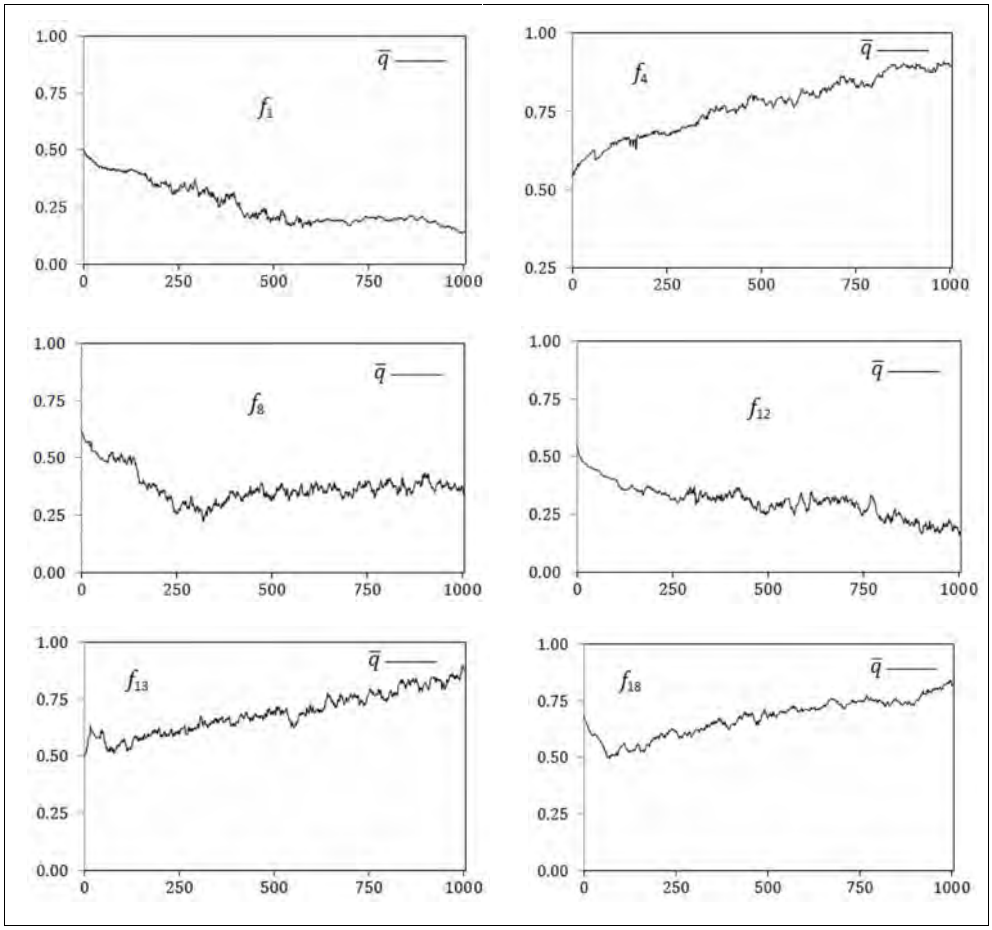


Figure 8.13: Evolution of the control parameter q_i by ABC-AX² along the ongoing cycles (iterations). The vertical axis shows the mean value of the control parameter q_i (i.e., $\overline{q}_i$), while the horizontal axis shows the number of cycles elapsed.

Table 8.26: Mean values of the control parameter q_i (i.e., $\overline{q}$), averaged over all the candidate solutions of the population after the final generation of ABC-AX². Here, C is the function characteristics with values — *U*: Unimodal, *M*: Multimodal, *S*: Separable and *N*: Non-Separable.

Function	C	$\overline{q}$	**Function**	C	$\overline{q}$
f_1	*US*	0.14	f_{10}	*MS*	0.19
f_2	*UN*	0.88	f_{11}	*MS*	0.12
f_3	*US*	0.16	f_{12}	*MS*	0.17
f_4	*UN*	0.89	f_{13}	*MN*	0.88
f_5	*UN*	0.84	f_{14}	*MN*	0.84
f_6	*UN*	0.85	f_{15}	*MS*	0.15
f_7	*UN*	0.79	f_{16}	*MS*	0.21
f_8	*US*	0.25	f_{17}	*MN*	0.77
f_9	*US*	0.14	f_{18}	*MN*	0.82
Average		**0.55** (Unimodal)	**0.46** (Multimodal)		**0.51** (Overall)
		0.17 (Separable)	**0.84** (Non-Separable)		

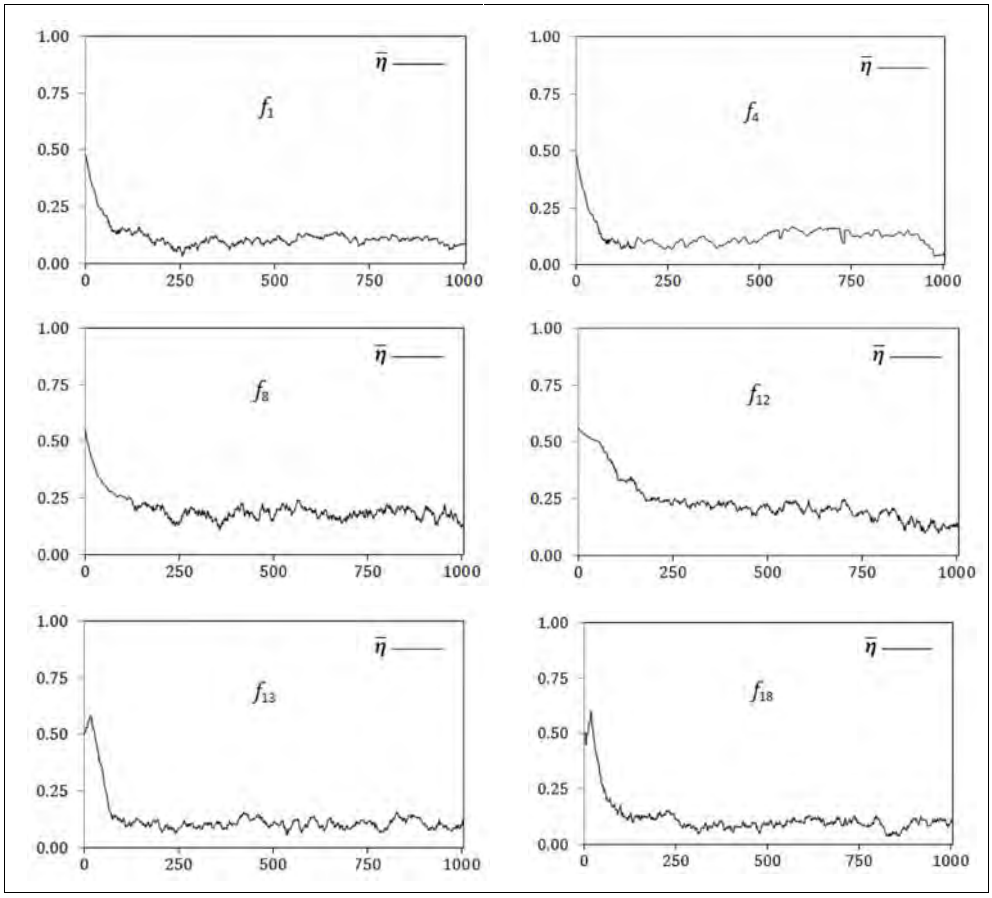


Figure 8.14: Evolution of the control parameter $\boldsymbol{\eta}_i$ by ABC-AX^2 along the ongoing cycles (iterations). The vertical axis shows the mean value of the control parameter (i.e., $\bar{\eta}$), while the horizontal axis shows the number of cycles elapsed.

Table 8.27: Mean values of the control parameter $\boldsymbol{\eta}_i$ (i.e., $\bar{\eta}$), averaged over all the candidate solutions of the population after the final generation of ABC-AX^2. Here, C is the function characteristics with values — U: Unimodal, M: Multimodal, S: Separable and N: Non-Separable.

Function	C	$\bar{\eta}$	**Function**	C	$\bar{\eta}$
f_1	US	0.09	f_{10}	MS	0.10
f_2	UN	0.08	f_{11}	MS	0.14
f_3	US	0.06	f_{12}	MS	0.12
f_4	UN	0.05	f_{13}	MN	0.10
f_5	UN	0.05	f_{14}	MN	0.13
f_6	UN	0.08	f_{15}	MS	0.07
f_7	UN	0.06	f_{16}	MS	0.08
f_8	US	0.22	f_{17}	MN	0.11
f_9	US	0.06	f_{18}	MN	0.10
Average		**0.08** (Unimodal)		**0.11** (Multimodal)	**0.095** (Overall)

Thirdly, we observe and analyse the evolution of the control parameter vector $\boldsymbol{\eta}_i$ in Fig. 8.14 and Table 8.27. In ABC-AX^2, each candidate solution $\boldsymbol{x}_i$ maintains its own control parameter vector $\boldsymbol{\eta}_i = [\eta_{i1}, \eta_{i2}, \ldots, \eta_{iD}]^T$, where each component η_{ij} controls the distribution of the scaling factor values (i.e., φ_{ij} values in (2.6)) during perturbing the corresponding parameter x_{ij} of the candidate solution $\boldsymbol{x}_i$. The φ_{ij} values in (2.6) are generated randomly using the Gaussian$(0, \eta_{ij})$ distribution, for $j = 1, 2, \ldots, D$. To observe the general trend of the η_{ij} values, we consider their population-wide average value $\bar{\eta}$, computed by eq. (8.5), in Fig. 8.14 and Table 8.27. For all the functions in Fig. 8.14, it is observed that the $\bar{\eta}$ quickly drops towards smaller values, indicating that exploitative perturbations with small φ_{ij} values (i.e., small, exploitative step sizes) become gradually more and more effective along the course of the optimization process. Table 8.27 shows this general trend for both unimodal (average $\bar{\eta}$=0.08) and multimodal (average $\bar{\eta}$=0.11) functions. However, the slightly smaller value (i.e., 0.08) of average $\bar{\eta}$ for the unimodal functions indicates that they need overall more exploitations, especially during their final cycles, compared to the multimodal functions. However, the multimodal functions also require more exploitations than explorations during their final cycles, as demonstrated by the plots of f_{12}, f_{13} and f_{18} in Fig. 8.14 and by the quite low value of the average $\bar{\eta}$ (i.e., 0.11) in Table 8.27.

Now, we experiment on whether the adaptation and self-adaptation of the control parameters p_i, q_i and $\boldsymbol{\eta}_i$ have effectively improved the rate of successful perturbations by ABC-AX^2. To find out, we have measured the average successful perturbation rate achieved by ABC and ABC-AX^2 for the high dimensional functions f_1–f_{18} with SN=100, MCN=1000 and $limit$=100. Table 8.28 shows that the percentage of perturbations that are successful (i.e., the new trial solution $\boldsymbol{v}_i$ has higher fitness value than the original solution $\boldsymbol{x}_i$) is always higher by ABC-AX^2 compared to the basic ABC algorithm. For some functions (e.g., f_{14} and f_{18}) the success rate of ABC-AX^2 is much higher than ABC, which also coincides with the much improved results of ABC-AX^2 for the same functions (f_{14} and f_{18}) in Table 7.1, indicating a strongly positive correlation between the improved results and the higher success rates of perturbations by ABC-AX^2. This indicates that the proposed adaptive and self-adaptive strategies of ABC-AX^2 for p_i, q_i and $\boldsymbol{\eta}_i$ can effectively produce more successful perturbations than can produce better trial solutions from the existing ones and thus result in significantly improved performance over the basic ABC algorithm.

Table 8.28: Comparison between ABC and ABC-AX[2], based on their successful perturbation rates on the high dimensional functions f_1–f_{18}. The best result for each function is marked with boldface font.

Function	*D*	Successful Perturbation Rate (%)	
		ABC	ABC-AX[2]
f_1	30	17.23	**24.75**
f_2	30	14.35	**20.66**
f_3	30	10.96	**14.16**
f_4	30	18.82	**24.09**
f_5	24	11.31	**18.60**
f_6	30	12.92	**16.84**
f_7	30	**17.75**	14.20
f_8	30	**19.20**	18.41
f_9	30	18.46	**28.34**
f_{10}	30	18.30	**22.94**
f_{11}	30	15.42	**19.53**
f_{12}	30	12.64	**18.12**
f_{13}	30	13.38	**19.30**
f_{14}	30	12.75	**23.44**
f_{15}	30	13.26	**18.62**
f_{16}	30	10.42	**17.47**
f_{17}	30	16.01	**19.52**
f_{18}	30	11.64	**22.87**
+		16	
–		2	
≈		0	

8.8 Comparison Among the Proposed Algorithms

In this section, we will compare our algorithms against each other to better understand their strengths and weaknesses. Along the course of this thesis, we have developed two improved EP-variants — the RTEP and DGEP (chapters 3–4), as well as three improved variants of the standard ABC algorithm — the ABC-SAM, ABC-IX and ABC-AX[2] (chapters 5–7). In the following paragraphs, each of these algorithms is compared against relevant other algorithms using a number of experiments.

In Tables 8.29–8.32, we make comparisons between RTEP and DGEP, based on their final solution quality (Tables 8.29, 8.31), population diversity (Table 8.30) and resilience against premature convergence (Table 8.32). For the results in Table 8.29, both the algorithms are run with population size = 50 and the number of function evaluations (*FE*) = 150,000. The other parameters of RTEP and DGEP are set to the same values as in Table 3.1 and Table 4.1 (DGEP[1]), respectively. A summary of the results observed in Tables 8.29–8.32 are presented in the following few points.

- Out of the 18 standard benchmark functions f_1–f_{18}, DGEP performs better than the RTEP variants on seven or eight functions, performs equally good on one or two, while both the RTEP variants perform better than DGEP on nine functions. Thus, the overall performance of DGEP is slightly worse than the RTEP variants on the standard benchmark functions f_1–f_{18}.
- Among the 18 standard benchmark functions, there are only three functions (i.e., f_3, f_7 and f_{12}) for which the RTEP variants face some difficulties. Interestingly, DGEP outperforms the RTEP variants on all of these three functions. This indicates that DGEP might be more effective than RTEP for more complex optimization tasks.

Table 8.29: Comparison between RTEP and DGEP variants, based on their final solution quality (i.e., mean error of their final results), on the high dimensional functions f_1–f_{18}. The best result for each function is marked with boldface font.

Function	Mean Error ± Standard Deviation			*t*-Test	
	DGEP	RTEP (2,4)	RTEP (4,8)	DGEP vs. RTEP (2,4)	DGEP vs. RTEP (4,8)
f_1	5.4e–08 ± 1.5e–08	7.5e–18 ± 4.4e–18	**2.4e–20 ± 7.4e–21**	–	–
f_2	6.6e–12 ± 3.6e–12	1.7e–09 ± 1.5e–09	**2.9e–12 ± 6.8e–13**	–	–
f_3	**1.06 ± 0.32**	1.7e+00 ± 6.2e–01	1.9e+00 ± 1.4e+00	+	+
f_4	4.1e–08 ± 9.2e–09	2.4e–15 ± 6.2e–16	**2.1e–15 ± 4.1e–16**	–	–
f_5	**2.5e–04 ± 1.3e–04**	2.6e–03 ± 7.7e–04	2.0e–03 ± 4.9–e04	+	+
f_6	**4.1e–02 ± 7.9e–03**	1.3e–01 ± 3.9e–02	1.2e–01 ± 3.5e–02	+	+
f_7	**1.1 ± 0.76**	2.7e+00 ± 9.0e–01	3.5e+00 ± 6.6e–01	+	+
f_8	0 ± 0	0 ± 0	0 ± 0	≈	≈
f_9	1.9e–12 ± 8.2e–13	**8.0e–38 ± 6.4e–39**	2.5e–34 ± 1.4e–35	–	–
f_{10}	1.5e–12 ± 5.1e–13	2.5e–14 ± 5.0e–15	**1.9e–14 ± 6.1e–15**	–	–
f_{11}	2.8e–05 ± 4.1e–06	2.9e–07 ± 5.1e–08	**1.1e–06 ± 9.1e–08**	–	–
f_{12}	**2.10 ± 1.38**	7.1e+02 ± 4.9e+02	3.6e+02 ± 9.9e+01	+	+
f_{13}	2.4e–16 ± 6.8e–17	**2.0e–11 ± 6.5e–12**	2.4e–09 ± 9.2e–10	–	–
f_{14}	7.5e–14 ± 2.5e–14	**2.7e–25 ± 6.3e–26**	8.4e–20 ± 3.3e–20	–	–
f_{15}	**9.5e–13 ± 2.5e–13**	7.8e–10 ± 9.4e–11	2.9e–12 ± 4.1e–13	+	+
f_{16}	**5.5e–08 ± 8.2e–09**	2.2e–07 ± 9.1e–08	6.1e–08 ± 2.1e–08	+	≈
f_{17}	**3.3e–14 ± 1.9e–14**	3.2e–13 ± 8.5e–14	1.7e–13 ± 2.7e–14	+	+
f_{18}	1.7e–04 ± 2.6e–05	**7.1e–08 ± 7.3e–09**	7.2e–05 ± 2.2e–05	–	–
Summary (*t*-Test)	DGEP +	8	7		
	DGEP –	9	9		
	DGEP ≈	1	2		

- Between RTEP and DGEP, which one performs more search space explorations? To find out, we have executed both the algorithms with population size = 50 and FE = 100,000. At the end of the run, the genetic diversity of their final generation population is measured and presented in Table 8.30. Results show that DGEP maintains higher level of genetic diversity than RTEP on seven out of nine unimodal (i.e., f_1–f_9) and eight out of nine multimodal (f_{10}–f_{18}) functions. Also, for both the unimodal and multimodal functions, the overall average amount of diversity maintained by DGEP is higher than RTEP, which indicates that DGEP performs an overall higher level of search space explorations than RTEP. This might be the reason for the better performance of DGEP on the more difficult and challenging problems, e.g., f_3, f_7 and f_{12}, as mentioned in the previous point.
- The previous two points suggest that DGEP is more explorative than RTEP and is more suitable for complex optimization tasks. Therefore, DGEP should be better effective on the more challenging CEC2005 benchmark functions. To find out whether this does really happen, we have run both the algorithms on the CEC2005 benchmark functions F_1–F_{25}, with D = 30, population size = 50 and FE = 3.0e+05. Results (Table 8.31) show that DGEP outperforms RTEP on as many as 18 (out of 25) CEC2005 functions, while RTEP performs better only on the remaining seven. Also, the mean absolute error of DGEP over F_1–F_{25} is 2.36e+02, which is smaller than the mean error of RTEP (4.35e+02). This further consolidates our assumption that DGEP is more suitable than RTEP for more complex and more challenging optimization problems.

Table 8.30: Comparison between RTEP and DGEP, based on their population diversity, measured at the end of their runs. Diversity is measured as the average Euclidean distance of the individuals from the centroid of the population. The higher diversity value for each function is marked with boldface font.

Function	Population Diversity		Function	Population Diversity	
	RTEP(4,8)	DGEP		RTEP(4,8)	DGEP
f_1	8.29e–06	**4.21e–04**	f_{10}	7.91e–07	**2.29e–04**
f_2	8.40e–04	**7.76e–03**	f_{11}	4.66e–03	**7.40e–03**
f_3	**6.52e–02**	5.11e–02	f_{12}	5.20e–04	**1.13e–01**
f_4	3.83e–04	**7.09e–02**	f_{13}	3.95e–05	**3.32e–03**
f_5	2.29e–03	**8.91e–01**	f_{14}	7.99e–05	**2.14e–04**
f_6	**8.94e–02**	5.52e–02	f_{15}	**3.54e–05**	1.77e–05
f_7	8.49e–01	**1.37e+00**	f_{16}	8.04e–04	**2.05e–03**
f_8	9.28e–04	**3.64e–01**	f_{17}	1.67e–06	**4.28e–05**
f_9	2.15e–03	**1.56e–02**	f_{18}	4.75e–07	**5.06e–05**
Average f_1–f_9	1.12e–01	**3.54e–01**	**Average** f_{10}–f_{18}	6.82e–04	**1.40e–02**

Table 8.31: Comparison between RTEP and DGEP, based on their final solution quality (i.e., mean error of their final results), on the CEC2005 benchmark functions. The best result for each function is marked with boldface font.

Function	Mean Error		Function	Mean Error	
	DGEP	RTEP (2,4)		DGEP	RTEP (2,4)
F_1	7.56e–08	**7.62e–09**	F_{13}	**1.03e–01**	1.54e–01
F_2	3.98e–10	**2.44e–11**	F_{14}	**5.13e+00**	8.94e+00
F_3	**2.24e+01**	9.25e+01	F_{15}	**4.88e+02**	5.05e+03
F_4	9.76e+02	**8.99e+02**	F_{16}	**3.72e+01**	5.84e+01
F_5	**5.11e+00**	7.64e+00	F_{17}	3.62e+01	**2.38e+01**
F_6	**2.19e+01**	5.01e+02	F_{18}	8.49e+01	**7.41e+01**
F_7	**7.23e–06**	2.85e–05	F_{19}	**8.25e+02**	9.96e+02
F_8	**2.00e+01**	1.47e+02	F_{20}	**7.91e+02**	1.80e+03
F_9	**1.87e–04**	3.69e–04	F_{21}	**5.00e+02**	5.11e+02
F_{10}	**2.08e+00**	5.92e+00	F_{22}	8.41e+02	**1.42e+02**
F_{11}	**5.15e+00**	7.39e+00	F_{23}	5.22e+02	**8.01e+01**
F_{12}	**3.09e+02**	4.73e+02	F_{24}	**2.00e+02**	7.98e+02
			F_{25}	**2.05e+02**	1.01e+02
Summary	DGEP +	9	**Summary**	DGEP +	9
	DGEP –	3		DGEP –	4
Mean Absolute Error		(DGEP) **2.36e+02**		(RTEP) 4.35e+02	

Between DGEP and RTEP, which one is more resilient against premature convergence? To find out, we have run both the algorithms 100 times on each of the standard benchmark functions f_1–f_{30}, each time starting from a very poor initial population that is randomly initialized within only 1% of the search space along each search dimension. The 1% initialization region along each dimension is picked uniformly at random along that particular search dimension. We employed such a poorly initialized population to represent a prematurely converged state from which both RTEP and DGEP starts their execution. In addition to RTEP(4,8), we also employed two more explorative RTEP variants — RTEP(8,4) and RTEP(20,4). Similarly, a more explorative DGEP variant — DGEP-II is employed, which sets the parameter values for more explorations, e.g., $u = 25$, $l = 0.05$ and $|A| = 15$. In Table 8.32, a particular run of DGEP or RTEP is considered 'successful' if it can reach sufficiently close to the global optimum. More precisely, a run is considered successful if and only if it can find a candidate solution $\boldsymbol{x}^*$ such that $|f(\boldsymbol{x}^{**}) - f(\boldsymbol{x}^*)| \leq 0.1$, where $\boldsymbol{x}^{**}$ is the globally minimum point. Table 8.32 shows that both the DGEP variants can produce more successful runs (i.e., with overall approx. 81% and 87% success rates) in comparison to the RTEP variants (overall 64%, 73% and 79% success rates). This indicates that DGEP is relatively more resilient than RTEP

Table 8.32: Comparison among the RTEP and DGEP variants based on their strength against premature convergence, measured as their number of successful runs (out of 100 test runs) to locate the global optimum, starting from a prematurely converged initial population. The best result for each function is marked with boldface font.

Function	No. of Successful Runs				
	RTEP(4,8)	**RTEP(8,4)**	**RTEP(20,4)**	**DGEP**	**DGEP-II**
f_1	87	98	**100**	**100**	**100**
f_2	76	88	97	**100**	**100**
f_3	81	86	98	93	**100**
f_4	73	82	95	92	**98**
f_5	12	19	34	40	**58**
f_6	59	66	82	75	**86**
f_7	17	24	32	35	**47**
f_8	69	74	78	75	**92**
f_9	86	96	**100**	**100**	**100**
f_{10}	65	71	88	83	**91**
f_{11}	37	52	62	58	**70**
f_{12}	8	11	18	17	**28**
f_{13}	53	82	88	93	**95**
f_{14}	80	83	**96**	93	**96**
f_{15}	58	86	86	90	**93**
f_{16}	52	79	87	84	**95**
f_{17}	34	54	69	67	**75**
f_{18}	24	28	48	45	**62**
f_{19}	95	99	100	100	100
f_{20}	96	99	100	100	100
f_{21}	100	100	100	100	100
f_{22}	100	100	100	100	100
f_{23}	100	100	100	100	100
f_{24}	100	100	100	100	100
f_{25}	58	63	73	76	87
f_{26}	50	62	66	81	85
f_{27}	57	65	72	89	92
f_{28}	45	52	60	70	76
f_{29}	100	100	100	100	100
f_{30}	48	60	66	79	86
Average Unimodal (f_1–f_9)	62.22	70.33	79.56	78.89	**86.78**
Average Multimodal (f_{10}–f_{18})	45.67	60.67	71.33	70	**78.33**
Average Multimodal (f_{19}–f_{30})	79.17	83.33	86.42	91.25	**93.83**
Average Overall (f_1–f_{30})	64.03	72.63	79.33	81.17	**87.07**

against premature convergence. However, with more increased values of the control parameter K_1 (i.e., length of the exploration stage), RTEP(K_1, K_2) might improve its results further, because the strength of RTEP against premature convergence seems to increase directly with the higher values of its control parameter K_1. This becomes apparent when we observe the results of RTEP(K_1, K_2) in Table 8.32 with $(K_1, K_2) = (4, 8)$, $(8, 4)$ and $(20, 4)$ — the overall success rates of these RTEP variants increase from (approx.) 64% to 73% to 79% for the gradually increased values of $K_1 = 4$, 8 and 20, respectively.

Now we compare the three improved ABC-variants that we have developed so far — ABC-SAM, ABC-IX and ABC-AX^2. Tables 8.33–8.36 compare them based on their final solution quality and explorative capability. In all these experiments, their common parameters are set as — colony size $(SN) = 100$, maximum cycle number $(MCN) = 1000$ and $limit = 100$. The other parameters of ABC-SAM, ABC-IX and ABC-AX^2 are the same as in Table 6.2 and Table 7.1. Our observations on the experimental results are briefly summarized in the following few points.

- In Table 8.33, the results of ABC-SAM, ABC-IX and ABC-AX^2 are presented side-by-side, for both 30-D and 60-D variants of the standard benchmark functions f_1–f_{18}. Results show that both ABC-IX and ABC-AX^2 outperform ABC-SAM on most of these functions. Out of the 18 functions, ABC-IX and ABC-AX^2 perform better than ABC-SAM on as many as 14 and 16 functions, respectively, while ABC-SAM performs equally well or better only on the remaining few (2∼4) functions.
- Between ABC-IX and ABC-AX^2, the overall better performance is shown by ABC-AX^2. Out of the 18 standard benchmark functions, ABC-AX^2 outperforms ABC-IX on 11 functions, while ABC-IX performs equally good or better on the remaining seven. However, the strength of ABC-IX is not in fine-tuning, rather in its more explorative search capacity than ABC-AX^2, which is explained in the next two points, using the population diversity and results on the more complex CEC2005 benchmark functions.
- Table 8.34 compares the explorative search capacity of ABC-IX and ABC-AX^2, based on their existing amount of population diversity, measured at the end of their runs. The population diversity is computed as the mean distance of the candidate solutions from the centroid of the population. Table 8.34 shows that ABC-IX generally maintains higher level of population diversity than ABC-AX^2. Out of the 18 standard functions, the diversity of ABC-IX is higher than ABC-AX^2 on as many as 14 functions, which clearly indicates that ABC-IX, with its simulated annealing based selection scheme that can probabilistically make both uphill and downhill movements, can perform more search space explorations than ABC-AX^2.

Table 8.33: Comparison among ABC-SAM, ABC-IX and ABC-AX², based on their final solution quality (mean error of their final results), on the standard benchmark functions f_1–f_{18}. The best result for each function is marked with boldface font.

No	f_{min}	D	G	ABC-SAM		ABC-IX		ABC-AX²	
				Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
f_1	0	30	1000	4.18e–14	5.37e–15	**2.84e–38**	**1.03e–38**	5.51e–24	3.73e–25
		60	2000	6.09e–13	7.24e–14	**6.07e–30**	**8.55e–31**	9.43e–28	7.26e–29
f_2	0	30	1000	2.47e–08	2.35e–09	8.30e–15	3.14e–15	**4.23e–15**	**3.54e–16**
		60	2000	5.06e–07	2.97e–07	3.01e–12	4.74e–13	**2.98e–17**	**1.07e–17**
f_3	0	30	1000	1.69e+01	1.43	8.37e+00	2.64	**6.60e–02**	**5.21e–03**
		60	2000	3.10e+01	5.12	1.41e+01	5.15	**2.78**	**0.77**
f_4	0	30	1000	3.95e–12	5.77e–13	**6.43e–31**	**7.37e–32**	3.42e–16	8.83e–18
		60	2000	7.54e–11	2.14e–11	**1.18e–26**	**4.95e–27**	8.84e–20	5.45e–21
f_5	0	24	1000	9.24e–01	2.08e–01	3.63e–01	6.18e–02	**2.23e–02**	**3.75e–03**
f_6	0	30	1000	2.16e–03	6.37e–04	5.86e–04	7.31e–05	**5.91e–05**	**5.67e–06**
		60	2000	7.76e–02	1.63e–02	9.88e–03	8.59e–03	**8.33e–05**	**1.71e–05**
f_7	0.	30	1000	2.28e+01	3.75	**8.96e+00**	**3.63e+00**	2.39e+01	3.66
		60	2000	4.96e+01	7.80	**1.35e+00**	**8.93e–01**	5.15e+01	7.69
f_8	0.	30	1000	0	0	0	0	0	0
		60	2000	0	0	0	0	0	0
f_9	0	30	1000	3.66e–16	1.44e–17	**8.31e–52**	**2.19e–52**	8.87e–34	6.78e–35
		60	2000	4.76e–15	5.32e–16	**6.09e–41**	**2.43e–42**	6.31e–32	2.16e–33
f_{10}	0	30	1000	1.26e–16	2.11e–17	**2.21e–35**	**1.60e–36**	4.68e–24	9.03e–26
		60	2000	8.55e–15	3.15e–16	**2.91e–32**	**6.44e–33**	6.12e–31	8.67e–33
f_{11}	0	30	1000	4.60e–10	8.85e–11	5.10e–11	9.78e–12	**1.04e–13**	**3.16e–14**
		60	2000	6.80e–09	8.77e–10	3.63e–10	1.35e–10	**4.25e–13**	**7.32e–14**
f_{12}	–12569.5	30	1000	–12416.19	4.02e+01	–12473.13	1.52e+01	**–12569.48**	**1.50e–02**
	–25138.9	60	2000	–23805.93	2.84e+02	–24281.80	1.35e+02	**–25016.6**	**1.89e+01**
f_{13}	0	30	1000	9.26e–08	1.89e–08	8.13e–13	6.71e–14	**4.20e–15**	**5.23e–16**
		60	2000	2.07e–08	3.55e–08	8.84e–14	1.40e–14	**3.62e–14**	**1.15e–15**
f_{14}	0	30	1000	8.36e–10	5.08e–11	**5.12e–33**	**1.55e–33**	5.63e–23	7.35e–25
		60	2000	1.56e–10	6.90e–11	**4.94e–32**	**9.98e–33**	7.04e–31	5.77e–32
f_{15}	0.	30	1000	2.22e–08	3.93e–09	8.75e–08	2.38e–08	**8.56e–13**	**1.56e–13**
		60	2000	1.17e–08	2.35e–09	7.08e–08	1.33e–08	**5.37e–13**	**1.25e–13**
f_{16}	0	30	1000	5.78e–04	6.31e–05	5.24e–06	2.04e–06	**6.46e–09**	**8.32e–10**
		60	2000	9.20e–03	4.03e–03	2.72e–05	5.12e–06	**5.38e–08**	**9.19e–10**
f_{17}	0.	30	1000	9.78e–12	3.89e–12	3.22e–10	7.88e–11	**3.85e–14**	**4.93e–15**
		60	2000	1.32e–11	5.15e–11	6.46e–10	1.59e–10	**3.50e–14**	**2.60e–15**
f_{18}	0.	30	1000	3.06e–02	8.59e–03	7.95e–02	2.14e–02	**2.33e–21**	**7.55e–22**
		60	2000	5.11e–02	7.39e–03	9.70e–02	2.05e–02	**7.52e–26**	**1.29e–26**
Summary		ABC-AX² +		16		11			
		ABC-AX² –		0		6			
		ABC-AX² ≈		2		1			

Table 8.34: Comparison between ABC-IX and ABC-AX², based on their population diversity, measured at the end of their run, on the high dimensional functions f_1–f_{18}. Diversity is measured as the average Euclidean distance of the individuals to the centroid of the population. The higher diversity value for each function is marked with boldface font.

Function	Population Diversity		Function	Population Diversity	
	ABC-IX	ABC-AX²		ABC-IX	ABC-AX²
f_1	**8.08e–04**	4.56e–04	f_{10}	**9.87e–03**	1.36e–04
f_2	**5.78e–04**	2.22e–05	f_{11}	6.26e–04	**6.82e–04**
f_3	**5.99e–01**	5.80e–02	f_{12}	2.88e–01	**3.04e–01**
f_4	**8.75e–03**	1.65e–06	f_{13}	**7.10e–05**	4.35e–05
f_5	**8.30e–03**	5.85e–03	f_{14}	**1.57e–04**	6.94e–06
f_6	7.06e–02	**7.85e–02**	f_{15}	**7.25e–02**	3.03e–03
f_7	6.84e–02	**6.95e–02**	f_{16}	**8.40e–01**	4.64e–02
f_8	**5.23e–01**	9.26e–03	f_{17}	**3.50e–05**	1.88e–08
f_9	**2.56e–05**	4.48e–07	f_{18}	**9.95e–04**	4.59e–04
Average f_1–f_9	**1.42e–01**	2.46e–02	**Average** f_{10}–f_{18}	**1.35e–01**	3.94e–02

The previous point suggests that ABC-IX is more explorative than ABC-AX². Does it make ABC-IX more effective on the more complex optimization tasks, such as the more challenging CEC2005 functions? To find out, we have run both the algorithms on the CEC2005 benchmark functions F_1–F_{25}, with $D = 30$, population size = 50 and $FE = 3.0e+05$, and the results are presented in Tables 8.35–8.36. The results on the non-composite functions F_1–F_{14} (Table 8.35) show that ABC-IX outperforms ABC-AX² on as many as seven (out of 14) functions, shows similar performance on two, and worse performance on the remaining five functions. For the hybrid composite functions F_{15}–F_{25} in Table 8.36 ABC-IX performs slightly worse than ABC-AX² (ABC-IX is found to be better on four, similar on two and worse on five functions). Summarizing both the Tables 8.35–8.36, ABC-IX performs better than ABC-AX² on 11 functions and worse on 10; so the overall performance of ABC-IX is slightly better than ABC-AX² on the complex CEC2005 benchmark functions, which might be due to its more explorative search capacity from its more population diversity, as demonstrated in the previous point.

8.9 Developing New and Improved Variants

Along the course of this thesis, we developed five improved EAs and SIAs — two of them (i.e., RTEP and DGEP) belong to the evolutionary family of algorithms, while the remaining three (i.e., ABC-SAM, ABC-IX and ABC-AX²) belong to the swarm intelligence based algorithm family. The techniques that we have employed in these algorithms are based on adaptation, self-adaptation,

Table 8.35: Comparison among ABC, ABC-SAM, ABC-IX and ABC-AX², based on their final solution quality, on the non-composite functions F_1–F_{14} of the CEC2005 benchmark suite [76]. The best result for each function is marked with boldface font.

Function		Mean Error			
		ABC	**ABC-SAM**	**ABC-AX²**	**ABC-IX**
F_1		4.89e–17	3.73e–21	5.63e–16	**5.92e–26**
F_2		4.81e–14	7.03e–16	1.12e–15	**8.73e–18**
F_3		2.50e+03	7.80e+01	**4.42e–06**	6.22e+00
F_4		1.50e–16	6.55e–18	**3.49e–19**	2.82e–18
F_5		5.82e+01	4.72e+00	3.62e–03	**1.04e–03**
F_6		3.31e+00	5.61e–02	6.89e–02	**9.28e–03**
F_7		2.52e–01	3.13e+00	**2.02e–04**	5.25e–01
F_8		2.03e+01	2.03e+01	**2.00e+01**	**2.00e+01**
F_9		4.87e–17	**8.58e–22**	1.47e–17	3.82e–20
F_{10}		2.22e+01	3.11e+00	**2.10e+00**	**2.11e+00**
F_{11}		5.46e+00	9.27e+00	**7.08e–02**	3.17e+00
F_{12}		9.85e+01	4.43e–01	1.19e+01	**6.99e–02**
F_{13}		2.96e–02	7.09e–01	**6.56e–03**	4.72e–02
F_{14}		3.41e+00	2.23e+00	2.24e+00	**2.05e+00**
Summary	ABC-IX +	13	13	7	
	ABC-IX –	1	1	5	
	ABC-IX ≈	0	0	2	

Table 8.36: Comparison among ABC, ABC-SAM, ABC-IX and ABC-AX² on the hybrid composition functions F_{15}–F_{25} of the CEC2005 benchmark suite [76]. The best result for each function is marked with boldface font.

Function		Mean Error			
		ABC	**ABC-SAM**	**ABC-AX²**	**ABC-IX**
F_{15}		1.53e+01	4.46e+00	3.78e+00	**3.17e+00**
F_{16}		1.75e+02	8.20e+01	1.69e+02	**7.37e+01**
F_{17}		1.96e+02	2.13e+02	2.02e+02	**1.51e+02**
F_{18}		4.46e+02	4.32e+02	**3.59e+02**	4.12e+02
F_{19}		4.51e+02	4.18e+02	**3.07e+02**	3.92e+02
F_{20}		4.38e+02	5.11e+02	**3.92e+02**	4.49e+02
F_{21}		4.87e+02	4.69e+02	**4.58e+02**	4.74e+02
F_{22}		8.59e+02	7.65e+02	7.62e+02	**6.98e+02**
F_{23}		5.98e+02	5.63+02	**5.57e+02**	**5.57e+02**
F_{24}		2.02e+02	2.01e+02	**2.00e+02**	**2.00e+02**
F_{25}		3.38e+02	3.48e+02	**3.27e+02**	3.53e+02
Summary	ABC-IX +	9	9	4	
	ABC-IX –	2	2	5	
	ABC-IX ≈	0	0	2	

hybridization, recurring alternations and diversity feedback control. Each of these techniques can be suitably altered, extended and incorporated into other algorithms to (possibly) improve their performance. In the following few points, we present some examples of how each of our proposed algorithms might be improved and extended further by employing some simple, yet effective techniques.

(i) RTEP (chapter 3) currently uses fixed values of its control parameters K_1 and K_2, which are the lengths of the explorative and exploitative stages, respectively. A more appropriate design would be to automatically adapt the values of K_1 and K_2 to ensure a dynamically appropriate adaptive length for the explorative and exploitative stages. To accomplish this, one simple strategy could be to estimate the improvements made by the explorative and exploitative stages and use this estimate to allow more execution length for the more successful stage (explorative or exploitative).

(ii) DGEP (chapter 4) tries to put an equal (rather than adaptive) emphasis on explorations and exploitations. This is why, during each mutation, DGEP picks one neighbor (for exploitations) and one non-neighbor (for explorations). But using two individuals for each single mutation involves two (instead of one) function evaluations, which is wasteful. A more appropriate strategy would be to probabilistically pick either a neighbor or a non-neighbor (but not both), either for exploitations or for explorations, based on an adaptive probability value p_i, maintained separately for every candidate solution $\boldsymbol{x}_i$. A somewhat similar strategy has been adopted by ABC-AX^2 (chapter 7), which introduces and automatically adapts a probability value p_i (i.e., probability of exploitative mutation on $\boldsymbol{x}_i$) for every candidate solution $\boldsymbol{x}_i$.

(iii) Both RTEP and DGEP perform explorations and exploitations using dissimilar and similar individuals, respectively. However, exploration and exploitation could be performed in many different ways, such as by employing simulated annealing based selection and/or perturbation schemes (as done by ABC-IX in chapter 6) or by using some other metaheuristic techniques (e.g., tabu search [28], iterated local search [77]) that allows some mechanism of control over the degree of explorations and exploitations around a candidate solution. Hybridizing such a metaheuristic technique with RTEP and DGEP might further improve their final solution quality and convergence speed.

(iv) In chapters 5–7, we have introduced three improved variants of ABC — ABC-SAM, ABC-IX and ABC-AX^2. Each one of them follows a single stage execution model. But a recurring execution model, like RTEP, might be more effective than the monotonous single stage model to balance between global explorations and local exploitations. To

accomplish this, each of ABC-SAM, ABC-IX and ABC-AX^2 may use two different settings of their operators and parameters — one for explorative stage, and the other one for exploitative stage. Then alternations between these two stages, either periodically or with adaptive stage lengths, may be more effective to improve their performance and to effectively balance between global explorations and local exploitations, as demonstrated by RTEP in chapter 3.

(v) None of our improved ABC-variants — ABC-SAM, ABC-IX and ABC-AX^2, makes any use of the population diversity information. In chapter 4, the experimental results on DGEP has demonstrated that employing the diversity information can be very useful to effectively control and guide the mutation operations. Since perturbation (or, mutation) is the sole variation operator in each of ABC-SAM, ABC-IX and ABC-AX^2, their performance possibly could be improved by estimating their population diversity during each cycle, then employing this information to dynamically control the perturbation rate and step size, as demonstrated by the DGM (diversity guided mutation) scheme of DGEP in chapter 4.

(vi) Each of the techniques we proposed along the course of this thesis is based on adaptation, self-adaptation, hybridization, recurring alternations and diversity feedback control to guide the perturbation operations. These techniques can often be hybridized, after suitable modifications and extensions, with many other existing metaheuristic algorithms to possibly improve their performance. This has been demonstrated in our another recent work — Scatter Search with Adaptive Diversity Control (SS-ADC) [235]. SS-ADC hybridizes the techniques of adaptation and diversity feedback control with the standard Scatter Search algorithm [236]. The overall performance of SS-ADC [235] is better than the standard scatter search algorithm [236], which indicates the effectiveness of our proposed techniques based on adaptation and diversity feedback control to improve the performance of many other existing metaheuristic algorithms.

The points mentioned above indicate that there exist many possibilities and directions along which each of our algorithms might be extended and improved. But it is beyond the scope of our thesis to explore all these possibilities. However, we have explored, to some extent, only one of the above design proposals — the proposal (i) above, which tries to automatically adapt the lengths of the exploration and exploitation stages (i.e., K_1 and K_2, respectively) of RTEP, based on the relative success and failure of each stage. The success of a particular stage (exploration or exploitation) is estimated by its Fitness Gain per Generation (*FGPG*), which is defined as follows. Suppose, an explorative stage executes for the last K_1 generations and causes

the best fitness value of the population to progress from f^* to f^{**}. Then, the total fitness gain (Δf_{expr}) and the *FGPG* of the current explorative stage is computed as $\Delta f_{\text{expr}} = f^{**} - f^*$, and $FGPG_{\text{expr}} = \Delta f_{\text{expr}} / K_1$. Similarly the *FGPG* of the current exploitative stage is computed as $FGPG_{\text{expt}} = \Delta f_{\text{expt}} / K_2$. If we find that $FGPG_{\text{expr}} > FGPG_{\text{expt}}$, then more exploration is promoted by increasing the value of K_1 and decreasing K_2 by setting $K_1 = K_1 * (1 + r)$ and $K_2 = K_2 * (1 - r)$. But in case we find out that $FGPG_{\text{expt}} \geq FGPG_{\text{expr}}$, the opposite is done — K_1 is decreased and K_2 is increased for more exploitations by setting $K_1 = K_1 * (1 - r)$ and $K_2 = K_2 * (1 + r)$. However, the lengths of both the stages (i.e., K_1 and K_2) are always kept within some predefined limit [K_{min}, K_{max}] to avoid the complete domination by either mode (explorative or exploitative) of operations. In our implementation of this adaptive RTEP variant (ada-RTEP), we have used $r = 0.10$, $K_{min} = 2$ and $K_{max} = 40$. Ada-RTEP is compared with RTEP using the same population size (i.e., 50) and the same number of function evaluations (150,000) for the high dimensional functions f_1–f_{18}. Table 8.37 compares the performance of ada-RTEP with RTEP(2,4) and RTEP(4,8). Our observations on these results are summarized in the following few points.

- On the unimodal functions f_1-f_9, ada-RTEP outperforms both the RTEP variants on five (out of nine) functions, and showed similar performance on the remaining four.
- For the high dimensional multimodal functions f_{10}–f_{18}, ada-RTEP performs better than the RTEP variants on most (seven out of nine) functions. However, on the remaining two functions, ada-RTEP performs worse than RTEP(2,4), but performs equally well to RTEP(4,8).
- For most of the low dimensional multimodal functions f_{19}–f_{30}, ada-RTEP shows similar performance to RTEP(2,4) and RTEP(4,8). For the remaining few functions, the performance of ada-RTEP is usually better.
- Combining all the three points above, we can conclude that the overall performance of ada-RTEP is better than both of its RTEP counterparts, which indicates the effectiveness of our proposed technique of ada-RTEP.

Ada-RTEP automatically adapts the values of K_1 and K_2. Does this automatic adaptation procedure lead towards an optimal K_2/K_1 (i.e., exploitation-to-exploration) ratio? In the previous Tables 8.1–8.9, we have manually evaluated RTEP for several different values of K_1 and K_2 and found an optimal K_2/K_1 ratio for each of the 30 standard benchmark functions. Tables 8.3, 8.6 and 8.9 show these manually found optimal K_2/K_1 ratios for the unimodal, high dimensional multimodal and the low dimensional multimodal functions, respectively. But does the

Table 8.37: Performance comparison (based on mean error of the final results) between RTEP and Ada-RTEP on the standard benchmark functions f_1–f_{30}. Results have been averaged over 50 independent runs. A '+' or '–' in the *t*-test between Ada-RTEP vs. RTEP indicates that Ada-RTEP is significantly better or worse, respectively than RTEP with 95% certainty, while a '≈' means that the difference is not statistically significant.

Function	Mean Error ± Standard Deviation			*t*-Test (Ada-RTEP vs.)	
	Ada-RTEP	RTEP (2,4)	RTEP (4,8)	RTEP (2,4)	RTEP (4,8)
f_1	**3.0e–26 ± 6.4e–27**	7.5e–18 ± 4.4e–18	2.4e–20 ± 7.4e–21	+	+
f_2	**8.3e–13 ± 2.5e–13**	1.7e–09 ± 1.5e–09	2.9e–12 ± 6.8e–13	+	+
f_3	1.9e+00 ± 8.2e–01	1.7e+00 ± 6.2e–01	1.9e+00 ± 1.4e+00	≈	≈
f_4	**2.1e–18 ± 8.6e–19**	2.4e–15 ± 6.2e–16	2.1e–15 ± 4.1e–16	+	+
f_5	2.1e–03 ± 6.6–e04	2.6e–03 ± 7.7e–04	2.0e–03 ± 4.9–e04	≈	≈
f_6	1.4e–01 ± 7.5e–02	1.3e–01 ± 3.9e–02	1.2e–01 ± 3.5e–02	≈	≈
f_7	1.5e+00 ± 4.1e–01	1.1e+00 ± 9.0e–01	1.7e+00 ± 6.6e–01	≈	≈
f_8	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	≈	≈
f_9	**3.4e–47 ± 7.4e–48**	8.0e–38 ± 6.4e–39	2.5e–34 ± 1.4e–35	+	+
f_{10}	**9.1e–20 ± 8.2e–21**	2.5e–14 ± 5.0e–15	1.9e–14 ± 6.1e–15	+	+
f_{11}	**7.4e–08 ± 1.6e–08**	2.9e–07 ± 5.1e–08	1.1e–06 ± 9.1e–08	+	+
f_{12}	**2.4e+01 ± 8.8e+00**	7.1e+02 ± 4.9e+02	3.6e+02 ± 9.9e+01	+	+
f_{13}	2.9e–10 ± 7.5e–11	**2.0e–11 ± 6.5e–12**	2.4e–09 ± 9.2e–10	–	+
f_{14}	**4.4e–31 ± 9.3e–32**	2.7e–25 ± 6.3e–26	8.4e–20 ± 3.3e–20	+	+
f_{15}	**6.2e–13 ± 2.0e–13**	7.8e–10 ± 9.4e–11	2.9e–12 ± 4.1e–13	+	+
f_{16}	6.4e–08 ± 2.8e–08	2.2e–07 ± 9.1e–08	6.1e–08 ± 2.1e–08	+	≈
f_{17}	**1.7e–13 ± 5.7e–14**	3.2e–13 ± 8.5e–14	**1.7e–13 ± 2.7e–14**	+	≈
f_{18}	4.6e–07 ± 1.6e–07	**7.1e–08 ± 7.3e–09**	7.2e–05 ± 2.2e–05	–	+
f_{19}	0.002 ± 4.0e–06	0.002 ± 5.8e–04	0.002 ± 3.3e–05	≈	≈
f_{20}	0.0008 ± 2.9e–07	**0.0004 ± 6.2e–07**	0.0008 ± 3.6e–04	–	≈
f_{21}	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	≈	≈
f_{22}	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	≈	≈
f_{23}	0.00 ± 0.00	0.03 ± 0.02	0.00 ± 0.00	+	≈
f_{24}	0 ± 0	0 ± 0	0 ± 0	≈	≈
f_{25}	**0.38 ± 0.12**	0.55 ± 0.08	0.39 ± 0.10	+	≈
f_{26}	0.40 ± 0.16	0.38 ± 0.12	0.40 ± 0.12	≈	≈
f_{27}	0.21 ± 0.07	0.25 ± 0.10	0.22 ± 0.10	≈	≈
f_{28}	**0.28 ± 0.05**	0.85 ± 0.20	0.42 ± 0.11	+	+
f_{29}	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	≈	≈
f_{30}	0.28 ± 0.09	0.29 ± 0.09	0.26 ± 0.08	≈	≈
Summary (*t*-Test)	Ada-RTEP +	14	12		
	Ada-RTEP –	3	0		
	Ada-RTEP ≈	13	18		

Table 8.38: Comparison between RTEP and Ada-RTEP, based on their optimal K_2/K_1 ratio (i.e., the exploitation-to-exploration ratio) for the standard benchmark functions f_1–f_{30}. RTEP finds these optimal K_2/K_1 ratio values manually, by trial-and-error, in the previous Tables 8.2, 8.5 and 8.8, while Ada-RTEP reaches these optimal K_2/K_1 values through its automatic adaptation schemes for the stage lengths K_1 and K_2. The strong positive correspondence between the two series of K_2/K_1 ratio values is demonstrated by their high, positive correlation values 0.94, 0.75 and 0.92 for the three different function families.

Function	**Ada-RTEP**			**RTEP**	**Correlation between $(K_2/K_1)_{\text{RTEP}}$ and $(K_2/K_1)_{\text{Ada-RTEP}}$**
	K_1	K_2	$(K_2/K_1)_{\text{Ada-RTEP}}$	$(K_2/K_1)_{\text{RTEP}}$	
f_1	6	21	3.50	3.75	0.94
f_2	6	17	2.83	2.88	
f_3	5	14	2.80	2.50	
f_4	7	19	2.71	2.50	
f_5	5	14	2.80	2.50	
f_6	7	19	2.71	2.75	
f_7	6	15	2.50	2.38	
f_9	6	17	2.83	2.75	
f_{10}	8	13	1.63	1.88	0.75
f_{11}	5	9	1.80	1.75	
f_{12}	12	19	1.58	1.75	
f_{13}	9	16	1.78	1.63	
f_{14}	9	22	2.44	2.25	
f_{15}	10	22	2.20	1.88	
f_{16}	9	16	1.78	1.75	
f_{17}	8	15	1.88	2.00	
f_{18}	7	13	1.86	1.88	
f_{19}	8	15	1.88	2.00	0.92
f_{20}	10	18	1.80	2.00	
f_{25}	10	16	1.60	1.75	
f_{26}	9	16	1.80	2.00	
f_{27}	9	15	1.70	1.88	
f_{28}	8	15	1.88	2.13	
f_{30}	8	14	1.75	2.00	

automatically adapted K_2/K_1 ratio values from ada-RTEP agree with those manually found optimal K_2/K_1 ratio values? Table 8.38 makes a comparison between these two sets of K_2/K_1 ratio values and discovers that there exists strong positive correlation between these two sets

of K_2/K_1 ratios (i.e., correlation = 0.94, 0.75 and 0.92 for the three function families — unimodal, high dimensional multimodal and low dimensional multimodal functions, respectively). This indicates that ada-RTEP does really make an automatic and appropriate adaptation of the K_1 and K_2 values based on the relative effectiveness of its explorative and exploitative stages.

8.10 Conclusion

In this chapter we have carried out in-depth experiments on each of the five algorithms that we have developed so far along the entire course of our thesis (across the chapters 3 to 7) — RTEP, DGEP, ABC-SAM, ABC-IX and ABC-AX^2. These experiments provide us with a better understanding and insight on several aspects of the proposed algorithms, such as how they work, how they improve their results over the basic EP-based or ABC-based algorithms, how their performance gets affected by their control parameters, how their final solution quality and explorative capacity are affected by the proposed techniques and so on. Then we have compared these algorithms against each other to identify their specific strengths and weaknesses, based on their final solution quality, explorative search capacity and their strength against premature convergence. We have also made a number of suggestions on how to improve their performance by incorporating a few simple, yet effective techniques. Finally, we have developed an improved and adaptive variant of one of our proposed EP-based algorithms (i.e., RTEP) by adapting the lengths of its explorative and exploitative stages, in order to demonstrate and emphasize the fact that each of our algorithms might be extended and improved further by employing some simple techniques, such as adaptation, self-adaptation and hybridization.

Chapter 9

Conclusions and Future Work

9.1 Introduction

Along the course of this thesis, we have developed a number of improved evolutionary and swarm intelligence algorithms, evaluated them on several benchmark problems on continuous optimization, compared them with some other relevant state-of-the-art metaheuristic algorithms and carried out some experiments on each of them to analyze and examine their components, characteristics, the role and effect of their parameters and so on. In this concluding chapter, we try to summarize all the works done so far, highlight our achievements and then provide some suggestions and directions for further research based on our works.

9.2 Summary and Conclusion

The objective of this thesis was to study and development of novel evolutionary and swarm intelligence algorithms that try to balance between global explorations and local exploitations and to maintain sufficient population diversity to avoid premature convergence around the local optima. With this end in view, we have developed a number of novel evolutionary and swarm intelligence algorithms by employing several techniques, such as regular alternations between exploitative and explorative operations (e.g., RTEP in chapter 3), using the population diversity information more effectively (e.g., DGEP in chapter 4), using the techniques of adaptation and self-adaptation (e.g., ABC-SAM and ABC-AX^2 in chapters 5 and 7, respectively) and hybridization with other meta-heuristic algorithms (e.g., ABC-IX in chapter 6). All these works are briefly summarized in the following paragraphs.

Chapter 3 introduces RTEP (Recurring Two-Stage Evolutionary Programming) — an improved EP-based algorithm that is established on the idea that explorations and exploitations can be orthogonal and complementary (rather than opposing and conflicting) operations. Therefore, RTEP tries to intensify both the explorative and exploitative operations by frequently alternating between its two stages — the explorative and exploitative stages. Two control parameters — K_1 and K_2 provide an adequate control on the degree of explorations, exploitations and the frequency of alternations between these two modes of operations. In chapter 8 (experiments), we have tested several different values of K_1, K_2 and the K_2/K_1 ratio, and suggested an optimal exploitation-to-exploration ratio for each standard benchmark function, as well as prescribed an overall optimal K_2/K_1 ratio for the three different function families — the unimodal functions f_1–f_9, the high dimensional multimodal functions f_{10}–f_{18} and the low dimensional functions f_{19}–f_{30}. The experimental results show that RTEP often performs significantly better than many other recent evolutionary and swarm intelligence algorithms.

In Chapter 4, we have developed DGEP (Diversity Guided Evolutionary Programming) — an improved evolutionary algorithm that employs the existing amount of population diversity to adaptively control and guide the degrees of explorations and exploitations during each mutation operation. To accomplish this, DGEP introduces DGM (Diversity Guided Mutation) — a novel mutation scheme that controls the mutation step size using the current population diversity information. Besides, DGEP employs some basic, yet effective diversity preserving techniques to maintain a sufficient amount of population diversity in order to assist the diversity-dependent DGM mutation scheme. Empirical results and comparison with several other relevant EAs and SIAs (chapter 4) have empirically established the effectiveness of the DGEP and DGM mutation scheme. Chapter 8 (experiments) has demonstrated that DGEP is more explorative and better resilient against premature convergence in comparison to RTEP, because DGEP maintains relatively higher amount of population diversity throughout the entire optimization process. This is why DGEP is more effective than RTEP on complex optimization problems, such as the more challenging CEC2005 benchmark problems on continuous optimization.

The contribution of chapter 5 is to introduce ABC-SAM (ABC with Self-Adaptive Mutation), which is an improved variant of the basic ABC algorithm. ABC-SAM tries to control the degree of explorations and exploitations, separately for every candidate solution $\boldsymbol{x}_i$ of the population, by adaptively controlling the mutation step size on $\boldsymbol{x}_i$. ABC-SAM maintains a scaling factor value SF_i, separately for each candidate solution $\boldsymbol{x}_i$ to customize the degree of explorations and exploitations around $\boldsymbol{x}_i$. The value of SF_i is automatically adapted at regular interval, either towards more explorations or towards better exploitations, based on the current

explorative/exploitative requirements of $\boldsymbol{x}_i$. ABC-SAM has a number of control parameters for which suitable values are suggested, through some evaluations, in chapters 5 and 8 (experiments). Experimental results on ABC-SAM indicate that the self-adaptation of mutation step size can effectively ensure better search space explorations and improve the performance significantly over the basic ABC algorithm, as well as several other state-of-the-art evolutionary and swarm intelligence algorithms.

Chapter 6 introduces ABC-IX (ABC with Improved eXplorations) — a novel ABC-variant that tries to improve the explorative capacity of both the perturbation and selection operations of the standard ABC algorithm. ABC-IX proposes a simulated annealing based probabilistic selection scheme that can accept both better and worse candidate solutions, thus allowing both uphill and downhill movements in the fitness landscape. Besides, ABC-IX employs a self-adaptive perturbation strategy that can adapt and customize the perturbation rate, and thus the degree of explorations and exploitations, separately for every candidate solution of the population. Evaluations and experimental results show that both the improved selection and perturbation operations of ABC-IX can work together cooperatively and synergistically to significantly improve its results over the standard ABC algorithm, as well as several other recent EAs and SIAs, including a number of improved, state-of-the-art variants of the ABC algorithm.

In chapter 7, we have introduced ABC-AX^2 — another novel algorithm that improves the basic ABC algorithm by employing a few techniques of adaptation and self-adaptation. ABC-AX^2 extends each candidate solution with three more control parameters that control the degree of explorations and exploitations, perturbation scaling factors and the rate of perturbations, separately for every candidate solution of the population. The value of each control parameter is automatically adapted, separately for every candidate solution, following some adaptive and self-adaptive rules. ABC-AX^2 has been extensively tested on two different benchmark suites and results have been compared with several other recent, improved variants of the ABC algorithm. The experimental results show that the performance of ABC-AX^2 is often better than its counterparts, which indicates the effectiveness of the proposed adaptive and self-adaptive techniques for better optimization.

Chapter 8 carries out an in-depth experimental study on each of the algorithms developed so far — the RTEP, DGEP, ABC-SAM, ABC-IX and ABC-AX^2. The experiments are intended to study the effect of the control parameters on the algorithms, the role, contribution and synergy of their components, the validity of their design choices, how their design components affect the perturbation success rate, population diversity and the degree of explorations, comparison among the algorithms based on their final results, explorative search capacity, strength against premature convergence and so on. Chapter 8 concludes by leaving a few suggestions on how

each of our proposed algorithms might be improved and extended for better performance by incorporating some simple techniques, such as adaptation, self-adaptation and hybridization.

The development and analysis of the algorithms along the course of this entire thesis has provided us with a number of important insights on how the different characteristics of the function may require different behaviors from the algorithm. In the following paragraphs, we present a few points on how the properties of the function (e.g., separability, modality and complexity) may necessitate different settings and actions from the optimizing algorithm for better performance.

Separability of the function: The suitable perturbation rate of the algorithm depends primarily on the separability of the function, as shown by the Tables 8.23, 8.26 and Figs. 8.11, 8.13 in chapter 8. For a separable function, small perturbation rates are usually more effective, while large perturbation rates are generally more suitable for non-separable functions. If the user has some domain knowledge about the separability of the problem, then an initial input may be sought by the algorithm from the user about whether the function is separable or not. This may lead towards the development of an improved user-interactive algorithm. Also, the algorithm may itself try to guess the separability of the problem by observing how the small and large perturbation rates are affecting the success rates of perturbations. This may lead to an improved self-adaptive or hybrid machine learning algorithm.

Modality of the function: The required exploitation-to-exploration ratio (i.e., the K_2/K_1 ratio of RTEP) for the multimodal functions is significantly smaller than the unimodal functions, as revealed by the experiments on RTEP (Tables 8.1–8.10, Figs. 8.1–8.3). If the modality of the function is known beforehand by the user, then the algorithm can be effectively guided using this knowledge as input from the user. Even without this domain knowledge, a machine learning algorithm may be developed that performs some analysis on the distribution of fitness values along each of the search dimensions to guess the modality of the function and thus set the value of the exploitation-to-exploration ratio to some suitable range. This initial ratio value may be adapted gradually using some strategy, such as the strategy adopted by ada-RTEP, as described in section 8.9 (results in Tables 8.37–8.38).

Complexity of the function: For complex high dimensional multimodal functions, the preservation of the population diversity is often considered necessary for avoiding premature convergence. The experimental results in Tables 8.30–8.36 reveal that the EAs and SIAs that can maintain more diversity (e.g., DGEP and ABC-IX, as shown in Tables 8.30, 8.34) usually perform much better on the more complex CEC2005 suite functions. However, the same algorithms (e.g., DGEP and ABC-IX) perform worse on the simpler functions, such as the standard benchmark suite functions, as shown in Tables 8.29, 8.33. This indicates that the explorative capacity of the

algorithm should match the necessary degree of explorations and diversity requirements of the function being optimized. If the user has some domain knowledge about the complexity of the problem at hand, he can either assist the algorithm with this knowledge input or pick the suitable algorithm for the problem, such as picking DGEP and/or ABC-IX for more complex functions, while RTEP and ABC-AX^2 for simpler functions. This may lead to an improved interactive algorithm, based on human-computer interactions.

9.3 Future Research Directions

Each of the algorithms developed along the course of this thesis can be improved and extended further for better final solution quality, more search space explorations, improved convergence speed, more resilience against local optima and premature convergence and so on. Therefore, further research on these algorithms is necessary to analyze, understand, extend and improve their performance. In the following few points, we leave a few suggestions and directions for further research on each of these algorithms.

Further research on RTEP: The current version of RTEP uses fixed values for its control parameters K_1 and K_2, which are the lengths of the recurring explorative and exploitative stages, respectively. An RTEP-variant could be developed that tries to make the values of K_1 and K_2 adaptive along the optimization process. The variant ada-RTEP, introduced in chapter 8, accomplishes this (i.e., adapts the values of K_1 and K_2) based on the fitness improvements of the explorative and exploitative stages. However, some other criteria, e.g., the convergence speed, genetic diversity, distribution of the candidate solutions and local neighborhoods across the search space, may also be considered for the adaptation of K_1 and K_2. Secondly, RTEP presents a very generic framework where any selection, recombination and/or mutation operation can be employed. It would be interesting to experiment how some other existing genetic operators could be incorporated within RTEP and how effectively they could perform the recurring explorations and exploitations using the framework of RTEP. For example, some genetic operators are known for their particular strengths or weaknesses, e.g., the SBX crossover, polynomial mutation [219] and the BLX-α crossover [95], each of which may be tested and compared using RTEP. Thirdly, it would be interesting to hybridize RTEP with many other existing continuous optimization algorithms. Since RTEP demonstrates outstanding capability to locate the global optimum, it might be applied on a problem that is partially solved by some other algorithm, say A, up to the point where A becomes prematurely converged or trapped around some locally optimal point(s) with prolonged fitness stagnation. It would be interesting to examine whether (and, how) RTEP can break free from the premature convergence and fitness stagnation and get rid of the locally optimal points. Fourthly, RTEP has

been employed only on the continuous optimization problems. In the future we would be looking forward to extend RTEP for the problems on dynamic optimization, multi-objective optimization, as well as find some new, innovative real world applications of RTEP.

Further research on DGEP: DGEP uses a number of control parameters, such as K, u, l, $|N|$ and $|A|$, most of which uses fixed values that are kept constant during the entire optimization process. The value of K is set at random during each mutation, but always within some fixed, predefined interval of values (Fig. 4.1). An adaptive DGEP-variant could be developed that uses an automatic adaptation strategy for some (or, all) of these control parameters. Such an adaptive approach would be more suitable than any fixed, predefined strategy to dynamically deal with the continuously evolving explorative/exploitative requirements of the candidate solutions across the population. Secondly, the distance-based DGM mutation scheme of DGEP puts an equal amount of emphasis on both exploitations (using the distance between neighbors) and explorations (using distance between non-neighbors). However, explorations are usually more effective during the early stage of evolution, while exploitations should be the major and dominant operation during the final generations. Therefore, an improved, adaptive DGEP variant might be developed than uses intensive explorations during the initial generations, but gradually turns to more and more exploitations with the ongoing generations. Thirdly, the distance-based DGM mutation scheme might be further improved by considering some more information, such as the genetic diversity, density of individuals around the current individual, properties of the fitness landscape, current rate of convergence towards some local or global optima and so on. Fourthly, the experimental studies (chapter 8) show that DGEP possesses an excellent explorative capacity and robustness against premature convergence, but its exploitative characteristics are not as good as RTEP or DPGA [146]. A hybrid DGEP-variant might be developed that tries to hybridize some efficient local search method (e.g., the Hooke-Jeeves pattern search technique [65]) with DGEP to improve its exploitative and fine-tuning characteristics. Fifthly, we have employed DGEP only to solve the continuous optimization problems. An interesting research topic would be to test and evaluate DGEP on many other existing problems, especially the discrete and the real world ones.

Further research on ABC-SAM: Incorporation of a few more techniques within ABC-SAM might further improve its performance. For example, during adaptation of the scaling factor values, ABC-SAM tries to emphasize both explorations and exploitations equally, rather than adaptively. But the relative necessity of explorations and exploitations usually do not remain equal throughout the optimization process. Explorations are considered highly necessary during the initial stage, while exploitations become gradually more and more effective, especially during the late generations. Besides, the relative necessity of exploitations and explorations also depend on the function characteristics, especially around the current

candidate solution, the existing amount of genetic diversity, the current maturity of the on-going optimization process and so on. Based on these facts, ABC-SAM could re-allocate and skew its emphasis during the adaptation cycles, either towards more explorations or towards more exploitations. Secondly, ABC-SAM gradually adapts only the scaling factors, i.e., the SF_i values, for every candidate solution $\boldsymbol{x}_i$. However, the performance of ABC-SAM is also affected by some other control parameters, such as K, α and β. Currently ABC-SAM assigns fixed, predefined values to all these control parameters, as well as to its other control parameters — τ_1, τ_2 and SF_{min}. An important research direction would be to make these control parameters adaptive and/or self-adaptive along with the on-going optimization process. Thirdly, ABC-SAM currently uses simple exponential and negative-exponential distributions for producing explorative and exploitative scaling factors, respectively during the adaptation cycles. There exists some research opportunity to test, evaluate and compare many other existing probability distributions, such as the Lévy [56] and Cauchy [57] distributions, to examine how they can assist the self-adaptive mutation strategy of ABC-SAM. Fourthly, the self-adaptive mutation scheme of ABC-SAM can be incorporated and hybridized with many other recently introduced evolutionary and swarm intelligence based algorithms, such as [107]–[129]. Fifthly, so far ABC-SAM has been employed only on the continuous optimization problems. It would be interesting to find out how well ABC-SAM performs on many other existing problems, especially the discrete, dynamic, multi-objective and real-world problems.

Further research on ABC-IX: ABC-IX might be extended and improved in a number of ways. For example, ABC-IX currently employs an explorative selection scheme that is based on simulated annealing [233]. Some other explorative meta-heuristic technique, other than simulated annealing, such as the tabu search [28] or iterated local search [77] might be used to devise an explorative selection and/or perturbation technique. Secondly, ABC-IX currently employs the simple and straightforward exponential cooling schedule [233] for its system temperature T, which might be replaced by some more sophisticated cooling policy, such as some policy adaptively controlled by the population diversity or genetic distribution or some other metric that can estimate the current maturity of the ongoing optimization process. Thirdly, ABC-IX has some additional control parameters, such as T_0, α and t, for which we currently use fixed, predefined values. This might be improved by employing some adaptive strategy that dynamically considers the current explorative/exploitative needs of the population and sets the values of these control parameters accordingly. Fourthly, ABC-IX is currently biased towards more explorations, rather than exploitations. Putting some more efforts for the exploitations, especially around the best candidate solutions found so far and during the final cycles of ABC-IX may further improve its results. Some more suggestions and directions for further research with ABC-IX are — hybridizing ABC-IX with many other existing

EAs and SIAs, finding new problems and application domains for ABC-IX, especially the discrete, dynamic and multi-objective optimization problems, as well as many other existing real-world application problems.

Further Research on ABC-AX²: There exist several possible future research directions along which ABC-AX² might be further extended and improved. Firstly, ABC-AX² uses several control parameters, such as τ_1, τ_2, u_1, u_2, t_1, t_2, t_3, s_1 and s_2. Currently each of these parameters is simply set to a fixed, predefined value by ABC-AX². To improve the performance of ABC-AX², we could develop some adaptive and/or self-adaptive strategy that would dynamically adapt the values of these control parameters based on the current explorative/exploitative requirements. Secondly, ABC-AX² uses simple strategies (i.e., eqs. (7.3)–(7.6)) to adjust the values of the control parameters — p_i, q_i and $\boldsymbol{\eta_i}$ for each candidate solution $\boldsymbol{x}_i$. Some more sophisticated strategy, such as considering the properties of fitness landscape around the current candidate solution $\boldsymbol{x}_i$, taking into account the existing population diversity, convergence speed and/or the maturity of the optimization process may be more effective for balancing exploitations with explorations around $\boldsymbol{x}_i$. Thirdly, after the execution of ABC-AX² is over, employing some exploitative and efficient local searcher might improve the results further by precisely pinpointing the global optimum. Fourthly, the adaptive and self-adaptive strategies of ABC-AX² can be employed with many other existing mutation-based swarm intelligence algorithms, such as [113]–[120]. Some more suggestions on further research with ABC-AX² are — hybridizing ABC-AX² with other suitable evolutionary, swarm intelligence and/or machine learning techniques, employing ABC-AX² to solve many other existing problems, especially the discrete and real world ones, improving and extending ABC-AX² for dynamic, noisy and multi-objective optimization problems, finding new and novel application domains for ABC-AX² and so on.

Appendix A

Benchmark Functions

Along the course of this thesis, we have developed a number of improved evolutionary and swarm intelligence algorithms (chapters 3–7). Each of these algorithms is evaluated on two different suites of benchmark functions on continuous optimization — the standard benchmark suite consisting of 30 benchmark functions, and the recently introduced CEC2005 benchmark suite consisting of 25 benchmark functions. Both the benchmark suite functions have been widely used in many recent studies on evolutionary and swarm intelligence algorithms, e.g., [2], [15]–[18], [20], [21] and [26]–[28]). In the following two sections, we present an overview on the 30 standard benchmark suite functions (section A.1), followed by the 25 CEC2005 functions (section A.2). More details on these functions can be found in [2], [21], [28] and [76].

A.1 Standard Benchmark Functions

The standard benchmark suite contains 30 benchmark functions. Based on their properties, these functions can be roughly categorized into two groups — the unimodal functions f_1–f_9 and the multimodal ones f_{10}–f_{30}. The multimodal functions (i.e., f_{10}–f_{30}) can further be categorized into two groups — the high dimensional multimodal functions f_{10}–f_{18}, each one of which contains several local minima, and the relatively low dimensional multimodal functions f_{19}–f_{30}, each one of which has dimensionality $D \leq 10$ and possesses only a few local minima. The suite also contains both separable (e.g., f_1, f_3, f_{15}, f_{16}) and non-separable (e.g., f_2, f_4, f_{14}, f_{18}) functions. The modality and separability of a function often significantly control the difficulty that the function presents to any algorithm during optimization. For example, the highly multimodal functions are usually more challenging to optimize by any algorithm, because the algorithm must possess both exploitative and explorative characteristics so that it can explore the locally optimal points without being trapped around any of them. Some of the multimodal functions in f_{10}–f_{18} have tens or hundreds of local minima, even when the dimensionality is just two or three (e.g., Rastrigin function f_{10}, Schwefel function f_{12}, Ackley function f_{13}, Griewank functions f_{14}, as illustrated in the Figs. A.1.7–A.1.9). For these functions, their number of local optima increases exponentially with the number of dimensions. This often makes their optimization extremely difficult for any algorithm. For example, the Ackley function f_{13} possesses one narrow basin

containing the global minimum, but surrounded with exponentially many local minima (Fig. A.1.8). The Griewank function f_{14} has a component which creates linkage among the search variables, which complicates the goal of reaching the global minimum by perturbing any subset of the variables. The major difficulty of the Schwefel function f_{12} arises from its second best local minima which are very far from the single global minimum. The low dimensional functions f_{19}–f_{30} have only a few local minima, but their locally minimal points often have significant in-between distance over the parameter space (e.g., Michalewicz function f_{29}, Fig. A.1.13). This makes it difficult for any algorithm to break free from their deep local valleys, because it requires very large, more explorative perturbations on the existing candidate solutions to break free from the strong locally minimal points. In the following three subsections A.1.1–A.1.3, we present a brief introduction on the three groups of standard benchmark functions — the unimodal functions f_1–f_9 (in subsection A.1.1), the high dimensional multimodal functions f_{10}–f_{18} (subsection A.1.2) and the low dimensional multimodal functions f_{19}–f_{30} (subsection A.1.3).

A.1.1 Unimodal Functions f_1–f_9

f_1: Sphere Function It is also known as De Jong's function 1. The Sphere function is very simple and is widely used for various demonstrations. It is defined by eq. (A.1.1).

$$f_1(\boldsymbol{x}) = \sum_{i=1}^{D} x_i^2 \tag{A.1.1}$$

where D is the number of dimensions and $\boldsymbol{x} = [x_1, x_2, \ldots, x_D]^T$ is a D-dimensional vector. The search area is usually restricted to: $-100 \le x_i \le 100$.

Global minimum: $\min(f_1) = f_1(0, 0, \ldots, 0) = 0$.

Properties:

- Unimodal
- Continuous
- Convex
- Separable

f_2: Schwefel's Problem 2.22 It is a unimodal, non-separable function, defined by eq. (A.1.2).

$$f_2(\boldsymbol{x}) = \sum_{i=1}^{D} |x_i| + \prod_{i=1}^{D} |x_i| \tag{A.1.2}$$

The search area is usually restricted to $-10 \le x_i \le 10$, for $i = 1, 2, \ldots, D$.

Global minimum: $\min(f_2) = f_2(0, 0, \ldots, 0) = 0$.

Properties:

- Unimodal
- Non-Separable
- Easily optimized dimension by dimension

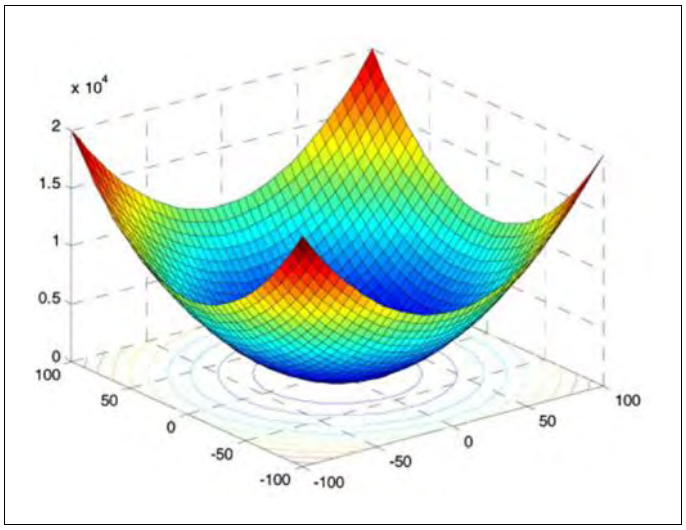


Figure A.1.1: 3-D shaded surface plot of 2-D sphere function (f_1)

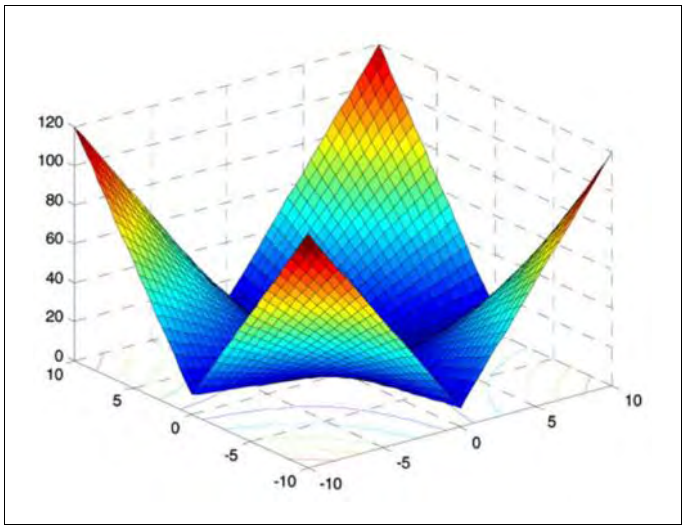


Figure A.1.2: 3-D shaded surface plot of 2-D Schwefel's problem 2.22 (f_2)

***f*$_3$: Schwefel's Problem 2.21** It is a unimodal and separable function, defined by eq. (A.1.3).

$$f_3(\boldsymbol{x}) = \max_i \{|x_i|, 1 \le i < D\} \tag{A.1.3}$$

Test area is usually restricted to $-10 \le x_i \le 10$, for $i = 1, 2, \ldots, D$.

Global minimum: min (f_3) = f_3(0, 0, …, 0) = 0.

Properties:

- Unimodal
- Separable

***f_4*: Schwefel's Problem 1.2** Schwefel's function 1.2 is an extension of the axis parallel hyper-ellipsoid, because this function produces rotated hyper-ellipsoids with respect to the coordinate axes. It is defined by eq. (A.1.4).

$$f_4(\boldsymbol{x}) = \sum_{i=1}^{D} \left(\sum_{j=1}^{i} x_j \right)^2 \qquad \text{(A.1.4)}$$

Test area is often restricted to the hypercube: $-65.536 \le x_i \le 65.536$, for $i = 1, 2, \ldots, D$.

Global minimum: min $(f_4) = f_4\,(0, 0, \ldots, 0) = 0$.

Properties:

- Unimodal
- Continuous
- Convex
- Non-separable

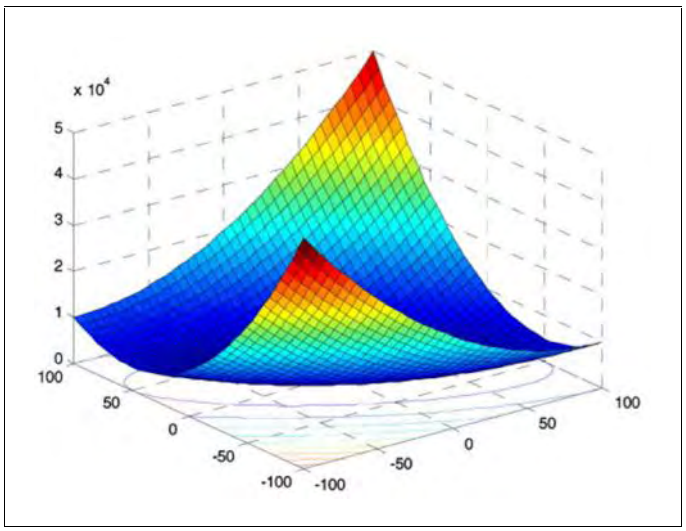


Figure A.1.3: 3-D shaded surface plot of 2-D Schwefel's problem 1.2 (f_4)

***f_5*: Powell's Function** It is a unimodal, non-separable function, defined by eq. (A.1.5).

$$f_5(\boldsymbol{x}) = \sum_{i=1}^{D/k} \left(x_{4i-3} + 10x_{4i-2} \right)^2 + 5\left(x_{4i-1} - x_{4i} \right)^2 + \left(x_{4i-2} - x_{4i-1} \right)^4 + 10\left(x_{4i-3} - x_{4i} \right)^4 \qquad \text{(A.1.5)}$$

The value of k is usually set to 4. The dimensionality D must be a multiple of k. In the suite, $D = 24$ has been used. Test area is often restricted to the search area: $-4 \le x_i \le 5$, for $i = 1, 2, \ldots, D$.

Global minimum: $\boldsymbol{x}^* = (3,-1,0,1, \ldots, 3,-1,0,1)$, $f_5(\boldsymbol{x}^*) = 0$.

Properties:

- Unimodal
- Non-separable

***f_6*: Dixon-Price Function** It is a unimodal, non-separable function, defined by eq. (A.1.6).

$$f_6(\boldsymbol{x}) = (x_1 - 1)^2 + \sum_{i=2}^{D} i\left(2x_i^2 - x_{i-1}\right)^2 \tag{A.1.6}$$

Test area is often restricted to the search area: $-10 \le x_i \le 10$, for $i = 1, 2, \ldots, D$.

Global minimum: min $(f_6) = f_6(0, 0, \ldots, 0) = 1$

Properties:

- Unimodal
- Non-separable

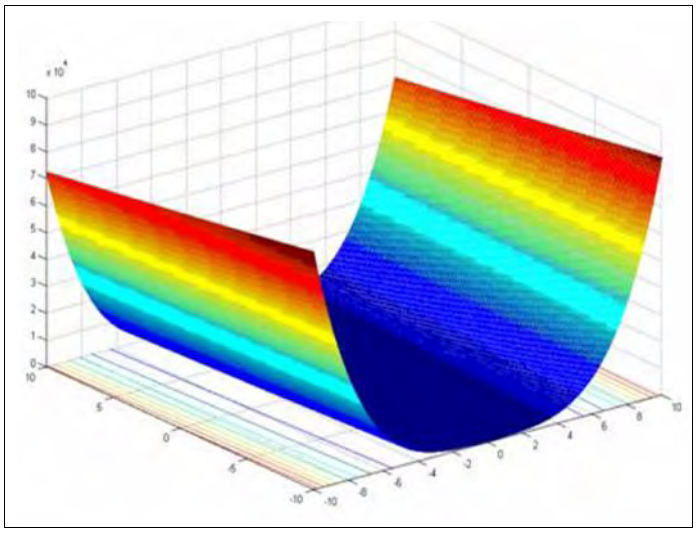


Figure A.1.4: 3-D shaded surface plot of 2-D Dixon-Price function (f_6)

***f_7*: Rosenbrock Function** Rosenbrock valley function is a classic optimization problem, which is also known as the Banana function. The global minimum resides inside a parabolic shaped valley (Fig. A.1.5). Finding the valley is not difficult, but precisely pinpointing the global optimum inside the almost flat, long, narrow valley is extremely difficult. Hence this problem has been widely used to assess the performance of many optimization algorithms. The general Rosenbrock function is defined by eq. (A.1.7).

$$f_7(\boldsymbol{x}) = \sum_{i=1}^{D-1}\left[100\left(x_{i+1} - x_i^2\right)^2 + \left(x_i - 1\right)^2\right] \tag{A.1.7}$$

Although the Rosenbrock function is often considered a unimodal function, as in [2], [28], there is some evidence [30] that it contains several minima in high dimensional instances. The search area is usually restricted to: $-30 \le x_i \le 30$, for $i = 1, 2, \ldots, D$.

Global minimum: min $(f_6) = f_6(1, 1, \ldots, 1) = 0$

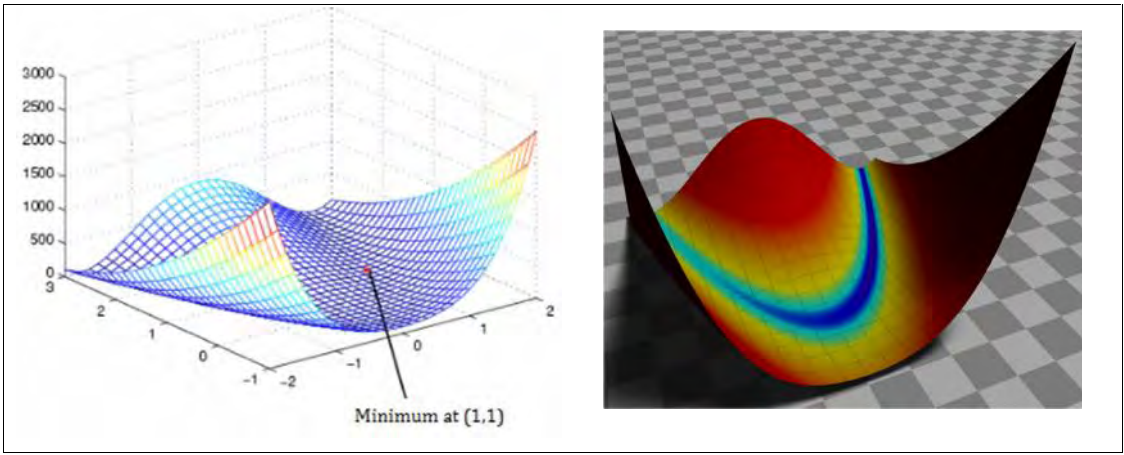


Figure A.1.5: 3-D shaded surface plot of 2-D Rosenbrock function, rendered with Matlab (on the left) and using a ray tracing program: POV-Ray (on the right)

Properties:

- Unimodal
- Separable
- High dimensional instances contain several minima

***f*₈: Step Function** It is a unimodal, separable function, defined by eq. (A.1.8).

$$f_8(\boldsymbol{x}) = \sum_{i=1}^{D} \left(\lfloor x_i + 0.5 \rfloor \right)^2 \qquad \text{(A.1.8)}$$

Test area is often restricted to the search area: $-100 \le x_i \le 100$, for $i = 1, 2, \ldots, D$.

Global minimum: $\min(f_8) = f_8(-0.5 \le x_i < 0.5) = 0$

Properties:

- Unimodal
- Separable

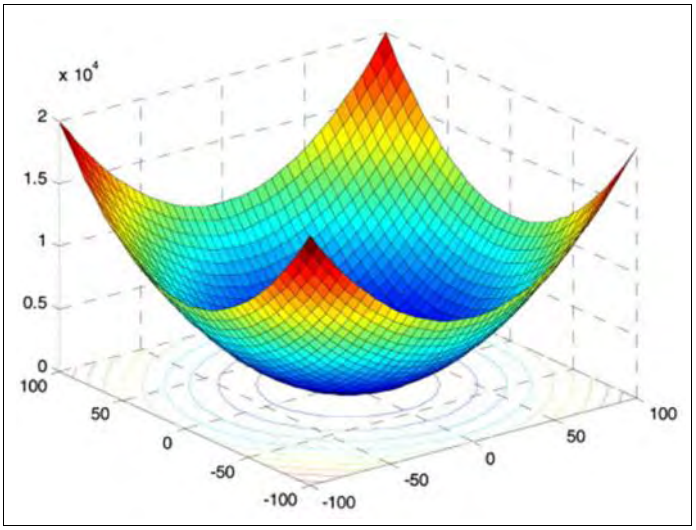


Figure A.1.6: 3-D shaded surface plot of 2-D Step function

***f*₉: Quartic Function with Noise** This function is unimodal, separable and noisy. The noise is simulated with the *random*[0,1) term in eq. (A.1.9).

$$f_9(\boldsymbol{x}) = \sum_{i=1}^{D} i x_i^4 + random[0,1) \tag{A.1.9}$$

Test area is often restricted to the search area: $-1.28 \le x_i \le 1.28$, for $i = 1, 2, \ldots, D$.

Global minimum: min $(f_9) = f_9(0, 0, \ldots, 0) = 0$

Properties:

- Unimodal
- Separable

A.1.2 High Dimensional Multimodal Functions f_{10}–f_{18}

***f*₁₀: Rastrigin Function** The Rastrigin function is based on Sphere function with the addition of cosine modulation to produce many local minima. For this reason, this function is highly multimodal and the location of the minima is regularly distributed. The function definition is given by eq. (A.1.10).

$$f_{10}(\boldsymbol{x}) = \sum_{i=1}^{D} \left[x_i^2 - 10\cos(2\pi x_i) + 10 \right] \tag{A.1.10}$$

Test area is usually restricted to the hypercube: $-5.12 \le x_i \le 5.12$, for $i = 1, 2, \ldots, D$.

Global minimum: min $(f_{10}) = f(0, 0, \ldots, 0) = 0$

Properties:

- Multimodal
- Separable
- Regularly distributed, exponentially many local minima

***f*₁₁: Non-continuous Rastrigin Function** It is a non-continuous, multimodal, separable function, with the definition given by eq. (A.1.11).

$$f_{11}(\boldsymbol{x}) = \sum_{i=1}^{D} \left[y_i^2 - 10\cos(2\pi y_i) + 10 \right] \tag{A.1.11}$$

$$\text{where } y_i = \begin{cases} x_i & |x_i| < 0.5 \\ 0.5 * round(2x_i) & |x_i| \ge 0.5 \end{cases}$$

Test area is usually restricted to the hypercube: $-5.12 \le x_i \le 5.12$, for $i = 1, 2, \ldots, D$.

Global minimum: min $(f_{11}) = f_{11}(0, 0, \ldots, 0) = 0$

Properties:

- Multimodal
- Separable
- Exponentially many local minima

***f*$_{12}$: Schwefel's Problem 2.26** Schwefel's function 2.26 is a deceptive function, because the global minimum is geometrically very distant, over the parameter space, from the second best local minima. Therefore, the search algorithms are potentially prone to convergence towards the wrong direction. The definition of the function is given by eq. (A.1.12).

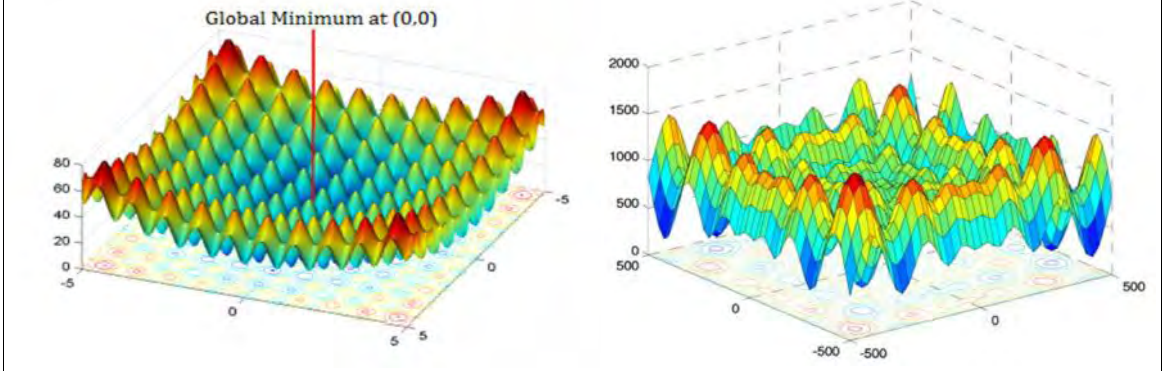
$$f_{12}(\boldsymbol{x}) = -\sum_{i=1}^{D} x_i \sin\left(\sqrt{|x_i|}\right) \qquad \text{(A.1.12)}$$

Test area is usually restricted to the hypercube: $-500 \le x_i \le 500$, for $i = 1, 2, \ldots, D$.

Global minimum: min (f_{12}) = f_{12}(420.9687, 420.9687, ..., 420.9687) = –12569.48

Properties:

- Multimodal
- Separable
- Exponentially many local minima



(a) Rastrigin function (b) Schwefel's problem 2.26

Figure A.1.7: 3-D shaded surface plot of 2-D Rastrigin function (f_{10}) and Schwefel's problem 2.26 (f_{12})

***f*$_{13}$: Ackley Function** It is a multimodal, non-separable function, with the definition of the function given by eq. (A.1.13).

$$f_{13}(\boldsymbol{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} x_i^{\,2}}\right) - \exp\left(\frac{1}{D}\sum_{i=1}^{D}\cos\left(2\pi x_i\right)\right) + 20 + e \qquad \text{(A.1.13)}$$

Test area is usually restricted to the hypercube: $-32 \le x_i \le 32$, for $i = 1, 2, \ldots, D$.

Global minimum: min (f_{13}) = f_{13}(0, 0, ..., 0) = 0

Properties:

- Multimodal
- Non-Separable
- Exponentially many local minima

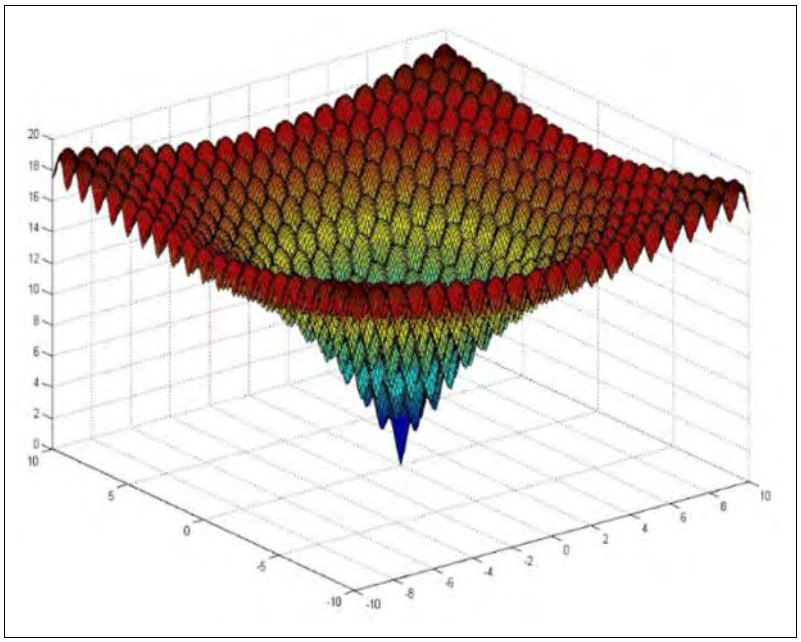


Figure A.1.8: 3-D shaded surface plot of 2-D Ackley function f_{13}

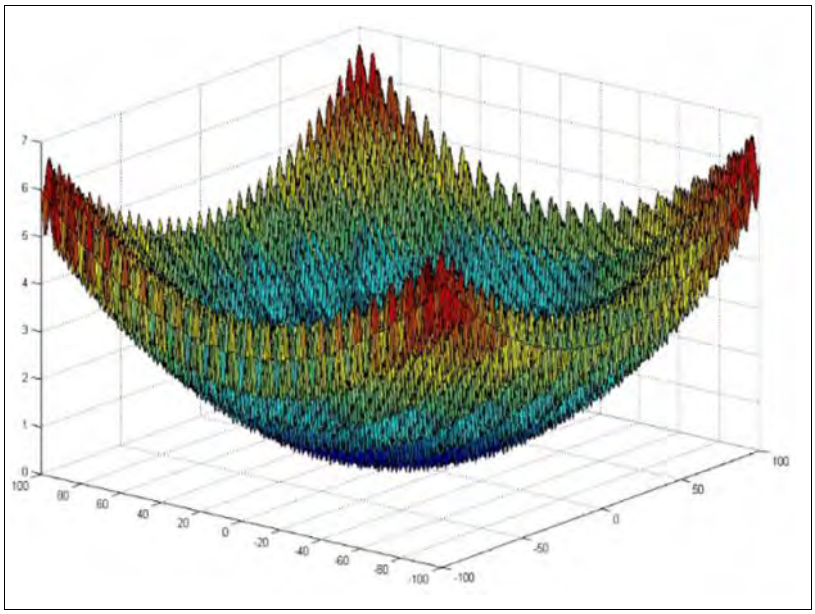


Figure A.1.9: 3-D shaded surface plot of 2-D Griewank function f_{14}

f_{14}: Griewank Function The Griewank function is a multimodal, non-separable continuous function similar to the Rastrigin function. It has exponentially many widespread local minima that are regularly distributed across the parameter space. The definition of the Griewank function is given by the following eq. (A.1.14).

$$f_{14}(\boldsymbol{x}) = \frac{1}{4000}\sum_{i=1}^{D} x_i^{\,2} - \prod_{i=1}^{D}\cos\left(\frac{x_i}{\sqrt{i}}\right) + 1 \qquad \text{(A.1.14)}$$

Test area is usually restricted to the hypercube: $-600 \le x_i \le 600$, for $i = 1, 2, \ldots, D$.

Global minimum: min $(f_{14}) = f_{14}(0, 0, \ldots, 0) = 0$

Properties:

- Multimodal
- Non-Separable
- Regularly distributed, exponentially many local minima

***f*$_{15}$: Alpine Function** The Alpine function is a multimodal, separable function, defined by the following eq. (A.1.15).

$$f_{15}(\boldsymbol{x}) = \sum_{i=1}^{D} \left| x_i \sin(x_i) + 0.1 x_i \right| \qquad \text{(A.1.15)}$$

The test area is usually restricted to: $-10 \le x_i \le 10$, for $i = 1, 2, \ldots, D$.

Global minimum: min $(f_{15}) = f_{15}(0, 0, \ldots, 0) = 0$

Properties:

- Multimodal
- Separable
- Exponentially many local minima

***f*$_{16}$: Weierstrass Function** The Weierstrass Function is a multimodal, separable function, defined by the following eq. (A.1.16).

$$f_{16}(\boldsymbol{x}) = \sum_{i=1}^{D}\left(\sum_{k=0}^{k\max}\left[a^k \cos\left(2\pi b^k (x_i + 0.5)\right)\right]\right) - D\sum_{k=0}^{k\max}\left[a^k \cos\left(2\pi b^k 0.5\right)\right] \qquad \text{(A.1.16)}$$

Here, the constants a, b and $kmax$ are set as: $a = 0.5$, $b = 3.0$ and $kmax = 20$. The test area is usually restricted to: $-0.5 \le x_i \le 0.5$, for $i = 1, 2, \ldots, D$.

Global minimum: min $(f_{16}) = f_{16}(0, 0, \ldots, 0) = 0$

Properties:

- Multimodal
- Separable
- Exponentially many local minima

***f*$_{17}$–*f*$_{18}$: Generalized Penalized Functions** Both these functions are multimodal, separable and have exponentially many locally optimal points. They are defined by eqs. (A.1.17) and (A.1.18).

$$f_{17}(\boldsymbol{x}) = \frac{\pi}{D}\left\{ 10\sin^2(\pi y_1) + \sum_{i=1}^{D}(y_i - 1)^2\,[1 + 10\sin^2(\pi y_{i+1})] + (y_n - 1)^2 \right\} + \sum_{i=1}^{D} u(x_i, 10, 100, 4) \quad \text{(A.1.17)}$$

$$f_{18}(x) = 0.1\left\{ \sin^2(\pi 3x_1) + \sum_{i=1}^{D}(x_i - 1)^2\,[1 + \sin^2(3\pi x_{i+1})] + (x_n - 1)^2[1 + \sin^2(2\pi x_D)] \right\} + \sum_{i=1}^{D} u(x_i, 5, 100, 4) \quad \text{(A.1.18)}$$

where: $$u(x_i, a, k, m) = \begin{cases} k(x_i - a)^m, & x_i > a \\ 0, & -a \le x_i \le a \\ k(-x_i - a)^m, & x_i < -a \end{cases}$$

$$y_i = 1 + \frac{1}{4}(x_i + 1)$$

The search area is usually restricted to: $-50 \le x_i \le 50$, for $i = 1, 2, \ldots, D$.

Global minimum: min (f_{17}) = min (f_{18}) = f (0, 0, …, 0) = 0

Properties:

- Multimodal
- Nonseparable

A.1.3 Low Dimensional Multimodal Functions f_{19}–f_{30}

***f_{19}*: Shekel's Foxholes Function** It is multimodal separable function, defined by the following eq. (A.1.19).

$$f_{19}(\boldsymbol{x}) = \left[\frac{1}{500} + \sum_{j=1}^{25}\left(\frac{1.0}{j + \sum_{i=1}^{2}(x_i - a_{ij})^6} \right) \right]^{-1} \quad \text{(A.1.19)}$$

The search area is usually restricted to: $-65.536 \le x_i \le 65.536$, for $i = 1, 2, \ldots, D$. For the usual value of $D = 2$, the constant (a_{ij}) is defined as:

$$(a_{ij}) = \begin{pmatrix} -32 & -16 & 0 & 16 & 32 & -32 & \ldots & \ldots & \ldots & 0 & 16 & 32 \\ -32 & -32 & -32 & -32 & -32 & -16 & \ldots & \ldots & \ldots & 32 & 32 & 32 \end{pmatrix}$$

Global minimum: min (f_{19}) = f_{19} (−32, −32, …, −32) ≈ 1

Properties:

- Multimodal
- Separable

***f*$_{20}$: Kowalik Function** It is multimodal, non-separable function, defined by the eq. (A.1.20).

$$f_{20}(\boldsymbol{x}) = \sum_{i=1}^{11}\left(a_i - \frac{x_1\left(b_i^2 + b_i x_2\right)}{b_i^2 + b_i x_3 + x_4}\right)^2 \qquad \text{(A.1.20)}$$

The search area is usually restricted to: $-5 \le x_i \le 5$, for $i = 1, 2, \ldots, D$. For the usual value of $D=4$, the constants a_i and b_i are defined in Table A.1.

Global minimum: min (f_{20}) ≈ f_{20}(0.1928, 0.1908, 0.1231, 0.1358) ≈ 0.0003075

Properties:

- Multimodal
- Non-separable

***f*$_{21}$: Six Hump Camel Back Function** The 2-D Six Hump Camel Back function has a total of six locally minimal points (within the bounded search region), two of which are global minima. The function is defined by the following eq. (A.1.21).

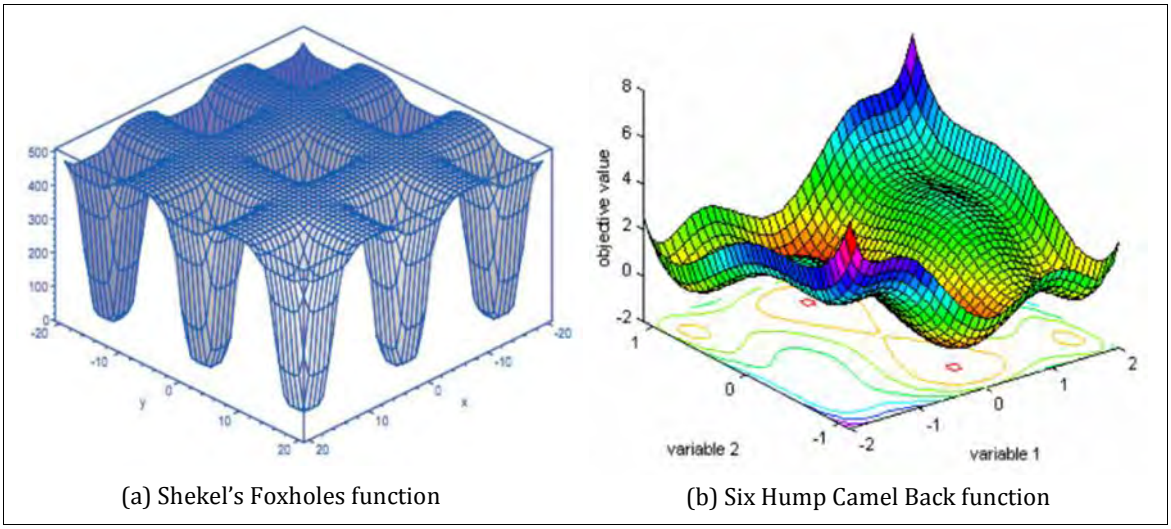
$$f_{21}(\boldsymbol{x}) = 4x_1^2 - 2.1x_1^4 + \frac{1}{3}x_1^6 + x_1x_2 - 4x_2^2 + 4x_2^4 \qquad \text{(A.1.21)}$$

The search area is usually restricted to: $-5 \le x_i \le 5$, for $i = 1, 2$.

Global minimum: Two different global minima are (0.0898, –0.7126) and (–0.0898, 0.7126).
min (f_{21}) = –1.0316285

Properties:

- Multimodal
- Non-separable
- Number of local minima: six local minima (within the bounded search space), two of which are global minima.



(a) Shekel's Foxholes function (b) Six Hump Camel Back function

Figure A.1.10: 3-D surface plot of Shekel's Foxholes function f_{19} and Six Hump Camel Back function f_{21}

Table A.1: Coefficients for Kowalik function f_{20}

i	a_i	b_i^{-1}
1	0.1957	0.25
2	0.1947	0.5
3	0.1735	1
4	0.1600	2
5	0.0844	4
6	0.0627	6
7	0.0456	8
8	0.0342	10
9	0.0323	12
10	0.0235	14
11	0.0246	16

***f*$_{22}$: Branin Function** The 2-D Branin function is a multimodal, separable test function that has three equal sized global minima, no local minimum and has the following definition.

$$f_{22}(\boldsymbol{x}) = \left(x_2 - \frac{5.1}{4\pi^2} x_1^2 + \frac{5}{\pi} x_1 - 6 \right)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos(x_1) + 10 \qquad \text{(A.1.22)}$$

The search area is usually restricted to: $-5 \le x_1 \le 10$, $0 \le x_2 \le 15$.

Global minimum: Three global minima: (–3.142, 2.275), (3.142, 2.275), (9.425, 2.425)

min (f_{22}) = 0.397887

Properties:

- Multimodal
- Separable
- Number of local minima: No local minimum except the global ones

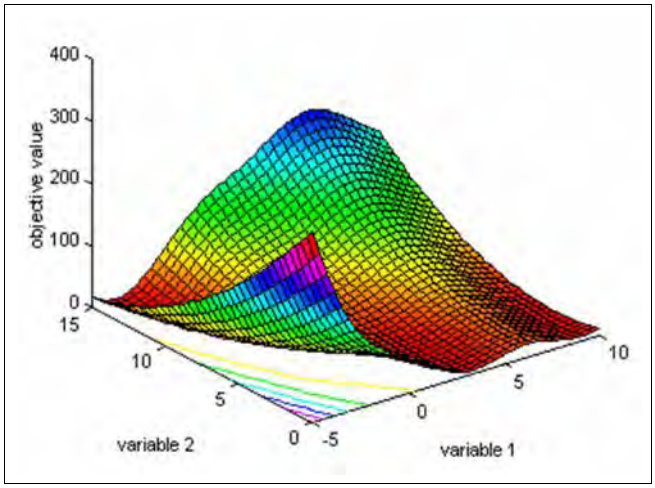


Figure A.1.11: 3-D shaded surface plot of the Branin function f_{22}

***f*$_{23}$: Hartman3 Function** This is a 3-D multimodal, non-separable test function that has the following definition (A.1.23).

$$f_{23}(\boldsymbol{x}) = -\sum_{i=1}^{4} c_i \, exp\left[-\sum_{j=1}^{3} a_{ij}\left(x_j - p_{ij}\right)^2\right] \qquad \text{(A.1.23)}$$

The search area is usually restricted to: $0 \le x_i \le 1$, for $i = 1, 2, 3$. The constants (a_{ij}), c_i and (p_{ij}) are defined in Table A.2.

Global minimum: $\boldsymbol{x}^* = (0.114614, 0.555649, 0.852547)$

$\min(f_{23}) = f_{23}(\boldsymbol{x}^*) = -3.86278$

Properties:

- Multimodal
- Non-separable
- Number of local minima: Four local minima

***f*$_{24}$: Hartman6 Function** This is a multimodal, non-separable test function with the dimensionality $D = 6$. It has the following definition (A.1.24).

$$f_{24}(\boldsymbol{x}) = -\sum_{i=1}^{4} c_i \, exp\left[-\sum_{j=1}^{6} a_{ij}\left(x_j - p_{ij}\right)^2\right] \qquad \text{(A.1.24)}$$

The search area is usually restricted to: $0 \le x_i \le 1$, for $i = 1, 2, \ldots, 6$. The constants (a_{ij}), c_i and (p_{ij}) are defined in Table A.3.

Global minimum: $\boldsymbol{x}^* = (0.20169, 0.150011, 0.476874, 0.275332, 0.311652, 0.6573)$

$\min(f_{24}) = f_{24}(\boldsymbol{x}^*) = -3.32237$

Properties:

- Multimodal
- Non-separable
- Number of local minima: Four local minima

Table A.2: Coefficients for the Hartman function f_{23}

i	a_{ij}, $j = 1, 2, 3$			c_i	p_{ij}, $j = 1, 2, 3$		
1	3	10	30	1	0.3689	0.1170	0.2673
2	0.1	10	35	1.2	0.4699	0.4387	0.7470
3	3	10	30	3	0.1091	0.8732	0.5547
4	0.1	10	35	3.2	0.038150	0.5743	0.8828

Table A.3: Coefficients for the Hartman function f_{24}

i	a_{ij}, $j = 1, \ldots, 6$						c_i	p_{ij}, $j = 1, \ldots, 6$					
1	10	3	17	3.5	1.7	8	1	0.1312	0.1696	0.5569	0.0124	0.8283	0.5886
2	0.05	10	17	0.1	8	14	1.2	0.2329	0.4135	0.8307	0.3736	0.1004	0.9991
3	3	3.5	1.7	10	17	8	3	0.2348	0.1415	0.3522	0.2883	0.3047	0.6650
4	17	8	0.05	10	0.1	14	3.2	0.4047	0.8828	0.8732	0.5743	0.1091	0.0381

***f*25–*f*27: Shekel's Family:** These three functions are non-separable multimodal functions, each one with dimensionality $D = 4$ and having five, seven and ten local minima, respectively. They are defined by the following general eq. with $m = 5, 7$ and 10, respectively.

$$f(\boldsymbol{x}) = -\sum_{i=1}^{m}\left[(x-a_i)(x-a_i)^T + c_i\right]^{-1};\ m = 5, 7, 10 \text{ for } f_{25},\ f_{26} \text{ and } f_{27}, \text{ respectively.}$$

$$f_{25}(\boldsymbol{x}) = -\sum_{i=1}^{5}\left[(x-a_i)(x-a_i)^T + c_i\right]^{-1} \quad \text{(A.1.25)}$$

$$f_{26}(\boldsymbol{x}) = -\sum_{i=1}^{7}\left[(x-a_i)(x-a_i)^T + c_i\right]^{-1} \quad \text{(A.1.26)}$$

$$f_{27}(\boldsymbol{x}) = -\sum_{i=1}^{10}\left[(x-a_i)(x-a_i)^T + c_i\right]^{-1} \quad \text{(A.1.27)}$$

The search area is usually restricted to: $0 \le x_i \le 10$, for $i = 1, 2, 3, 4$. The constants a_i and c_i are defined in Table A.4.

Global minimum: $\boldsymbol{x}^* = (4, 4, 4, 4)$

$\min(f_{25}) = f(\boldsymbol{x}^*) = -10.1532$

$\min(f_{26}) = f(\boldsymbol{x}^*) = -10.4029$

$\min(f_{27}) = f(\boldsymbol{x}^*) = -10.5364$

Properties:

- Multimodal
- Non-separable
- Number of local minima: Five, seven and ten local minima for f_{25}, f_{26} and f_{27}, respectively

Table A.4: Coefficients for the Shekel functions f_{25}, f_{26}, f_{27}

i	a_{ij}, $j = 1, \ldots, 4$				c_i
1	4	4	4	4	0.1
2	1	1	1	1	0.2
3	8	8	8	8	0.2
4	6	6	6	6	0.4
5	3	7	3	7	0.4
6	2	9	2	9	0.6
7	5	5	3	3	0.3
8	8	1	8	1	0.7
9	6	2	6	2	0.5
10	7	3.6	7	3.6	0.5

***f*$_{28}$: Fletcher-Powell Function** This is a multimodal, non-separable test function with the following definition.

$$f_{28}(\boldsymbol{x}) = -\sum_{i=1}^{D}\left(A_i - B_i\right)^2 \qquad \text{(A.1.28)}$$

$$A_i = \sum_{j=1}^{D}\left(a_{ij}\sin\sim_j + b_{ij}\cos\sim_j\right)$$

$$B_i = \sum_{j=1}^{D}\left(a_{ij}\sin x_j + b_{ij}\cos x_j\right)$$

The search area is usually restricted to: $-\pi \le x_i \le \pi$, for i = 1, 2, …, D. The values of the constants (a_{ij}), (b_{ij}) and α_j can be found at the Tables 5–7 in [11].

Global minimum: min (f_{28}) = f_{28}(0, 0, …, 0) = 0

Properties:

- Multimodal
- Non-separable

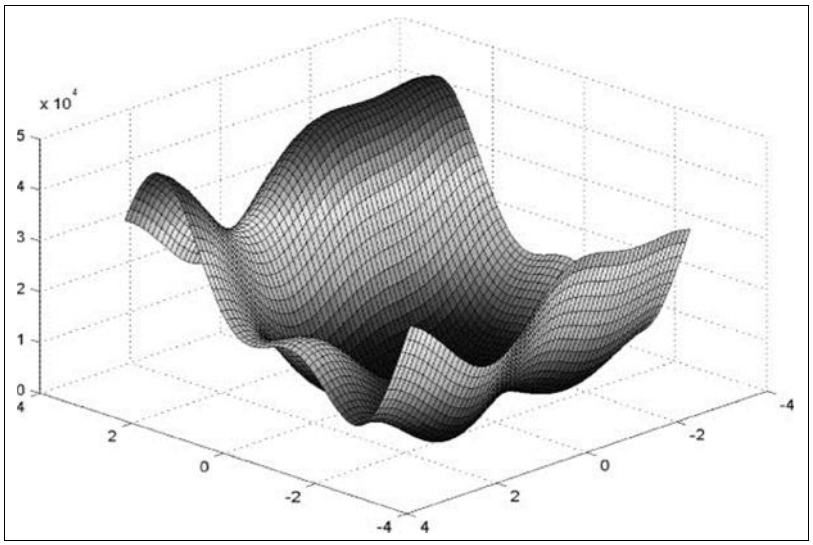


Figure A.1.12: 3-D surface plot of the 2-D Fletcher-Powell function f_{28}

***f*$_{29}$: Michalewicz Function** This is a highly multimodal test function with D! local optima. The parameter m defines the steepness of the valleys or edges, so larger values of m leads to more difficult search. For very large value of m, the function behaves like a needle in the haystack (i.e., the function values at the points outside the narrow peaks provide very little direction information towards the global optimum). The function is defined by the following eq. (A.1.29), and the search area is usually restricted to: $0 \le x_i \le \pi$, for i = 1, 2, …, D.

$$f_{29}(\boldsymbol{x}) = -\sum_{i=1}^{D}\sin\left(x_i\right)\left(\sin\left(ix_i^{\,2}/\text{®}\right)\right)^{2m} \qquad \text{(A.1.29)}$$

Global minimum: For $D = 10$, min $(f_{29}) = -9.66015171$

Properties:

- Multimodal
- Non-separable

f_{30}: Langerman Function This is a multimodal, non-separable test function with unevenly distributed local minima. It is defined by the following eq. (A.1.30).

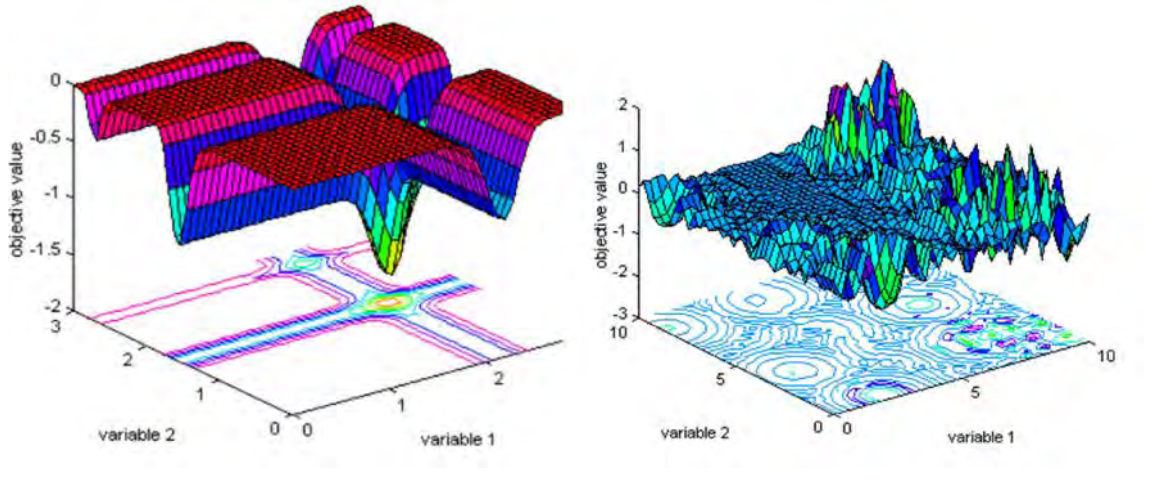
$$f_{30}(\boldsymbol{x}) = -\sum_{i=1}^{10} c_i \exp\left(-\frac{1}{\pi}\sum_{j=1}^{D}\left(x_j - a_{ij}\right)^2\right)\cos\left(\pi\sum_{j=1}^{D}\left(x_j - a_{ij}\right)^2\right) \qquad \text{(A.1.30)}$$

The search area is usually restricted to: $0 \le x_i \le 10$, for $i = 1, 2, \ldots, D$. The constants (a_{ij}) and (c_i) are defined in Table A.5.

Global minimum: For $D = 10$, min $(f_{30}) = -1.4$

Properties:

- Multimodal
- Non-separable



(a) Michalewicz function (b) Langerman function

Figure A.1.13: 3-D shaded surface plot of 2-D Michalewicz function f_{29} and Langerman function f_{30}

A.2 CEC2005 Benchmark Functions

This section briefly presents the CEC2005 benchmark functions which was proposed for the "Special Session on Continuous Optimization" held at the 2005 IEEE Congress on Evolutionary Computation (CEC-2005), held on 2-4 September 2005 at Edinburgh, UK. For this special session, a set of 25 continuous functions that are very difficult and challenging to optimize were proposed. These functions present many of the characteristics that can make a continuous function very hard to be solved, such as multimodality (presence of many local optima that mislead the search process), non-separability (dependencies among the variables that make it impossible

Table A.5: Coefficients (a_{ij}) and c_i for the Langerman function f_{30}

i	a_{ij}, j = 1, …, 10										c_i
1	9.681	0.667	4.783	9.095	3.517	9.325	6.544	0.211	5.122	2.020	0.806
2	9.400	2.041	3.788	7.931	2.882	2.672	3.568	1.284	7.033	7.374	0.517
3	8.025	9.152	5.114	7.621	4.564	4.711	2.996	6.126	0.734	4.982	1.500
4	2.196	0.415	5.649	6.979	9.510	9.166	6.304	6.054	9.377	1.426	0.908
5	8.074	8.777	3.467	1.863	6.708	6.349	4.534	0.276	7.633	1.567	0.965
6	7.650	5.658	0.720	2.764	3.278	5.283	7.474	6.274	1.409	8.208	0.669
7	1.256	3.605	8.623	6.905	0.584	8.133	6.071	6.888	4.187	5.448	0.524
8	8.314	2.261	4.224	1.781	4.124	0.932	8.129	8.658	1.208	5.762	0.902
9	0.226	8.858	1.420	0.945	1.622	4.698	6.228	9.096	0.972	7.637	0.531
10	7.305	2.228	1.242	5.928	9.133	1.826	4.060	5.204	8.713	8.247	0.876
11	0.652	7.027	0.508	4.876	8.807	4.632	5.808	6.937	3.291	7.016	0.462
12	2.699	3.516	5.874	4.119	4.461	7.496	8.817	0.690	6.593	9.789	0.491
13	8.327	3.897	2.017	9.570	9.825	1.150	1.395	3.885	6.354	0.109	0.463
14	2.132	7.006	7.136	2.641	1.882	5.943	7.273	7.691	2.880	0.564	0.714
15	4.707	5.579	4.080	0.581	9.698	8.542	8.077	8.515	9.231	4.670	0.352
16	8.304	7.559	8.567	0.322	7.128	8.392	1.472	8.524	2.277	7.826	0.869
17	8.632	4.409	4.832	5.768	7.050	6.715	1.711	4.323	4.405	4.591	0.813
18	4.887	9.112	0.170	8.967	9.693	9.867	7.508	7.770	8.382	6.740	0.811
19	2.440	6.686	4.299	1.007	7.008	1.427	9.398	8.480	9.950	1.675	0.828
20	6.306	8.583	6.084	1.138	4.350	3.134	7.853	6.061	7.457	2.258	0.964
21	0.652	2.343	1.370	0.821	1.310	1.063	0.689	8.819	8.833	9.070	0.789
22	5.558	1.272	5.756	9.857	2.279	2.764	1.284	1.677	1.244	1.234	0.360
23	3.352	7.549	9.817	9.437	8.687	4.167	2.570	6.540	0.228	0.027	0.369
24	8.798	0.880	2.370	0.168	1.701	3.680	1.231	2.390	2.499	0.064	0.992
25	1.460	8.057	1.336	7.217	7.914	3.615	9.981	9.198	5.292	1.224	0.332
26	0.432	8.645	8.774	0.249	8.081	7.461	4.416	0.652	4.002	4.644	0.817
27	0.679	2.800	5.523	3.049	2.968	7.225	6.730	4.199	9.614	9.229	0.632
28	4.263	1.074	7.286	5.599	8.291	5.200	9.214	8.272	4.398	4.506	0.883
29	9.496	4.830	3.150	8.270	5.079	1.231	5.731	9.494	1.883	9.732	0.608
30	4.138	2.562	2.532	9.661	5.611	5.500	6.886	2.341	9.699	6.500	0.326

for the algorithm to independently optimize each variable), shifting of the global optimum (to prevent algorithms to take advantage of a global optimum centered at zero), etc. Before formulating and describing each of the proposed functions, we first introduce the notation used to describe the functions — D represents the dimensionality of the problem, i.e., the number of variables to optimize, z is a candidate solution to the problem, o is the optimal solution (i.e., global optimum) to the problem and M are linear transformation matrices with an associated condition number (a measure that tells how numerically well-conditioned a problem is, i.e., how small variations of the input data can affect the output value). The functions in the CEC205 benchmark suite are classified into the following four groups.

i. Unimodal functions (F1–F5)

ii. Basic multimodal functions (F6–F12)

iii. Expanded functions (F13–F14)

iv. Hybrid composition functions (F15–F25)

The first two groups are simple unimodal and multimodal functions, respectively. The third group is composed of functions constructed in the following way — given a 2-D function $F(x; y)$ as a starting function, the corresponding expanded function $EF(x_1, x_2, \ldots, x_D)$ is defined as follows.

$$EF\left(x_1, x_2, \ldots, x_{D-1}, x_D\right) = F\left(x_1, x_2\right) + F\left(x_2, x_3\right) + \ \ldots \ + F\left(x_{D-1}, x_D\right) + F\left(x_D, x_1\right)$$

Finally, the fourth group of functions is composed of hybrid functions constructed from multiple functions from the second group. Hybrid functions apply each of the simple functions to the candidate solution and then carry out a weighted average of the result values. Additionally, some stretching and compressing values are applied, as well as shifting of the global optima.

A.2.1 Unimodal Functions (F1–F5)

F1: Shifted Sphere Function This is a unimodal, shifted, separable and scalable function, defined by eq. (A.2.1).

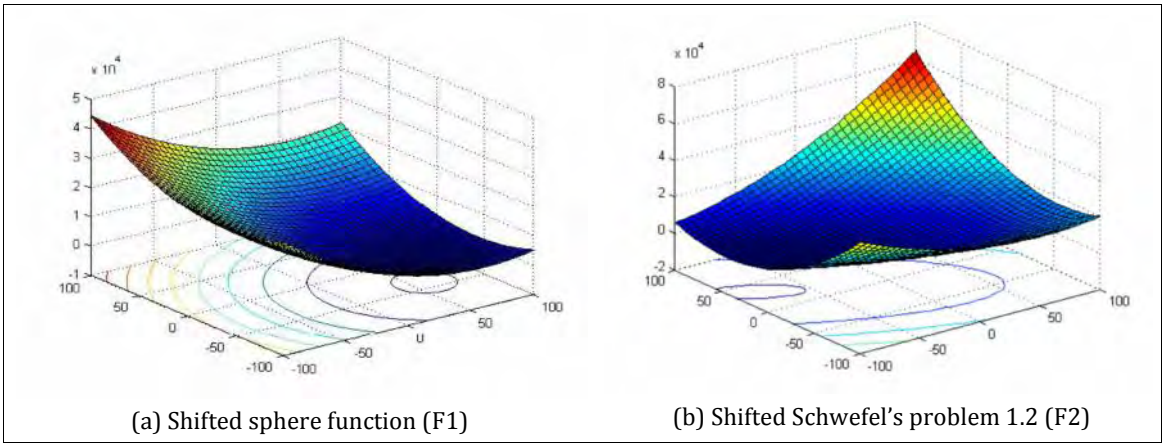
$$F_1(x) = \sum_{i=1}^{D} z_i^{\,2} \tag{A.2.1}$$

where: $z = x - o, \quad x = [x_1, x_2, \ldots, x_D] \qquad x \in [-100, 100]^D$

F2: Shifted Schwefel's Problem 1.2 This is a unimodal, shifted, non-separable and scalable function, defined by eq. (A.2.2).

$$F_2(x) = \sum_{i=1}^{D} \left(\sum_{j=1}^{i} z_j^{\,2} \right) \tag{A.2.2}$$

where: $z = x - o, \quad x = [x_1, x_2, \ldots, x_D] \qquad x \in [-100, 100]^D$



(a) Shifted sphere function (F1)

(b) Shifted Schwefel's problem 1.2 (F2)

Figure A.2.1: 3-D shaded surface plots of the 2-D shifted sphere function (F1) and shifted Schwefel's problem 1.2 (F2)

F3: Shifted Rotated High Conditioned Elliptic Function This is a unimodal, shifted, rotated, scalable and non-separable function, defined by eq. (A.2.3).

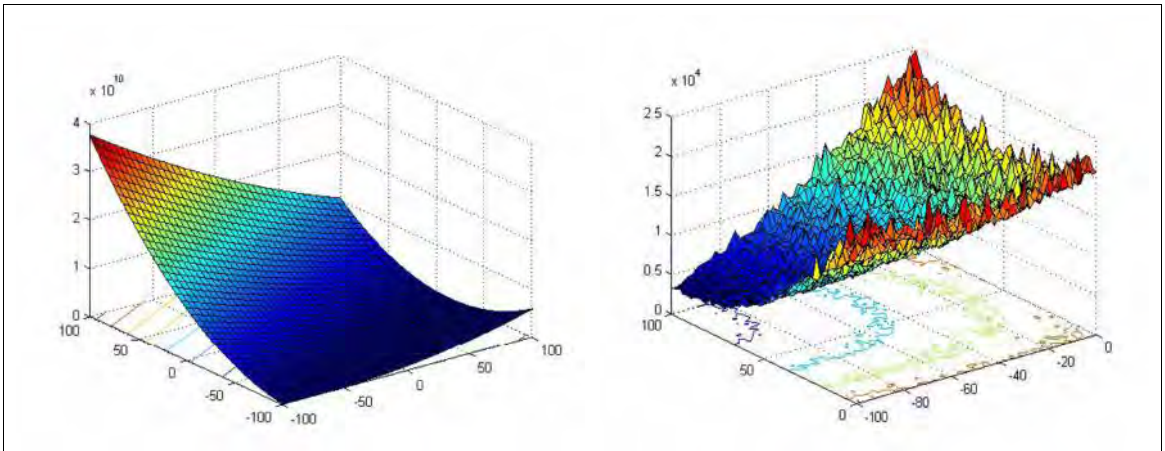
$$F_3(x) = \sum_{i=1}^{D} \left(10^6\right)^{\frac{i-1}{D-1}} z_i^{\,2} \qquad \text{(A.2.3)}$$

where: $z = (x - o) * M, \quad x = [x_1, x_2, \ldots, x_D] \qquad x \in [-100, 100]^D$

F4: Shifted Schwefel's Problem 1.2 with Noise in Fitness This is a unimodal, shifted, non-separable and scalable function with noise in fitness, defined by eq. (A.2.4).

$$F_4(x) = \sum_{i=1}^{D} \left(\sum_{j=1}^{i} z_j^{\,2} \right) * (1 + 0.4 \,|\, N(0,1) \,|) \qquad \text{(A.2.4)}$$

where: $z = x - o, \quad x = [x_1, x_2, \ldots, x_D] \qquad x \in [-100, 100]^D$



(a) High conditioned elliptic function (F3)

(b) Shifted Schwefel's problem 1.2 with noise (F4)

Figure A.2.2: 3-D shaded surface plots of High conditioned elliptic function (F3) and Shifted Schwefel's problem 1.2 with noise (F4)

F5: Schwefel's Problem 2.6 with Global Optimum on Bounds This is a unimodal, non-separable and scalable function with the global optimum located at the bounds of the search space, defined by eq. (A.2.5).

$$F_5(x) = \max \left| A_i x - B_i \right| \qquad \text{(A.2.5)}$$

Here, A is a $D*D$ matrix of random integer numbers, $a_{ij} \in [-500, 500] \ \forall_{i,j} \in [1, D]$ and $|A| \neq 0$. Also, A_i is the i-th row of A and $B_i = A_i * o$, where o is a $D*1$ vector of random integer numbers within the interval $[-100, 100]$. Finally, $x \in [-100, 100]^D$

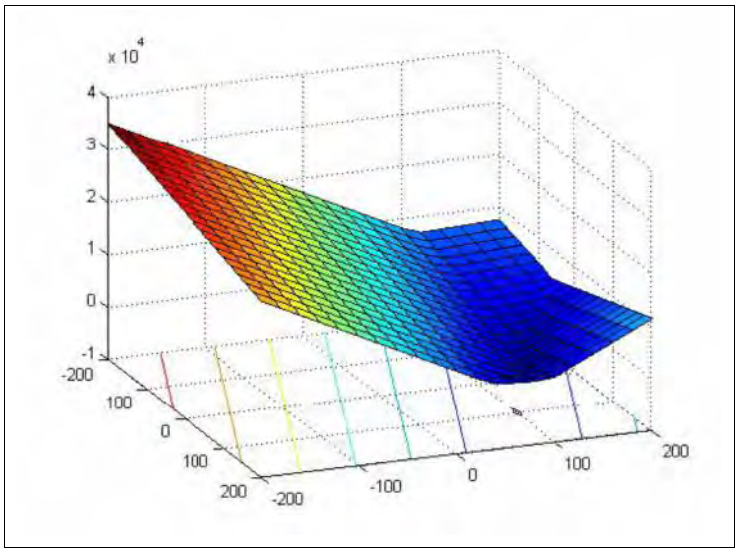


Figure A.2.3: 3-D shaded surface plot of Schwefel's problem 2.6 with global optimum on bounds (F5)

A.2.2 Basic Multimodal Functions (F6–F12)

F6: Shifted Rosenbrock Function This is a multimodal, shifted, non-separable and scalable function with a very narrow valley from the local to the global optimum. It is defined by eq. (A.2.6). Fig. A.2.4 shows a 3-D shaded surface plot of this function.

$$F_6(x) = \sum_{i=1}^{D-1} (100\left(z_i^{\,2} - z_{i+1}\right)^2 + (z_i - 1)^2)) + 390 \qquad \text{(A.2.6)}$$

where: $z = x - o + 1, \ \ x = [x_1, x_2, \ \ldots, \ x_D] \quad x \in [-100, 100]^D$

F7: Shifted Rotated Griewank Function without Bounds This is a multimodal, non-separable, shifted, rotated and scalable function, with no bounds for the variable x. The population is initialized in the interval $[0, 600]^D$, but the global optimum is outside of this initialization range. This function is defined by eq. (A.2.7). Fig. A.2.4 shows a 3-D shaded surface plot of this function.

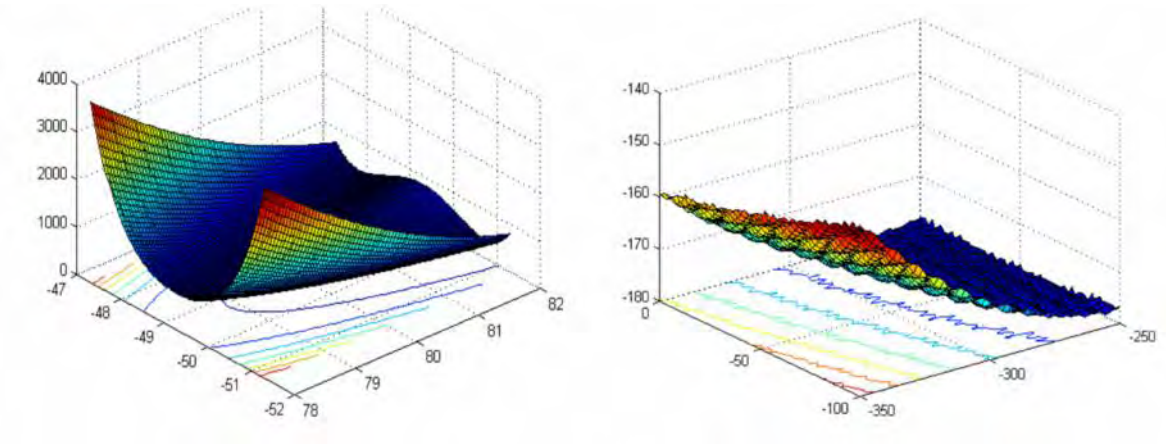
$$F_7(x) = \sum_{i=1}^{D} \frac{z_i^{\,2}}{400} - \prod_{i=1}^{D} \cos\left(\frac{z_i}{\sqrt{i}}\right) + 1 - 180 \qquad \text{(A.2.7)}$$

where: $z = (x - o) * M, \ \ x = [x_1, x_2, \ \ldots, \ x_D] \ \ x \in [0, 600]^D$

F8: Shifted Rotated Ackley's Function with Global Optimum on Bounds This is a multimodal, rotated, shifted, non-separable, scalable function. The global optimum is located at one bound of the search space. It is defined by eq. (A.2.8). Fig. A.2.5 shows its 3-D surface plot.

$$F_8(x) = -20\exp\left(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} z_i^{\,2}}\right) - \exp\left(\frac{1}{D}\sum_{i=1}^{D}(2\pi z_i)\right) + 20 + e - 140 \qquad \text{(A.2.8)}$$

where: $z = (x - o) * M, \;\; x = [x_1, x_2, \ldots, x_D] \quad x \in [-32, \; 32]^D$



(a) Shifted Rosenbrock function (F6) (b) Shifted Rotated Griewank function (F7)

Figure A.2.4: 3-D shaded surface plot of the Shifted Rosenbrock function (F6) and Shifted Rotated Griewank function (F7)

F9: Shifted Rastrigin's Function This is a multimodal, shifted, separable and scalable function, with a huge number of local optima. It is defined by eq. (A.2.9) and Fig. A.2.5 shows its 3-D plot.

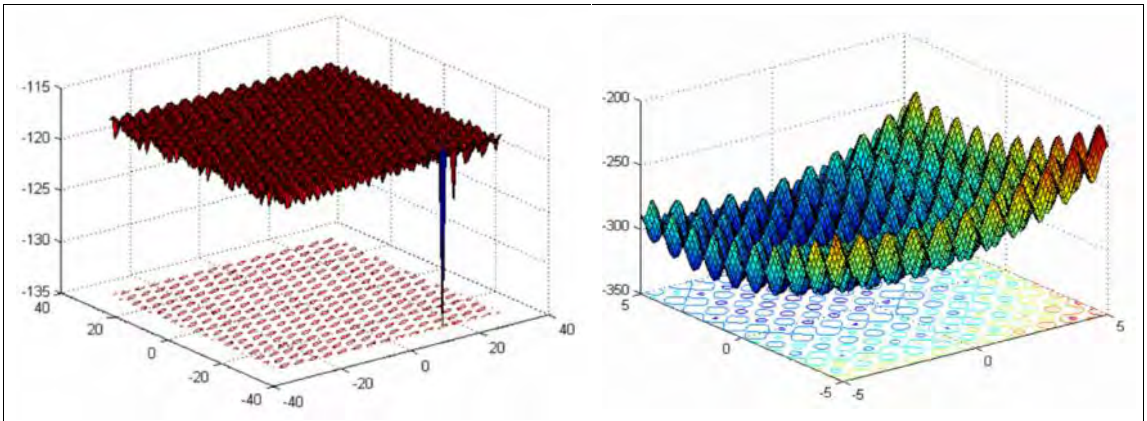
$$F_9(x) = \sum_{i=1}^{D}(z_i^{\,2} - 10\cos(2\pi z_i) + 10) - 330 \qquad \text{(A.2.9)}$$

where: $z = (x - o), \;\; x = [x_1, x_2, \ldots, x_D] \quad x \in [-5, \; 5]^D$

F10: Shifted Rotated Rastrigin Function This is a multimodal, shifted, rotated, non-separable and scalable function, with a huge number of local optima. It is the same function as the previous one, with the only difference that, in this case, $z = (x - o) * M$.

$$F_9(x) = \sum_{i=1}^{D}(z_i^{\,2} - 10\cos(2\pi z_i) + 10) - 330 \qquad \text{(A.2.10)}$$

where: $z = (x - o) * M, \;\; x = [x_1, x_2, \ldots, x_D] \quad x \in [-5, \; 5]^D$



(a) Shifted Rotated Ackley function (F8) (b) Shifted Rastrigin function (F9)

Figure A.2.5: 3-D shaded surface plots of shifted Rotated Ackley function with global optimum on bounds (F8) and Shifted Rastrigin function (F9)

F11: Shifted Rotated Weierstrass Function This is a multimodal, shifted, rotated, non-separable and scalable function. This function has the particularity of being continuous in all the domains but only differentiable in a particular set of points. It is defined by eq. (A.2.11), whereas a 3-D shaded surface plot of this function is provided in Fig. A.2.6.

$$F_{11}(x) = \sum_{i=1}^{D}\left(\sum_{k=0}^{km\max}[a^k \cos(2\pi b^k(z_i + 0.5))]\right) - D\sum_{k=0}^{k\max}[a^k \cos(2\pi b^k * 0.5)] + 90 \quad \text{(A.2.11)}$$

where: $z = (x - o) * M, \quad x = [x_1, x_2, \ldots, x_D] \quad x \in [-0.5, 0.5]^D$

F12: Schwefel's Problem 2.13 This is a multimodal, shifted, non-separable and scalable function. It is defined by eq. (A.2.12).

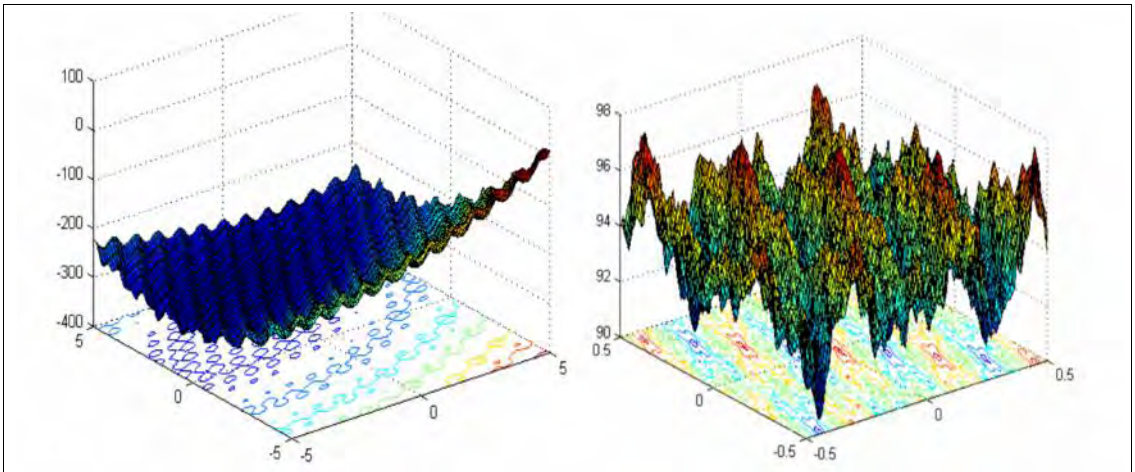
$$F_{12}(x) = \sum_{i=1}^{D}(A_i - B_i(x))^2 - 460 \quad \text{(A.2.12)}$$

Where A_i and B_i are defined as two $D*D$ matrices, and a and b are two random numbers within the interval $[-100,100]$.

$$A_i = \sum_{j=1}^{D}(a_{ij}\sin\alpha_j - b_{ij}\cos\alpha_j)$$

$$B_i(x) = \sum_{j=1}^{D}(a_{ij}\sin x_j - b_{ij}\cos x_j), \ \forall_i \in [1, D]$$

$$\alpha = [\alpha_1, \alpha_2, \ldots, \alpha_D], \ \alpha_1 \in [-\pi, \pi]$$



(a) Shifted Rotated Rastrigin function (F10) (b) Shifted Rotated Weierstrass function (F11)

Figure A.2.6: 3-D shaded surface plots of the Shifted Rotated Rastrigin function (F10) and Shifted Rotated Weierstrass function (F11)

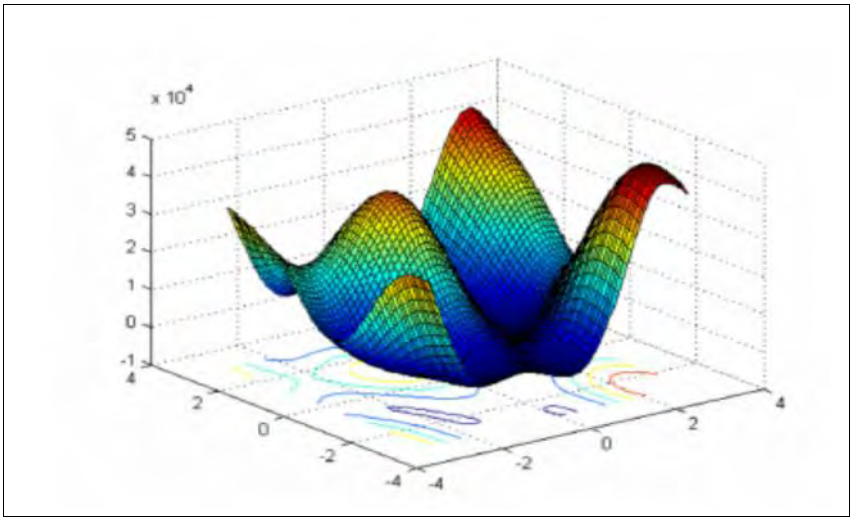


Figure A.2.7: 3-D shaded surface plot of Schwefel's problem 2.13 (F12)

A.1.3 Expanded Functions (F13–F14)

F13: Shifted expanded Griewank's plus Rosenbrock's Function This function is an expanded composition of Griewank function (eq. (A.2.7)) and Rosenbrock function (eq. (A.2.6)). The shifted function is defined by the following eq. (A.2.13).

$$F_{13}(x) = F8(F2(z_1, z_2)) + F8(F2(z_2, z_3)) + \ldots + F8(F2(z_{D-1}, z_D)) + F8(F2(z_D, z_1)) \qquad \text{(A.2.13)}$$

where: $z = x - o + 1, \quad x = [x_1, x_2, \ldots, x_D] \quad x \in [-3, 1]^D$

F14: Shifted Rotated Scaffer's F6 Function This is an expanded version of Scaffer's F6 function, which is defined by the following eq. (A.2.14).

$$F_{(x,y)} = 0.5 + \frac{(\sin^2(\sqrt{x^2 + y^2}) - 0.5)}{(1 + 0.001(x^2 + y^2))} \qquad \text{(A.2.14a)}$$

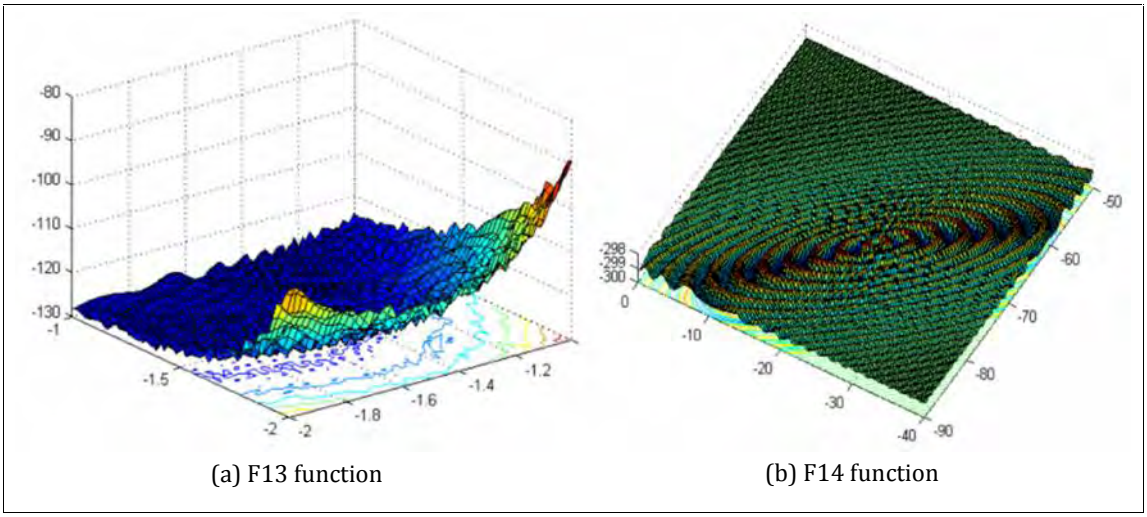
It is then expanded as the following eq. (A.2.14b).

$$F(x) = EF(z_1, z_2, \ldots, z_{D-1}, z_D) = F(z_1, z_2) + F(z_2, z_3) + \ldots + F(z_{D-1}, z_D) + F(z_D, z_1) \qquad \text{(A.2.14b)}$$

where: $z = (x - o) * M, \quad x = [x_1, x_2, \ldots, x_D] \quad x \in [-100, 100]^D$

A.1.3 Hybrid Composition Functions (F15–F25)

These functions are compositions of multiple individual. For their sheer complexity, their equations are not presented here. A detailed description on each of these hybrid composition functions, as well as the necessary pseudocode to compute their values, is provided in [76].



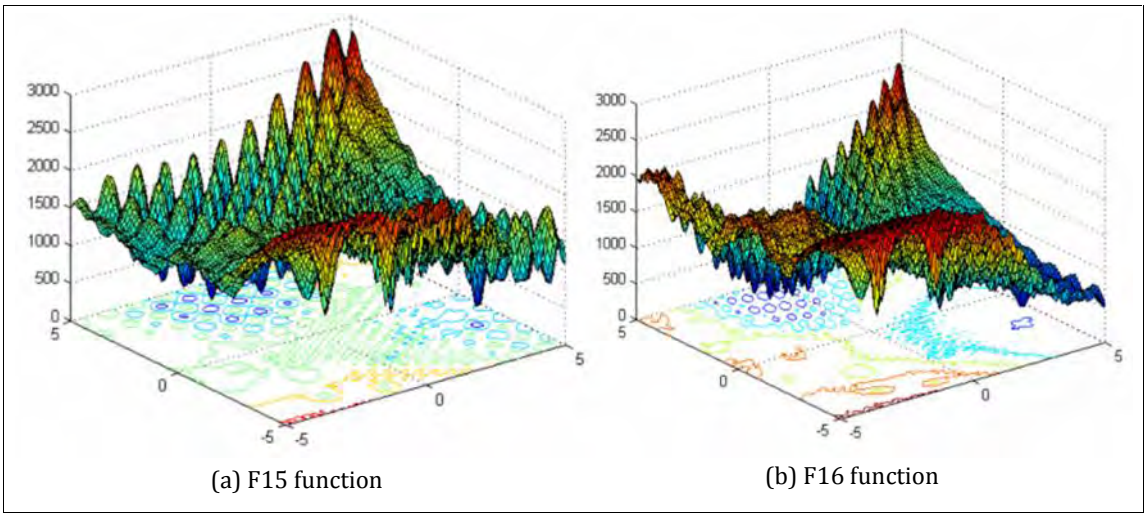
(a) F13 function (b) F14 function

Figure A.2.8: 3-D shaded surface plots of the expanded functions F13 and F14

F15: Hybrid Composition Function This is a multimodal, separable (near the global optimum) and scalable hybrid function with a huge number of local optima where characteristics of different constituent functions, such as the Rastrigin, Weierstrass, Griewank, Ackley and Sphere functions, are mixed together. The separability near the global optimum is due to Rastrigin's function, whereas the two flat areas present in the function are due to the Sphere function. This function is defined within the interval $[-5, 5]^D$.

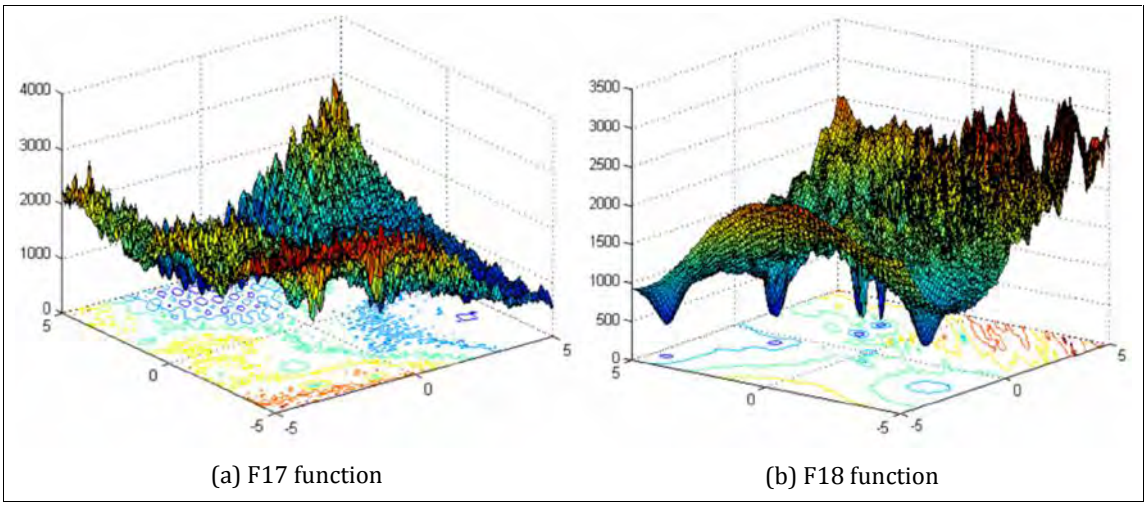
F16: Rotated Version of the Hybrid Composition Function F15 This is a rotated version of the previous function (F15) with similar characteristics. It uses the same constituent functions

(i.e., the Rastrigin, Weierstrass, Griewank, Ackley and Sphere functions) and is defined within the same search area, i.e., $[-5,\ 5]^D$.



(a) F15 function (b) F16 function

Figure A.2.9: 3-D shaded surface plots of the hybrid composition functions F15 and F16



(a) F17 function (b) F18 function

Figure A.2.10: 3-D shaded surface plots of the hybrid composition functions F17 and F18

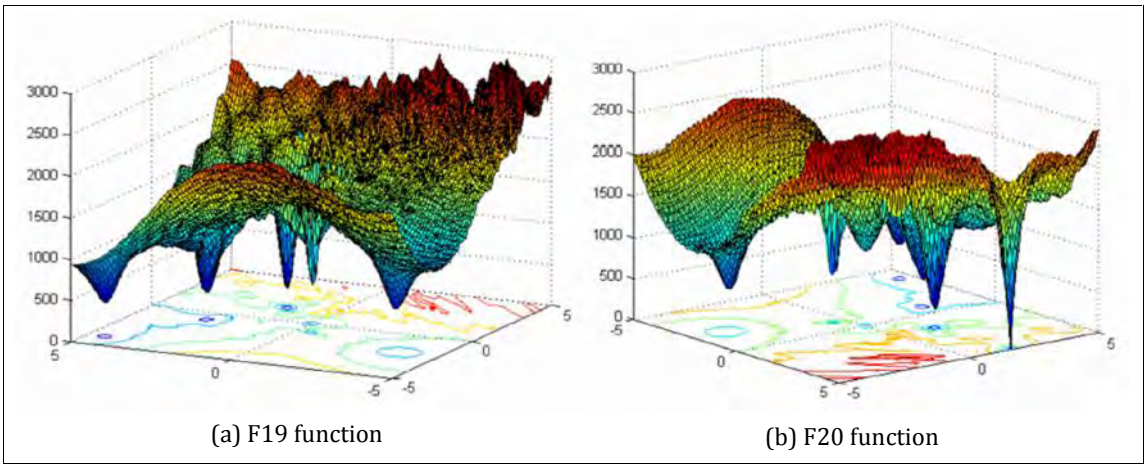
F17: F16 with Noise in Fitness This is the same function as the previous one with the only difference that a Gaussian Noise is introduced in the fitness function, as shown in eq. (A.2.15).

$$F_{17}(x) = F_{16}(x) * (1 + 0.2 \mid N(0,1) \mid) \qquad \text{(A.2.15)}$$

F18: Rotated Hybrid Composition Function This is a multimodal, rotated, non-separable and scalable hybrid function with a huge number of local optima in which properties of different functions such as Ackley, Rastrigin, Sphere, Weierstrass and Griewank are mixed together. Two flat areas are present in the function due to the Sphere function, and a local optimum is set in the origin. This function is defined within the interval $[-5,5]^D$.

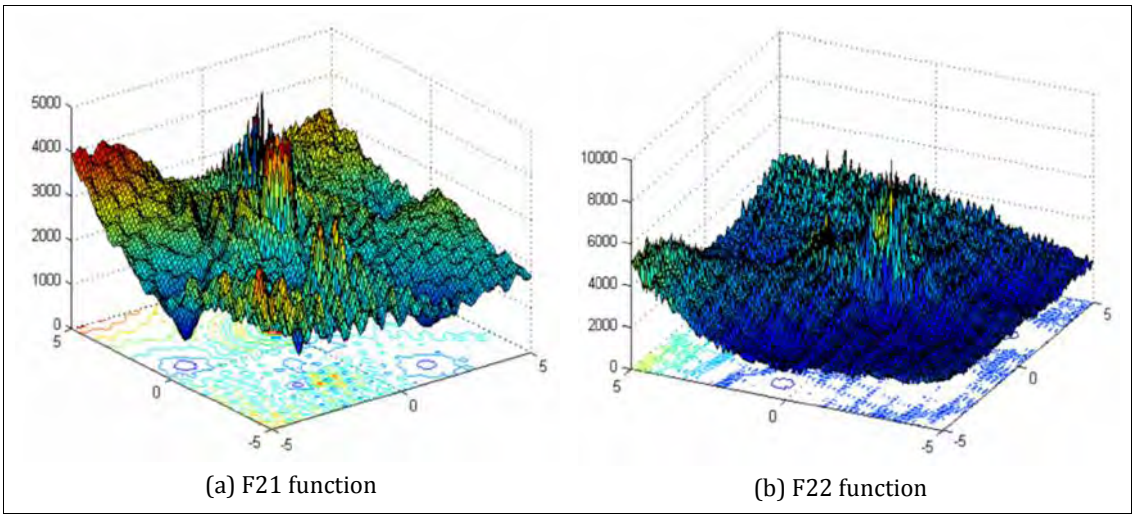
F19: Rotated Hybrid Composition Function with Narrow Basin Global Optimum This is the same function as the previous one, but with different weights for composing the different constituent functions that make the global optimum become a small basin.

F20: Rotated Hybrid Composition Function with Global Optimum at the bounds This is the same function as F18, but with the global optimum shifted to the bounds of the search space.



(a) F19 function (b) F20 function

Figure A.2.11: 3-D shaded surface plots of the hybrid composition functions F19 and F20



(a) F21 function (b) F22 function

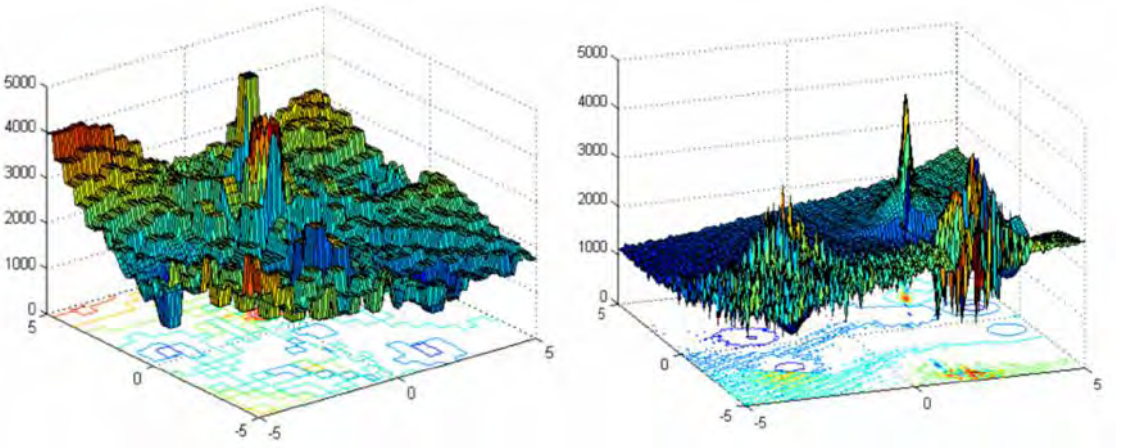
Figure A.2.12: 3-D shaded surface plots of the hybrid composition functions F21 and F22

F21: Rotated Hybrid Composition Function This is a multimodal, rotated, non-separable and scalable hybrid function with a huge number of local optima in which properties of functions such as F13, F14, Rastrigin, Weierstrass and Griewank are mixed together. This function is defined within the interval $[-5, 5]^D$.

F22: Rotated Hybrid Composition Function with High Condition Number Matrix This is the same function as the previous one but with higher condition numbers for linear transformation matrices. This makes the fitness landscape rougher and more difficult to search.

F23: Non-Continuous Rotated Hybrid Composition Function This is a non-continuous neutralized version of the previous F21 function with the global optimum located at the bounds of the search space.

F24: Rotated Hybrid Composition Function This is a multimodal, rotated, non-separable and scalable hybrid function with a huge number of locally optimal points that have properties of



(a) F23 function (b) F24 function

Figure A.2.13: 3-D shaded surface plots of the hybrid composition functions F23 and F24

different functions such as Weierstrass, F13, F14, Ackley, Rastrigin, Griewank, non-continuous versions of F14 and Rastrigin, the High Conditioned Elliptic function and the Sphere function with noise in fitness are mixed together. It presents several flat areas due to the unimodal functions. This function is defined within the interval $[-5, 5]^D$.

F25: Rotated Hybrid Composition Function without Bounds This is the same function as F24 except for the open (unbounded) search range that is set for it. The population should be initialized within the interval $[2, 5]^D$.

Appendix B

List of Publications

1. M. S. Alam, M. M. Islam, X. Yao and K. Murase, "Recurring Two-Stage Evolutionary Programming: A novel approach for numeric optimization," IEEE Transactions on Systems, Man and Cybernetics, Part B: Cybernetics, vol. 41, no. 5, pp. 1352–1365, Oct. 2011.
2. M. S. Alam, M. M. Islam, X. Yao and K. Murase, "Diversity Guided Evolutionary Programming: A novel approach for continuous optimization," Applied Soft Computing, vol. 12, no. 6, pp. 1693–1707, Jun. 2012.
3. M. S. Alam and M. M. Islam, "Artificial bee colony algorithm with self-adaptive mutation: A novel approach for numeric optimization," in Proceedings of the IEEE 2011 International Conference on Trends and Developments in Converging Technology (TENCON), Bali, Indonesia, Nov. 21–24, 2011, pp. 49–53.
4. M. S. Alam, M. M. Islam and K. Murase, "Artificial bee colony algorithm with improved explorations for numerical function optimization," in Proceedings of the 13th International Conference on Intelligent Data Engineering and Automated Learning (IDEAL), Natal, Brazil, Aug. 29–31, 2012, pp. 1–8.
5. M. S. Alam, M. M. Islam and K. Murase, "Artificial bee colony algorithm with adaptive explorations and exploitations: A novel approach for continuous optimization," manuscript submitted for publication to Applied Mathematics and Computation (Imprint: ELSEVIER, ISSN: 0096-3003).
6. M. S. Alam, M. M. Islam and K. Murase, "Artificial bee colony algorithm with improved explorations: A novel approach for numerical optimization," manuscript submitted for publication to International Journal of Computational Intelligence and Applications (Imprint: World Scientific Publishing Co., Print ISSN: 1469-0268, Online ISSN: 1757-5885).

References

[1] Holland, J., Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, 1975.

[2] Koza, J., Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA, 1992.

[3] Fogel, D., Evolutionary Computation: Toward a New Philosophy of Machine Intelligence. IEEE Press, New York, 1995.

[4] Schwefel, H.-P., Numerical Optimization of Computer Models. John Wiley & Sons, Chichester, UK, 1981.

[5] Dorigo, M. and Stützle, T., Ant Colony Optimization. MIT Press, Cambridge, MA, 2004.

[6] Parsopoulos, K. and Vrahatis, M., "Recent approaches to global optimization problems through particle swarm optimization", Natural Computing, Vol-1, No. 2-3, pp 235-306, 2002.

[7] Das, S., Biswas, A., Dasgupta, S. and Abraham, A., "Bacterial foraging optimization algorithm: Theoretical foundations, analysis, and applications", Foundations of Computational Intelligence, Vol-3: Global Optimization, pp 23–55, Springer, 2009.

[8] Karaboga, D., Gorkemli, B., Ozturk, C. and Karaboga, N., "A comprehensive survey: artificial bee colony (ABC) algorithm and applications", Artificial Intelligence Review, pp 1–37, 2012.

[9] Lim, C. P., Jain, L. C. and Dehuri, S., Eds., Innovations in Swarm Intelligence. Springer Publishing Inc., 2009.

[10] Yao, J., Kharma, N., and Grogono, P., "Bi-objective multipopulation genetic algorithm for multimodal function optimization", *IEEE Transactions on Evolutionary Computation.*, Vol–14, No. 1, pp 80–102, 2010.

[11] Karaboga, D. and Akay, B., "A comparative study of artificial bee colony algorithm", Applied Mathematics and Computation, Vol-214, No. 1, pp 108–132, 2009.

[12] Oliveto, P., He, J. and Yao, X., "Analysis of the (1+1)–EA for finding approximate solutions to vertex cover problems", IEEE Transactions on Evolutionary Computation, Vol–13, No. 5, pp 1006–1029, 2009.

[13] Bai, Q. and Yun, X., "A new hybrid artificial bee colony algorithm for the traveling salesman problem", Proceedings of the 3rd International Conference on Communication Software and Networks (ICCSN), 2011, pp 155–159.

[14] Arnold, D. and Hausen, N., "A (1+1)–CMA-ES for constrained optimization", Proceedings of the 21st Genetic and Evolutionary Computation Conference (GECCO), 2012, pp 297---304.

[15] Stanarevic, N., M. Tuba, N. Bacanin, “Modified artificial bee colony algorithm for constrained problems optimization”, International Journal of Mathematical Models and Methods in Applied Sciences, Vol-5, No. 3, pp 644–651, 2011.

[16] Soylu, B. and Koksalan, M., “A favorable weight–based evolutionary algorithm for multiple criteria problems”, IEEE Transactions on Evolutionary Computation, Vol–14, No. 2, pp 191–205, 2010.

[17] Omkar, S., Senthilnath, J., Khandelwal, R., Naik, G. and Gopalakrishnan, S., “Artificial bee colony (ABC) for multi-objective design optimization of composite structures", Applied Soft Computing, Vol–11, No. 1, pp 489–499, 2011.

[18] Xin, B., Chen, J., Zhang, J., Fang, H. and Z. Peng, “Hybridizing differential evolution and particle swarm optimization to design powerful optimizers: A review and taxonomy”, IEEE Transactions on Systems, Man and Cybernetic–Part C: Applications and Reviews, Vol–42, No. 5, pp 744–767, 2012.

[19] Kang, F., Li, J. and Xu, Q., “Structural inverse analysis by hybrid simplex artificial bee colony algorithms”, Computers and Structures, Vol–87, No. 13–14, pp 861–870, 2009.

[20] Aguilar–Ruiz, J., Riquelme, J. and Toro, M., “Evolutionary learning of hierarchical decision rules”, IEEE Transactions on Systems, Man and Cybernetic–Part B: Cybernetics, Vol–33, pp 324–331, 2009.

[21] Schneider, G., Wersing, H., Sendhoff, B. and Körner, E., “Evolutionary optimization of a hierarchical object recognition model”, IEEE Transactions on Systems, Man and Cybernetic–Part B: Cybernetics, Vol–35, pp 426–437, 2008.

[22] Mukhopadhyay, A., Maulik, U. and Bandyopadhyay, S., “Multiobjective genetic algorithm–based fuzzy clustering of categorical attributes”, IEEE Transactions on Evolutionary Computation, Vol–13, No. 5, pp 991–1005, 2009.

[23] Han, S. and Cho, S., “Evolutionary neural networks for anomaly detection based on the behavior of a program”, IEEE Transactions on Systems, Man and Cybernetic–Part B: Cybernetics, Vol–36, pp 559–570, 2008.

[24] Irani, R. and Nasimi, R., “Application of artificial bee colony–based neural network in bottom hole pressure prediction in underbalanced drilling”, Journal of Petroleum Science and Engineering, Vol–78, No. 1, pp 6–12, 2011.

[25] Karaboga, N., “A new design method based on artificial bee colony algorithm for digital IIR filters”, Journal of the Franklin Institute, Vol–346, No. 4, pp 328–348, 2009.

[26] Karaboga, D. and Akay, B., “PID controller design by using artificial bee colony, harmony search and bees algorithms”, Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering, Vol–224, No. 7, pp 869–883, 2010.

[27] Rao, R. and Pawar, P., “Parameter optimization of a multi–pass milling process using non–traditional optimization algorithms”, Applied Soft Computing, Vol–10, No. 2, pp 445–456, 2010.

[28] Brownlee, J., Clever Algorithms: Nature–Inspired Programming Recipes, 1st ed. Raleigh, NC: Lulu Enterprises, Inc., 2011.

[29] Lewis, R., Torczon, V. and Trosset, M., "Direct search methods: Then and now", Journal of Computational and Applied Mathematics, Vol–124, pp 191–207, 2000.

[30] Chen, Y. and Chen, C., "Enabling the extended compact genetic algorithm for real-parameter optimization by using adaptive discretization", Evolutionary Computation, Vol–18, No. 2, pp 199–228, 2010.

[31] Yao, J., "A bi-objective multi-population genetic algorithm with applications to function optimization and ellipse detection", Doctoral Dissertation, Concordia University, Montreal, Canada, 2008.

[32] Korošec, P. and J. Šilc, "A distributed ant-based algorithm for numerical optimization", Proceedings of the 1st Workshop on Bio-inspired Algorithms for Distributed Systems (BADS), 2009, pp 37–44.

[33] Sato, S., Otori, K., Takizawa, A., Sakai, H., Ando, Y. and Kawamura, H., "Applying genetic algorithms to the optimum design of a concert hall", Journal of Sound and Vibration, Vol–258, No. 3, pp 517–526, 2002.

[34] Obayashi, S., Sasaki, D., Takeguchi, Y. and Hirose, N., "Multi-objective evolutionary computation for supersonic wing-shape optimization", IEEE Transactions on Evolutionary Computation, Vol–4, No. 2, pp 182–187, 2000.

[35] Charbonneau, P., "Genetic algorithms in astronomy and astrophysics", The Astrophysical Journal Supplement Series, Vol–101, pp 309–334, 1995.

[36] Haas, O., Burnham, K. and Mills, J., "On improving physical selectivity in the treatment of cancer: A systems modelling and optimisation approach", Control Engineering Practice, Vol–5, No.12, pp 1739–1745.

[37] Gillet, V., "Reactant and product-based approaches to the design of combinatorial libraries", Journal of Computer-Aided Molecular Design, Vol–16, p.371–380, 2002.

[38] Au, W., Chan, K. and Yao, X., "A novel evolutionary data mining algorithm with applications to churn prediction", IEEE Transactions on Evolutionary Computation, Vol–7, No. 6, pp 532–545, 2003.

[39] Mohammadi, B. and Pironneau, O., "Shape Optimization in Fluid Mechanics", Annual Review of Fluid Mechanics, Vol–36, pp 255–279, 2004.

[40] Andreas, A., Georgopoulos, E. and Likothanassis, S., "Exchange-rates forecasting: A hybrid algorithm based on genetically optimized adaptive neural networks", Computational Economics, Vol–20, No.3, pp 191–210, 2002.

[41] Bayer, P., Bürger, C. and Finkel, M., "Solving demanding reliability based design problems in hydrogeology", Proceedings of the 6th International Conference on Calibration and Reliability in Groundwater Modeling: Credibility in Modeling (ModelCARE), 2007, pp 22–26.

[42] Yonggon, L. and Zak, S., "Designing a genetic neural fuzzy antilock-brake-system controller", IEEE Transactions on Evolutionary Computation, Vol–6, No.2, pp 198–211, 2002.

[43] Assion, A., Baumert, T., Bergt, M., Brixner, T., Kiefer, B., Seyfried, V., Strehle, M. and Gerber, G., "Control of chemical reactions by feedback-optimized phase-shaped femtosecond laser pulses", Science, Vol–282, pp 919–922, 1998.

[44] Gálvez, A. and Iglesias, A., "Efficient particle swarm optimization approach for data fitting with free knot B-splines", Computer-Aided Design, Vol–43, No. 12, pp 1683—1692, 2011.

[45] Rizki, M., Zmuda, M. and Tamburino, L., "Evolving pattern recognition systems", IEEE Transactions on Evolutionary Computation, Vol–6, No. 6, pp 594–609, 2002.

[46] Glen, R. and Payne, A., "A genetic algorithm for the automated generation of molecules within constraints", Journal of Computer-Aided Molecular Design, Vol–9, pp. 181–202, 2005.

[47] Leite, P., Carneiro, A. and Carvalho, A., "Hybrid genetic algorithm applied to the determination of the optimal operation of hydrothermal systems", Proceedings of the 9th Brazilian Symposium on Neural Networks (SBRN), 2006, pp 15–22.

[48] Haupt, R. and Haupt, S., Practical Genetic Algorithms. Wiley-Interscience, 2004.

[49] Wales, D. and Scheraga, H., "Global optimization of clusters, crystals and biomolecules", Science, Vol–285, No. 5432, pp 1368–1372, 1999.

[50] Doye, J., "Physical perspectives on the global optimization of atomic clusters", Global Optimization, pp 103–139, 2006.

[51] Andre, D. and Teller, A., "Evolving team Darwin united", RoboCup-98: Robot Soccer World Cup II, pp 346–351, 1999.

[52] Grosan, C. and Abraham, A., "A new approach for solving nonlinear equations systems", IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans, Vol–38, No. 3, pp 698–714, 2008.

[53] Bürger, C., Bayer, P. and Finkel, M., "Algorithmic funnel–and–gate system design optimization", Water Resources Research, Vol–43, No. 8, 2007.

[54] Lampinen, J. and Zelinka, I., "On stagnation of the differential evolution algorithm", Proceedings of the 6th International Mendel Conference on Soft Computing, 2000, pp 76–83.

[55] Akay, B. and Karaboga, D., "A modified artificial bee colony algorithm for real–parameter optimization", Information Sciences Vol–192, pp 120–142, 2012.

[56] Lee, C. and Yao, X., "Evolutionary programming using mutations based on the Lévy probability distribution", IEEE Transactions on Evolutionary Computation, Vol–8, No. 1, pp 1–13, 2004.

[57] Yao, X., Liu, Y. and Lin, G., "Evolutionary programming made faster", IEEE Transactions on Evolutionary Computation, Vol–3, No. 2, pp 82–102, 1999.

[58] Nieberg, S. and Beyer, H., “Self-adaptation in evolutionary algorithms”, Parameter Setting in Evolutionary Algorithm, pp 47–76, 2007.

[59] Alam, M. S. and Islam, M. M., “Artificial bee colony algorithm with self-adaptive mutation: A novel approach for numeric optimization”, Proceedings of the 2011 IEEE International Conference on Trends and Developments in Converging Technology (TENCON), 2011, pp 49–53.

[60] Abd, M., “A cooperative approach to the artificial bee colony algorithm”, in Proceedings of the 2010 IEEE Congress on Evolutionary Computation (CEC), 2010, pp 1–5.

[61] Lee, W. and Cai, W., “A novel artificial bee colony algorithm with diversity strategy”, Proceedings of the 7th International Conference on Natural Computation (ICNC), 2011, pp 1441–1444.

[62] Wu, B. and Fan, S., “Improved Artificial Bee Colony Algorithm with Chaos”, Proceedings of the Computer Science for Environmental Engineering and Eco–Informatics, Part I, Communications in Computer and Information Science, Yu, Y., Yu, Z., Zhao, J., Eds. Vol—158, pp 51–56, 2011.

[63] Fenglei, L., Haijun, D. and Xing, F., “The parameter improvement of bee colony algorithm in TSP problem”, Science Paper Online, November 2007.

[64] Zhu, G. and Kwong, S., “Gbest–guided artificial bee colony algorithm for numerical function optimization”, Applied Mathematics and Computation, Vol–217, No. 7, 3166—3173, 2010.

[65] Kang, F., Li, J., Ma, Z. and Li, H., “Artificial bee colony algorithm with local search for numerical optimization”, Journal of Software, Vol–6, No. 3, pp 490–497, 2011.

[66] Qingxian, F. and Haijun, D., “Bee colony algorithm for the function optimization”, Science Paper Online, August 2008.

[67] Huan, H. and Shi, X., “On the analysis of performance of the improved artificial bee colony algorithm”, Proceedings of the 4th IEEE International Conference on Natural Computation (ICNC), 2008, pp 654–658.

[68] Montes, E. and Koeppel, R., “Elitist artificial bee colony for constrained real-parameter optimization”, IEEE Congress on Evolutionary Computation, Vol–11, pp 1–8, 2010.

[69] Liang, J., Qin, A., Suganthan, P. and Baskar, S., “Comprehensive learning particle swarm optimizer for global optimization of multimodal functions”, IEEE Transactions on Evolutionary Computation, Vol–10, No. 3, pp 281–295, 2006.

[70] Felice, M., Meloni, S. and Panzieri, S., “Effect of topology on diversity of spatially-structured evolutionary algorithms”, Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, 2011, pp 1579–1586.

[71] Li, C. and Yang, S., “An island based hybrid evolutionary algorithm for optimization”, Proceedings of the 7th International Conference on Simulated Evolution and Learning, 2008, pp 180–189.

[72] Ursem, R., “Multinational evolutionary algorithms”, Proceedings of the IEEE Congress on Evolutionary Computation (CEC), 1999, pp 1633–1640.

[73] Guo, Y. and Liu, D., “Multi-population cooperative particle swarm cultural algorithms”, Proceedings of the 7th International Conference on Natural Computation, Shanghai, China, 26–28 Jul. 2011, pp 1351–1355.

[74] Yen, G. and Leong, W., “Dynamic multiple swarms in multiobjective PSO”, IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans, Vol–39, No.4, pp 890–911, 2009.

[75] Karaboga, D., “An idea based on honey bee swarm for numerical optimization”, Erciyes University, Kayseri, Turkey, Technical Report–TR06, 2005.

[76] Suganthan, P., Hansen, N., Liang, J., Deb, K., Chen, Y., Auger, A. and Tiwari, S., “Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization”, Nanyang Technological University, Singapore, Technical Report, 2005 and IIT Kanpur, India, KanGAL Report No. 2005005, 2005.

[77] Luke, S., Essentials of Metaheuristics. Lulu, 2009.

[78] Alam, M. S., Islam, M. M. and Murase, K., “Artificial Bee Colony Algorithm with Improved Explorations for Numerical Function Optimization”, Proceedings of the 13th International Conference on Intelligent Data Engineering and Automated Learning (IDEAL), 2012, pp 1–8.

[79] Parpinelli, R. and Lopes, H., “New inspirations in swarm intelligence: a survey”, International Journal on Bio-Inspired Computation, Vol–3, No. 1, pp 1–16, 2011.

[80] Chaudhuria, S. and Deb, K., “An interactive evolutionary multi-objective optimization and decision making procedure”, Applied Soft Computing, Vol–10, No. 2, pp 495–511, 2010.

[81] Mühlenbein, H. and Schlierkamp-Voosen, D., “Predictive Models for the Breeder Genetic Algorithm: Continuous Parameter Optimization”, Evolutionary Computation, Vol–1, No. 1, pp 25–49, 1993.

[82] Bäck, T., “Optimal mutation rates in genetic search”, Proceedings of the 5th International Conference on Genetic Algorithms (ICGA), 1993, pp 2–8.

[83] Bäck, T., Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms. Oxford University Press, USA, 1996.

[84] Mühlenbein, H., “The Breeder Genetic Algorithm – a provable optimal search algorithm and its application”, IEEE Colloquium on Applications of Genetic Algorithms, Digest No. 94/067, London, March 15, 1994.

[85] Ostermeier, A., Gawelczyk, A. and Hansen, N., “A derandomized approach to self–adaptation of evolution strategies”, Evolutionary Computation, Vol–2, No. 4, pp 369–380, 1994.

[86] Ostermeier, A., Gawelczyk, A. and Hansen, N., “Step size adaptation based on non-local use of selection information”, Proceedings of the 3rd Parallel Problem Solving from Nature (PPSN), 1994, pp 189–198.

[87] Hansen, N., Ostermeier, A. and Gawelczyk, A., “On the adaptation of arbitrary normal mutation distributions in evolution strategies: The generating set adaptation”, Proceedings of the 6th International Conference on Genetic Algorithms (ICGA), 1997, pp 57–64.

[88] Chen., L., “An adaptive genetic algorithm based on population diversity strategy”, Proceedings of the 3rd International Conference on Genetic and Evolutionary Computing, 2009, pp 93–96.

[89] Goldberg, D., Genetic Algorithms in Search, Optimization, and Machine Learning. Addison–Wesley, New York, 1989.

[90] Eshelman, L., Caruana, A. and Schaffer, J., “Biases in the Crossover Landscape”, Proceedings of the 3rd International Conference on Genetic Algorithms, 1989, pp 10—19.

[91] Syswerda, G., “Uniform Crossover in Genetic Algorithms”, Proceedings of the 3rd International Conference on Genetic Algorithms, 1989, pp 2–9.

[92] Radcliffe, N., “Equivalence Class Analysis of Genetic Algorithms”, Complex Systems, Vol-5, No. 2, pp 183–205, 1991.

[93] Herrera, F., Lozano, M. and Verdegay, J., “Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis”, Artificial intelligence review, Vol–12, No. 4, pp 265–319, 1998.

[94] Michalewicz, Z., Genetic Algorithms + Data Structures = Evolution Programs. Springer–Verlag, New York, USA, 1998.

[95] Herrera, F., Lozano, M., Pérez, E., Sánchez, A. and Villar, P., “Multiple crossover per couple with selection of the two best offspring: An experimental study with the BLX–α crossover operator for real-coded genetic algorithms”, Proceedings of the 8th Ibero-American Conference on AI (IBERAMIA), 2002, pp 392–401.

[96] Wright, A., “Genetic algorithms for real parameter optimization”, Foundations of Genetic Algorithms, Vol–1, pp 205–218, 1991.

[97] Schlierkamp–Voosen, D., “Strategy Adaptation by Competition”, Proceedings of the 2nd European Congress on Intelligent Techniques and Soft Computing, 1994, pp 1270—1274.

[98] Herrera, F., Herrera-Viedma, E., Lozano, M. and Verdegay, J., “Fuzzy tools to improve genetic algorithms”, Proceedings of the 2nd European Congress on Intelligent Techniques and Soft Computing, 1994, pp 1532–1539.

[99] Davis, L., “Adapting operator probabilities in genetic algorithms”, Proceedings of the 3rd International Conference on Genetic Algorithms, 1989, pp 61–69.

[100] Kelly, J. and Davis, L., "Hybridizing the GA and the *k*-nearest neighbors classification algorithm", Proceedings of the 4th International Conference on Genetic Algorithms, 1991, pp 377–383.

[101] Voigt, H. and Anheyer, T., "Modal mutations in evolutionary algorithms", Proceedings of the 1st IEEE Conference on Evolutionary Computation, 1994, pp 88–92.

[102] Bäck, T. and Schwefel, H.–P., "An overview of evolutionary algorithms for parameter optimization", Evolutionary Computation, Vol–1, No. 1, pp 1–23, 1993.

[103] Bonabeau, E. and Dorigo, M., Theraulaz, G., Swarm Intelligence: from Natural to Artificial Systems. Oxford University Press, New York, USA, 1999.

[104] Millonas, M., "Swarms, phase transitions, and collective intelligence", Artificial life III. Addison–Wesley, Reading, pp 417–445, 1994.

[105] Waibel, M., Floreano, D., Magnenat S. and Keller, L, "Division of labour and colony efficiency in social insects: Effects of interactions between genetic architecture, colony kin structure and rate of perturbations", Proceedings of the Royal Society B: Biological Sciences, Vol–273, No. 1595, pp 1815–1823, 2006.

[106] Grosan., C. and Abraham, A., "Stigmergic optimization: Inspiration, technologies and perspectives", Stigmergic optimization, pp 1–24, 2006.

[107] De Castro, L. and Von Zuben, F., "Learning and optimization using the clonal selection principle", IEEE Transactions on Evolutionary Computation, Vol–6, No. 3, pp 239–251, 2002.

[108] Forrest, S., Perelson, A., Allen, L. and Cherukuri, R., "Self-nonself discrimination in a computer", Proceedings of the IEEE Symposium on Research in Security and Privacy, 1994, pp 202–212.

[109] Timmis, J., Neal, M. and Hunt, J., "An artificial immune system for data analysis", BioSystems, Vol–55, No. 1, pp 143–150, 2000.

[110] Greensmith, J. and Aickelin, U., "Artificial dendritic cells: Multi-faceted perspectives", Human–Centric Information Processing Through Granular Modelling, pp 375–395, 2009.

[111] Passino, K., "Bacterial Foraging Optimization", International Journal of Swarm Intelligence Research, Vol. 1, No. 1, pp 1–16, 2010.

[112] Yang, X., "A new metaheuristic bat-inspired algorithm", Nature Inspired Cooperative Strategies for Optimization (NISCO), 2010, pp 65–74.

[113] Civicioglu, P. and Besdok, E., "A conception comparison of the cuckoo search, particle swarm optimization, differential evolution and artificial bee colony algorithms", Artificial Intelligence Review, pp 1–32, 2011.

[114] Civicioglu, P., "Transforming geocentric cartesian coordinates to geodetic coordinates by using differential search algorithm", Computers and Geosciences, Vol–46, pp 229—247, 2012.

[115] Yang, X. –S., Nature–Inspired Metaheuristic Algorithms. Luniver Press, 2008.

[116] Krishnanand K. and Ghose, D., "Detection of multiple source locations using a glowworm metaphor with applications to collective robotics", IEEE Swarm Intelligence Symposium, 2005, pp 84–91.

[117] Krishnanand, K., Ghose, D., "Theoretical foundations for rendezvous of glowworm-inspired agent swarms at multiple locations", Robotics and Autonomous Systems, Vol—56, No. 7, pp 549–569, 2008.

[118] Krishnanand, K., Ghose, D., "Glowworm swarm optimization for simultaneous capture of multiple local optima of multimodal functions", Swarm Intelligence, Vol–3, No. 2, pp 87–124, 2009.

[119] Hendtlass, T., "WoSP: A multi-optima particle swarm algorithm", Proceedings of the IEEE Congress on Evolutionary Computation, 2005, pp 727–734.

[120] Parsopoulos, K. and Vrahatis, M., "Recent approaches to global optimization problems through particle swarm optimization", Natural Computing, Vol–1, No. 2–3, pp 235–306, 2002.

[120] Rashedi, E., Nezamabadi-pour, H., Saryazdi, S., "GSA: a gravitational search algorithm", Information Science, Vol–179, No. 13, pp 2232–2248, 2009.

[121] Formato, R., "Central force optimization: A new metaheuristic with applications in applied electromagnetics", Progress In Electromagnetics Research, Vol–77, pp 425—491, 2007.

[122] Shah-Hosseini, H., "The intelligent water drops algorithm: A nature-inspired swarm-based optimization algorithm", International Journal of Bio-Inspired Computation, Vol—1, No. 1, pp 71–79, 2009.

[123] Waibel, M., Floreano, D. and Keller, L., "A quantitative test of Hamilton's rule for the evolution of altruism", PLoS Biology, Vol–9, No. 5, 2011.

[124] Tayarani, M. and Akbarzadeh, M., "Magnetic optimization algorithms: A new synthesis", IEEE Congress on Evolutionary Computation (CEC), 2008, pp 2659–2664.

[125] Gandomi, A. and Alavi, A., "Krill herd algorithm: A new bio-inspired optimization algorithm", Communications in Nonlinear Science and Numerical Simulation, 2012.

[126] Rabanal, P., Rodríguez, I. and Rubio, F., "Using river formation dynamics to design heuristic algorithms", Unconventional Computation, pp 163–177, 2007.

[127] Vicsek, T., Czirok, A., Ben-Jacob, E., Cohen, I., Shochet, O., "Novel type of phase transition in a system of self-driven particles", Physical Review Letters, Vol–75, pp 1226–1229, 1995.

[128] Nasuto, S., Bishop, J. and Lauria, S., "Time complexity analysis of the stochastic diffusion search", Neural Computation '98, Vienna, Austria, 1998, pp 260–266.

[129] Mehrabian, A. and Lucas, C., "A novel numerical optimization algorithm inspired from weed colonization", Ecological Informatics, Vol–1, pp 355–366, 2006.

[130] Holland, J., "Genetic algorithms", Scientific American, July 1992, pp 66–72.

[131] Mitchell, M., An Introduction to Genetic Algorithms. MIT Press, 1996.

[132] Koza, J., Martin, K., Streeter, M., Mydlowec, W., Yu J. and Lanza, G., Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers, 2003.

[133] Forrest, S., "Genetic algorithms: Principles of natural selection applied to computation", Science, Vol–261, pp 872–878, 1993.

[134] Dawkins, R., The Blind Watchmaker: Why the Evidence of Evolution Reveals a Universe Without Design. W.W. Norton, 1996.

[135] Kramer, O., "Evolutionary self-adaptation: A survey of operators and strategy parameters", Evolutionary Intelligence, Vol–3, No. 2, pp 51–65, 2010.

[136] Graham-Rowe, D., "Radio emerges from the electronic soup", New Scientist, Vol–175, No. 2358, pp 19, 2002.

[137] Fogarty, T., "Varying the probability of mutation in the genetic algorithm", Proceedings of the 3rd International Conference on Genetic Algorithms, 1989, pp 104–109.

[138] Hesser, J. and Männer, R., "Investigation of the metaheuristic for optimal mutation probabilities", Proceedings of the 2nd Parallel Problem Solving Nature (PPSN), 1992, pp 115–124.

[139] Thierens, D., "Adaptive mutation rate control schemes in genetic algorithms", Proceedings of the IEEE Congress on Evolutionary Computation (CEC), 2002, pp 980—985.

[140] Bäck, T., "The interaction of mutation rate, selection and self-adaptation within a genetic algorithm", Proceedings of the 2nd Parallel Problem Solving from Nature (PPSN), 1992, pp 85–94.

[141] Ursem, R., "Diversity guided evolutionary algorithms", Proceedings of the 7th Parallel Problem Solving from Nature (PPSN), 2002, pp 462–471.

[142] Bäck, T., Fogel, D. and Michalewicz, Z., Handbook on Evolutionary Computation. IOP Publishing Ltd. and Oxford University Press, 1997.

[143] Jumonji, T., Chakraborty, G., Mabuchi, H. and Matsuhara, M. "A novel distributed genetic algorithm implementation with variable number of islands", Proceedings of the IEEE Congress on Evolutionary Computation (CEC), 2007, pp 4698–4705.

[144] Thomsen, R., Rickers, P. and Krink, T., "A religion-based spatial model for evolutionary algorithms", Proceeding of the 6th Parallel Problem Solving from Nature (PPSN), 2000, pp 817–826.

[145] Tsutsui, S. and Fujimoto, Y., "Forking genetic algorithm with blocking and shrinking modes", Proceedings of the 5th International Conference on Genetic Algorithms, 1993, pp 206–215.

[146] Park, T. and Ryu, K., "A dual-population genetic algorithm for adaptive diversity control", IEEE Transactions on Evolutionary Computation, Vol–14, No. 6, pp 865–884, 2010.

[147] Cantúz–Paz, E., “Migration policies, selection pressure, and parallel evolutionary algorithms”, Journal of Heuristics, Vol–7, No. 4, pp 311–334, 2001.

[148] Rudolph, G., “On takeover times in spatially structured populations: Array and ring”, Proceedings of the 2nd Asia-Pacific Conference on Genetic Algorithms Applications, 2000, pp 144–151.

[149] Sekaj, I., “Robust parallel genetic algorithms with re-initialization”, Proceedings of the 8th Parallel Problem Solving Nature (PPSN), 2004, pp 411–419.

[150] Skolicki, Z. and Jong, K., “The inuence of migration sizes and intervals on island models”, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), 2005, pp 1295–1302.

[151] Sareni, B. and Krahenbuhl, L., “Fitness sharing and niching methods revisited”, IEEE Transactions on Evolutionary Computation, Vol–2, No. 3, pp 97–106, 1998.

[152] Deb, K. and Goldberg, D., “An investigation of niche and species formation in genetic function optimization”, in Proceedings of the 3rd International Conference on Genetic Algorithm (ICGA), 1989, pp 42–50.

[153] Della Cioppa, A., De Stefano, C. and Marcelli, A. “On the role of population size and niche radius in fitness sharing”, IEEE Transactions on Evolutionary Computation, Vol–8, No. 6, pp 580–592, 2004.

[154] Della Cioppa, A., De Stefano, C. and Marcelli, A. “Where are the niches? Dynamic fitness sharing”, IEEE Transactions on Evolutionary Computation, Vol–11, No. 4, pp 453–465, 2007.

[155] Mahfoud, S., “Crowding and preselection revisited”, Proceedings of the 2nd Parallel Problem Solving from Nature (PPSN), 1992, pp 27–36.

[156] Mahfoud, S., “Niching methods for genetic algorithms”, Ph.D. dissertation, Dept. of General Engineering, University of Illinois, Urbana-Champaign, USA, 1995.

[157] Mengsheol, O. and Goldberg, D., “Probabilistic crowding: Deterministic crowding with probabilistic replacement”, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), 1999, pp 409–416.

[158] Harick, G., “Finding multimodal solutions using restricted tournament selection”, Proceedings of the 6th International Conference on Genetic Algorithms (ICGA), 1995, pp 24–31.

[159] Shimodaira, H., “A diversity control oriented genetic algorithm: Development and experimental results”, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), 1999, pp 603–611.

[160] Lozano, M., Herrera, F., Krasnogor, N. and Molina, D., “Real-coded memetic algorithms with crossover hill-climbing”, Evolutionary Computation, Vol–12, pp 273–302, 2004.

[161] Rechenberg, I., “Evolutions strategie: Optimierung technischer systeme nach prinzipien der biologischen evolution”, Frommann-Holzboog Verlag, Stuttgart, 1973.

[162] Molina, D., Herrera, F. and Lozano, M., "Adaptive local search parameters for real-coded memetic algorithms", Proceedings of the IEEE Congress on Evolutionary Computation (CEC), 2005, pp 888–895.

[163] Yang, Z., Yao, X. and He, J., "Making a difference to differential evolution", Advances in Metaheuristics for Hard Optimization. Springer, pp 397–414, 2008.

[164] Yang, Z., Tang, K. and Yao, X., "Differential evolution for high-dimensional function optimization", Proceedings of the IEEE Congress on Evolutionary Computation, (CEC), 2007, pp 3523–3530.

[165] Hansen, N., Muller, S. and Koumoutsakos, P., "Reducing the time complexity of the de-randomized evolution strategy with covariance matrix adaptation (CMA-ES)", Evolutionary Computation, Vol–11, pp 1–18, 2003.

[166] Park, T. and Ryu, K., "Crew pairing optimization by a genetic algorithm with unexpressed genes", Journal of Intelligent Manufacturing., Vol–17, No. 4, pp 375–383, 2006.

[167] Kominami, M. and Hamagami, T., "A new genetic algorithm with diploid chromosomes by using probability decoding for non-stationary function optimization", Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, 2007, pp 1268—1273.

[168] Yoshida, Y. and Adachi, N., "A diploid genetic algorithm for preserving population diversity—Pseudo-Meiosis GA", Proceedings of the 3rd Parallel Problem Solving Nature (PPSN), 1994, pp 36–45.

[169] Collard, P. and Aurand, J., "DGA: An efficient genetic algorithm", Proceedings of the 11th European Conference on Artificial Intelligence, 1994, pp 487–491.

[170] Collard, P. and Escazut, C., "Genetic operators in a dual genetic algorithm", Proceedings of the 7th International Conference on Tools with Artificial Intelligence, 1995, pp 12—19.

[171] Yang, S., "PDGA: The primal-dual genetic algorithm", Design and Application of Hybrid Intelligent Systems. IOS Press, Amsterdam, The Netherlands, pp 214–223, 2003.

[172] Yang, S., "Nonstationary problem optimization using the primal-dual genetic algorithm", Proceedings of the IEEE Congress on Evolutionary Computation, Vol–3, pp 2246–2253, 2003.

[173] Tereshko, V., "Reaction-diffusion model of a honeybee colony's foraging behavior", Proceedings of the 6th Parallel Problem Solving from Nature (PPSN), 2000, pp 807—816.

[174] Tereshko, V. and Lee, T., "How information mapping patterns determine foraging behaviour of a honeybee colony", Open Systems and Information Dynamics, Vol–9, pp 181–193, 2002.

[175] Tereshko, V. and Loengarov, A., "Collective decision-making in honeybee foraging dynamics", Computing and Information Systems Journal, Vol–9, No. 3, 2005.

[176] Wedde, H., Farooq, M. and Zhang, Y., "BeeHive: An efficient fault-tolerant routing algorithm inspired by honey bee behavior", Ant Colony, Optimization and Swarm Intelligence, pp 83–94, 2004.

[177] Teodorovic, D. and Dell'Orco, M., "Bee Colony Optimization – A Cooperative Learning Approach to Complex Transportation Problems", in Advanced OR and AI Methods in Transportation: Proceedings of the 16th Mini–EURO Conference and 10th Meeting of EWGT, 2005, pp 51–60.

[178] Lucic, P. and Teodorovic, D., "Computing with bees: Attacking complex transportation engineering problems", International Journal on Artificial Intelligence Tools, Vol–12, No. 3, pp 375–394, 2003.

[179] Lucic, P. and Teodorovic, D., "Vehicle routing problem with uncertain demand at nodes: The bee system and fuzzy logic approach", Studies in Fuzziness and Soft Computing, Vol–126, pp 67–82, 2003.

[180] Lucic, P. and Teodorovic, D., "Bee system: Modeling combinatorial optimization transportation engineering problems by swarm intelligence", Preprints of the TRISTAN IV Triennial Symposium on Transportation Analysis, June, 2001, pp 441–445.

[181 Drias, H., Sadeg, S. and Yahi, S., "Cooperative bees swarm for solving the maximum weighted satisfiability problem", Proceedings of the International Work Conference on Artificial and Natural Neural Networks (IWAAN), 2005, pp 318–325.

[182] Benatchba, K., Admane and L., Koudil, M., "Using bees to solve a data-mining problem expressed as a max-sat one", Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach, pp 76–86, 2005.

[183] Fathian, M., Amiri, B. and Maroosi, A., "A honeybee-mating approach for cluster analysis", International Journal of Advanced Manufacturing Technology, Vol–38, pp 809–821, 2008.

[184] Wong, L., Low, M. and Chong, C., "A bee colony optimization algorithm for traveling salesman problem", Proceedings of the Second Asia International Conference on Modelling and Simulation (AMS), 3008, pp 818–823.

[185] Baykasoglu, A., Ozbakor, L. and Tapkan, P., "Artificial bee colony algorithm and its application to generalized assignment problem", Swarm Intelligence: Focus on Ant and Particle Swarm Optimization, pp 113–144, 2007.

[186] Chong, C., Sivakumar, A., Low, M. and Gay, K., "A bee colony optimization algorithm to job shop scheduling", Proceedings of the 38th International Conference on Winter Simulation, 2006, pp 1954–1961.

[187] Nakrani, S. and Tovey, C., "On honey bees and dynamic allocation in an internet server colony", Proceedings of 2nd International Workshop on the Mathematics and Algorithms of Social Insects, 2003, pp 826–833.

[188] Yang, X., "Engineering optimizations via nature-inspired virtual bee algorithms", Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach, pp 317–323, 2005.

[189] Pham, D., Koç, E., Ghanbarzadeh, A., Otri, S., Rahim, S. and Zaidi, M., "The Bees algorithm – A novel tool for complex optimization problems", Proceeding of the 2nd International Virtual Conference on Intelligent Production Machines and Systems (IPROMS), 2006, pp 454–461.

[190] Pham, D., Koç, E., Ghanbarzadeh, A. and Otri, S., "Optimization of the weights of multi-layered perceptions using the bees algorithm", Proceedings of the 5th International Symposium on Intelligent Manufacturing Systems, 2006, pp 38–46.

[191] Karaboga, D. and Basturk, B., "On the performance of artificial bee colony (ABC) algorithm", Applied Soft Computing, Vol–8, No. 1, pp 687–697, 2008.

[192] Karaboga, D. and Basturk, B., "Artificial bee colony (ABC) optimization algorithm for solving constrained optimization problems", Foundations of Fuzzy Logic and Soft Computing, pp 789–798, 2007.

[193] Karaboga, D. and Basturk, B., "A powerful and efficient algorithm for numerical function optimization: Artificial bee colony (ABC) algorithm", Journal of Global Optimization, Vol–39, No. 3, pp 459–471, 2007.

[194] Srinivasan, D. and Seow, T., "Evolutionary Computation", Proceedings of the IEEE Congress on Evolutionary Computation (CEC), 2003, pp 888–895.

[195] Corne, D., Dorigo, M. and Glover, F., New Ideas in Optimization. McGraw-Hill, 1999.

[196] Vesterstrom, J. and Thomsen, R., "A comparative study of differential evolution, particle swarm optimization and evolutionary algorithms on numerical benchmark problems", Proceedings of the IEEE Congress on Evolutionary Computation (CEC), 2004, Vol–2, pp 1980–1987.

[197] Srinivasan, D. and Seow, T., "Particle swarm inspired evolutionary algorithm (PS-EA) for multi-criteria optimization problems", Evolutionary Multiobjective Optimization, pp 147–165, 2005.

[198] Pawar, P., Rao, R. and Davim, J., "Optimization of process parameters of abrasive flow machining process using artificial bee colony algorithm", Proceedings of the Advances in Mechanical Engineering (AME), 2008, pp 520–530.

[199] Pawar, P., Rao, R. and Davim, J., "Optimization of process parameters of milling process using particle swarm optimization and artificial bee colony algorithm", Proceedings of the Advances in Mechanical Engineering (AME), 2008, pp 445–456.

[200] Pawar, P., Rao, R. and Shankar, R., "Multi-objective optimization of electro-chemical machining process parameters using artificial bee colony (ABC) algorithm", Proceedings of the Advances in Mechanical Engineering (AME), 2008, pp 463–472.

[201] Tsai, P., Pan, J., Liao, B. and Chu, S., "Enhanced artificial bee colony optimization", International Journal of Innovative Computing, Information and Control, Vol–5, No. 12, pp 5081–5092, 2009.

[202] Karaboga, D. and Basturk, B., "An artificial bee colony (ABC) algorithm on training artificial neural networks", Proceedings of the 15th IEEE Signal Processing and Communications Applications, 2007, pp 1–4.

[203] Karaboga, D., Ozturk, C. and Akay, B., "Training neural networks with ABC optimization algorithm on medical pattern classification", Proceedings of the International Conference on Multivariate Statistical Modelling and High Dimensional Data Mining, 2008, pp 137–141.

[204] Ozturk, C. and Karaboga, D., "Classification by neural networks and clustering with artificial bee colony (ABC) algorithm", Proceedings of the 6th International Symposium on Intelligent and Manufacturing Systems: Features, Strategies and Innovation, 2008, pp 342–349.

[205] Kang, F., Li, J. and Xu, Q., "Structural inverse analysis by hybrid simplex artificial bee colony algorithms", Computers and Structures, Vol–87, No. 13–14, pp 861–870, 2009.

[206] Mala, D. and Mohan, V., "ABC Tester — Artificial bee colony based software test suite optimization approach", International Journal of Software Engineering, Vol–2, No. 2, pp 15–43, 2009.

[207] Slowik, A., "Steering of balance between exploration and exploitation properties of evolutionary algorithms: Mix selection", Proceedings of the 10th International Conference on Artificial Intelligence and Soft Computing, 2010, pp 213–220.

[208] Naqi, A., Erdogan, A. and Arslan, T., "Balancing exploration and exploitation in an adaptive three-dimensional cellular genetic algorithm via a probabilistic selection operator", in Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2010, pp 258–264.

[209] Zhao, G., Luo, W., Nie, H. and Li, C., "A genetic algorithm balancing exploration and exploitation for the travelling salesman problem", Proceedings of the 4th International Conference Natural Computation (ICNC), 2008, pp 505–509.

[210] Culberson, J., "On the futility of blind search: An algorithmic view of no free lunch", Evolutionary Computation, Vol–6, No. 2, pp 109–128, 1998.

[211] Tsai, J., Liu, T. and Chou, J. "Hybrid Taguchi-genetic algorithm for global numerical optimization", IEEE Transactions on Evolutionary Computation, Vol–8, No. 4, pp 365—377, 2004.

[212] Krasnogor, N. and Smith, J., "A tutorial for competent memetic algorithms: Model, taxonomy and design issues", IEEE Transactions on Evolutionary Computation, Vol–9, No. 5, pp 474–488, 2005.

[213] Chellapilla, K., "Combining mutation operators in evolutionary programming", IEEE Transactions on Evolutionary Computation, Vol–2, No. 3, pp 91–96, 1998.

[214] Gehlhaar, D. and Fogel, D., "Tuning evolutionary programming for conformationally flexible molecular docking", Proceedings of the 5th Annual Conference on Evolutionary Programming, 1996, pp 419–429.

[215] Arnold, D. and Hansen, N., "Active covariance matrix adaptation for the (1+1)-CMAES", Proceedings of the 19th Genetic and Evolutionary Computation Conference (GECCO), 2010, pp 385–392.

[216] Ibaraki, T., "Combination with local search", Handbook of Genetic Algorithms. Oxford University Press, London, U.K., 1997.

[217] Liang, K., Yao, X. and Newton, C., "Adapting self-adaptive parameters in evolutionary algorithms", Applied Intelligence, Vol–15, No. 3, pp 171–180, 2001.

[218] Islam, M. M., Alam, M. S. and Murase, K., "A new recurring multistage evolutionary algorithm for solving problems efficiently", Proceedings of the Intelligent Data Engineering and Automated Learning (IDEAL), 2007, pp 97–106.

[219] Deb, K. and Goyal, M., "A combined genetic adaptive search (GeneAS) for engineering design", Computer Science and Informatics, Vol–26, No. 1, pp 30–45, 1996.

[220] Schaffer, J., Caruana, R., Eshelman, L. and Das, R., "A study of control parameters affecting on-line performance of genetic algorithms for function optimisation", Proceedings of the 3rd International Conference on Genetic Algorithms, 1989, pp 51—60.

[221] Spears, W., "Crossover or Mutation?", Proceedings of the 2nd Workshop on the Foundations of Genetic Algorithms, 1992, pp 221–237.

[222] Hoffmeister, F. and Bäck, T., "Genetic self-learning", Proceedings of the 1st European Conference on Artificial Life: Toward a Practice of Autonomous Systems, 1992, pp 227—235.

[223] Bäck, T., "Self adaptation in genetic algorithms" Proceedings of the 1st European Conference on Artificial Life: Toward a Practice of Autonomous Systems, 1992, pp 263—271.

[224] De Jong, K., "An Analysis of the Behavior of a Class of Genetic Adaptive Systems", Ph.D. dissertation, University of Michigan, Ann Arbor, MI, 1975.

[225] Goldberg, D. and Richardson, J., "Genetic algorithms with sharing for multimodal function optimization", Proceedings of the 2nd International Conference on Genetic Algorithms and their application, 1987, pp 41–49.

[226] Cobb, H. and Grefenstette, J., "Genetic algorithms for tracking changing environments", Proceedings of the 5th International Conference on Genetic Algorithms, 1993, pp 523—530.

[227] Coello, C., Pulido, G. and Lechuga, M., "Handling multiple objectives with particle swarm optimization", IEEE Transactions on Evolutionary Computation, Vol–8, No. 3, pp 256–279, 2004.

[228] Deb, K., Anand, A. and Joshi, D., "A computationally efficient evolutionary algorithm for real-parameter optimization", Evolutionary Computation, Vol–10, No. 4, pp 371–395, 2002.

[229] Shang, Y. and Qiu, Y., "A note on the extended Rosenbrock function", Evolutionary Computation, Vol–14, No. 1, pp 119–126, 2006.

[230] Qin, A. and Suganthan, P., "Self-adaptive differential evolution algorithm for numerical optimization", Proceedings of the IEEE Congress on Evolutionary Computation (CEC), 2005, pp 1785–1791.

[231] Liang, J. and Suganthan, P., "Dynamic multi-swarm particle swarm optimizer with local search", Proceedings of the IEEE Congress on Evolutionary Computation (CEC), 2005, pp 522–528.

[232] Jian, M., "Introducing recombination with dynamic linkage discovery to particle swarm optimization", Technical Report NCL-TR-2006006, Natural Computing Laboratory (NCLab), Department of Computer Science, National Chiao Tung University, 2006.

[233] Kirkpatrik, S. and Vecchi, M., "Optimization by Simulated Annealing", Science, Vol–220, No. 4598, pp 671–680, 1983.

[234] Auger, A. and Hansen, N., "A restart CMA evolution strategy with increasing population size", Proceedings of the IEEE Congress on Evolutionary Computation (CEC), 2005, pp 1769–1776.

[235] Rubya, S., Monir, S., Sharmin, M., Alam, M. S. and Rahman, M. S., "SS-ADC: Scatter search with adaptive diversity control", accepted in the 7th International Conference on Electrical and Computer Engineering (ICECE), Dhaka, Bangladesh, 20–22 December 2012.

[236] Herrera, F., Lozano, M. and Molina, D., "Continuous scatter search: An analysis of the integration of some combination methods and improvement strategies", European Journal of Operational Research, Vol–169, No. 2, pp 450–476, 2005.