An Analysis of Particle Swarm Optimizers

by

Frans van den Bergh

Submitted in partial fulfillment of the requirements for the degree Philosophiae Doctor

in the Faculty of Natural and Agricultural Science

University of Pretoria

Pretoria

November 2001

An Analysis of Particle Swarm Optimizers

by

Frans van den Bergh

Abstract

Many scientific, engineering and economic problems involve the optimisation of a set of parameters. These problems include examples like minimising the losses in a power grid by finding the optimal configuration of the components, or training a neural network to recognise images of people's faces. Numerous optimisation algorithms have been proposed to solve these problems, with varying degrees of success. The Particle Swarm Optimiser (PSO) is a relatively new technique that has been empirically shown to perform well on many of these optimisation problems. This thesis presents a theoretical model that can be used to describe the long-term behaviour of the algorithm. An enhanced version of the Particle Swarm Optimiser is constructed and shown to have guaranteed convergence on local minima. This algorithm is extended further, resulting in an algorithm with guaranteed convergence on global minima. A model for constructing cooperative PSO algorithms is developed, resulting in the introduction of two new PSO-based algorithms. Empirical results are presented to support the theoretical properties predicted by the various models, using synthetic benchmark functions to investigate specific properties. The various PSO-based algorithms are then applied to the task of training neural networks, corroborating the results obtained on the synthetic benchmark functions.

Thesis supervisor: Prof. A. P. Engelbrecht
Department of Computer Science
Degree: Philosophiae Doctor

An Analysis of Particle Swarm Optimizers

deur

Frans van den Bergh

Opsomming

Talle wetenskaplike, ingenieurs en ekonomiese probleme behels die optimering van 'n aantal parameters. Hierdie probleme sluit byvoorbeeld in die minimering van verliese in 'n kragnetwerk deur die optimale konfigurasie van die komponente te bepaal, of om neurale netwerke af te rig om mense se gesigte te herken. 'n Menigte optimeringsalgoritmes is al voorgestel om hierdie probleme op te los, soms met gemengde resultate. Die Partikel Swerm Optimeerder (PSO) is 'n relatief nuwe tegniek wat verskeie van hierdie optimeringsprobleme suksesvol opgelos het, met empiriese resultate ter ondersteuning. Hierdie tesis stel bekend 'n teoretiese model wat gebruik kan word om die langtermyn gedrag van die PSO algoritme te beskryf. 'n Verbeterde PSO algoritme, met gewaarborgde konvergensie na lokale minima, word aangebied met die hulp van dié teoretiese model. Hierdie algoritme word dan verder uitgebrei om globale minima te kan opspoor, weereens met 'n teoreties-bewysbare waarborg. 'n Model word voorgestel waarmee koöperatiewe PSO algoritmes ontwikkel kan word, wat gevolglik gebruik word om twee nuwe PSO-gebaseerde algoritmes mee te ontwerp. Empiriese resultate word aangebied om die teoretiese kenmerke, soos voorspel deur die teoretiese model, toe te lig. Kunsmatige toetsfunksies word gebruik om spesifieke eienskappe van die verskeie algoritmes te ondersoek. Die verskeie PSO-gebaseerde algoritmes word dan gebruik om neurale netwerke mee af te rig, as 'n kontrole vir die empiriese resultate wat met die kunsmatige funksies bekom is.

Tesis studieleier: Prof. A. P. Engelbrecht
Departement Rekenaarwetenskap
Graad: Philosophiae Doctor

Acknowledgements

I would like to thank the following people for their assistance during the production of this thesis:

- Professor A.P. Engelbrecht, my thesis supervisor, for his insight and motivation;
- Edwin Peer, Gavin Potgieter, Andrew du Toit, Andrew Cooks and Jacques van Greunen, UP Techteam members, for maintaining the computer infrastructure used to perform my research;
- Professor D.G. Kourie (UP), for providing valuable insight into some of the mathematical proofs;
- Nic Roets (Sigma Solutions), for showing me a better technique to solve recurrence relations;

I would also like to thank all the people who listened patiently when I discussed some of my ideas with them, for their feedback and insight.

‘Would you tell me, please, which way I ought to go from here?’

‘That depends a good deal on where you want to get to,’ said the Cat.

‘I don’t much care where—’ said Alice.

‘Then it doesn’t matter which way you go,’ said the Cat.

— *Alice’s Adventures in Wonderland*, by Lewis Carroll (1865)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

You awaken to the sound of your alarm clock. A clock that was manufactured by a company that tried to maximise its profit by looking for the optimal allocation of the resources under its control. You turn on the kettle to make some coffee, without thinking about the great lengths that the power company went to in order to optimise the delivery of your electricity. Thousands of variables in the power network were configured to minimise the losses in the network in an attempt to maximise the profit of your electricity provider. You climb into your car and start the engine without appreciating the complexity of this small miracle of engineering. Thousands of parameters were fine-tuned by the manufacturer to deliver a vehicle that would live up to your expectations, ranging from the aesthetic appeal of the bodywork to the specially shaped side-mirror cowls, designed to minimise drag. As you hit the gridlock traffic, you think "Couldn't the city planners have optimised the road layout so that I could get to work in under an hour?"

Optimisation forms an important part of our day-to-day life. Many scientific, social, economic and engineering problems have parameters that can be adjusted to produce a more desirable outcome.

Over the years numerous techniques have been developed to solve such optimisation problems. This thesis investigates the behaviour of a relatively new technique known as Particle Swarm Optimisation, a technique that solves problems by simulating swarm

behaviour.

1.1 Motivation

It is clear that there will always be a need for better optimisation algorithms, since the complexity of the problems that we attempt to solve is ever increasing. The Particle Swarm Optimiser was introduced in 1995 [38, 70], yet very few formal analyses of the behaviour of the algorithm have been published. Most of the published work was concerned with empirical results obtained by changing some aspect of the original algorithm.

Without a formal model of why the algorithm works, it was impossible to determine what the behaviour of the algorithm would be in the general case. If the algorithm has been shown to be able to solve 10 difficult optimisation problems, what could be said about the infinite number of problems that have not yet been studied empirically?

While the results obtained from empirical comparisons provided useful insights into the nature of the PSO algorithm, it was clear that a general, theoretical description of the behaviour of the algorithm was needed. This thesis constructs such a model, which is subsequently used to analyse the convergence behaviour of the PSO algorithm.

Several new PSO-based algorithms were subsequently developed, with the aid of the theoretical model of the PSO algorithm. These algorithms were constructed to address specific weaknesses of the PSO algorithm that only became apparent once the theoretical convergence behaviour of the PSO was understood.

1.2 Objectives

The primary objectives of this thesis can be summarised as follows:

- To develop a theoretical model for the convergence behaviour of the Particle Swarm Optimisation algorithm, and the various derived algorithms introduced in this thesis.

- To extend the PSO algorithm so that it becomes a global optimisation technique with guaranteed convergence on global optima.

- To develop and test cooperative Particle Swarm Optimisation algorithms, based on models that have proven to be successful when applied to other evolutionary algorithms.
- To obtain empirical results to support the predictions offered by the theoretical models.
- To investigate the application of various PSO-based algorithms to the task of training summation and product unit neural networks.

1.3 Methodology

The theoretical models developed in this thesis are used to characterise the behaviour of all the newly introduced algorithms. Each new algorithm is theoretically analysed to show whether it is guaranteed to converge on either a local or global minimum, depending on whether the algorithm is a local or global search algorithm, respectively.

Empirical results were obtained using various synthetic benchmark functions with well-known characteristics. These results are used as supporting evidence for the theoretical convergence characteristics of the various algorithms. Owing to the stochastic nature of all these algorithms, it is not always possible to directly observe the characteristics predicted by the theoretical model, *i.e.* a stochastic global optimisation algorithm may require an infinite number of iterations to guarantee that it will find the global minimiser. Therefore the probability of observing this algorithm locate a global minimiser in a finite number of iterations is very small. Despite this problem, it is still possible to see whether the algorithm is still making progress toward its goal, or whether it has become trapped in a local minimum.

The results of two Genetic Algorithm-based optimisation techniques are also reported for the same synthetic benchmark functions. These results provide some idea of the relative performance of the PSO-based techniques when compared to other stochastic, population-based algorithms.

A second set of experiments were performed on a real-world problem to act as a control for the results obtained on the synthetic functions. The task of training both

summation and product unit neural networks was selected as an example of a real-world optimisation problem. On these problems the results of the PSO-based algorithms were compared to that of the GA-based algorithms, as well as that of two efficient gradient-based algorithms.

1.4 Contributions

The main contributions of this thesis are:

- A theoretical analysis of the behaviour of the PSO under different parameter settings. This analysis led to the development of a model that can be used to predict the long-term behaviour of a specific set of parameters, so that these parameters can be classified as leading to convergent or divergent particle trajectories.
- The discovery that the original PSO algorithm is not guaranteed to converge on a local (or global) minimiser. An extension to the existing PSO algorithm is presented that enables the development of a formal proof of guaranteed local convergence.
- The development of a technique for extending the PSO algorithm so that it is guaranteed to be able to locate the global minimiser of the objective function, together with a formal proof of this property.
- The application of existing cooperative models to the PSO algorithm, leading to two new PSO-based algorithms. These new algorithms offer a significant improvement in performance on multi-modal functions. The existing cooperation model is then extended to produce a new type of cooperative algorithm that does not suffer from the same weaknesses as the original model.

1.5 Thesis Outline

Chapter 2 starts with an introduction to the theory of optimisation, followed by a brief review of existing evolutionary techniques for solving optimisation problems. This is

followed by a description of the Particle Swarm Optimiser, including a discussion of the numerous published modifications to the PSO algorithm. The focus then shifts slightly to the topic of coevolutionary algorithms, since these methods form the basis of the work presented in Chapter 4.

Chapter 3 presents a theoretical analysis of the behaviour of the PSO algorithm, including formal proofs of convergence for the various new PSO-based algorithms introduced there.

Several cooperative PSO algorithms, based on the models discussed in Chapter 2, are introduced in Chapter 4. The convergence properties of these cooperative algorithms are investigated, with formal proofs where applicable.

Chapter 5 presents an empirical analysis of the behaviour of the various PSO-based algorithms introduced in Chapters 3 and 4, applied to minimisation tasks involving synthetic benchmark functions. These synthetic functions allow specific aspects of PSO behaviour to be tested.

In Chapter 6, the same PSO-based algorithms are used to train summation and product unit networks. These results are presented to show that the new algorithms introduced in this thesis have similar performance on both real-world and synthetic minimisation tasks.

Chapter 7 presents a summary of the findings of this thesis. Some topics for future research are also discussed.

The appendices present, in order, a glossary of terms, a definition of frequently used symbols, a derivation of the closed-form PSO equations, a set of 3D-plots of the synthetic benchmark functions used in Chapter 5, a description of the gradient-based algorithms used in Chapter 6 and a list of publications derived from the work presented in this thesis.

Chapter 2

Background & Literature Study

This chapter reviews some of the basic definitions related to optimisation. A brief discussion of Evolutionary Algorithms and Genetic Algorithms is presented. The origins of the Particle Swarm optimiser are then discussed, followed by an overview of the various published modifications to the basic PSO algorithm. Next an introduction to coevolutionary and cooperative algorithms is presented, followed by a brief overview of the important issues that arise when cooperative algorithms are implemented.

2.1 Optimisation

The task of optimisation is that of determining the values of a set of parameters so that some measure of optimality is satisfied, subject to certain constraints. This task is of great importance to many professions, for example, physicists, chemists and engineers are interested in design optimisation when designing a chemical plant to maximise production, subject to certain constraints, *e.g.* cost and pollution. Scientists require optimisation techniques when performing non-linear curve or model fitting. Economists and operation researchers have to consider the optimal allocation of resources in industrial and social settings. Some of these problems involve only linear models, resulting in *linear optimisation* problems, for which an efficient technique known as linear programming [58] exists. The other problems are known as *non-linear optimisation* problems, which are generally very difficult to solve. These problems are the focus of the work

presented in this thesis.

The term optimisation refers to both minimisation and maximisation tasks. A task involving the maximisation of the function f is equivalent to the task of minimising $-f$, therefore the terms minimisation, maximisation and optimisation are used interchangeably.

This thesis deals mostly with *unconstrained minimisation* tasks, formally defined as

$$\begin{aligned} &\text{Given } f : \mathbb{R}^n \rightarrow \mathbb{R} \\ &\text{find } \mathbf{x}^* \in \mathbb{R}^n \text{ for which } f(\mathbf{x}^*) \leq f(\mathbf{x}),\ \forall \mathbf{x} \in \mathbb{R}^n \end{aligned} \tag{2.1}$$

Some problems require that some of the parameters satisfy certain constraints, *e.g.* all the parameters must be non-negative. These types of problems are known as *constrained minimisation* tasks. They are typically harder to solve than their equivalent unconstrained versions, and are not dealt with explicitly here.

Another class of optimisation problems are known as *least-squares* problems, which are of the form

$$\begin{aligned} &\text{Given } \mathbf{r} : \mathbb{R}^n \rightarrow \mathbb{R}^m, \quad n < m \\ &\text{find } \mathbf{x}^* \in \mathbb{R}^n \text{ for which } \sum_{i=1}^{m} (r_i(\mathbf{x}))^2 \text{ is minimised.} \end{aligned} \tag{2.2}$$

These optimisation problems present themselves when there are more non-linear requirements than there are degrees of freedom. Note that the least-squared problem can be solved using the same approach as used in solving (2.1), by defining

$$f(\mathbf{x}) = \sum_{i=1}^{m} (r_i(\mathbf{x}))^2$$

and minimising f. Neural Network training is sometimes solved as such a non-linear least-squares problem (see Chapter 6 for more details).

Techniques used to solve the minimisation problems defined above can be placed into two categories: Local and Global optimisation algorithms.

2.1.1 Local Optimisation

A *local minimiser*, $\mathbf{x}^*_B$, of the region B, is defined so that

$$f(\mathbf{x}^*_B) \leq f(\mathbf{x}),\ \forall \mathbf{x} \in B \tag{2.3}$$

where $B \subset S \subseteq \mathbb{R}^n$, and S denotes the search space. Note that $S = \mathbb{R}^n$ when dealing with unconstrained problems. More importantly, note that B is a proper subset of S. A given search space S can contain multiple regions B_i such that $B_i \cap B_j = \emptyset$ when $i \neq j$. It then follows that $\mathbf{x}^*_{B_i} \neq \mathbf{x}^*_{B_j}$, so that the minimiser of each region B_i is unique. Any of the $\mathbf{x}^*_{B_i}$ can be considered a minimiser of B, although they are merely local minimisers. There is no restriction on the value that the function can assume in the minimiser, so that $f(\mathbf{x}^*_{B_i}) = f(\mathbf{x}^*_{B_j})$ is allowed. The value $f(\mathbf{x}^*_{B_i})$ will be called the *local minimum*.

Most optimisation algorithms require a starting point $\mathbf{z}_0 \in S$. A local optimisation algorithm should guarantee that it will be able to find the minimiser $\mathbf{x}^*_B$ of the set B if $\mathbf{z}_0 \in B$. Some algorithms satisfy a slightly weaker constraint, namely that they guarantee to find a minimiser $\mathbf{x}^*_{B_i}$ of some set B_i, not necessarily the one closest to $\mathbf{z}_0$.

Many local optimisation algorithms have been proposed. A distinction will be made between deterministic, analytical algorithms and the stochastic algorithms discussed in Sections 2.2–2.4. The deterministic local optimisation[1] algorithms include simple Newton-Raphson algorithms, through Steepest Descent [11] and its many variants, including the Scaled Conjugate Gradient algorithm (SCG) [87] and the quasi-Newton [11, 30] family of algorithms. Some of the better known algorithms include Fletcher-Reeves (FR), Polar-Ribiere (PR), Davidon-Fletcher-Powell (DFP), Broyden-Fletcher-Goldfarb-Shanno (BFGS) [104, 11]. There's even an algorithm that was designed specifically for solving least-squares problems, known as the Levenberg-Marquardt (LM) algorithm [11].

2.1.2 Global Optimisation

The *global minimiser*, $\mathbf{x}^*$, is defined so that

$$f(\mathbf{x}^*) \leq f(\mathbf{x}), \ \forall \mathbf{x} \in S \tag{2.4}$$

where S is the search space. For unconstrained problems it is common to choose $S = \mathbb{R}^n$, where n is the dimension of $\mathbf{x}$. Throughout this thesis the term *global optimisation* will refer strictly to the process of finding $\mathbf{x}^*$ as defined in (2.4). The term *global minimum* will refer to the value $f(\mathbf{x}^*)$, and $\mathbf{x}^*$ will be called the *global minimiser*. A global optimisation

[1] see Section 2.1.2 for an explanation as to why these algorithms are classified as local methods here.

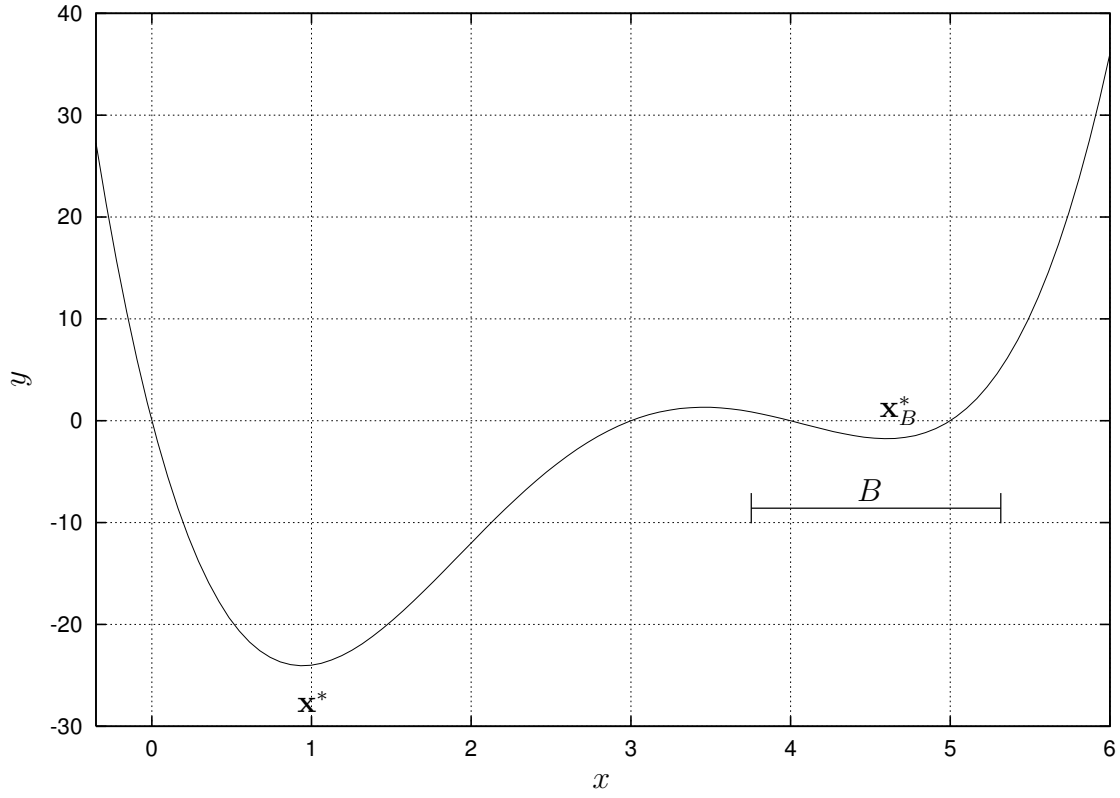


Figure 2.1: The function $f(x) = x^4 - 12x^3 + 47x^2 - 60x$, indicating the global minimiser $\mathbf{x}^*$, as well as a local minimiser $\mathbf{x}^*_B$.

algorithm, like the local optimisation algorithms described above, also starts by choosing an initial starting position $\mathbf{z}_0 \in S$.

Contrary to the definition above in (2.4), some texts (*e.g.* [30]) define a global optimisation algorithm differently, namely an algorithm that is able to find a (local) minimiser of $B \subset S$, regardless of the actual position of $\mathbf{z}_0$. These algorithms consist of two processes: "global" steps and "local" steps. Their local steps are usually the application of a local minimisation algorithm, and their "global" steps are designed to ensure that the algorithm will move into a region B_i, from where the "local" step will be able to find the minimiser of B_i. These methods will be referred to as *globally convergent* algorithms, meaning that they are able to converge to a local minimiser regardless of their starting position $\mathbf{z}_0$. These methods are also capable of finding the global minimiser, given that the starting position $\mathbf{z}_0$ is chosen correctly. There is no known reliable, general way of doing this, though.

Figure 2.1 illustrates the difference between the local minimiser $\mathbf{x}_B^*$ and the global minimiser $\mathbf{x}^*$. A true global optimisation algorithm will find $\mathbf{x}^*$ regardless of the choice of starting position $\mathbf{z}_0$. Dixon and Szegø have edited two collections of papers on the topic of true global optimisation algorithms [31, 32]. The topic of global optimisation algorithms will be revisited in Chapter 3.

2.1.3 No Free Lunch Theorem

One of the more interesting developments in optimisation theory was the publication of the "No Free Lunch" (NFL) theorem by Wolpert and Macready [144, 145]. This theorem states that the performance of all optimisation (search) algorithms, amortised over the set of all possible functions, is equivalent.

The implications of this theorem are far reaching, since it implies that no algorithm can be designed so that it will be superior to a linear enumeration of the search space, or even a purely random search. The theorem is only defined over finite search spaces, however, and it is as yet not clear whether the result applies to infinite search spaces. All computer implementations of search algorithms will effectively operate on finite search spaces, though, so the theorem is directly applicable to all existing algorithms.

Although the NFL theorem states that all algorithms perform equally well over the

set of *all* functions, it does not necessarily hold for all subsets of this set. The set of all functions over a finite domain includes the set of all the permutations of this domain. Many of these functions do not have compact descriptions, so that they appear to be largely "random". Most real-world functions, however, have some structure, and usually have compact descriptions. These types of functions form a rather small subset of the set of all functions. This concern lead to the development of sharpened versions of the NFL [117], showing that it holds for much smaller subsets than initially believed.

A more constructive approach is to try and characterise the set of functions over which the NFL does *not* hold. Christensen *et al.*proposed a definition of a "searchable" function [18], as well as a general algorithm that provably performs better than random search on this set of searchable functions.

This thesis will side with the latter approach, assuming that it is possible to design algorithms that perform, on average, better than others (*e.g.* random search) *over a limited subset* of the set of all functions. No further attempt will be made to characterise this subset. Instead, empirical results will be used to show that real-world applications can benefit from improved algorithms.

2.2 Evolutionary Computation

Evolutionary Computation (EC) defines a number of methods designed to simulate evolution. These methods are all population-based, and rely on a combination of random variation and selection to solve problems. Several different approaches exist within the field, including Evolutionary Algorithms (EAs), Evolution Strategies (ES), Evolutionary Programming (EP), Genetic Algorithms (GAs) and Genetic Programming (GP) [9, 8].

Although the ancient Greeks had some rudimentary grasp of the theory of evolution, it was Charles Darwin who first popularised the modern theory of evolution. Credit for this discovery is shared with Alfred Russel Wallace, who independently developed the same theory concurrently. Although their original work was presented simultaneously at a meeting of the Linnean Society of London in 1858, it was Darwin's book [26] that immortalised his name. The fundamental principle underlying evolution is one of optimisation, where the goal is survival of the species. This does not mean, however,

that EC methods can only be applied to optimisation problems. The EC paradigms named above helped solve problems that were previously considered computationally intractable. Several (somewhat overlapping) categories have been identified to which EC has successfully been applied [9]:

- planning
- design
- simulation and identification
- control
- classification.

The evolutionary process

The processes responsible for driving the evolutionary process include reproduction, mutation, competition and selection. Reproduction is effected through the transfer of an individual's genetic program to its progeny. This way, genetic traits that resulted in a successful organism are preserved. The transfer of the genetic program is, however, subject to error. These errors, called mutations, may either improve or impede the resulting organism. Competition results when the resources in the environment are limited — the population of organisms cannot expand without bound, thus an organism must strive to be better than its competitors in order to survive. In the presence of competition, the replication process leads to selection, so that the more successful individuals survive and produce offspring; the less successful ones face extinction.

The genetic program referred to above is called the *genotype* of the population. The genotype carries the genetic information that is passed from parent to offspring, representing the experiential evidence gathered by the parent. The population possesses a second, related set of properties called its *phenotype*. The phenotype is the behavioural expression of the genotype in a specific environment. The mapping between the genotype and the phenotype can be quite complex, owing to the influence of *pleiotropy* and *polygeny* [83]. Pleiotropy is when the random modification of one piece of information in the genotype can lead to unexpected variation in the phenotype, affecting more than

one phenotypic trait. Polygeny is observed when several genes (pieces of genotypic information) interact to produce a specific phenotypic trait. Thus, to change the phenotypic behaviour, all the relevant genes must be modified. Natural evolutionary systems have no one-to-one mappings between genotype and phenotype; the mapping is a non-linear function defining the interaction between the genes and the environment.

Several different paradigms to model evolutionary processes on digital computers have been proposed. A general framework for Evolutionary Algorithms, introduced by Bäck [7], will be presented next. Two algorithms belonging to this class, Evolutionary Programming and Evolution Strategies will be discussed in terms of this framework; the discussion of Genetic Algorithms is deferred until Section 2.3.

2.2.1 Evolutionary Algorithms

The term Evolutionary Algorithm refers to a family of algorithms that can all be described in terms of a general evolutionary framework. The exact form of the operators, as well as the relationship between the sizes of parent and offspring populations, define the specific instance of EA, *e.g.* EP, ES or GA. Genetic Programming is treated as a specialised GA.

Consider a population of μ individuals, $P(t) = (\mathbf{x}_1(t), \mathbf{x}_2(t), \ldots, \mathbf{x}_\mu(t))$ at time t, where each $\mathbf{x}_i \in S$ represents a potential solution (in the search space S) to the problem at hand. Let $f(\mathbf{x})$ be a function that determines the quality of a solution, called the *fitness function*. The fitness of the whole population can thus be expressed as $F(t) = (f(\mathbf{x}_1(t)), f(\mathbf{x}_2(t)), \ldots, f(\mathbf{x}_\mu(t)))$. Given arbitrary parameters μ, λ, Θ_r, Θ_m and Θ_s, the general EA framework (adapted from [9]) is shown in Figure 2.2. The parameters Θ_r, Θ_m and Θ_s are the probabilities of applying the operators to which they correspond, which are the recombination, mutation and selection operators, respectively.

The parameter μ is the size of the parent population; $\mu + \lambda$ denotes the total population size (parents plus offspring) after the recombination and mutation operators have been applied. The selection operator pares down the resultant population $P''(t)$ to the size of the parent population μ, according to some metric that can be controlled with the parameter Θ_s.

Popular choices for the selection operator are based on variants of *tournament se-*

$t \leftarrow 0$
$P(t) \leftarrow \text{initialise}(\mu)$
$F(t) \leftarrow \text{evaluate}(P(t), \mu)$
repeat:
  $P'(t) \leftarrow \text{recombine}(P(t), \Theta_r)$
  $P''(t) \leftarrow \text{mutate}(P''(t), \Theta_m)$
  $F(t) \leftarrow \text{evaluate}(P''(t), \lambda)$
  $P(t+1) \leftarrow \text{select}(P''(t), F(t), \mu, \Theta_s)$
  $t \leftarrow t + 1$
until stopping criterion is met

Figure 2.2: General framework for describing EAs

lection. For a given value, q, this operator compares the fitness of the member under consideration to that of q other population members. The individual scores one point whenever it possesses an equal or better fitness value than the other population member it is compared with. The population members can then be ranked according to how frequently they "win" the other members they were compared with. If $q = \lambda$, each member is compared with every other member, resulting in unique ordering. This results in very strong selection pressure, so that only the top μ members survive into the next generation. If $q = 1$, selection pressure is weak, leading to slower convergence. The ordering is no longer unique, since the rank of an individual depends on which other population member it was compared with. Whenever the value of q is less than λ the tournament selection process produces non-deterministic results, and is called a *probabilistic selection* operator.

The general framework allows either the recombination or the mutation operators to be identity operators. The purpose of the recombination operator is to take several (typically only two) elements from the parent population and to produce several offspring by combining (or mixing) their genotypic traits, and is controlled by the parameter Θ_r. The mutation operator takes an element from the population and produces offspring (usually only one) by perturbing the genotypic traits of its parent, subject to the value of the control parameter Θ_m.

The values μ, λ, Θ_r, Θ_m and Θ_s are called *strategy parameters*. Many implementations keep the values of these parameters fixed during a run, but they can be adapted dynamically, using a second EA if desired, or even by concatenating them to the search space parameters.

2.2.2 Evolutionary Programming (EP)

Evolutionary Programming was devised by L. J. Fogel in the context of evolving finite state-machines to be used in the prediction of time series [45, 46]. Fogel's original algorithm did not use recombination; it relied on mutation exclusively. The following mutation operators were used: change an output symbol, change a state transition, add a state, delete a state or change the initial state. Mutations were applied randomly using a uniform probability distribution. His implementation produced one offspring per parent in every iteration, so that $\lambda = 2\mu$ in the notation introduced above in Section 2.2.1. The selection operator discarded all the solutions with a fitness value below the median of the combined population $P''(t)$, so that only the best μ members were retained.

Later Evolutionary Programs were extended to include more general representations, including ordered lists (to solve the Traveling Salesman Problem) and real-valued vectors for continuous function optimisation [47].

Modern EPs are characterised as EAs without recombination, thus relying exclusively on mutation and selection. When EPs are applied to real-valued optimisation problems, they use normally-distributed mutations and usually evolve their strategy parameters concurrently. The selection operator is probabilistic, in contrast with Fogel's original implementation.

2.2.3 Evolution Strategies (ES)

Evolution Strategies, devised by Rechenberg [108, 109] and Schwefel [118, 119], are usually applied to real-valued optimisation problems. ES programs make use of both mutation and recombination, searching both the search space and the strategy parameter space simultaneously.

The parent and offspring population sizes usually differ, with the offspring population

at least as large as the parent population. Two different schemes are often encountered: the *comma* strategy and the *plus* strategy. The comma strategy is identified by the notation (μ, λ), meaning that μ parents are used to generate λ offspring. Of the λ offspring, μ of them will be selected to become the parent population in the next iteration of the algorithm. The implication of this strategy is that good solutions from the previous generation may be lost, since the original parent population is not preserved. On the other hand, this approach increases the diversity of the population. The plus strategy is denoted by $(\mu + \lambda)$. This strategy concatenates the μ parents and the λ offspring into a single large population. Selection is performed on this combined population, selecting as the parent population of the next iteration the best solutions found in both μ and λ. This method preserves the best solutions discovered so far, so that the fitness of the best individual of the population is a monotonic function.

The relationship between the parent population and the offspring population, especially in the case of the $(\mu + \lambda)$ strategy, makes selection deterministic.

2.3 Genetic Algorithms (GAs)

The Genetic Algorithm (GA), originally described by Holland [62, 63] (then called adaptive or reproductive plans), is another instance of an Evolutionary Algorithm. The emphasis of the GA is usually on recombination, with mutation treated as a 'background operator'. Only a brief overview of the Genetic Algorithm will be presented here; the reader is referred to [51] for an in-depth treatment of the subject.

The canonical GA makes use of a binary format to represent the genotypes. Using a mapping function, the genotype is converted into the equivalent phenotype, which is an element of the search space. A simple example will help to illustrate this process. Assume that the GA is to locate the minimum of the function $f(x) = x^2 - 10x + 25$. It is known that the minimiser is located in the interval $[0, 10)$. Assume that a 16-bit representation is used to represent the value of x, so that the genotypes of the elements in the population are 16-bit strings. Given such a 16-bit representation, b, the equivalent

phenotype can be obtained using

$$x = 10 \times \frac{1}{2^{16}} \sum_{i=0}^{15} 2^i b_i$$

where b_i denotes the value of bit i in the bit string b. Note that the factor 10 is to scale the value from its initial range of $[0, 1)$ to the range $[0, 10)$. The minimiser of this function is 5, so the genotypic representation, using 16 bits, is

$$1000000000000000$$

Although the minimiser has an exact representation in this example, this is not always the case. Using 16 bits to represent a value in the range $[0, 10)$ results in a granularity of 1.5258×10^{-4}, which may or may not be sufficient, depending on the particular problem. By increasing the number of bits in the genotype greater accuracy can be obtained, at the cost of increasing the time needed to perform the genotype-to-phenotype mapping, as well as increasing the effective search space that the GA has to examine.

Genotypic Representations

The notion of using separate genotypic and phenotypic representations used to be one of the defining differences between GAs and other types of evolutionary algorithms. This distinction has blurred somewhat with the advent of non-binary coded GAs. For example, a GA could use ordinary real-valued numbers to represent population members [42], using arithmetic crossover [29] rather than binary crossover. Other possible representations include permutations [27, 52] and tree-based representations, usually encountered in Genetic Programming (GP) [74]. The disadvantage of these non-binary coded GAs is that they require different recombination and mutation operators for each representation, whereas a single pair of recombination and mutation operators are sufficient for any binary coded problem.

When using binary coded strings to represent integers (which can be mapped to real values), there is a choice between either 'plain' binary encoding (as used in the example above), or Gray-coded values. Consider the example problem above, where the minimiser of the function had a binary value of $1000000000000000 = 2^{15}$. The next larger value in

this representation is $1000000000000001 = 2^{15} + 1$, differing only in the least significant bit. The *Hamming distance* between two binary strings is defined as the number of bits in which they disagree, so that these two strings have a Hamming distance of exactly one. The value directly preceding the minimiser has the encoding $0111111111111111 = 2^{15} - 1$. Note that this value is the complement of the minimiser itself, differing in *all* its bits, thus at a Hamming distance of 16 from the minimiser. This discontinuity in the genotypic representation is called a *Hamming cliff* [116], and may impede the progress of the GA because of the way in which the recombination and mutation operators function. Gray codes solve this problem by using a representation where any two consecutive values differ in exactly one bit, resulting in a Hamming distance of one. Although this encoding solves the problem associated with Hamming cliffs, it introduces an artificial nonlinearity in the relationship between the string representation and the decoded value.

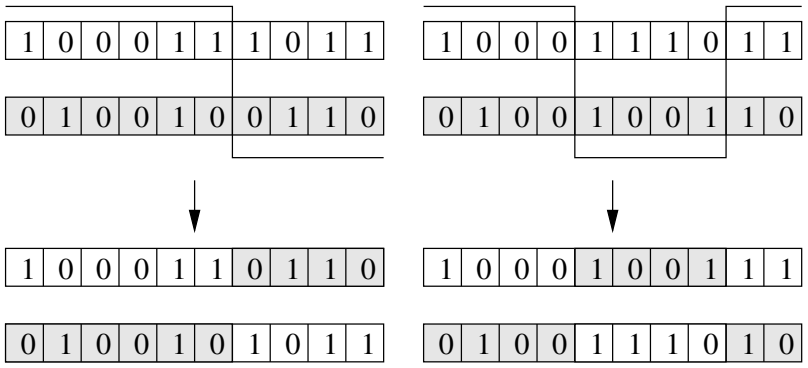
The following sections will discuss the recombination and mutation operators in more detail.

Recombination

Two popular recombination operators applied to binary-coded representations are the one- and two-point crossover operators. Two parents are selected for recombination, and segments of their bit strings are exchanged between the two parents to form the two offspring. The *one-point crossover* proceeds by picking a locus randomly in the bit string and exchanging all the bits after that locus. *Two-point crossover* is slightly more conservative, picking two random loci to demarcate the boundaries of the exchange, resulting on average in a smaller segment than that produced by single-point crossover. These two operators are graphically illustrated in Figure 2.3. *Uniform crossover* is yet another bit-string based operator. Every bit in the one parent string is exchanged with the corresponding bit in the other parent subject to some probability, usually set to 0.5. This operator is considered to be more disruptive than the two-point crossover operator.

Another popular recombination operator is called *arithmetic crossover*. This operator is used when dealing with real-valued GAs. Let $\mathbf{x}_a(t)$ and $\mathbf{x}_b(t)$ denote the two parents. Then the two offspring are obtained using

$$\mathbf{x}_a(t+1) = r_1\mathbf{x}_a(t) + (1.0 - r_1)\mathbf{x}_b(t)$$



(a) One-point crossover (b) Two-point crossover

Figure 2.3: Example recombination operators

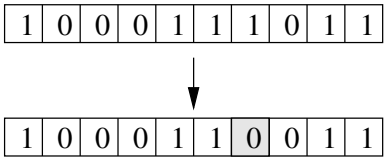


Figure 2.4: A simple mutation operator

$$\mathbf{x}_b(t+1) \quad = \quad r_1\mathbf{x}_b(t) + (1.0 - r_1)\mathbf{x}_a(t)$$

where $r_1 \sim U(0,1)$ is a uniform random variate.

Mutation

The mutation operator used on bit string representations is particularly simple: Just invert the value of each bit in the string subject to some small probability, called the *mutation rate*. The goal of the mutation operator is to introduce some diversity into the population, thereby extending the effective area of the search space that the algorithm considers. This process is illustrated in Figure 2.4. A high mutation rate may destabilise the population by disrupting the existing good solutions. Since GAs usually rely on their recombination operators, the mutation rate is usually set quite low. A good value for the mutation rate is the reciprocal of the string length in bits, so that a 10-bit representation should have a mutation rate of 0.1. Alternate strategies include starting

with a high mutation rate, and gradually decreasing it over time.

Selection

Many of the selection operators used in Evolution Strategies or Evolutionary Programming can be used successfully on Genetic Algorithms as well. Most GAs use a selection operator that selects the next generation only from the offspring population. This can clearly be seen by the way that the recombination operator functions, since it completely replaces the parents with the offspring. A GA using this approach is called a *generational genetic algorithm*, since none of the parents from the previous generation are explicitly preserved. This technique usually increases the diversity of the population, and helps to prevent premature convergence onto a local minimum. Naturally, this approach slows down the rate of convergence somewhat, since potentially good solutions from the current generation may not survive into the next generation.

An alternative is the so called *elitist strategy*. A copy of the parent population is made before applying the recombination and mutation operators. The algorithm then selects the next generation, ranked by fitness, from both the parent and the offspring populations, similar to the $(\mu + \lambda)$ mechanism found in Evolution Strategies. Variations on this theme include limiting the size of the parent population participating in this process to k, so that at most k parents are preserved.

A common way of implementing a generational GA is to use *fitness-proportionate selection*. The probability of including an individual i in the next generation, denoted by $P_i(t+1)$, is computed using

$$P_i(t+1) = \frac{f_i(t)}{\frac{1}{\mu}\sum_{j=1}^{\mu} f_j(t)}$$

where μ denotes the population size and $f_i(t)$ the fitness of the individual i. Clearly this method assigns a higher probability to individuals with higher fitness values, so that the probability of a highly fit individual appearing in the next generation is quite good. Note that due to the possibility of having both highly fit and highly unfit members in the population, the individual fitness values $f_i(t)$ are scaled so that the effective fitness values are in the range $[0, 1]$.

After the fitness-proportionate selection method has been used to make a fitness-biased copy of the previous generation, the recombination and mutation operators can be applied. This is usually implemented by selecting the elements from the copied population in pairs (based on their index numbers) and applying the recombination operator, subject to some probability called the *crossover probability*. The mutation operator is then probabilistically applied to the result.

This concludes the brief overview of genetic algorithms.

2.4 Particle Swarm Optimisers

The Particle Swarm Optimiser (PSO) is a population-based optimisation method first proposed by Kennedy and Eberhart [70, 38]. Some of the attractive features of the PSO include the ease of implementation and the fact that no gradient information is required. It can be used to solve a wide array of different optimisation problems, including most of the problems that can be solved using Genetic Algorithms; some example applications include neural network training [41, 135, 136, 34] and function minimization [121, 124].

Many popular optimisation algorithms are deterministic, like the gradient-based algorithms mentioned in Section 2.1.1. The PSO, similarly to the algorithms belonging to the Evolutionary Algorithm family, is a stochastic algorithm that does not need gradient information derived from the error function. This allows the PSO to be used on functions where the gradient is either unavailable or computationally expensive to obtain.

2.4.1 The PSO Algorithm

The origins of the PSO are best described as sociologically inspired, since the original algorithm was based on the sociological behaviour associated with bird flocking [70]. This topic will be discussed in more detail below after the basic algorithm has been described.

The algorithm maintains a population of particles, where each particle represents a potential solution to an optimisation problem. Let s be the size of the swarm. Each particle i can be represented as an object with several characteristics. These characteristics are assigned the following symbols:

$\mathbf{x}_i$: The *current position* of the particle;

$\mathbf{v}_i$: The *current velocity* of the particle;

$\mathbf{y}_i$: The *personal best position* of the particle.

The personal best position associated with particle i is the best position that the particle has visited (a previous value of $\mathbf{x}_i$), yielding the highest fitness value for that particle. For a minimisation task, a position yielding a smaller function value is regarded as having a higher fitness. The symbol f will be used to denote the objective function that is being minimised. The update equation for the personal best position is presented in equation (2.5), with the dependence on the time step t made explicit.

$$\mathbf{y}_i(t+1) = \begin{cases} \mathbf{y}_i(t) & \text{if } f(\mathbf{x}_i(t+1)) \geq f(\mathbf{y}_i(t)) \\ \mathbf{x}_i(t+1) & \text{if } f(\mathbf{x}_i(t+1)) < f(\mathbf{y}_i(t)) \end{cases} \tag{2.5}$$

Two versions of the PSO exist, called the *gbest* and *lbest* models [37]. The difference between the two algorithms is based on the set of particles with which a given particle will interact with directly, where the symbol $\hat{\mathbf{y}}$ will be used to represent this interaction. The details of the two models will be discussed in full below. The definition of $\hat{\mathbf{y}}$, as used in the *gbest* model, is presented in equation (2.6).

$$\begin{aligned} \hat{\mathbf{y}}(t) \in \{\mathbf{y}_0(t), \mathbf{y}_1(t), \ldots, \mathbf{y}_s(t)\} \mid f(\hat{\mathbf{y}}(t)) \\ = \min\{f(\mathbf{y}_0(t)), f(\mathbf{y}_1(t)), \ldots, f(\mathbf{y}_s(t))\} \end{aligned} \tag{2.6}$$

Note that this definition states that $\hat{\mathbf{y}}$ is the best position discovered by any of the particles so far.

The algorithm makes use of two independent random sequences, $r_1 \sim U(0,1)$ and $r_2 \sim U(0,1)$. These sequences are used to effect the stochastic nature of the algorithm, as shown below in equation (2.7). The values of r_1 and r_2 are scaled by constants $0 < c_1, c_2 \leq 2$. These constants are called the *acceleration coefficients*, and they influence the maximum size of the step that a particle can take in a single iteration. The velocity update step is specified separately for each dimension $j \in 1..n$, so that $v_{i,j}$ denotes the j^{th} dimension of the velocity vector associated with the i^{th} particle. The velocity update equation is then

$$\begin{aligned} v_{i,j}(t+1) &= v_{i,j}(t) + c_1 r_{1,j}(t)[y_{i,j}(t) - x_{i,j}(t)] + \\ &\quad c_2 r_{2,j}(t)[\hat{y}_j(t) - x_{i,j}(t)] \end{aligned} \tag{2.7}$$

From the definition of the velocity update equation is clear that c_2 regulates the maximum step size in the direction of the global best particle, and c_1 regulates the step size in the direction of the personal best position of that particle. The value of $v_{i,j}$ is clamped to the range $[-v_{max}, v_{max}]$ to reduce the likelihood that the particle might leave the search space. If the search space is defined by the bounds $[-x_{max}, x_{max}]$, then the value of v_{max} is typically set so that $v_{max} = k \times x_{max}$, where $0.1 \leq k \leq 1.0$ [23].

The position of each particle is updated using the new velocity vector for that particle, so that

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1) \tag{2.8}$$

The algorithm consists of repeated application of the update equations presented above. Figure 2.5 lists the pseudo-code for the basic PSO algorithm. Note that the two *if*-statements are the equivalent of applying equations (2.5) and (2.6), respectively. The initialisation mentioned in the first step of the algorithm consists the following:

1. Initialise each coordinate $x_{i,j}$ to a value drawn from the uniform random distribution on the interval $[-x_{max}, x_{max}]$, for all $i \in 1..s$ and $j \in 1..n$. This distributes the initial positions of the particles throughout the search space. Many of the pseudo-random number generators available have flaws leading to low-order correlations when used to generate random vectors this way, so care must be exercised in choosing a good pseudo-random algorithm. Alternatively, the initial positions can be distributed uniformly through the search space using sub-random sequences, for example Sobol's sequence or a Latin hypercube distribution ([107], chapter 7.7).

2. Initialise each $v_{i,j}$ to a value drawn from the uniform random distribution on the interval $[-v_{max}, v_{max}]$, for all $i \in 1..s$ and $j \in 1..n$. Alternatively, the velocities of the particles could be initialised to $\mathbf{0}$, since the starting positions are already randomised.

3. Set $\mathbf{y}_i = \mathbf{x}_i$, $\forall i \in 1..s$. Alternatively, two random vectors can be generated for each particle, assigning the more fit vector to $\mathbf{y}_i$ and the less fit one to $\mathbf{x}_i$. This would require additional function evaluations, so the simpler method described first is usually used.

Create and initialise an n-dimensional PSO : S
repeat:
  for each particle $i \in [1..s]$ **:**
    if $f(S.\mathbf{x}_i) < f(S.\mathbf{y}_i)$
      then $S.\mathbf{y}_i = S.\mathbf{x}_i$
    if $f(S.\mathbf{y}_i) < f(S.\hat{\mathbf{y}})$
      then $S.\hat{\mathbf{y}} = S.\mathbf{y}_i$
  endfor
  Perform PSO updates on S using equations (2.7–2.8)
until stopping condition is true

Figure 2.5: Pseudo code for the original PSO algorithm

The stopping criterion mentioned in Figure 2.5 depends on the type of problem being solved. Usually the algorithm is run for a fixed number of function evaluations (thus a fixed number of iterations) or until a specified error bound is reached.

It is important to realise that the velocity term models the rate of change in the position of the particle. The changes induced by the velocity update equation (2.7) therefore represent acceleration, which explains why the constants c_1 and c_2 are called acceleration coefficients.

A brief description of how the algorithm works is as follows: Initially, some particle is identified as the best particle in a neighbourhood of particles, based on its fitness. All the particles are then accelerated in the direction of this particle, but also in the direction of their own best solutions that they have discovered previously. Occasionally the particles will overshoot their target, exploring the search space beyond the current best particles. All particles also have the opportunity to discover better particles *en route*, in which case the other particles will change direction and head towards the new 'best' particle. Since most functions have some continuity, chances are that a good solution will be surrounded by equally good, or better, solutions. By approaching the current best solution from different directions in search space, the chances are good that these neighbouring solutions will be discovered by some of the particles.

2.4.2 Social Behaviour

Many interpretations of the operation of the PSO have been suggested. Kennedy strengthened the socio-psychological view by performing experiments to investigate the function of the different components in the velocity update equation [67]. The task of training a neural network to correctly classify the XOR problem was used to compare the performance of the different models. Kennedy made use of the *lbest* model (see Section 2.4.6 for a complete description of this model), rather than the *gbest* model outlined above.

Consider the velocity update equation, repeated here for convenience

$$\begin{aligned} v_{i,j}(t+1) &= v_{i,j}(t) + c_1 r_{1,j}(t)[y_{i,j}(t) - x_{i,j}(t)] + \\ & \quad c_2 r_{2,j}(t)[\hat{y}_j(t) - x_{i,j}(t)] \end{aligned}$$

The term $c_1 r_{1,j}(t)[y_{i,j}(t) - x_{i,j}(t)]$ is associated with *cognition* since it only takes into account the particle's own experiences. If a PSO is constructed making use of the cognitive term only, the velocity update equation will become

$$v_{i,j}(t+1) = v_{i,j}(t) + c_1 r_{1,j}(t)[y_{i,j}(t) - x_{i,j}(t)]$$

Kennedy found that the performance of this 'cognition only' model was inferior to that of the original swarm, failing to train the network within the maximum allowed number of iterations for some parameter settings. One of the reasons for the poor behaviour of this version of the PSO is that there is no interaction between the different particles.

The third term in the velocity update equation, $c_2 r_{2,j}(t)[\hat{y}_j(t) - x_{i,j}(t)]$, represents the *social* interaction between the particles. A 'social only' version of the PSO can be constructed by using the following velocity update equation

$$v_{i,j}(t+1) = v_{i,j}(t) + c_2 r_{2,j}(t)[\hat{y}_{i,j}(t) - x_{i,j}(t)]$$

The performance of this model was superior to that of the original PSO on the specific problem that Kennedy investigated.

In summary, the PSO velocity update term consists of both a cognition component and a social component. Little is currently known about the relative importance of these two terms, although initial results seem to indicate that the social component may be

more significant on some problems. The social interaction between the particles is a form of cooperation, roughly following the model discussed by Clearwater *et al.* [19]. Cooperation will be discussed in more detail in Section 2.8.2.

2.4.3 Taxonomic Designation

The PSO is clearly related to some of the evolutionary algorithms. For one, the PSO maintains a population of individuals representing potential solutions, a property common to all EAs. If the personal best positions ($\mathbf{y}_i$) are treated as part of the population, then there is clearly a weak form of selection [2]. In a $(\mu+\lambda)$ ES algorithm, the offspring compete with the parents, replacing them if they are more fit. The update equation (2.5) resembles this mechanism, with the difference that each personal best position (parent) can only be replaced by its *own* current position (offspring), should the current position be more fit than the old personal best position. To summarise, there appears to be some weak form of selection present in the PSO.

The velocity update equation resembles the arithmetic crossover operator found in real-valued GAs. Normally, the arithmetic crossover produces two offspring that are linear blends of the two parents involved. The PSO velocity update equation, without the $v_{i,j}(t)$ term (see equation 2.7), can be interpreted as a form of arithmetic crossover involving two parents, returning a single offspring. Alternatively, the velocity update equation, without the $v_{i,j}(t)$ term, can be seen as a mutation operator, with the strength of the mutation governed by the distance that the particle is from its two 'parents'. This still leaves the $v_{i,j}(t)$ term unaccounted for, which can be interpreted as a form of mutation dependent on the position of the individual in the previous iteration.

A better way of modeling the $v_{i,j}(t)$ term is to think of each iteration not as a process of replacing the previous population with a new one (death and birth), but rather as a process of adaption [35]. This way the $\mathbf{x}_i$ values are not replaced, but rather adapted using the velocity vectors $\mathbf{v}_i$. This makes the difference between the other EAs and the PSO more clear: the PSO maintains information regarding position *and* velocity (changes in position); in contrast, traditional EAs only keep track of positions.

Therefore it appears that there is some degree of overlap between the PSO and most other EAs, but the PSO has some characteristics that are currently not present in other

EAs, especially the fact that the PSO models the velocity of the particles as well as the positions.

2.4.4 Origins and Terminology

The movement of the particles has been described as "flying" through n-dimensional space [37]. This terminology is in part due to experiments with bird flocking simulations which led to the development of the original PSO algorithm [70]. In fact, studying a paper by Reynolds [111] (cited in Kennedy and Eberhart's original PSO paper) reveals some interesting insights. Reynolds was mainly interested in simulating the flight patterns of birds for visual computer simulation purposes, observing that the flock *appears* to be under central control. It is clear that a natural flock (or school) is unaffected by the number of individuals participating in the formation, since schools of fish of up to 17 miles in length have been observed. Considering that the birds (or fish) must have finite 'processing power', one would expect a sharp upper limit on the size that a natural flock can reach, if an individual had to track all the members of the flock. Since no such upper bound is observed in nature, one must conclude that a bird (or fish) only pays attention to a limited number of its neighbours, implying local rather than global control.

Several reasons have been forwarded for the flocking behaviour observed in nature. Some evolutionary advantages include: protection from predators, improved survival of the gene pool, and profiting from a larger effective search area with respect to food. This last property is invaluable when the food is unevenly distributed over a large region.

Reynolds proceeded to model his flocks using three simple rules: collision avoidance, velocity matching and flock centering. Note that the flock centering drive will prompt a bird to fly closer to its neighbours (carefully, so that the velocity matching is not jeopardised) but still maintaining a safe distance, as governed by the collision avoidance rule. Reynolds decided to use a flock centering drive calculated by considering only the nearest neighbours of a bird, instead of using the centroid of the whole swarm, which he called the "central force model". This corresponds roughly to the *lbest* model of the PSO (described below). It is interesting to note Reynolds's observation [111]:

> "Before the current implementation of localised flock centering behaviour was implemented, the flocks used a central force model. This leads to un-

> usual effects such as causing all the members of a widely scattered flock to simultaneously converge toward the flock's centroid."

This describes quite accurately what happens in the *gbest* PSO model (also described in more detail below).

Reynolds's flocking *boids* (a word derived from bird-oid, denoting a generic bird-like object) were a popular example of some of the principles of Artificial Life. The flocking dynamics were a convincing example of *emergent behaviour*: complex global behaviour arising from the interaction of simple rules. This is one of the features that makes the PSO such a successful optimisation algorithm: a simple implementation that results in complex (and effective) search behaviour.

Even though the particle movement visually looks like flocking, it does not strictly comply with certain definitions of flocking behaviour. Matarić defines the following concepts [81]:

Safe-Wandering: The ability of a group of agents to move about while avoiding collisions with obstacles and each other.

Dispersion: The ability of a group of agents to spread out in order to establish and maintain some minimum inter-agent distance.

Aggregation: The ability of a group of agents to gather in order to establish and maintain some maximum inter-agent distance.

Homing: The ability to find a particular region or location.

Based on these definitions, Matarić contends that flocking behaviour consists of *homing*, *safe-wandering*, *dispersion* and *aggregation*. The PSO only implements homing and aggregation, lacking safe-wandering and dispersion. Safe-wandering is not important to the PSO, since it only applies to entities that can collide physically. Dispersion means that the particles will fan out when they get too close to one another, something which is not currently in the PSO model — the PSO encourages the particles to cluster.

The terms "swarm" and "swarming" are much more appropriate. This term was used by Millonas to describe artificial life models [85]. Millonas suggested that swarm intelligence is characterised by the following properties:

Proximity: Carrying out simple space and time computations.

Quality: Responding to quality factors in the environment.

Diverse Response: Not falling into a restricted subset of solutions.

Stability: Being able to maintain modes of behaviour when the environment changes.

Adaptability: Being able to change behavioural modes when deemed profitable.

Eberhart *et al.* [37] presented arguments indicating that the particles in the PSO possess these properties.

Lastly, the term "particle" requires some justification. The members of the population lack mass and volume, thus calling them "points" would be more accurate. The concepts of velocity and acceleration, however, are more compatible with the term particle (alluding to a small piece of matter) than they are with the term point. Some other research fields, notably computer graphics, also use the term "particle systems" to describe the models used for rendering effects like smoke or fire [110].

2.4.5 Gbest Model

The *gbest* model offers a faster rate of convergence [37] at the expense of robustness. This model maintains only a single "best solution," called the *global best particle*, across all the particles in the swarm. This particle acts as an attractor, pulling all the particles towards it. Eventually all particles will converge to this position, so if it is not updated regularly, the swarm may converge prematurely. The update equations for $\hat{\mathbf{y}}$ and $\mathbf{v}_i$ are the ones presented above, repeated here for completeness.

$$\begin{aligned}\hat{\mathbf{y}}(t) \in \{\mathbf{y}_0(t), \mathbf{y}_1(t), \ldots, \mathbf{y}_s(t)\} \mid f(\hat{\mathbf{y}}(t)) \\ = \min\{f(\mathbf{y}_0(t)), f(\mathbf{y}_1(t)), \ldots, f(\mathbf{y}_s(t))\}\end{aligned} \tag{2.9}$$

$$\begin{aligned}v_{i,j}(t+1) &= v_{i,j}(t) + c_1 r_{1,j}(t)[y_{i,j}(t) - x_{i,j}(t)] + \\ &\quad c_2 r_{2,j}(t)[\hat{y}_j(t) - x_{i,j}(t)]\end{aligned} \tag{2.10}$$

Note that $\hat{\mathbf{y}}$ is called the *global best position*, and belongs to the particle referred to as the *global best particle*.

2.4.6 Lbest Model

The *lbest* model tries to prevent premature convergence by maintaining multiple attractors. A subset of particles is defined for each particle from which the the *local best particle*, $\hat{\mathbf{y}}_i$, is then selected. The symbol $\hat{\mathbf{y}}_i$ is called the *local best position*, or the *neighbourhood best*. Assuming that the particle indices wrap around at s, the *lbest* update equations for a neighbourhood of size l are as follows:

$$\begin{aligned} N_i &= \{\mathbf{y}_{i-l}(t), \mathbf{y}_{i-l+1}(t), \ldots, \mathbf{y}_{i-1}(t), \mathbf{y}_i(t), \\ &\quad \mathbf{y}_{i+1}(t), \ldots, \mathbf{y}_{i-1}(t), \mathbf{y}_{i+l}(t)\} \end{aligned} \tag{2.11}$$

$$\hat{\mathbf{y}}_i(t+1) \in N_i | \ f(\hat{\mathbf{y}}_i(t+1)) = \min\{f(\mathbf{a})\}, \ \forall \mathbf{a} \in N_i \tag{2.12}$$

$$\begin{aligned} v_{i,j}(t+1) &= v_{i,j}(t) + c_1 r_{1,j}(t)[y_{i,j}(t) - x_{i,j}(t)] + \\ &\quad c_2 r_{2,j}(t)[\hat{y}_{i,j}(t) - x_{i,j}(t)] \end{aligned} \tag{2.13}$$

Note that the particles selected to be in subset N_i have no relationship to each other in the search space domain; selection is based purely on the particle's index number. This is done for two main reasons: it is computationally inexpensive, since no clustering has to be performed, and it helps promote the spread of information regarding good solutions to all particles, regardless of their current location in search space.

Lastly, note that the *gbest* model is actually a special case of the *lbest* model with $l = s$. Experiments with $l = 1$ have shown the *lbest* algorithm to converge somewhat more slowly than the *gbest* version, but it is less likely to become trapped in an inferior local minimum [37].

2.5 Modifications to the PSO

Numerous improvements to the Particle Swarm Optimiser have been proposed. The improvements listed below are grouped according to the specific shortcoming of the PSO that they aim to address. Before the improvements are discussed, though, a section describing a binary version of the PSO is presented.

2.5.1 The Binary PSO

A binary version of the PSO was introduced by Kennedy and Eberhart [71]. The binary version is useful for making comparisons between binary coded GAs and the PSO, as well as for representing problems that are binary by nature. One typical application may be to represent a neural network's connection graph, where a '1' represents a connection and a '0' represents the absence of a connection between two nodes in the network. A binary PSO can then be used to evolve the network architecture.

The binary version restricts the component values of $\mathbf{x}_i$ and $\mathbf{y}_i$ to be elements taken from the set $\{0, 1\}$. There is no such restriction on the value of the velocity, $\mathbf{v}_i$, of a particle, though. When using the velocity to update the positions, however, the velocity is thresholded to the range $[0, 1]$ and treated as a probability. This can be accomplished by using the sigmoid function, defined as

$$\text{sig}(x) = \frac{1}{1 + \exp(-x)} \tag{2.14}$$

The update equation for the velocity term used in the binary swarm is then

$$v_{i,j}(t+1) = v_{i,j}(t) + c_1 r_{1,j}(t)[y_{i,j} - x_{i,j}(t)] + c_2 r_{2,j}(t)[\hat{y}_j - x_{i,j}(t)] \tag{2.15}$$

Note that this velocity update equation does not differ from that used in the original PSO. Instead of the usual position update equation (*e.g.* equation 2.8), a new probabilistic update equation is used, namely

$$x_{i,j}(t+1) = \begin{cases} 0 & \text{if } r_{3,j}(t) \geq \text{sig}(v_{i,j}(t+1)) \\ 1 & \text{if } r_{3,j}(t) < \text{sig}(v_{i,j}(t+1)) \end{cases} \tag{2.16}$$

where $r_{3,j}(t) \sim U(0, 1)$ is a uniform random variate. By studying equation (2.16), it becomes clear that the value of $x_{i,j}$ will remain 0 if $\text{sig}(v_{i,j}) = 0$. This will happen when $v_{i,j}$ is approximately less than -10. Likewise, the sigmoid function will saturate when $v_{i,j} > 10$. To prevent this it is recommended to clamp the value of $v_{i,j}$ to the range ± 4 [35], resulting in a state-change probability of $\text{sig}(4) \approx 0.018$. The original paper describing the binary PSO recommended a slightly larger v_{max} threshold of ± 6, resulting in a probability of approximately 0.0025 [71].

Note that the velocity update equation corresponds to the original velocity update equation without the inertia weight or constriction coefficients (see Section 2.5.2). This

is because the paper describing the binary PSO was published before these modifications were introduced. A later paper used the binary PSO in a comparison with a GA on a multi-modal test-function generator [72]. That binary PSO made use of a constriction coefficient, showing that the techniques usually applied to the continuous PSO are applicable to the binary PSO as well. The results reported by Kennedy and Spears show that the binary PSO reached the solution to the problems faster than the GAs on most of the functions tested, especially when the problem dimensionality was increased.

It is possible to use both binary and continuous values in the same vector simultaneously. This version has been called a *Hybrid swarm* [35], but the term 'Hybrid', like the term 'Modified', has been applied to more than one modified PSO already. To prevent any confusion, a swarm using both binary and continuous variables will be called a 'binary+continuous swarm' in this work.

Recent papers extended the abilities of the PSO to include arbitrary discrete representations [50] by simply discretising the relevant quantities when necessary.

2.5.2 Rate of Convergence Improvements

Several techniques have been proposed for improving the rate of convergence of the PSO. These proposals usually involve changes to the PSO update equations, without changing the structure of the algorithm otherwise. This usually results in better local optimisation performance, sometimes with a corresponding decrease in performance on functions with multiple local minima.

Inertia weight

Some of the earliest modifications to the original PSO were aimed at further improving the rate of convergence of the algorithm. One of the most widely used improvements is the introduction of the *inertia weight* by Shi and Eberhart [124]. The inertia weight is a scaling factor associated with the velocity during the previous time step, resulting in a new velocity update equation, so that

$$\begin{aligned} v_{i,j}(t+1) &= wv_{i,j}(t) + c_1 r_{1,j}(t)[y_{i,j}(t) - x_{i,j}(t)] + \\ &\quad c_2 r_{2,j}(t)[\hat{y}_j(t) - x_{i,j}(t)] \end{aligned} \tag{2.17}$$

The original PSO velocity update equation can be obtained by setting $w = 1$. Shi and Eberhart investigated the effect of w values in the range $[0, 1.4]$, as well as varying w over time [124]. Their results indicate that choosing $w \in [0.8, 1.2]$ results in faster convergence, but that larger w values (> 1.2) result in more failures to converge.

The inertia weight governs how much of the previous velocity should be retained from the previous time step. To briefly illustrate the effect of w, let $c_1 = c_2 = 0$. Now, a w value greater than 1.0 will cause the particle to accelerate up to the maximum velocity v_{max} (or $-v_{max}$), where it will remain, assuming the initial velocity was non-zero. A w value less than 1.0 will cause the particle to slowly decelerate until its velocity reaches zero. When $c_1, c_2 \neq 0$, the behaviour of the algorithm is harder to predict, but based on the results of Shi and Eberhart [124] it would appear that w values close to 1.0 are preferable.

Another set of experiments were performed to investigate the interaction between v_{max} and the inertia weight [120]. For the single function studied in this experiment, it was found that an inertia weight of 0.8 produced good results, even when $v_{max} = x_{max}$. The best performance, however, was again obtained by using an inertia weight that decreased from 0.9 to 0.4 during the first 1500 iterations.

Further empirical experiments have been performed with an inertia weight set to decrease linearly from 0.9 to 0.4 during the course of a simulation, this time using four different objective functions [121]. This setting allows the PSO to explore a large area at the start of the simulation run (when the inertia weight is large), and to refine the search later by using a smaller inertia weight. The inertia weight can be likened to the temperature parameter encountered in Simulated Annealing [73, 89]. The Simulated Annealing algorithm has a process called the *temperature schedule* that is used to gradually decrease the temperature of the system. The higher the temperature, the greater the probability that the algorithm will explore a region outside of basin of attraction of the current local minimum. Therefore an adaptive inertia weight can be seen as the equivalent of a temperature schedule in the Simulated Annealing algorithm.

Fuzzy Inertia Weight

Shi and Eberhart recently proposed a technique for adapting the inertia weight dynamically using a fuzzy controller [122, 123]. A fuzzy controller is a mechanism that can be used to translate a linguistic description of a problem into a model that can be used to predict a numeric variable, given numeric inputs [37]. In other words, if a human can describe roughly how a variable should be adjusted when observing the input, that knowledge can be captured by the fuzzy controller. Some understanding of the effect of the inertia weight has been accumulated over the many experimental studies that have been performed, making a fuzzy inertia weight controller a good choice.

The controller proposed by Shi and Eberhart uses as input the current inertia weight and the function value corresponding to the best solution found so far, $f(\hat{\mathbf{y}})$. Since most problems have function values on differing scales, the value of $f(\hat{\mathbf{y}})$ must be normalised. Equation (2.18) presents one possible technique for scaling the function values:

$$f_{norm}(\hat{\mathbf{y}}) = \frac{f(\hat{\mathbf{y}}) - f_{min}}{f_{max} - f_{min}} \tag{2.18}$$

The values f_{max} and f_{min} are problem dependent, and must be known in advance or some estimate must be available.

Shi and Eberhart chose to use three fuzzy membership functions, corresponding to three fuzzy sets (*low, medium, high*) that the input variables can belong to. The output of the fuzzy controller is the suggested change in the value of the inertia weight. Some fine-tuning is required to specify the critical values for the fuzzy membership functions, but that's the whole point of using a fuzzy controller in the first place: these parameters are known approximately from previous experience.

The fuzzy adaptive inertia weight PSO was compared to a PSO using a linearly decreasing inertia weight. The results indicated that the fuzzy inertia weight PSO exhibited improved performance on some of the functions tested, for certain parameter settings [122]. It is interesting to note that the fuzzy inertia weight method had a greater advantage on the unimodal function in the test suite. This behaviour is easily explained: unimodal functions have no local minima, so an optimal inertia weight can be determined at each iteration. Logically, at the start of the run a large inertia weight will allow the PSO to locate the approximate region in which the minimiser is situated more rapidly.

Upon reaching this area the inertia weight should be gradually decreased to slow down the movement of the particles, allowing them to locate smaller features on the function's surface. This process can be approximated by linearly decreasing the inertia weight over time, but this mechanism does not 'know' whether the PSO has located the region where a smaller inertia weight should be used yet. Sometimes the PSO may take longer to reach this region, sometimes it finds it very quickly. The adaptive fuzzy controller is able to predict approximately what type of behaviour is more suitable. The rules in the fuzzy controller are effectively reducing the inertia weight at a rate proportional to how close the PSO is to the minimum, which is measured by how close $f_{norm}(\hat{\mathbf{y}})$ is to zero.

When dealing with a function containing multiple local minima, however, it is more difficult to find an optimal inertia weight. If a particle has already 'stumbled upon' the basin containing the global minimum, the inertia weight can be decreased significantly to allow the PSO to perform a fine-grained search in that basin. On the other hand, if the swarm has only managed to find a good nearby local minimum, the correct setting for the inertia weight would be a somewhat larger value to allow the PSO to escape from the local minimum's basin of attraction. The adaptive fuzzy inertia weight controller cannot tell the difference between being trapped in a good (*i.e.* small value for $f_{norm}(\hat{\mathbf{y}})$) local minimum, and being close to the minimum of a unimodal function.

The adaptive fuzzy inertia weight controller is a promising technique for optimising the inertia weight, but implementation difficulties, like knowing the values of f_{max} and f_{min}, makes it hard to implement in a generic fashion.

Constriction Factor

Recently, work by Clerc [20, 23] indicated that a *constriction factor* may help to ensure convergence. The constriction factor model describes, amongst other things, a way of choosing the values of w, c_1 and c_2 so that convergence is ensured. By choosing these values correctly, the need for clamping the values of $v_{i,j}$ to the range $[-v_{max}, v_{max}]$ is obviated. A discussion of the different models that Clerc proposed, as well as the analytical insights gained from his work, follows in Section 2.7. A specific instance of the constriction model, related to the results discussed below, will be described next.

A modified velocity update equation, corresponding to one of several constriction

models [23, 20], is presented in equation (2.19).

$$v_{i,j}(t+1) = \chi\Big(v_{i,j}(t) + c_1 r_{1,j}(t)(y_{i,j}(t) - x_{i,j}(t)) + c_2 r_{2,j}(t)(\hat{y}_j(t) - x_{i,j}(t))\Big), \quad (2.19)$$

where

$$\chi = \frac{2}{\left|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}\right|}, \quad (2.20)$$

and $\varphi = c_1 + c_2, \quad \varphi > 4$.

Let $c_1 = c_2 = 2.05$. Substituting $\varphi = c_1 + c_2 = 4.1$ into (2.20) yields $\chi = 0.7298$. Substitution into equation (2.19), and dropping the explicit reference to t, results in

$$v_{i,j}(t+1) = 0.7298\Big(v_{i,j} + 2.05 \times r_{1,j}(y_{i,j} - x_{i,j}) + 2.05 \times r_{2,j}(\hat{y}_j - x_{i,j})\Big),$$

Since $2.05 \times 0.7298 = 1.4962$, this is equivalent to using the values $c_1 = c_2 = 1.4962$ and $w = 0.7298$ in the modified PSO velocity update equation (2.17).

Eberhart and Shi compared the performance of a swarm using the v_{max} clamping to one using only the constriction factor [39]. Their results indicated that using the constriction factor (without clamping the velocity) usually resulted in a better rate of convergence. On some of the test functions, however, the PSO with the constriction factor failed to reach the specified error threshold for that problem within the allocated number of iterations. The problem, according to Eberhart and Shi, is that the particles stray too far from the desired region of search space. To mitigate this effect they decided to apply clamping to the constriction factor implementation as well, setting the v_{max} parameter equal to x_{max}, the size of the search space. This led to improved performance for almost all the functions they used during testing — both in terms of the rate of convergence and the ability of the algorithm to reach the error threshold.

Selection

Angeline introduced a version of the PSO that borrows the concept of selection from the field of Evolutionary Computation [2] (refer to Section 2.2 for more detail on Evolutionary Computation). Angeline argues that the current PSO has a weak, implicit form of selection if one considers the personal best position as additional population members. In the *gbest* model (the one used by Angeline for comparison), a particle only has access

to its own personal best and that of the global best particle. This means the possible interaction between members from one half of the population (the current positions) and the other half (the personal best positions) is severely restricted.

The purpose of selection in an Evolutionary Algorithm is to focus the effort of the algorithm on a specific region of the search space, usually one that delivered promising solutions in the recent past. A more thorough search of this region then ensues. Angeline proposed the following method for adding selection to the PSO [2]:

1. Pick an individual from the population. Compare the fitness of this individual with k other individuals in the population, awarding it with one mark every time that the current individual has a fitness value superior to that of the one it is compared with. Repeat this process for every individual.

2. Rank the particles by sorting them according to the marks accumulated in the previous step.

3. Select the top half of the population, and copy their current positions onto the current positions of the bottom half of the population. The personal best values are left untouched.

This process is applied before the PSO velocity update equations are executed.

Angeline presented results comparing the original PSO (without a constriction factor and without an inertia weight) to the PSO with selection. It was found that the modified PSO performed significantly better than the original on the unimodal functions as well as Rastrigin's function, but worse on Griewank's function. Note that Griewank's function contains many local minima, which means that the selection mechanism actually promotes convergence onto a local minimum. If some particles discover a reasonable minimum, the other half of the population could be moved into the basin of the same local minimum. This affects the ability of the algorithm to explore large regions of search space, thus preventing it from finding the global minimum.

Selection thus improves the local search abilities of the PSO, but simultaneously hampers its global search abilities.

Breeding

Following the work of Angeline [2], Løvbjerg *et al.* applied further Evolutionary Computation mechanisms to the PSO algorithm [79]. They chose to investigate the effect of *reproduction* and *recombination*, of GA parlance, which they collectively referred to as *breeding*.

The proposed modification to the PSO proceeds as follows:

1. Calculate the new particle velocities and positions, using, for example, equations (2.7) and (2.8).

2. Mark each particle as a potential parent, with probability P_b (breeding probability).

3. From the pool of marked particles, select two candidates and perform the arithmetic crossover operation as detailed in equations (2.21)–(2.24), yielding two new children, replacing the original parents.

4. The personal best position of each of the particles involved are set to their current positions, *i.e.* $\mathbf{y}_i = \mathbf{x}_i$.

Note that the selection of parents is effected in a purely stochastic fashion, no fitness-based selection is performed. This prevents some of the potential problems associated with fitness-based selection on functions containing many local minima.

Let a and b denote the indices of the two particles selected as parents. Then the arithmetic crossover proceeds as follows:

$$\mathbf{x}_a(t+1) = r_1\mathbf{x}_a(t) + (1.0 - r_1)\mathbf{x}_b(t) \tag{2.21}$$

$$\mathbf{x}_b(t+1) = r_1\mathbf{x}_b(t) + (1.0 - r_1)\mathbf{x}_a(t) \tag{2.22}$$

$$\mathbf{v}_a(t+1) = \frac{\mathbf{v}_a(t) + \mathbf{v}_b(t)}{||\mathbf{v}_a(t) + \mathbf{v}_b(t)||}||\mathbf{v}_a(t)|| \tag{2.23}$$

$$\mathbf{v}_b(t+1) = \frac{\mathbf{v}_a(t) + \mathbf{v}_b(t)}{||\mathbf{v}_a(t) + \mathbf{v}_b(t)||}||\mathbf{v}_b(t)|| \tag{2.24}$$

where $r_1 \sim U(0,1)$. The arithmetic crossover of the positions yields two new positions at random locations within the hypercube of which the parents form the corners. The velocity crossover normalises the length of the sum of the two parent's velocities, so that only the direction and not the magnitude is affected.

The results presented by Løvbjerg *et al.* [79] show that the breeding slows down the rate of convergence on unimodal functions, thus making the PSO with breeding a less efficient local optimiser than the original PSO. The situation is reversed on functions with multiple local minima, so that the PSO with breeding takes the lead. No comparison with the *lbest* model was presented, so it is not clear whether the breeding performs better than the original *lbest* algorithm on the multiple-minima functions.

2.5.3 Increased Diversity Improvements

The modifications presented in this section are aimed at increasing the diversity of solutions in the population, a technique often applied to Genetic Algorithms. Most of these approaches are based on the *lbest* model, where the neighbourhood of a particle is smaller than the whole swarm. These improvements usually slow down the rate of convergence, but produce better results when faced with multiple local minima.

Spatial Neighbourhoods

The original *lbest* PSO (see Section 2.4.6) partitions the swarm into neighbourhoods based on their index numbers, that is, particles $\mathbf{x}_1$ and $\mathbf{x}_2$ are considered to be neighbours in a neighbourhood of radius 1, regardless of their spatial positions. A different partitioning scheme, based on the spatial location of the particles, has been proposed by Suganthan [132]. During each iteration of the algorithm the distance from each particle to every other particle in the swarm is computed, keeping track of the largest distance between any two particles with a variable called d_{max}. For each particle the ratio $||\mathbf{x}_a - \mathbf{x}_b||/d_{max}$ is computed, where $||\mathbf{x}_a - \mathbf{x}_b||$ is the distance from the current particle, a, to another particle b. This ratio can be used to select neighbouring particles, corresponding to small ratios, or particles further away, corresponding to larger ratios. Suganthan proposed that the selection threshold should be varied over the number of iterations, starting with a small ratio (*i.e.* an *lbest* model), and gradually increasing the ratio. Once the ratio reaches 1, the algorithm will effectively be using the *gbest* model.

Suganthan suggests that the threshold, called $frac$, should be computed as follows

$$frac = \frac{3 \times k + 0.6 \times k_{max}}{k_{max}}$$

where k is the current iteration number, and k_{max} is the maximum allowed number of iterations. Another particle b is considered to be in the neighbourhood of the current particle if $||\mathbf{x}_a - \mathbf{x}_b||/d_{max} < frac$. Using the notation introduced in Section 2.4, the neighbourhood of particle i is defined as

$$N_i = \{\mathbf{y}_l\} \mid \frac{||\mathbf{x}_i - \mathbf{x}_l||}{d_{max}} < frac,\ l \in 1..s$$

The local best particle is then selected using equation (2.12), after which equation (2.13) can be used to update the particle's velocity.

Suganthan also linearly decayed the values of c_1, c_2 and w over time, but states that fixed values of c_1 and c_2 produced better results. It was also found that setting $c_1 = 2.5$ and $c_2 = 1.5$ produced better results on some of the test functions.

The modified neighbourhood rule, combined with time-varying w values, resulted in improved performance (compared to *gbest*) on almost all of the test configurations, even on the unimodal test functions. This last property is somewhat surprising, since *gbest* is expected to perform better than *lbest* on unimodal functions. Judging from the pseudo-code provided in [132], however, it appears that the *gbest* algorithm did not enjoy the benefits of a time-varying w value, which would certainly affect the results significantly.

Neighbourhood Topologies

The *lbest* model obtained through equations (2.11) and (2.12) with an l value of 1 describes a ring topology, so that every particle considers its two immediate neighbours (in index space) to be its entire neighbourhood. All the particles can exchange information indirectly, since particle $i+1$ is the neighbour of both particles i and $i+2$, who in turn have neighbours $i-1$ and $i+3$, and so on. The relatively long path between particles i and $2i$ slows down the exchange of information between them. This allows them to explore different regions of search space, but still be able to share information.

Kennedy has constructed alternative topologies through which the rate of information flow can be varied [68]. Sociology researchers use the term "small worlds" to describe the well-known phenomenon that a person indirectly shares information with a vast number of other persons. Research conducted by Milgram [84] indicated that people in the United States were only five persons apart, that is, given two random individuals,

the first could locate the second through as few as five people in between. Further, research by Watts and Strogetz [142] shows that changing a few randomly-selected edges in a ring topology dramatically reduces the average path length, while still maintaining a high degree of clustering. Kennedy exploits these findings to construct alternative neighbourhood topologies for an *lbest* PSO.

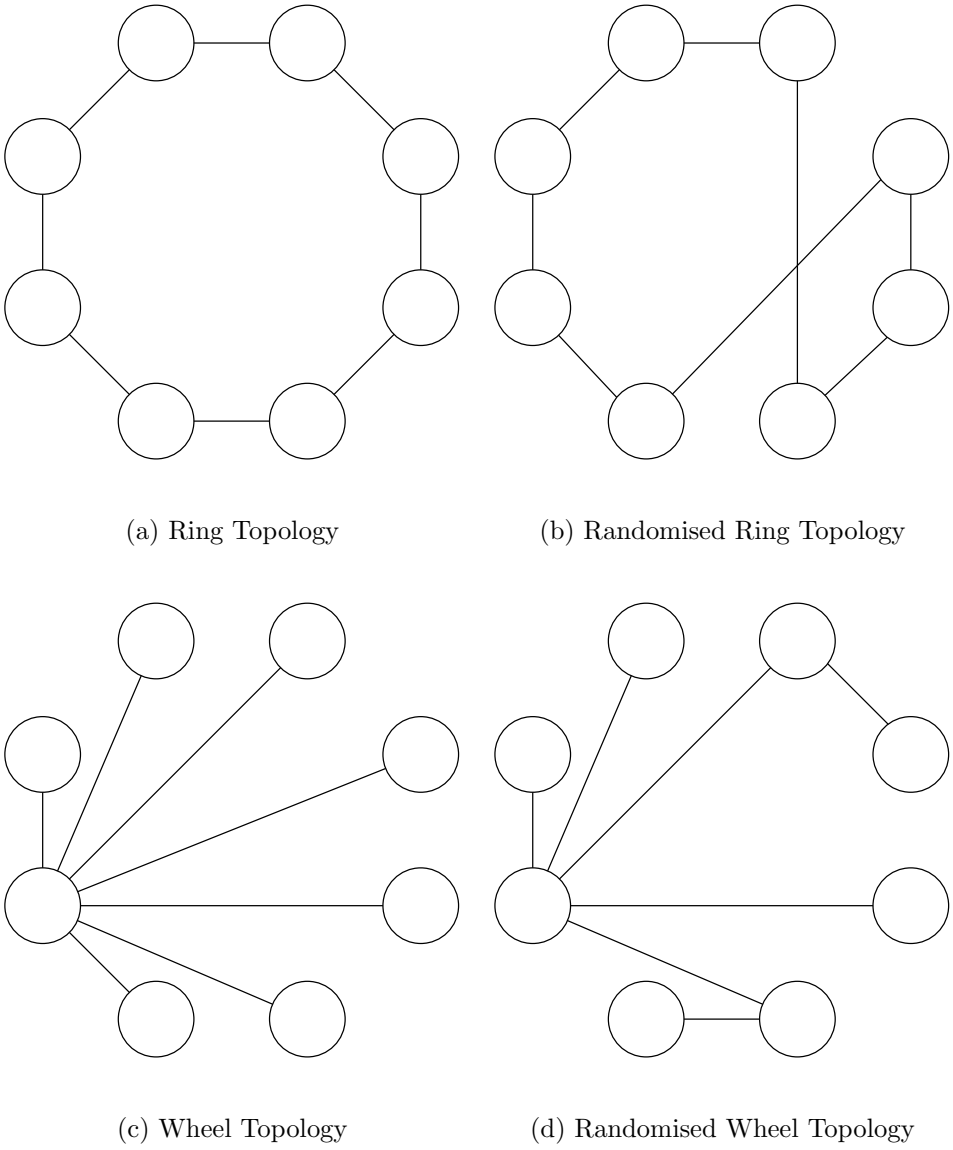
The first topology tested by Kennedy is the original ring-structure, but with a varying number of randomly interchanged connections. A "wheel" topology is also considered, where all the particles are connected to a single "hub" particle, but not directly to each other. Figure 2.6 illustrates these two topologies, before and after some links have been randomly exchanged. The last two topologies that were considered were a randomly-connected topology and a star topology. The star topology represents a fully connected swarm, thus it is actually the *gbest* model. Kennedy hypothesized that highly connected topologies (like the star topology) may have difficulty in finding good optima when the function contains a large number of local optima.

The experimental results presented by Kennedy indicates that the topology significantly affects the performance of the algorithm, but it appears that the optimal topology depends on the specific problem. For example, the wheel topology produced the best results when applied to a function with many local minima. Kennedy postulated that this can be attributed to the slower spread of information through the topology, resulting in a more robust algorithm in the face of many local optima. On the unimodal functions, however, the star topology (*gbest*) produced better solutions than the less-interconnected topologies, owing to the faster spread of information.

Note that all of the topologies considered here were structured in the particle index space, not in search space.

Social Stereotyping

Kennedy proposed a version of the *lbest* PSO that is a mixture of the spatial neighbourhood and the ring-topology approaches, called social stereotyping [69]. The particles in the original PSO are attracted to previous best positions discovered by themselves (the $\mathbf{y}_i$'s) or other particles in the swarm (the $\hat{\mathbf{y}}_i$'s). Human social interaction studies indicate that people often attempt to follow the collective beliefs of a group, rather than



(a) Ring Topology

(b) Randomised Ring Topology

(c) Wheel Topology

(d) Randomised Wheel Topology

Figure 2.6: A diagrammatic representation of two possible neighbourhood topologies, before and after edges have been randomly exchanged.

the beliefs of a specific individual in the group. This notion can be implemented in the PSO as follows: each particle is a member of a cluster in search space. The centroid of that cluster is analogous to the collective beliefs of the cluster. To simulate the effect of a human tending toward the group's collective belief, the particles can be drawn to the centroids of such clusters, rather than individual best positions.

It is possible to modify the PSO so that the cognitive (relating to the particle's personal previous best position) or the social (relating to the best previous position in the neighbourhood) component, or both, is replaced with the appropriate cluster centroid. A brief description of the process follows.

A number of clusters $\mathbf{C}_l,\ l \in 1..\#clusters$ are formed from the personal best values of the particles, using a k-means clustering algorithm. The number of clusters, $\#clusters$, is selected beforehand. Let $\mathbf{C}^*$ denote the set of all the clusters. Then $\text{map}(\mathbf{C}^*, i)$ is a function returning the value l so that $l = \text{map}(\mathbf{C}^*, i) \Rightarrow \mathbf{y}_i \in \mathbf{C}_l$. With this function it is possible to find the cluster to which particle i belongs.

The centroid for the cluster containing particle i is defined to be

$$\overline{\mathbf{C}(i)} = \frac{1}{|\mathbf{C}_l|} \sum_{\mathbf{a} \in \mathbf{C}_l} \mathbf{a}, \quad l = \text{map}(\mathbf{C}^*, i) \tag{2.25}$$

where $|\mathbf{C}_l|$ denotes the number of elements in cluster l. Using the neighbourhood of radius 1, defined as in equation (2.11), a best particle g is selected from the neighbourhood so that

$$\mathbf{y}_g \ \in N_i \mid f(\mathbf{y}_g) \leq f(\mathbf{y}_b)\ \forall \mathbf{y}_b \in N_i$$

The centroid of the cluster containing g is thus $\overline{\mathbf{C}(g)}$, using the definition of the centroid from equation (2.25). With these definitions, three new variations of the *lbest* update equation (2.13) are possible (explicit reference to t omitted on the right-hand side of the equations):

$$v_{i,j}(t+1) = wv_{i,j} + c_1 r_{1,j}(\overline{C(i)}_j - x_{i,j}) + c_2 r_{2,j}(\hat{y}_{i,j} - x_{i,j}) \tag{2.26}$$

$$v_{i,j}(t+1) = wv_{i,j} + c_1 r_{1,j}(y_{i,j} - x_{i,j}) + c_2 r_{2,j}(\overline{C(g)}_j - x_{i,j}) \tag{2.27}$$

$$v_{i,j}(t+1) = wv_{i,j} + c_1 r_{1,j}(\overline{C(i)}_j - x_{i,j}) + c_2 r_{2,j}(\overline{C(g)}_j - x_{i,j}) \tag{2.28}$$

The first version, presented in equation (2.26), can be interpreted as follows: Instead of using its own personal experience, the particle uses the collective experience of the

spatial cluster to which it belongs instead. The neighbourhood influence is unchanged from the original *lbest* algorithm.

The second version uses the opposite approach: equation (2.27) shows that the neighbourhood influence is now based on the centroid of the cluster to which the most successful particle in the neighbourhood belongs. Note that the neighbourhood is still defined in terms of the particle's index (*i.e.* no spatial relationship). The cognition component of the update equation is unaffected.

Lastly, equation (2.28) replaces both the cognition and the social component with their respective centroids. The particle is now drawn to the average of its own cluster, as well as the average of the cluster to which the most successful particle in its neighbourhood belongs.

The calculations involved with the computation of the cluster centroids take a non-negligible amount of time, so that the social stereotyping approach is slightly slower than the original *lbest* PSO.

Kennedy reported that the first version, corresponding to equation (2.26), was able to produce better solutions than the original *lbest* algorithm on some of the problems. A second set of experiments, where the algorithms were timed to see how long they took to reach a specified error bound, indicated that the stereotyping algorithms were generally slower than the original *lbest* algorithm. The overhead of forming clusters during each iteration slows down the new algorithms significantly.

The other two algorithms, based on equations (2.27) and (2.28), generally performed worse, indicating that a particle should not attempt to emulate the centroid of a distant cluster.

Subpopulations

The idea of using subpopulations to increase the diversity of solutions maintained by an algorithm has previously been used in Genetic Algorithms [130]. To create subpopulations the original population is partitioned into smaller populations. The algorithm (*e.g.* the GA) is applied to the elements in the subpopulation in the usual manner. From time to time members are exchanged between subpopulations, or some other interaction scheme is used to facilitate the sharing of information between the subpopulations. The

idea is to allow each population to thoroughly search a smaller region of search space without the possibly deleterious influence of solutions from a far-removed region of search space.

Løvbjerg *et al.* applied the notion of subpopulations to the PSO [79]. The breeding operator (an arithmetic crossover operator) that they applied to the PSO, as described above, is also used to effect the inter-subpopulation communication. They partition the original swarm into blocks, where each block maintains its own global best particle. When they select the parents for the crossover operator there is a small probability that one of the parents will be selected from a different subpopulation. If this probability is sufficiently small the subpopulations will have time to discover their own solutions, which they can then share with other subpopulations through inter-subpopulation breeding.

In their actual implementation, the authors chose to keep the number of particles fixed at 20 while forming subpopulations, so that a 2-subpopulation configuration would consist of two swarms of 10 particles each. The results reported in [79] indicate that this method of creating subpopulations does not lead to better performance. The rate of convergence slowed down as the number of subpopulations was increased.

It seems like the subpopulation technique is of little benefit to the PSO algorithm. The way in which the subpopulations were formed could possibly be to blame, however, since the idea is sound and works well when applied to other evolutionary algorithms.

2.5.4 Global Methods

This section describes methods that attempt to find all the global minima of a function. These approaches can be used in conjunction with most local search algorithms, although they are easier to implement in conjunction with evolutionary approaches like GAs and PSOs.

Sequential Niche Technique

Beasley *et al.* introduced an approach called the *sequential niche technique* [10] to systematically visit each global minimum of the objective function in turn. The local optimiser that they chose was a GA, not a PSO, but the technique is applicable to the PSO as well, and would be an interesting topic for future research.

Theoretically, a search algorithm like the PSO can be applied repeatedly to the same objective function, to eventually yield all the desired minima. This technique only has an asymptotic probability of one to find all the minima, though, so it could take an unacceptable number of attempts in a practical application before finding all the desired minima. The idea proposed by Beasley *et al.* is to progressively adapt the fitness function after each minimum has been discovered, so that the algorithm will not return to this minimum again. By applying this technique repeatedly, each time suppressing one new minimum, it will allow the search algorithm to enumerate all the minima.

The method used to adapt the fitness function is crucial to the success of the algorithm. Beasley *et al.* chose to use a set of "derating" functions to suppress the maxima in the fitness landscape (to a GA, a high fitness value indicates a good solution). A sample derating function is shown in equation (2.29).

$$G(\mathbf{x}, \mathbf{x}_B^*) = \begin{cases} \left(\frac{||\mathbf{x}-\mathbf{x}_B^*||}{r}\right)^\alpha & \text{if } ||\mathbf{x}-\mathbf{x}_B^*|| < r \\ 1 & \text{otherwise} \end{cases} \tag{2.29}$$

where $\mathbf{x}_B^*$ is the position of the previously discovered maximum (in the fitness landscape), α is a tunable parameter, indicating the strength of the attenuation, and r is the radius associated with the derating function.

The derating function results in a radial depression, with radius r, of the fitness function around the point $\mathbf{s}$. If the strength of the depression is precisely the correct magnitude, and r is chosen correctly, then the derating function will effectively remove the presence of the local maximum in the fitness landscape located at $\mathbf{x}_B^*$.

Several problems remain with this technique:

1. The derating function assumes that the local maximum in the fitness landscape has a form matching the inverse of the derating function, *i.e.* the local maximum must be a radial "hill";

2. The position $\mathbf{s}$ must be centred exactly on top of the centre of the local maximum;

3. The value of α must be chosen, or determined, correctly;

4. The value of r, called the *niche radius*, must be chosen or calculated correctly.

The last two issues were addressed by choosing default values for α and r. Clearly this leads to suboptimal removal of the local maximum, since not all the maxima will have the same shape and size.

Choosing a value of r that is smaller than the actual radius of the local maximum (assuming it is has a radial shape) will result in the introduction of more local maxima. These maxima will usually have a small enough magnitude so that other valid maxima will be discovered first, but clearly there is no guarantee of this.

On the other hand, choosing a value of r that is larger than the actual radius of the local maximum may lead to the suppression of a neighbouring local maximum. Even if this hypothetical neighbouring maximum is not significantly attenuated by the derating function, the true location of the maximum will shift slightly because of the derating function's presence.

At first it would appear that this technique introduces more problems than it solves, but Beasley *et al.* have shown that the sequential niche technique can successfully locate all the maxima, even on so-called "trap" functions designed to mislead Genetic Algorithms. Their experimental results further shown that locating p maxima takes roughly p times as long as locating a single maximum, implying linear time scaling with the number of maxima.

One of the major strengths of this technique is that it allows the enumeration of all the global minima of a multi-modal function, a feature that is especially valuable in a multi-objective optimisation problem.

Objective Function Stretching

The idea of adapting the objective function to reduce the effort expended to locate all the global minima has been successfully applied to the PSO by Parsopoulos *et al.* [101, 99, 98, 97]. Their approach is reminiscent of that of Beasley *et al.* [10], although the details differ significantly.

Parsopoulos *et al.* propose a two-step "Stretching" process through which the objective function is modified to prevent the PSO from returning to the just-discovered local minimum. The goal of the stretching function is to eliminate all local minima located *above* the current local minimum, without affecting the minima *below* it. This

implies that the true location of the global minimum is unaffected, a property that is not necessarily shared by the approach suggested by Beasley *et al.*

Let $\mathbf{x}_B^*$ be the location in search space of a recently discovered local minimum, so that a neighbourhood B exists where $f(\mathbf{x}_B^*) \leq f(\mathbf{x}),\ \forall \mathbf{x} \in B$. Application of the stretching technique produces a new objective function H, so that

$$G(\mathbf{x}) = f(\mathbf{x}) + \frac{\gamma_1}{2}||\mathbf{x} - \mathbf{x}_B^*||\text{sign}(f(\mathbf{x}) - f(\mathbf{x}_B^*)) + 1), \tag{2.30}$$

$$H(\mathbf{x}) = G(\mathbf{x}) + \frac{\gamma_2(\text{sign}(f(\mathbf{x}) - f(\mathbf{x}_B^*)) + 1)}{2\tanh(\mu(G(\mathbf{x}) - G(\mathbf{x}_B^*)))} \tag{2.31}$$

where γ_1, γ_2 and μ are arbitrarily chosen positive constants, and the sign function is defined as

$$\text{sign}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

Note that the sign function can also be approximated by the tanh function, so that sign(x) $\approx \tanh(\lambda x)$, for a suitably large constant λ. Parsopoulos *et al.* speculate that the tanh function may result in slightly better performance than the sign function. Recommended values for the parameters are as follows: $\gamma_1 = 10^4$, $\gamma_2 = 1$, and $\mu = 10^{-10}$.

The stretching algorithm can be added to the PSO with little difficulty. Specifically, whenever the PSO converges (presumably on a local minimum), the value of $\mathbf{x}_B^*$ is recorded. The function $f(x)$ is set equal to $H(x)$ (as defined in equation 2.31), and all the particles in the swarm are re-initialised. The process of building a new H function and reinitialising the swarm whenever another minimum is discovered is repeated until some stopping criterion is met. Re-initialising the swarm involves choosing new random positions (and personal best positions) for all the particles. This step is required since the particles will not be able to discover new minima once they have all converged onto the global best particle, so they are simply scattered through the search space like at the start of the algorithm. If it is assumed that the algorithm only terminates once the global minimiser has been found, then $\mathbf{x}_B^*$ is equal to the value of the global minimiser at the end of the run.

This technique is not without problems, though. Consider the function f(x) = $x^4 - 12x^3 + 47x^2 - 60x$, with a local minimiser located at approximately 4.60095589. If

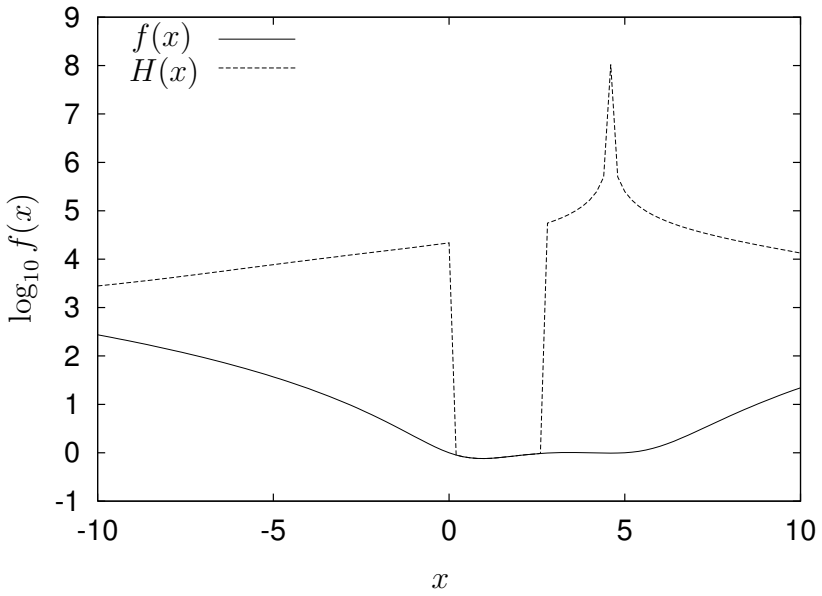


Figure 2.7: The function $H(x)$ represents the objective function, $f(x)$, after stretching. Parameters settings: $\gamma_1 = 10^3$, $\gamma_2 = 1$, $\mu = 10^{-10}$.

the PSO has converged on this local minimum, the stretching technique will thus set $x_B^* = 4.6009558$. Figure 2.7 shows the stretched function $H(x)$ with parameter γ_1 set to 10^3. The local minimum located at $x = 4.60095589$ is not clearly visible on the scale at which the graph is presented, though. Note that the global minimiser, located at $x = 0.94345$, is situated in what appears to be a deep trench after stretching. All the function values outside of this trench have been "stretched" higher, with the previously discovered local minimum located at 4.60095589 now forming a maximum.

The problem with the stretching technique is the introduction of false local minima, as well as misleading gradients. Even though the PSO does not make use of gradient information, it still has a tendency to move 'down' a slope. Note that, in Figure 2.7, the slope in the interval $-10 < x < 0$ leads away from the region in which the global minimiser is contained. Further, note that the slope of the original $f(x)$ in this region leads *to* the global minimum. If the PSO is bounded to the region $-10 < x < 10$, then it may converge at either of the edges since the slope would lead them to these boundaries.

Different parameter settings change the shape of $H(x)$, as can be observed in Fig-

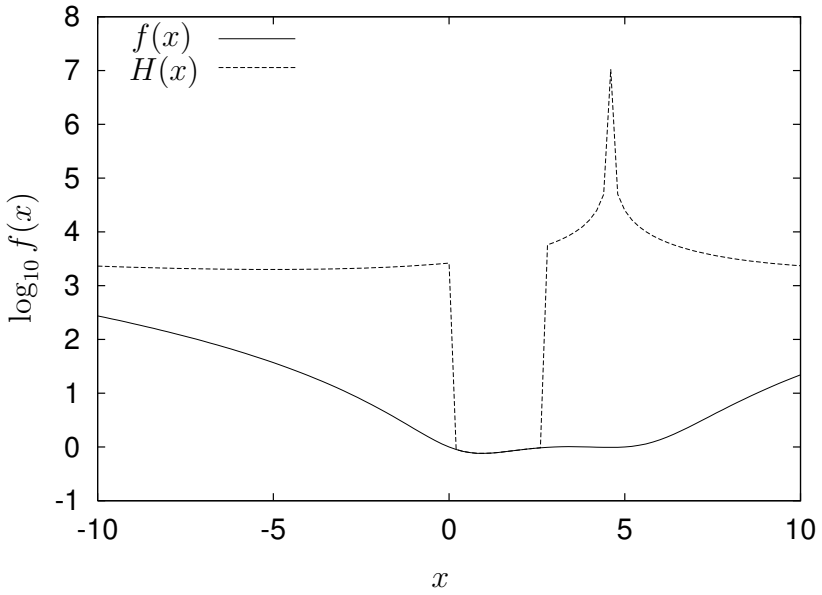


Figure 2.8: The function $H(x)$ represents the objective function, $f(x)$, after stretching. Parameters settings: $\gamma_1 = 10^4$, $\gamma_2 = 1$, $\mu = 10^{-10}$.

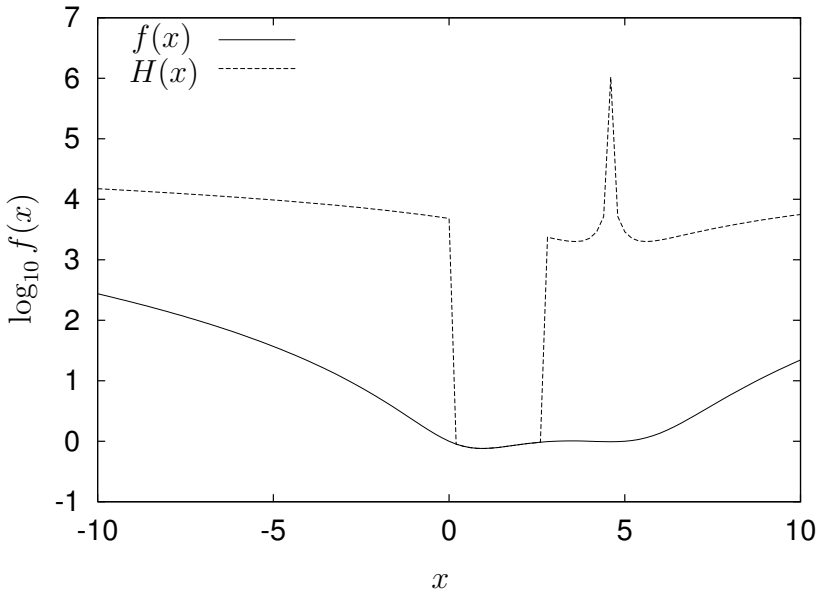


Figure 2.9: The function $H(x)$ represents the objective function, $f(x)$, after stretching. Parameters settings: $\gamma_1 = 10^5$, $\gamma_2 = 1$, $\mu = 10^{-10}$.

ure 2.8, where γ_1 is set to 10^4. Note that $H(x)$ is now slightly concave in the interval $-10 < x < 0$, thus introducing a false minimum to which the PSO may converge. The boundaries at ± 10 still present a problem.

Increasing the value of γ_1 even further produces the graph shown in Figure 2.9. Note that the slope in the interval $-10 < x < 0$ now leads to the interval containing the global minimiser, but that two local minima have been introduced in the interval $3.5 < x < 7$.

Although this example does not test every possible combination of parameter settings, it does show that the stretching technique appears to introduce false minima regardless of the parameter settings, or that the optimal parameter settings may be problem-specific.

Lastly, the technique developed by Beasley *et al.* can be used to locate all minima in the function sequentially; in contrast, the technique proposed by Parsopoulos *et al.* is at best only able to locate one global minimiser. This presents a problem when the error function is truly multi-modal, *i.e.* there exists several minimisers with the same optimal function value. These different minimisers might reflect different design choices, so that they may all be of interest. Once the stretching technique has located the first global minimiser for such a function, it will effectively reduce all other minima with the same function value to isolated, discontinuous points in search space, making it almost impossible to locate them. This implies that the stretching technique can not be used to enumerate the minima of a function.

2.5.5 Dynamic Objective Functions

Much of the research regarding optimisation algorithms is performed in controlled environments, specifically with respect to the objective functions used. Usually new algorithms are compared to existing algorithms on a set of benchmark objective functions with known properties. These comparisons can provide useful insights into the performance of an algorithm, but the objective functions usually considered are all static over the duration of the experiment. Many real-world optimisation problems have objective functions that vary over time, sometimes because of changing environments, measurement error or both. Clearly an algorithm designed to be used in a real-world system should be tested in an environment most closely resembling the real-world situation.

Dynamic objective functions can be classified based on the nature of the changes over

time. Two main classes can be identified:

Noisy functions: In some applications the objective function is inherently imprecise, for example when the function value is obtained through numerical integration over a convoluted high-dimensional volume. Other examples include measurement or discretisation errors, making it impossible to approximate the gradient using finite difference methods. This type of function can be simulated by adding noise to the function values *without changing the location of the minima*. For example, a noisy function $f_\sigma(\mathbf{x})$ can be defined so that

$$f_\sigma(\mathbf{x}) = f(\mathbf{x})(1+\eta), \quad \eta \sim N(0,\sigma^2) \tag{2.32}$$

where σ can be used to control the degree of distortion.

Non-stationary functions: This class represents functions where the actual location of the minimiser in search space changes over time. A typical example might be a chemical process where the purity of some of the ingredients varies rapidly, shifting the relative concentrations of other chemicals added to the process. Maximising the yield of the reaction requires that the optimal solution is found every time that the relative purity of the input chemicals changes. Since the location of the optimiser is moving through the search space, an optimisation algorithm is expected to track it over time. This class of functions can easily be simulated by defining a function $f_{\mathbf{h}}(\mathbf{x},t)$ so that

$$f_{\mathbf{h}}(\mathbf{x},t) = f(\mathbf{x}+\mathbf{h}(t)) \tag{2.33}$$

where $\mathbf{h}$ is some vector-valued function, possibly a random walk.

Parsopoulos and Vrahatis investigated the performance of the PSO on noisy functions [100]. They used three 2-dimensional benchmark functions, transformed by rotation of the coordinate axes and by the addition of Gaussian noise in the form of equation (2.32). The PSO was not seriously affected on these functions over values of σ ranging from 0 to 0.9, indicating that the standard PSO is able to function in noisy environments without modification.

Experiments performed by Carlisle and Dozier indicated that the PSO may have trouble tracking a non-stationary minimiser [16]. They set up an experiment involving

a very simple error function of the form

$$f(\mathbf{x}) = ||\mathbf{x} - \mathbf{g}||$$

where $\mathbf{g}$ was the "goal" vector. The value of $\mathbf{g}$ was changed linearly at different fixed velocities. Using a linearly decreasing inertia weight in the range 0.65 down to 0.15, they found that the PSO was only able to track the goal for velocities up to 0.002, failing almost completely at velocities greater than 0.1.

Carlisle and Dozier proposed a method for forcing the PSO to forget the outdated information stored in the personal best position vectors $\mathbf{y}_i$. This is achieved by setting $\mathbf{y}_i = \mathbf{x}_i$ (which they referred to as "resetting" a particle) either at regular intervals, or triggered by some event. Their reasoning was that completely restarting the PSO discards too much useful information, and that a milder form of "forgetting" can be achieved by the aforementioned method. They found that resetting all the particles at regular intervals did improve the ability of the swarm to track minima (up to a velocity of 0.01), but that too-frequent resets affected the ability of the swarm to successfully locate the minimum.

They also proposed a "trigger" based method for detecting when the objective function has changed appreciably, signalling that the particles should be reset. Under some strong assumptions it was shown that the trigger method performed more consistently than the periodic approach, but still was only able to track the goal up to a velocity of 0.01.

In a further paper [17] Carlisle and Dozier implemented a version of Clerc's constriction factor instead of a linearly decreasing inertia weight. They also refined the trigger mechanism for detecting when the objective function has changed. The new method marks a particle as a *sentry*, comparing the value of the objective function of the sentry particle's position with a stored copy of the function value at this position (from the previous iteration). If these two values differ significantly, the particles in the swarm are reset.

The results obtained using the constriction factor version of the PSO were significantly better than those obtained with the linearly decreasing inertia weight. In fact, this raises the question whether the particles still have to be reset in order for the PSO to be able to track the goal. Unfortunately, Carlisle and Dozier did not present any results

of a PSO using a constriction coefficient without the particle-resetting mechanism, so this question cannot be answered based on their experiments.

This question is partially answered in a paper by Eberhart and Shi, also studying the ability of the PSO to track a dynamic minimum [36]. Unfortunately, a different objective function is used in this paper, so a direct comparison to Carlisle and Dozier's work is not possible. Eberhart and Shi used the 3-dimensional version of the spherical function in order to allow comparisons to Angeline and Bäck's earlier studies involving EP and ES algorithms, respectively. The minimiser of the function was moved linearly at velocities ranging from 0.01 up to 0.5, resulting in a total displacement of the minimiser of between 0.5 and 250 along each axis over the duration of the experiment.

Eberhart and Shi chose to use a different type of inertia weight for these experiments. The value of w is set so that

$$w = 0.5 + r(t)/2$$

where $r(t) \sim U(0, 1)$. This means that the expected value of w is 0.75, which is close to the constant value 0.729 used previously [39]. The acceleration coefficients were both set to 1.494.

The results obtained from Eberhart and Shi's experiments show that the PSO is able to track a non-stationary minimiser following a linear path to a much greater accuracy than that reported in the literature for the EP and ES algorithms. These positive results further call to question the necessity of Carlisle and Dozier's "resetting" method.

All the experiments described above involving a non-stationary minimum have one subtle flaw, though. They all assess the performance of the PSO when tracking a minimum that starts moving when the experiment starts, rather than waiting for the PSO to converge before moving the minimum. It is believed that the PSO will have trouble tracking the minimum if it had been allowed to converge, since the velocity terms of the particles may be too small. Especially in Carlisle and Dozier's case, the "resetting" technique will be of no benefit if the velocity of the particle is too small, since then $\mathbf{x}_i \approx \mathbf{y}_i$. One way of breaking free from this scenario is to randomise the positions of some (say half) of the particles in the swarm at regular intervals, or use some mechanism to determine a suitable time for re-initialisation.

2.6 Applications

The PSO has been applied to a vast number of problems, though not all of these applications have been described in published material yet. This section will briefly describe some of the applications that can be found in the literature.

Neural network training was one of the first applications of the PSO. Kennedy and Eberhart reported that the PSO was successful in training a network to correctly classify the XOR problem, a process involving the minimisation of function in a 13-dimensional search space [70]. They also reported that the PSO could train a neural network to classify Fisher's Iris Data set [44] (also available from [12]), although few details were provided. Salerno also applied the PSO to the task of training a neural network to learn the XOR problem, reporting significantly better performance than that obtained with a Gradient Descent algorithm [114]. He also showed that the PSO was able to train a simple recurrent neural network.

In fact, most PSO applications reported in the literature involve neural network training. Earlier versions of the PSO, before the introduction of the inertia weight or constriction factor, did not have the ability to perform a fine-grained search of the error surface [1]. This lead to experiments involving a hybrid between PSO and traditional gradient techniques. Van den Bergh used the PSO to find a suitable starting position for the Scaled Conjugate Gradient algorithm [135]. Results showed that the hybrid method resulted in significantly better performance on both classification problems, using the UCI Ionosphere problem [12] as example, and function approximation problems, using the henon-curve time series as example.

Later examples of neural network training include [34]. Here Eberhart and Hu used the PSO to train a network to correctly classify a patient as exhibiting *essential tremor*, or suffering from Parkinson's Disease. Their PSO implementation used an inertia weight that decreased linearly from 0.9 to 0.4 over 2000 iterations. An interesting feature of the neural network they used was that they trained the slope of the sigmoidal activation functions along with the weights of the network. The slope of a sigmoidal unit is determined by the parameter γ, where

$$\text{sig}(x) = \frac{1}{1 + \exp(-\gamma x)} \tag{2.34}$$

If γ is very large, the sigmoid function approximates the Heaviside step function, and can thus be replaced with one, resulting in a computationally more efficient network. Another benefit of the slope parameter is that the inputs to the neural network no longer have to be rescaled, since the γ parameter performs the required scaling. This technique is discussed in more detail in [35].

The PSO has also been used to evolve the architecture of a neural network in tandem with the training of the weights of the network. Zhang and Shao report that their PSONN algorithm [146] was able to successfully evolve a network for estimating the quality of jet fuel. Part of the PSONN system involves the optimisation of the number of hidden units used in the network — a task that is handled by a PSO. Whenever a new hidden node is added to the network, only the newly added nodes are trained (again using a PSO) in an attempt to reduce the error, greatly improving the speed of the algorithm. Unfortunately they do not provide details on which type of PSO they used, or how they represented the discrete numbers required for the description of the network architecture.

Product Unit Neural Networks (see Chapter 6) are a type of neural network with an error function that is notoriously hard to train. Many of the gradient-based optimisation techniques become trapped in the numerous local minima present in the search space. Engelbrecht and Ismail studied the ability of various optimisation algorithms to train product unit networks [41]. In this study, the PSO was found to perform better than random search, a GA and the LeapFrog algorithm [126, 127] on several product unit network training problems.

Eberhart *et al.* describe several other applications of the PSO in [35], including some more neural network training applications. Tandon used the PSO to train a neural network used to simulate and control an end milling process [133]. End milling involves the removal of metal in a manufacturing process, using computer numerically controlled (CNC) machine tools. Another neural network training application described by Eberhart *et al.* is that of training a network to estimate the state-of-charge of a battery pack [65].

Yet another neural network training application, this time using a Fuzzy Neural Network, was studied by He *et al.* [60]. One of the more interesting points regarding

their research was the fact that they modified the velocity update equation so that it is no longer accumulative, *i.e.*

$$\begin{aligned} v_{i,j}(t+1) &= c_1 r_{1,j}(t)[y_{i,j}(t) - x_{i,j}(t)] + \\ & \quad c_2 r_{2,j}(t)[\hat{y}_j(t) - x_{i,j}(t)] \end{aligned}$$

Results are presented showing that this modified velocity update equation gave rise to improved performance on some benchmark functions. Note that this modification makes their version of the PSO very similar to an Evolution Strategies algorithm. They also showed that their fuzzy neural network was able to produce a set of 15 rules with a classification accuracy of 97% on the Iris data set. This is a rather large number of rules for such a simple classification task, compared to other efficient algorithms [139].

At least one application unrelated to neural network training is found in [35]. The PSO was used to optimise the ingredient mix of chemicals used to facilitate the growth of strains of microorganisms, resulting in a significant improvement over the solutions found by previous optimisation methods. One of the strengths of the PSO is the ability to explore a large area of search space relatively efficiently; this property led the PSO to discover a better solution in a location in search space very different from the solutions discovered by other existing techniques.

Another application unrelated to neural network training was published by Fukuyama and Yoshida [50]. They have shown that the PSO is very effective at optimising both continuous and discrete variables simultaneously. The PSO velocity update equation can be adapted for use with discrete variables by simply discretising the values before using them in the velocity update step (using for example equation 2.7). The position of the particle is also discretised after equation (2.8) has been used to update it. These discrete variables can be mixed freely with the continuous variables, as long as the appropriate (discretised or continuous) update equations are applied to them. The application on which Fukuyama and Yoshida demonstrated their modified PSO was that of calculating the correct response to changes in the load on an electric power grid. This problem requires the simultaneous optimisation of numerous discrete and continuous variables, and was traditionally solved using the Reactive Tabu Search (RTS) algorithm. The RTS

algorithm scales very poorly with problem dimensionality, since the number of candidates for evaluation increases exponentially with the dimension of the problem. To illustrate, the RTS algorithm required 7.6 hours to solve a problem consisting of 1217 busses. The same problem was solved in only 230 seconds using a PSO with a linearly decreasing inertia weight in the range $[0.9, 0.4]$.

Fukuyama *et al.* also applied Angeline's Hybrid PSO [2] to the same problem [49], finding that the Hybrid algorithm appears to be even more effective. They further report that the same parameter settings for the PSO consistently produced high quality solutions for different power system problems.

2.7 Analysis of PSO Behaviour

This section will briefly discuss various previous publications concerned with formal analyses of the behaviour of the PSO algorithm. The focus of much of this work was the convergence behaviour of the PSO, since no formal proof of convergence existed at the time of their publication.

Currently there are numerous versions of the PSO that can be classified according to their convergence behaviour, including the original PSO, the PSO with an inertia weight and the PSO with a constriction coefficient. Several other PSO variants have been described in Section 2.5, but the focus here will be on the simpler models that have been analysed in terms of their convergence behaviour.

Before the different models and their analyses are presented, a definition of convergence is in order. Many of the authors of the papers cited below use the term 'convergence' to mean that the algorithm makes progress toward a terminal state. In effect, this means that the magnitude of the changes in the positions of the particles in the swarm diminishes over time. This should not be confused with the concept of "convergence onto a local minimum." Formally, a sequence $\{\mathbf{z}_k\}_{k=1}^{+\infty}$ is said to converge onto a minimiser (local or global) $\mathbf{x}^*$ if

$$\lim_{k \to +\infty} \mathbf{z}_k = \mathbf{x}^*$$

For example, to show that the PSO can successfully locate a local minimum $\mathbf{x}_B^*$ in the

region B, one would have to show that

$$\lim_{k \to +\infty} \hat{\mathbf{y}}_k = \mathbf{x}_B^*$$

That is, the sequence of global best positions of the swarm, over time, must form a sequence that converges onto $\mathbf{x}_B^*$. None of the analyses presented in the sections below prove anything regarding this type of convergence.

Instead, some of the analyses found in the literature prove a considerably weaker condition, namely

$$\lim_{k \to +\infty} \hat{\mathbf{y}}_k = \mathbf{p}$$

where $\mathbf{p}$ is an arbitrary point in the search space *not* guaranteed to coincide with the global minimum, nor any local minimum, for that matter. While this type of convergence is not sufficient to show that the PSO is a local (or global) minimisation algorithm, it is a necessary condition in order for the algorithm to terminate in a finite number of steps.

Original PSO particle trajectory

Ozcan and Mohan have published the first mathematical analyses regarding the trajectory of a PSO particle [91, 90]. The model that they based their analysis on was the original PSO, without an inertia weight or a constriction coefficient.

To make the problem more tractable, the stochastic components of the update equations, as well as the personal best position of the particle, y_i, were held constant. If the analysis is restricted to one dimension, the subscript j can be omitted, thus simplifying the notation. From this simplified model Ozcan and Mohan derived a recursive form of the PSO position update equation:

$$x_i(t) - (2 - \phi_1 - \phi_2)x_i(t-1) + x_i(t-2) = \phi_1 y_i + \phi_2 \hat{y}$$

Note that $\phi_1 = r_2(t)c_1$, and $\phi_2 = r_2(t)c_2$. That is, they are specific instances of the stochastic variables, held constant during the analysis. The resulting non-homogeneous recursion relation can be solved to obtain a closed form equation of the form

$$x_i(t) = \Lambda_i \sin(\theta_i t) + \Upsilon_i \cos(\theta_i t) + \kappa_i \tag{2.35}$$

where Λ_i, θ_i, Υ_i and κ_i are constants derived from the initial conditions and the values of ϕ_1 and ϕ_2, where $0 < \phi_1, \phi_2 < 2$. The reader is referred to [91] for the details of the derivation.

What Ozcan and Mohan discovered is that the trajectory of a particle (in the absence of stochastic influences) traces out a sinusoidal waveform. The frequency and the amplitude of the waveform depends on the initial position, initial velocity and the values of ϕ_1 and ϕ_2. When the stochastic component is taken into account, the values of ϕ_1 and ϕ_2 change during every iteration, thus affecting the amplitude and frequency of the waveform. To quote Ozcan and Mohan [91]:

> "We have shown that in the general case, a particle does not 'fly' in the search space, but rather 'surfs' it on sine waves. A particle seeking an optimal location attempts to 'catch' another wave randomly, manipulating its frequency and amplitude."

Note that equation (2.35) does not take into account the influence of clamping the velocity to the interval $[-v_{max}, v_{max}]$. The equation does, however, help in understanding the purpose of v_{max}. If the values of ϕ_1 and ϕ_2 are chosen randomly, then the amplitude of the sine wave may become excessively large, allowing the particle to wander too far from the intended search space. By applying the v_{max} clamping the particle cannot move too far in a single iteration, however, over several iterations it may still escape.

Ozcan and Mohan did not present an in-depth analysis of the influence of the inertia weight, nor did they address the issue of convergence. Chapter 3 presents an analysis, involving an inertia weight, that has been derived independently.

Constricted PSO particle trajectory

Clerc and Kennedy published a paper describing several different versions of the PSO, all guaranteed to converge [21]. Convergence for these models is guaranteed through the use of a *constriction coefficient*. In order to simplify the derivation of the equations below, Clerc chose to use an alternate representation for the PSO update equations. Initially, the same assumptions that Ozcan and Mohan made apply, *i.e.* a single particle in one dimension, with a constant ϕ_1, ϕ_2, y and $\hat{y}$ is considered. To shorten the notation even

further, Clerc combined y and $\hat{y}$ into a single point p, so that

$$p = \frac{\phi_1 y + \phi_2 \hat{y}}{\phi_1 + \phi_2} \tag{2.36}$$

The derivation of this property is not provided in [21], however, an independently derived proof of this fact is presented in Chapter 3. Using this simplified notation, the PSO equations are reduced to

$$v(t+1) = v(t) + \phi(p - x(t)) \tag{2.37}$$

$$x(t+1) = x(t) + v(t+1) \tag{2.38}$$

where $\phi = \phi_1 + \phi_2$. This representation is then transformed to the discrete-time dynamic system

$$v_{t+1} = v_t + \phi z_t \tag{2.39}$$

$$z_{t+1} = -v_t + (1 - \phi) z_t \tag{2.40}$$

where $z_t = p - x_t$. Clerc shows that the convergence behaviour of the system is governed by the two eigenvalues, denoted e_1 and e_2, of this system. The system converges when $\max(|e_1|, |e_2|) < 1$. A second system of equations can be defined so that its eigenvalues, denoted e_1' and e_2', are always less than one. This is achieved by defining the system so that $e_1' = \chi_1 e_1$ and $e_2' = \chi_2 e_2$, where several methods exist for determining the values of χ_1 and χ_2. In order to gain more control over the convergence behaviour of the system, Clerc introduced five control coefficients $\alpha, \beta, \gamma, \delta$, and η, resulting in the generalised system

$$v_{t+1} = \alpha v_t + \beta \phi z_t \tag{2.41}$$

$$z_{t+1} = -\gamma v_t + (\delta - \eta \phi) z_t \tag{2.42}$$

Different classes of systems have been identified, based on the choice of parameters. They are:

Class 1 model: This class is characterised by the following relations

$$\alpha = \delta$$

$$\beta\gamma = \eta^2$$

Note that these conditions are satisfied by the relation $\alpha = \beta = \gamma = \delta = \eta$. The constricted version of this model can be implemented using

$$\begin{aligned} v(t+1) &= \chi\Big(v(t) + \phi z(t)\Big) \\ z(t+1) &= -\chi\Big(v(t) + (1-\phi)z(t)\Big) \end{aligned}$$

where

$$\chi = \frac{2\kappa}{\left|1 - \phi - \sqrt{\phi^2 - 4\phi}\right|}, \ \kappa \in (0,1)$$

Class $1'$ model: Related to the class 1 model, this class is defined by the relations

$$\begin{aligned} \alpha &= \beta \\ \gamma &= \delta = \eta \end{aligned}$$

Under the condition $\chi_1 = \chi_2 = \chi$, this system has solutions in the form

$$\alpha = (2-\phi)\chi + \phi - 1$$

$$\gamma = \chi \text{ or } \gamma = \frac{\chi}{\phi - 1}$$

The constricted version of this model can be implemented using

$$\begin{aligned} v(t+1) &= \chi\Big(v(t) + \phi z(t)\Big) \\ z(t+1) &= -v(t) + (1-\phi)z(t) \end{aligned}$$

where

$$\chi = \frac{2\kappa}{\left|1 - \phi - \sqrt{\phi^2 - 4\phi}\right|}, \ \kappa \in (0,1), \ \phi \in (0,2)$$

Class $1''$ model: Another related model, defined by

$$\alpha = \beta = \gamma = \eta$$

where

$$\alpha = \frac{2\delta + (\chi_1 + \chi_2)(\phi - 2) - (\chi_1 - \chi_2)\sqrt{\phi^2 - 4\phi}}{2(\phi - 1)}$$

Usually, the additional constraint $\delta = 1$ is used with this system. The constricted class $1''$ model follows:

$$\begin{aligned} v(t+1) &= \chi\Big(v(t) + \phi z(t)\Big) \\ z(t+1) &= -\chi v(t) + (1 - \chi\phi) z(t) \end{aligned}$$

where

$$\chi = \begin{cases} \sqrt{\frac{2\kappa}{\left|\phi - 2 + \sqrt{\phi^2 - 4\phi}\right|}} & \text{if } \phi > 4 \\ \sqrt{\kappa} & \text{otherwise} \end{cases}$$

where $\kappa \in (0, 1)$.

Class 2 model: The second class of models is defined by

$$\begin{aligned} \alpha &= \beta = 2\delta \\ \eta &= 2\gamma \end{aligned}$$

When $\chi_1 = \chi_2 = \chi$, and $2\gamma\phi \geq \delta$, then

$$\delta = \chi \frac{2 - \phi + \sqrt{\phi^2 - 4\phi}}{2}$$

$$\gamma = \chi \frac{2 - \phi - 3\sqrt{\phi^2 - 4\phi}}{4\phi}$$

These different constriction models all have the same aim: Prevent the velocity from growing without bound, causing the systems to "explode". The differences between the models described by Clerc (reproduced above) lie mainly in their rates of convergence. On a unimodal function a higher rate of convergence usually improves performance; in contrast, a function with multiple local minima requires more exploration, thus slightly slower convergence. A major benefit of using any of the constricted models above is that the velocity of a particle no longer has to be clamped to the range $[-v_{max}, v_{max}]$ in order for the PSO to converge. Since the optimal value of v_{max} is problem dependent, this constriction approach generalises the PSO.

Clerc proposed the following explicit forms for the PSO update equations, derived from the recurrence relations presented in equations (2.39) and (2.40).

$$v(t) = k_1 e_1^t + k_2 e_2^t \tag{2.43}$$

$$z(t) = \frac{1}{\phi}\Big(k_1 e_1^t (e_1 - 1) + k_2 e_2^t (e_2 - 1)\Big) \tag{2.44}$$

where k_1 and k_2 are constants determined by the initial conditions. Note that these forms place no restriction on the value of t. That is, t does not have to be an integer. Since the eigenvectors are complex numbers, a non-integer value of t results in a $v(t)$ and a $z(t)$ with non-zero imaginary components. The trajectory of a particle in continuous time is thus a curve in a 5-dimensional space defined by $(\mathcal{R}e(v(t)), \mathcal{I}m(v(t)), \mathcal{R}e(z(t)), \mathcal{I}m(z(t)), t)$. The sinusoidal waveforms observed by Ozcan and Mohan correspond to the trajectory of the particle for integer values of t.

Several issues regarding PSO convergence remain unaddressed, though. The model proposed by Clerc does not fully explain the interaction between the different particles. Further, the convergence obtained by using the constriction coefficients does not guarantee that the PSO will converge on a global (or even local) minimum (as will be shown in Section 3.3), in other words, Clerc's constriction coefficients produce a sequence of $\{\hat{\mathbf{y}}_k\}_{k=1}^{+\infty}$ values so that

$$\lim_{k \to +\infty} \hat{\mathbf{y}}_k = \mathbf{p}$$

where $\mathbf{p}$ is not guaranteed to be a minimiser of the objective function.

2.8 Coevolution, Cooperation and Symbiosis

This section briefly introduces the topic of coevolutionary and cooperative evolutionary algorithms, serving as background material for the development of the new algorithms introduced in Chapter 4.

The evolutionary metaphor used in Evolutionary Algorithms can be extended to model several different types of organisms (species) living in the same environment, *i.e. sympatric* populations. When there's more than one type of organism they can have several types of relationships:

Competition: This type of inter-species interaction is sometimes referred to as the *Predator-prey* relationship. This name clearly alludes to the fact that one of the species will improve itself at the cost of the other species. If all the species are actively trying to improve their performance relative to the others, the fitness of all the species may improve — unless a species is driven to extinction. An example from nature would be the relationship between the Cheetah and the Impala: if the Cheetah evolves to run even faster, the Impala have to respond by either also evolving to run faster, or to become more alert (*e.g.* improved hearing). If the predators become too plentiful, the prey will be exhausted, so that an increase in the number of predators does not imply an increase in the number of prey, but rather *requires* an increase in the number of prey. Artificial predator-prey relationships (as used in EAs) do not necessarily possess this last property.

Symbiosis: When an increase in fitness of one species also leads to an increase in fitness of another species, they are said to have a symbiotic relationship. An example from nature would be the relationship between some insects and flowers: the insects help the flowers to distribute their pollen, while the insects obtain nourishment from the nectar provided by the flowers. When there are more insects, more flowers can be pollinated, leading to a larger population of flowers that can in turn sustain a larger population of insects.

The following sections will review some implementations of coevolutionary algorithms.

2.8.1 Competitive Algorithms

The design of a competitive coevolutionary algorithm usually starts with the design of a *competitive fitness function* [3]. Such a competitive fitness function measures how well a solution scores when competing with another individual. When dealing with a single population, the following competition patterns have been suggested:

'All versus all': Each individual is tested on all other population members;

Random competition: This pattern tests each individual on a number of randomly-chosen population members;

Tournament competition: Another form of probabilistic competition, where members compete in rounds using a relative fitness function. This type of fitness function rates the fitness of an individual relative to the other, and is therefore not a global measure of an individual's fitness;

'All versus best': All population members are tested with the current best (most fit) individual.

Since a single invocation of the fitness function only ranks the two participating individuals, one of the above strategies has to be used to rank the whole population. One of the best-known uses of a competitive fitness function is Axelrod's experiments with the *iterated prisoner's dilemma* [5, 6].

All these competition patterns, except tournament competition, can be extended to deal with multiple species. Hillis presented one example of such a multi-species implementation, where he used predator-prey coevolution to evolve sorting networks [61]. The first population he used consisted of sorting networks, which were pitted against the test-lists evolved by the second population. The sorting networks received as a score the percentage of test lists they were able to sort correctly; the test lists are scored according to the number of sorting networks that they've defeated. Hillis has shown that the coevolutionary approach produced better sorting networks than comparable non-coevolutionary algorithms. The algorithm was also more efficient, since the test-list population was typically smaller than the test set that would have been used in a non-coevolutionary implementation. This was because the test-list population focused on the regions of search space that were harder for the sorting networks to solve.

Coevolutionary Genetic Algorithms

Paredis introduced the term "test-solution problems" [95] as a generalisation of the approach that Hillis followed for evolving his sorting networks. This term refers to a problem that has a solution that must satisfy certain *a priori* criteria. One of the earliest implementations of a Coevolutionary Genetic Algorithm (CGA) is also due to Paredis. He used the CGA to train a neural network to perform a classification task [93]. The first population represents the network; the second population represents the test

data for the classification problem. Each time the network misclassified a test pattern, that individual in the test data population was awarded a fitness point. This way the CGA procedure automatically pares down the test set to the patterns that were hardest to classify (these usually represent the boundary cases).

Other Constraint Satisfaction Problems (CSP) have also been successfully solved using a CGA, again finding high-quality solutions more efficiently than traditional single-population GAs [92].

Niching Genetic Algorithms

Fitness sharing, developed by Goldberg and Richardson [53], is another approach to preserve population diversity, eventually leading to the evolution of different competing species. Fitness sharing encourages the individuals to find their own ecological niches by modifying the fitness function to incorporate information regarding individuals already inhabiting that region of search space. A sharing function is defined as

$$\text{share}(d_{ij}) = \begin{cases} 1 & \text{if } d_{ij} = 0 \\ 1 - \left(\frac{d_{ij}}{\sigma_s}\right)^{\alpha} & \text{if } d_{ij} < \sigma_s \\ 0 & \text{otherwise} \end{cases}$$

where σ_s is an adjustable parameter indicating the radius (in the search space) of the niche, d_{ij} represents the distance in search space between individuals i and j, and α is an adjustable decay rate.

The modified fitness of an individual can then be computed using

$$f_i' = \frac{f_i}{\sum_{j=1}^{n} \text{share}(d_{ij})} \tag{2.45}$$

The effect of fitness sharing is that an individual will score lower when it approaches another population member already occupying that region of search space. Depending on the decay parameter α and the niche radius σ_s, several individuals may occupy a specific niche, however. By forcing the individuals to look elsewhere (when a niche is already filled), diversity is increased. From equation (2.45) it is clear that a niche containing individuals with very high fitness values will be able to accommodate more

individuals, which is intuitively satisfying when one compares this phenomenon to its biological counterpart.

Note that this is a competitive fitness scheme, since the individuals compete for positions in the existing niches, or they have to go and establish their own niches. If an individual joins a niche, all the other members of that niche will suffer a decrease in their effective fitness values, as computed by equation (2.45).

Another related niching technique, known as *crowding*, has been introduced by De Jong [28]. This approach uses a *steady state* GA, which is similar to a $(\mu + 1)$ Evolution Strategy. One offspring is produced and compared with every element in the existing population, replacing the individual it most closely resembles. This resemblance measure can be computed in genotypic or phenotypic space. The idea behind the crowding technique is to preserve the existing diversity in the population. Note that crowding does not take into account the number of individuals already occupying that region of space, so that there is no competition or associated drop in effective fitness. This implies that crowding is not a competitive scheme.

Lastly, Beasley *et al.* [10] have suggested a sequential niching technique. This approach is discussed in more detail in Section 2.5.4.

2.8.2 Symbiotic Algorithms

Paredis's CGA has also been employed in a symbiotic environment to co-evolve the representation used by the GA concurrently with the process of solving the problem itself [94, 96]. This is a particularly useful approach, since the optimal representation for a specific problem is not usually known in advance. Many other symbiotic algorithms have been proposed, some of which are discussed next.

Parallel Genetic Algorithms

Parallel GAs sprung forth from the desire to implement GAs on multi-processor machines. Two approaches to parallelising a GA have been proposed by Gordon and Whitley [56]. The first is a straightforward implementation on a multi-processor, shared memory system. This model, sometimes called a *global* model [15], allows recombination between any two individuals, and is thus semantically equivalent to a normal GA.

A second model is called the *island model* (or, alternatively, the migration or coarse-grained model), first introduced by Grosso [59]. This model maintains separate subpopulations (on different machines) where each subpopulation functions as a normal GA population within itself, *i.e.* crossover and mutation operators are applied on elements in the subpopulation. From time to time individuals are allowed to migrate from one subpopulation to another, thus sharing information between different subpopulations. This model reduces the required inter-processor communication bandwidth significantly, and manages to maintain greater diversity throughout all the subpopulations. This model is cooperative, since the different subpopulations do not affect each other negatively, nor do they compete directly for resources.

A third model, known as the *neighbourhood model* (or, alternatively, as the diffusion or fine-grained model), makes use of overlapping subpopulations [88, 80]. Different neighbourhood topologies can be used, but the idea is a straightforward extension of the island model with some individuals belonging to more than one subpopulation at any given moment. This model clearly increases the inter-processor communication requirements, but it may converge faster than the island model [55]. Because there's more interaction between the subpopulations, they all benefit whenever another subpopulation discovers a good solution, resulting in cooperative behaviour.

In these last two models the subpopulations are sometimes referred to as *demes*, a term used in biology to describe such closely-related subpopulations.

Lastly, a fourth model, called the *hybrid* model, has been described in [15]. This model uses any combination of the first three methods mentioned above, and has been shown to offer some advantages. It should be noted that the level of symbiosis in the parallel GA is quite low, since the subpopulations are arguably of the same species.

Cooperative Coevolutionary Genetic Algorithms

The Cooperative Coevolutionary Genetic Algorithm (CCGA) was first introduced by Potter [106, 105]. Potter hypothesised that to evolve more complex structures the notion of modularity should be made explicit. The model he proposed was that complex solutions should be evolved in the form of interacting co-adapted subcomponents. Each subcomponent is represented by a separate species, evolving in its own private GA.

Each species represents only a partial solution to the more complex problem, in fact, the fitness of a subcomponent cannot be evaluated unless there is some associated context. The context is built up by selecting representative members of each of the other subpopulations and constructing a template solution vector missing exactly one subcomponent. Using this template, each individual from a specific species can be evaluated in the context of the template solution vector to compute its fitness.

Consider, for example, a problem with a 10-dimensional parameter space, where the vector can be partitioned into 10 populations corresponding to the 10 parameters. Each population contains a number of individuals representing possible solutions for that specific parameter. A potential solution to the 10-dimensional problem can be constructed by selecting one individual from each of the 10 populations, each one corresponding to a dimension of the vector. The fitness of this vector can then be computed using the objective function of the problem. Now consider, for example, the 8^{th} dimension of the search space, corresponding to the 8^{th} population. To compute the fitness of each individual in this population, we substitute the individual into the 8^{th} dimension of the template vector, which is then evaluated. In the simplest case, the fitness of each individual can be set to the fitness obtained by substituting it into the template solution vector (see Section 2.9 for a detailed discussion of alternatives).

How to build a good template vector, representing a favourable context, is not immediately clear. A greedy approach is to use the best individual from each population as a representative for the corresponding parameter in the template vector. Potter called this algorithm CCGA-1. A second algorithm, CCGA-2, considered two contexts for each fitness evaluation: one using the 'greedy' template vector as context, and a second template vector constructed by selecting one random individual from each population. The individual's fitness is then set to the better of that obtained using the greedy and random contexts.

This dimension-based partitioning technique was applied to compare the CCGA-1 algorithm to a standard GA when optimising functions with multiple local minima [106]. On separable functions the CCGA-1 algorithm succeeded in locating the minima of the functions significantly faster than the standard GA. The performance of CCGA-1 was mixed on non-separable functions, performing less well on one of two such functions

tested (Rosenbrock's function). The CCGA-2 algorithm, using a randomly constructed context vector, performed better than CCGA-1 on Rosenbrock's function.

Note that the approach proposed by Potter is more general than simply partitioning the parameter vector into its constituent components. In later work, Potter applied the CCGA algorithm to a string covering problem [105]. The purpose of this experiment was to determine whether the different species can cooperate so that the best solution produced by each population is distinct from that of another population. In other words, if the problem requires 10 different strings to cover (match) the target set, will the CCGA algorithm be able to find the 10 targets using only 10 populations? The results obtained by Potter indicated that CCGA was able to locate many of the required environmental niches, showing that evolutionary pressure alone was sufficient to coerce the species to cooperate. The CCGA technique has also been applied successfully to evolve a cascade neural network [105].

Note that cooperation is achieved by partitioning the search space into disjoint subspaces. The results obtained by one subpopulation in one of the subspaces influences the characteristics of the search space as seen by the other subpopulations in their own subspaces. The subpopulations thus cooperate by mutually focusing their efforts on more promising regions of the search space.

Blackboard Information Sharing

Clearwater *et al.* [19] define cooperation as follows:

> *Cooperation involves a collection of agents that interact by communicating information to each other while solving a problem.*

and they further state

> *The information exchanged between agents may be incorrect, and should sometimes alter the behaviour of the agent receiving it.*

This is a rather general framework within which a wide range of algorithms can be classified as being cooperative. For example, even a standard GA can be viewed as a cooperative algorithm, rather than a competitive algorithm, using Clearwater's definition. If individuals in a GA population are seen as agents, and the crossover operator

as information exchange, then the GA is certainly a cooperative algorithm. This view is shared by other researchers as well, showing that under the assumption of certain parameter settings, the GA is indeed a cooperative learner [22].

Another form of cooperation, as used by Clearwater *et al.* [19], is the use of a "blackboard". This device is a shared memory where agents can post hints, or read hints from. It is possible that an agent can combine the hints read from the blackboard with its own knowledge to produce a better partial solution, or hint, that may lead to the solution more quickly than the agent would have been able to discover on its own. Using the blackboard model they have been able to show a super-linear increase in speed by increasing the number of agents, given that the agents were sufficiently diverse. The increase is attributed to the large "jumps" in search space that cooperation facilitates, sometimes with detrimental effect, but usually improving the exploration ability of the search algorithm. If the agents were non-diverse, only a linear speed-up was observed.

2.9 Important Issues Arising in Coevolution

New challenges are introduced when a problem is to be solved using a cooperative coevolutionary approach. Potter suggested several categories of issues that must be addressed by the algorithm [105]. These categories are briefly outlined next.

2.9.1 Problem Decomposition

The divide-and-conquer strategy used in some sorting algorithms (*e.g.* merge-sort) is a prime example of a successful *problem decomposition*. For example, using merge-sort, the list of values is split into to shorter sublists, recursively, until only two elements remain in each sublist. These sublists are easy to sort, and it's almost a trivial task to combine two sorted lists so that their union remains sorted. Experience and knowledge of the domain led to the discovery of the merge-sort algorithm's successful decomposition into simpler subtasks.

An example of a decomposition in an optimisation problem is that of the *relaxation method* [129, 48]. This method can be used to solve an optimisation problem in n variables by holding $n-1$ of the variables constant while optimising the remaining one.

This process is repeated by cycling through the combinations of $n-1$ variables that can be held constant. Effectively the problem of optimising n variables has been reduced to n subtasks of optimising a single variable, thereby reducing the complexity of the original problem.

Unfortunately, little is known about many of the complex problems that we may encounter, making it difficult to find such highly effective decompositions. When the variables involved in a multi-dimensional optimisation problem are independent a component-wise decomposition like the relaxation method can easily be applied. Many optimisation problems, however, involve variables with non-linear dependencies, making decomposition a difficult if not impossible task.

Ideally, the optimisation algorithm must be able to evolve the most efficient decomposition as part of the learning process.

2.9.2 Interdependencies Between Components

A function f in an n-dimensional optimisation problem is said to be separable if it can be rewritten as the sum of n functions, each involving only a single component of the vector, so that

$$f(\mathbf{x}) = g_1(x_1) + g_2(x_2) + \cdots + g_n(x_n)$$

Such problems can easily be decomposed into n independent problems, where the solution of f will coincide with the vector obtained by concatenating the n solutions of the g_i functions. Separable functions are the one extreme, representing problems with no interdependencies between the components. For example, the function

$$f(\mathbf{x}) = x_1^2 + x_2^2$$

clearly has no interdependencies between its components. It can be decomposed into two separate problems, *e.g.*

$$\begin{aligned} f(\mathbf{x}) &= g_1(\mathbf{x}) + g_2(\mathbf{x}) \\ g_1(\mathbf{x}) &= x_1^2 \\ g_2(\mathbf{x}) &= x_2^2 \end{aligned}$$

Clearly, the minimum of g_1 can be found independently of g_2, so that the value of x_1 that minimises g_1 will coincide with the value of x_1 that forms part of the minimiser of f.

A problem that appears no more difficult to solve (by looking at the graphical plots) can be constructed by adding a product term, so that

$$f(\mathbf{x}) = x_1^2 + x_2^2 + 0.25(2 - x_1)(1 - x_2)$$

This is still a relatively easy problem, but note that a change in the value of x_1 affects the term $0.25(2 - x_1)(1 - x_2)$, possibly requiring that a new optimal value for x_2 must be computed. One possible attempt at decomposing the function is

$$\begin{aligned} f(\mathbf{x}) &= g_1(\mathbf{x}) + g_2(\mathbf{x}) \\ g_1(\mathbf{x}) &= x_1^2 + 0.25(2 - x_1)(1 - x_2) \\ g_2(\mathbf{x}) &= x_2^2 + 0.25(2 - x_1)(1 - x_2) \end{aligned}$$

where x_2 and x_1 are treated as constants in g_1 and g_2 respectively. This implies that the minimiser of g_1 is influenced by the (constant) value of x_2, and g_2's minimiser will be influenced similarly by x_1. Clearly the minimiser of f cannot be found in one iteration by simply minimising g_1 followed by g_2; multiple iterations will be required.

The interdependencies between variables can be significantly more complex than illustrated in this simple example. It is usually not possible to determine the relationship between the variables beforehand, especially if the interdependencies are highly non-linear in nature.

2.9.3 Credit Assignment

The credit assignment problem is an important hurdle to overcome in the design of a coevolutionary problem, since it increases the complexity of measuring the fitness of an individual solution. The *credit assignment* problem can be defined as follows:

> Given a solution $\mathbf{x}$ consisting of n subcomponents, and a fitness f associated with $\mathbf{x}$, what percentage of the credit associated with f must each of the subcomponents receive for their contribution towards $\mathbf{x}$?

Consider, for example, a chemical reaction yielding y units of the desired product per time unit per unit measure of reacting substances. If the temperature of the reaction is increased and a catalyst is added, production is increased to $2y$. Did the reaction produce a higher yield because of the increased temperature, the addition of the catalyst, or both? Assuming that both the increased temperature and the catalyst are responsible, what is the relationship between the addition of the catalyst and the resulting increase in reaction yield? How much of the increased yield can be attributed to the increased temperature?

Clearly, these questions cannot be answered without further knowledge about the reaction or without running controlled experiments. It may even be the case that the combination of the catalyst and the increased temperature results in a larger yield than either of them produces alone.

Decomposing a complex solution into interacting subcomponents results in a credit assignment problem similar to the chemical reaction example described above. Many evolutionary algorithms (*e.g.* GAs) base their decisions on the fitness of an individual, typically by allowing more copies of the fit individual to survive into the next generation. Consider the following problem: Individuals of a population represent subcomponents of a more complex problem. The fitness of a subcomponent cannot be evaluated directly, but only when viewed in the context of the complete solution vector, consisting of several subcomponents. How much credit should an individual, representing only one possible choice for a specific subcomponent, receive for its contribution to the solution?

Usually this information is not available in advance, nor is it feasible to run controlled experiments to determine the exact solutions for a specific credit assignment problem. Some acceptable approximate solution to the credit assignment problem must be considered in order to construct a practical algorithm for coevolving subcomponents.

2.9.4 Population Diversity

It is important that the population of potential solutions remain sufficiently diverse in order for them to remain representative. Consider the coevolution of sorting networks and test-lists, like those studied by Hillis [61]. If the test-list population discovers a particular pattern, say pattern a, that is difficult for the sorting networks to sort, then the

test-list population will gradually evolve to contain mostly minor variations on pattern a. The sorting networks evolve so that they can defeat as many individuals in the test-list population as possible. The test-list population, however, is no longer representative of the original problem (that of sorting arbitrary lists), but has converged on a small subset of the original problem space surrounding pattern a. Even though the sorting networks will appear to continue to improve over previous generations, they will no longer be quality solutions to the original problem.

To prevent the coevolutionary system from converging on a subset of the problem space the diversity of all the populations involved must be maintained. Several techniques have been proposed for GA, including crowding and niching, as described in Section 2.8.

2.9.5 Parallelism

Several models suggested for extending the GA to exploit multi-processor machines have been discussed in Section 2.8. The island model, in particular, appears to be ideally suited to a problem that can be decomposed into subcomponents. Each species, corresponding to a subcomponent of the solution vector, can be evolved on a separate processor. Note that the island model, in this sense, is not limited to genetic algorithms, but can be applied to the PSO or other evolutionary algorithms.

The original island model allowed migration between islands relatively infrequently. The island model can be adapted to support a coevolutionary approach where each population represents a subcomponent of the solution. In order to evaluate the fitness of an individual in such a subcomponent population, a template vector consisting of representatives from the other populations must be available. The vector can be maintained centrally, but this would result in unwanted overhead since each species would have to access it every time they want to compute the fitness of an individual. A better approach would be to keep a copy of this template vector on every machine, updating all template vectors when necessary using broadcasts.

If the template vector has to be updated frequently the resulting broadcast traffic could become a problem. A topic for future research is an investigation of the influence that network delays will have on the performance of such a cooperative algorithm implemented in a networked, multi-computer environment.

2.10 Related Work

Much of the work in the field of optimisation is related to the topics investigated in this thesis. A vast number of new algorithms can be constructed by simply connecting different optimisation algorithms in the correct way. For example, a GA can be used during the early stages of the optimisation process, switching over to say the Scaled Conjugate Gradient (SCG) algorithm to perform a rapid local search to refine a solution when necessary. Replacing the Scaled Conjugate Gradient algorithm with some other local method yields yet another new hybrid algorithm. This example illustrates how a local method (the SCG algorithm) can be made a global optimisation algorithm with the aid of a GA. Such hybrid algorithms are closely related to Memetic Algorithms [24]. One algorithm's weakness may be another's strength, so that no method can be said to be irrelevant in the study of optimisation algorithms.

Chapter 3 investigates the convergence characteristics of the PSO algorithm. Some of the analysis techniques used in that chapter are based on previous work by Ozcan and Mohan [91], as well as work by Clerc [21]. Chapter 3 further introduces a new PSO algorithm with guaranteed convergence on local minima. This algorithm is extended, resulting in a PSO algorithm with guaranteed convergence on global minima. Formal proofs are presented to support these claims, using the technique for proving convergence properties of stochastic algorithms used by Solis and Wets [128].

Chapter 4 investigates several techniques for constructing cooperative PSO algorithms. Potter introduced a novel technique for designing a cooperative Genetic Algorithm, called CCGA [106]. Potter postulated that his model for cooperation could be extended to include many other Evolutionary Algorithms. The first cooperative PSO introduced in Chapter 4 was inspired by Potter's model. Other researchers have since extended the CCGA-approach to other types of search algorithm, including a cooperative algorithm based on generation stochastic search methods [131].

Clearwater *et al.* investigated the "blackboard" model of cooperation [19]. Their approach models a generic communication mechanism between cooperating agents. This model was used in Chapter 4 to construct a second type of cooperative PSO algorithm.

Chapter 3

Particle Swarm Optimiser Convergence Characteristics

This chapter presents an analysis of the convergence behaviour of the PSO. First an explicit equation describing the trajectory of a particle is developed. This explicit equation is then used to show for which parameter settings the movement of the PSO will not "explode". A modification to the PSO is then presented, for which a proof of guaranteed convergence onto a local minimiser is developed. A technique for extending the convergent PSO to locate global minimisers is developed, with a corresponding proof of guaranteed convergence onto these global minimisers.

3.1 Analysis of Particle Trajectories

This section presents an analysis of the trajectory of a particle in the PSO algorithm, as well as providing further insights regarding the choice of the parameters c_1, c_2 and w. The term *convergence* will be used in this section to refer to the property that the limit

$$\lim_{t \to +\infty} \mathbf{x}(t) = \mathbf{p}$$

exists, where $\mathbf{p}$ is an arbitrary position in search space, and $\mathbf{x}(t)$ is the position of a particle at time t. The ability of the PSO algorithm to locate minima (local or global) is discussed in Section 3.3.

For convenience, the velocity and position update equations for the PSO with an inertia weight are repeated here:

$$\begin{aligned} v_{i,j}(t+1) &= wv_{i,j}(t) + c_1 r_{1,j}(t)[y_{i,j}(t) - x_{i,j}(t)] + \\ &\quad c_2 r_{2,j}(t)[\hat{y}_j(t) - x_{i,j}(t)] \end{aligned} \tag{3.1}$$

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1) \tag{3.2}$$

The implicit form of the velocity update equation, presented in equation (3.1), is used for multi-particle PSOs working in a multi-dimensional search space. For ease of notation, the analysis below will be restricted to a single dimension so that the subscript j is dropped. This can be done without loss of generality since there is no interaction between the different dimensions in the PSO. The notation can be simplified even more by looking at the trajectory of a single particle in isolation, thus dropping the subscript i. This simplification assumes that the other particles in the swarm will remain "frozen" in space while the trajectory of a single particle is analyzed. The trajectory of a particle will be considered in discrete time steps, so that x_t denotes the value of $x(t)$.

Now, by substituting (3.1) into (3.2), the following non-homogeneous recurrence relation is obtained (details are provided in Appendix C):

$$x_{t+1} = (1 + w - \phi_1 - \phi_2)x_t - wx_{t-1} + \phi_1 y + \phi_2 \hat{y}, \tag{3.3}$$

where $\phi_1 = c_1 r_1(t)$ and $\phi_2 = c_2 r_2(t)$; ϕ_1, ϕ_2 and w are assumed to be constant. The values ϕ_1 and ϕ_2 are thus specific instances of $c_1 r_1(t)$ and $c_2 r_2(t)$, respectively.

When the initial conditions $x(0) = x_0$ and $x(1) = x_1$ have been specified, the closed form of (3.3) can be obtained using any suitable technique for solving non-homogeneous recurrence relations [125]. A complete derivation of the equations below can be found in Appendix C. The closed form equation is given by

$$x_t = k_1 + k_2 \alpha^t + k_3 \beta^t, \tag{3.4}$$

where

$$k_1 = \frac{\phi_1 y + \phi_2 \hat{y}}{\phi_1 + \phi_2} \tag{3.5}$$

$$\gamma = \sqrt{(1 + w - \phi_1 - \phi_2)^2 - 4w} \tag{3.6}$$

$$\alpha = \frac{1 + w - \phi_1 - \phi_2 + \gamma}{2} \tag{3.7}$$

$$\beta = \frac{1 + w - \phi_1 - \phi_2 - \gamma}{2} \tag{3.8}$$

$$x_2 = (1 + w - \phi_1 - \phi_2)x_1 - wx_0 + \phi_1 y + \phi_2 \hat{y} \tag{3.9}$$

$$k_2 = \frac{\beta(x_0 - x_1) - x_1 + x_2}{\gamma(\alpha - 1)} \tag{3.10}$$

$$k_3 = \frac{\alpha(x_1 - x_0) + x_1 - x_2}{\gamma(\beta - 1)} \tag{3.11}$$

Note that the above equations assume that y and $\hat{y}$ remain constant while t changes. The actual PSO algorithm will allow y and $\hat{y}$ to change through equations (2.5) and (2.6), respectively. Thus the closed form of the update equation presented above remains valid until a better position x (and thus y, $\hat{y}$) is discovered, after which the above equations can be used again after recomputing the new values of k_1, k_2 and k_3. The exact time step at which this will occur depends on the objective function, as well as the values of y and $\hat{y}$. To allow the extrapolation of the sequence it is convenient to rather keep y and $\hat{y}$ constant; by implication k_1, k_2 and k_3 will be constant as well. Although it is unlikely that this scenario will be encountered in a real PSO application, it aids us in the elucidation of the convergence characteristics of the PSO.

An important aspect of the behaviour of a particle concerns whether its trajectory (specified by x_t) converges or diverges. The conditions under which the sequence $\{x_t\}_{t=0}^{+\infty}$ will converge will thus be considered next.

3.1.1 Convergence Behaviour

The analysis of the convergence of a particle's trajectory can easily be performed for constant values of ϕ_1 and ϕ_2. It should be remembered, though, that the actual PSO algorithm uses pseudo-random values for these parameters, rather than constant values. As will be shown later, however, the behaviour of the system is usually specified by the upper bound associated with these values. Thus by using the largest values that ϕ_1 and ϕ_2 can assume, the worst-case behaviour (in terms of convergence) can be studied.

Equation (3.4) can be used to compute the trajectory of a particle, under the assumption that y, $\hat{y}$, ϕ_1, ϕ_2 and w remain constant. Convergence of the sequence $\{x_t\}_{t=0}^{+\infty}$

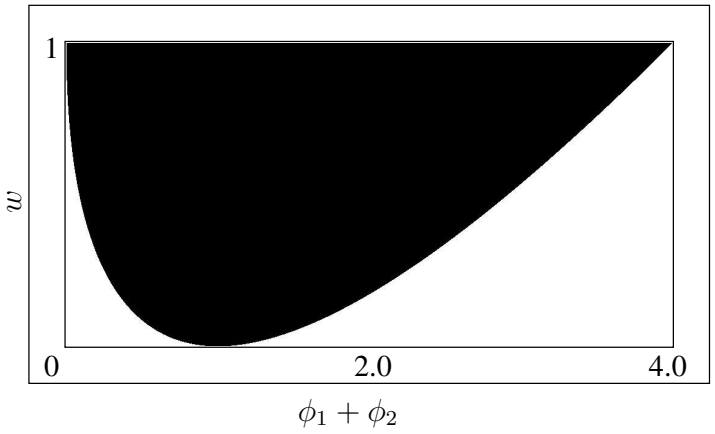


Figure 3.1: The dark paraboloid-shaped region represents the region for which $(1 + w - \phi_1 - \phi_2)^2 < 4w$, so that all the points in this region result in a γ (and thus also α and β) value with a non-zero imaginary component.

is determined by the magnitude of the values α and β, as computed using equations (3.7) and (3.8). From equation (3.6) it is clear that γ will be a complex number with a non-zero imaginary component whenever

$$(1 + w - \phi_1 - \phi_2)^2 < 4w \tag{3.12}$$

or equivalently, when

$$(\phi_1 + \phi_2 - w - 2\sqrt{w} - 1)(\phi_1 + \phi_2 - w + 2\sqrt{w} - 1) < 0$$

A complex γ results in α and β being complex numbers with non-zero imaginary components as well. Figure 3.1 depicts the range of ϕ_1, ϕ_2 and w values for which relation (3.12) holds. One might wonder at this point how the recurrence relation presented in equation (3.3) can yield complex numbers. The answer is that the values obtained using the explicit form of equation (3.4) are real when t is an integral number, since the imaginary components in k_2, k_3, α and β cancel out.

The magnitude of α and β is measured using the L_2 norm for vectors. For an arbitrary complex number, z, the L_2 norm is expressed as

$$\|z\| = \sqrt{(\Re e(z))^2 + (\Im m(z))^2} \tag{3.13}$$

Keep in mind that $\mathbb{R} \subset \mathbb{C}$, so that the above metric remains valid even if α is a real number.

Any complex number z^t can be written as

$$\begin{aligned} z^t &= \left(\|z\|e^{\hat{\imath}\theta}\right)^t \\ &= \|z\|^t e^{\hat{\imath}\theta t} \\ &= \|z\|^t(\cos(\theta t) + \hat{\imath}\sin(\theta t)) \end{aligned}$$

where $\theta = \arg(z)$. The limit,

$$\lim_{t\to+\infty} z^t = \lim_{t\to+\infty} \|z\|^t(\cos(\theta t) + \hat{\imath}\sin(\theta t))$$

exists only when $\|z\| < 1$, in which case the limits assumes the value 0.

Consider now the value of x_t (from equation 3.4) in the limit, thus

$$\lim_{t\to+\infty} x_t = \lim_{t\to+\infty} k_1 + k_2\alpha^t + k_3\beta^t \tag{3.14}$$

Clearly, equation (3.14) implies that the trajectory of a particle, $\{x_t\}_{t=0}^{+\infty}$, will diverge whenever $\max(\|\alpha\|, \|\beta\|) > 1$, since then the limit does not exist. Conversely, $\{x_t\}_{t=0}^{+\infty}$ will converge when $\max(\|\alpha\|, \|\beta\|) < 1$, so that

$$\lim_{t\to+\infty} x_t = \lim_{t\to+\infty} k_1 + k_2\alpha^t + k_3\beta^t = k_1 \tag{3.15}$$

since $\lim_{t\to+\infty} \alpha^t = 0$ if $\|\alpha\| < 1$ and $\lim_{t\to+\infty} \beta^t = 0$ if $\|\beta\| < 1$. Further, let z represent either α or β. Then, if $\|z\| = 1$, the limit

$$\lim_{t\to+\infty} c^t = \lim_{t\to+\infty} 1^t(\cos(\theta t) + \hat{\imath}\sin(\theta t)) \tag{3.16}$$

does not exist, so the sequence $\{x_t\}_{t=0}^{+\infty}$ diverges.

Note that the above calculations assumed that ϕ_1, ϕ_2 remained constant, which is not the case in the normal PSO. The values of c_1 and c_2 can, however, be considered an upper bound for ϕ_1 and ϕ_2. The average behaviour of the system can be observed by considering the expected values of ϕ_1 and ϕ_2, assuming uniform distributions:

$$E[\phi_1] = c_1 \int_0^1 \frac{x}{1-0} \mathrm{d}x = c_1 \left.\frac{x}{2}\right|_0^1 = \frac{c_1}{2} \tag{3.17}$$

and

$$E[\phi_2] = c_2 \int_0^1 \frac{x}{1-0} \mathrm{d}x = c_2 \left. \frac{x}{2} \right|_0^1 = \frac{c_2}{2} \tag{3.18}$$

Assume that ϕ_1, ϕ_2 and w were chosen so that $\max(\|\alpha\|, \|\beta\|) < 1$; in other words, they were chosen so that the sequence $\{x_t\}_{t=0}^{+\infty}$ converges. This means, from equations (3.15) and (3.5), that

$$\begin{aligned} \lim_{t \to +\infty} x_t &= k_1 \\ &= \frac{\phi_1 y + \phi_2 \hat{y}}{\phi_1 + \phi_2} \end{aligned} \tag{3.19}$$

If the expected values of ϕ_1 and ϕ_2 (from equations 3.17 and 3.18) are substituted in equation (3.19), the following results:

$$\begin{aligned} \lim_{t \to +\infty} x_t &= \frac{\frac{c_1}{2} y + \frac{c_2}{2} \hat{y}}{\frac{c_1}{2} + \frac{c_2}{2}} \\ &= \frac{c_1 y + c_2 \hat{y}}{c_1 + c_2} \end{aligned}$$

The trajectory of the particle thus converges onto a weighted mean of y and $\hat{y}$. To illustrate, if $c_1 = c_2$, then

$$\lim_{t \to +\infty} x_t = \frac{y + \hat{y}}{2}$$

A more general solution can be obtained for arbitrary values of c_1 and c_2, as follows

$$\begin{aligned} \lim_{t \to +\infty} x_t &= \frac{c_1 y + c_2 \hat{y}}{c_1 + c_2} \\ &= \frac{c_1}{c_1 + c_2} y + \frac{c_2}{c_1 + c_2} \hat{y} \\ &= \left(1 - \frac{c_2}{c_1 + c_2}\right) y + \frac{c_2}{c_1 + c_2} \hat{y} \\ &= (1 - a) y + a \hat{y} \end{aligned} \tag{3.20}$$

where $a = c_2/(c_1 + c_2)$, therefore $a \in [0, 1]$. Equation (3.20) implies that the particle converges to a value derived from the line connecting the personal best to the global best. This result is intuitively satisfying, since it implies that the particle will search for better solutions in the area between its personal best position and the global best position.

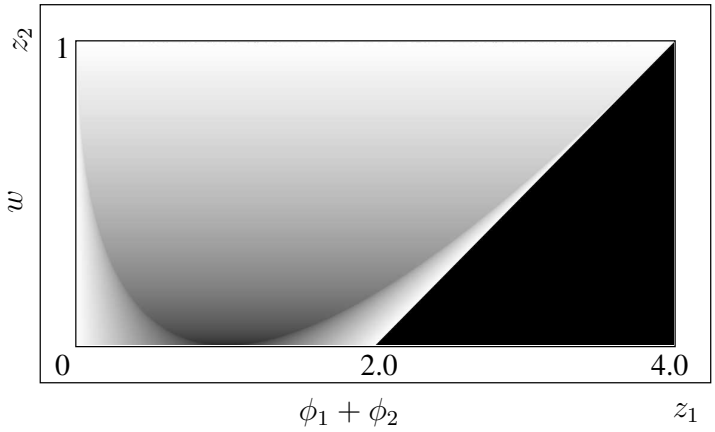


Figure 3.2: The black triangle to the bottom right represents the values for which the particle strictly diverges, *i.e.* $\max(\|\alpha\|, \|\beta\|) > 1$. This is the region for which $w < 0.5(\phi_1 + \phi_2) - 1$. The lighter regions represent the magnitude of $\max(\|\alpha\|, \|\beta\|)$, with white representing magnitude 1. The darker regions (outside of the divergent region) represent values leading to more rapid convergence.

Figure 3.2 is an experimentally obtained map visualising the ϕ_1, ϕ_2 and w values leading to convergence or divergence. The map was constructed by sampling the values of $\phi_1 + \phi_2$ and w on a regular grid, using 1000 horizontal samples and 500 vertical samples. The intensity of each point on the grid represents the magnitude $\max(\|\alpha\|, \|\beta\|)$, with lighter shades representing larger magnitudes, except for the black triangular shape observed in the bottom right corner of the map. This triangle corresponds to the values of ϕ_1, ϕ_2 and w resulting in $\max(\|\alpha\|, \|\beta\|) > 1$, which implies that the trajectory of the particle will diverge when using these values. Figure 3.2 should also be compared to Figure 3.1 to see the relationship between the magnitude $\max(\|\alpha\|, \|\beta\|)$ and whether γ has a non-zero imaginary component. Note that all the parameter values leading to a divergent trajectory have real-valued γ values, since the entire divergent triangle falls inside the white area of Figure 3.1. The parameters that correspond to real-valued γ values that do fall inside the convergent area of Figure 3.2 have relatively large magnitudes, as can be seen from their lighter shading.

The trajectory of a particle can be guaranteed to converge if the parameters ϕ_1, ϕ_2

and w are chosen so that the corresponding point on the map in Figure 3.2 always falls in the convergent region. Let z_1 represent the horizontal axis, associated with $\phi_1 + \phi_2$ and z_2 the vertical axis, associated with w. If we take into account that c_1 and c_2 represent the upper limits of ϕ_1 and ϕ_2, respectively, so that $\phi_1 \in [0, c_1]$ and $\phi_2 \in [0, c_2]$, then the range of values that $\phi_1 + \phi_2$ can assume (to ensure convergence) occur to the left of the vertical line $z_1 = c_1 + c_2$ in the figure. Searching vertically along line $z_1 = c_1 + c_2$ for the point where it exits the black divergent triangle yields the smallest w value that will result in a convergent trajectory (*i.e.* a point outside of the divergent triangular region). The coordinates of this intersection are $(c_1 + c_2, 0.5(c_1 + c_2) - 1)$. All w values larger than this critical value will also lead to convergent trajectories, so the general relation

$$w > \frac{1}{2}(c_1 + c_2) - 1 \tag{3.21}$$

can be defined to characterise these values.

Figure 3.3 is an alternate representation of Figure 3.2. Note that the magnitude $\max(\|\alpha\|, \|\beta\|)$ gradually increases from 0 to about 2.5 — this is especially visible in the furled right-hand bottom corner. Keep in mind that all magnitudes greater or equal to 1.0 lead to divergent trajectories.

3.1.2 Original PSO Convergence

The original particle swarm, with $c_1 = c_2 = 2$ and $w = 1$, is a boundary case, with $0.5(2 + 2) - 1 = 1 = w$. Further insight can be gained by directly calculating the value of $\max(\|\alpha\|, \|\beta\|)$ using explicit formulae (3.7) and (3.8). This is achieved by setting $\phi_1 = \phi_2 = 2$, since the maximum of these two values are determined by their respective upper bounds with values $c_1 = c_2 = 2$, yielding $\|\alpha\| = \|\beta\| = 1$. This would seem to imply that the original PSO equations resulted in divergent trajectories, according to equation (3.16). This verdict does not take into account the stochastic component, though. Now, considering the stochastic component with $\phi_1 = r_1(t)c_1$ and $\phi_2 = r_2(t)c_2$, where $r_1(t), r_2(t) \sim U(0, 1)$, it is clear that $0 < \phi_1, \phi_2 < 2$ when $c_1 = c_2 = 2$. Substituting $\phi = \phi_1 + \phi_2$ into equation (3.6) yields

$$\begin{aligned} \gamma &= \sqrt{(2 - \phi)^2 - 4} \\ &= \hat{\imath}\sqrt{4\phi - \phi^2} \end{aligned}$$

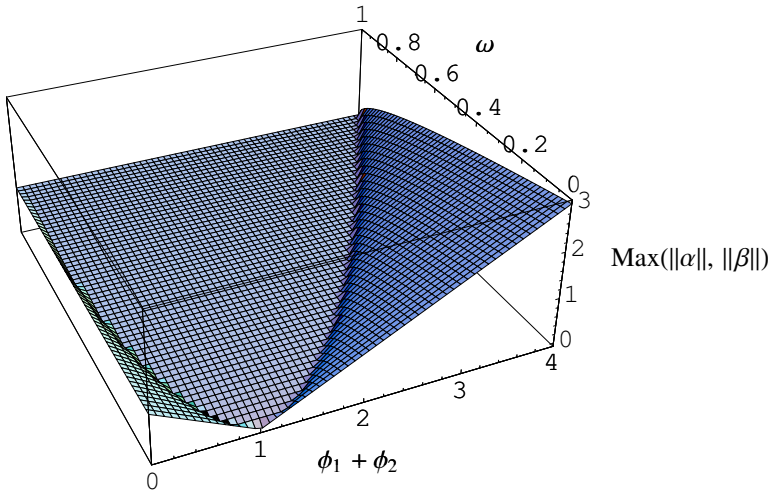


Figure 3.3: A 3-dimensional representation of the value $\max(\|\alpha\|, \|\beta\|)$. Note that the elevation of the furled right-hand bottom corner exceeds 1.0.

which in turn yields

$$\begin{aligned} \|\alpha\| &= \sqrt{\frac{(2-\phi)^2}{4} + \frac{4\phi - \phi^2}{4}} \\ &= \sqrt{\frac{4 - 4\phi + \phi^2}{4} + \frac{4\phi - \phi^2}{4}} \\ &= 1 \end{aligned} \tag{3.22}$$

Because α and β are complex conjugates when γ is complex, this implies that $\|\beta\| = 1$ as well, so that $\max(\|\alpha\|, \|\beta\|) = 1$. This means that the trajectory of the particle will be divergent regardless of the value of ϕ, which explains why the original PSO algorithm had to clamp the velocities to the range $[-v_{max}, v_{max}]$ to prevent the system from diverging.

Although the intuitive understanding of the concept of a divergent trajectory calls to mind the image of a sequence that grows without bound, a divergent trajectory need not be unbounded. A sequence may oscillate through a set of values without ever converging. This is exactly what happens in the case of the original PSO. Consider the value of α when $c_1 = c_2 = 2$, that is,

$$\alpha = \frac{(2-\phi)^2 + \hat{\imath}\sqrt{4\phi - \phi^2}}{2}$$

Recall that a complex number z^t can be represented in exponential form, so that

$$z^t = \|z\|^t(\cos(\theta t) + \hat{\imath}\sin(\theta t)) \tag{3.23}$$

where $\theta = \arg(z)$. Since $\|\alpha\| = 1$, as shown in equation (3.22), equation (3.23) can be reduced to (after substituting α for z)

$$\alpha^t = \cos(\theta t) + \hat{\imath}\sin(\theta t)$$

This implies that the trajectory of the particles is described by

$$\begin{aligned} x_t &= k_1 + k_2\alpha^t + k_3\beta^t \\ &= k_1 + k_2(\cos(\theta t) + \hat{\imath}\sin(\theta t)) + k_3(\cos(\theta t) - \hat{\imath}\sin(\theta t)) \\ &= k_1 + (k_2 + k_3)\cos(\theta t) + \hat{\imath}(k_2 - k_3)\sin(\theta t) \end{aligned}$$

where $\theta = \arg(\alpha)$, and $\arg(\beta) = -\theta$, since α and β are complex conjugates. The imaginary components cancel for integral values of t, as usual. The trajectory of a particle using the original PSO parameter settings thus traces out a superposition of two sinusoidal waves; their amplitudes and frequencies depend on the initial position and velocity of the particle. This is consistent with the findings of Ozcan and Mohan for equivalent parameter settings [91].

This clearly shows that the original PSO parameters led to divergent particle trajectories. The next section investigates the characteristics of trajectories obtained using parameter settings from the convergent region indicated in Figure 3.2.

3.1.3 Convergent PSO Parameters

Above it was shown that the parameter settings of the original PSO would cause the trajectories of its particles to diverge, were it not for the effect of the v_{max} clamping strategy. An infinite number of parameter settings exist that do ensure a convergent trajectory, so more information is needed to decide on a particular choice. A brief example will now show that certain parameter choices leads to convergent behaviour without having to clamp the velocities to the range $[-v_{max}, v_{max}]$.

One popular choice of parameters is $c_1 = c_2 = 1.49618$ and $w = 0.7298$ [39]. First, note that it satisfies relation (3.21) since $0.5(1.49618 + 1.49618) - 1 = 0.49618 < 0.7298$.

The stochastic behaviour can also be predicted using this relation, so that $0.5(\phi_1+\phi_2)-1<0.7298$, which is trivially true for $\phi_1+\phi_2\in(0,2\times 1.49618)$. Substitution into the explicit formulae for α and β, with $\phi=\phi_1+\phi_2$, confirms this. Two sets of calculations follow: one for the real-valued γ values, and another set for complex γ values. When $\phi\in[0,0.02122]$, implying a real-valued γ, then

$$\begin{aligned}\gamma &= \sqrt{(1+w-\phi)^2-4w}\\ &\approx \sqrt{0.073-3.4596\phi+\phi^2}\end{aligned}$$

$$\max(\|\alpha\|,\|\beta\|) \approx \frac{1.7298-\phi+\sqrt{0.073-3.4596\phi+\phi^2}}{2}<1$$

Otherwise, when $\phi\in(0.02122,2.992]$, resulting in complex γ values,

$$\begin{aligned}\gamma &= \sqrt{(1+w-\phi)^2-4w}\\ &\approx \hat{\imath}\sqrt{-0.073+3.4596\phi-\phi^2}\end{aligned}$$

$$\begin{aligned}\|\alpha\|=\|\beta\| &= \sqrt{\frac{(1+w-\phi)^2}{4}+\frac{-0.073+3.4596\phi-\phi^2}{4}}\\ &\approx 0.8754\end{aligned}$$

Again, note that $\|\beta\|=\|\alpha\|$ when γ is complex, since they are complex conjugates. Figure 3.4 is a cross-section of Figure 3.3 along the line $w=0.7298$. The figure clearly confirms the values derived above. Note that the figure indicates that the maximum value of ϕ can be increased to approximately 3.45 without causing the trajectory to diverge. The benefit of increasing ϕ will be discussed below. To summarise, this example shows that a popular choice of parameter settings leads to a convergent trajectory without having to clamp the velocities of the particles.

To ensure convergence, the value for w should thus be chosen so that it satisfies relation (3.21). This relation can also be reversed to calculate the values of c_1 and c_2 once a suitable w has been decided on. Note, however, that there is an infinite number of ϕ_1 and ϕ_2 values that satisfy $\phi=\phi_1+\phi_2$, all exhibiting the exact same convergence behaviour under the assumptions of this analysis, so it is customary to set $\phi_1=\phi_2$.

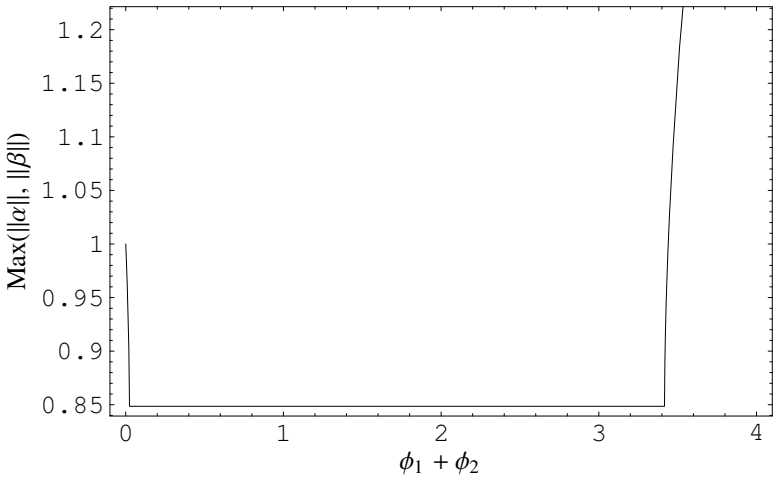


Figure 3.4: The magnitude $\max(\|\alpha\|, \|\beta\|)$ for $w = 0.7298$ and $\phi_1 + \phi_2 \in (0, 4)$.

The trajectory of a particle obtained by using a set of parameters leading to convergent behaviour can be associated with a physical phenomenon. First, equation (3.4) is re-written using the alternate representational form for complex numbers to yield

$$\begin{aligned} x(t) &= k_1 + k_2\alpha^t + k_3\beta^t \\ &= k_1 + k_2\|\alpha\|^t(\cos(\theta_1 t) + \hat{\imath}\sin(\theta_1 t)) + k_3\|\beta\|^t(\cos(\theta_2 t) + \hat{\imath}\sin(\theta_2 t)) \end{aligned}$$

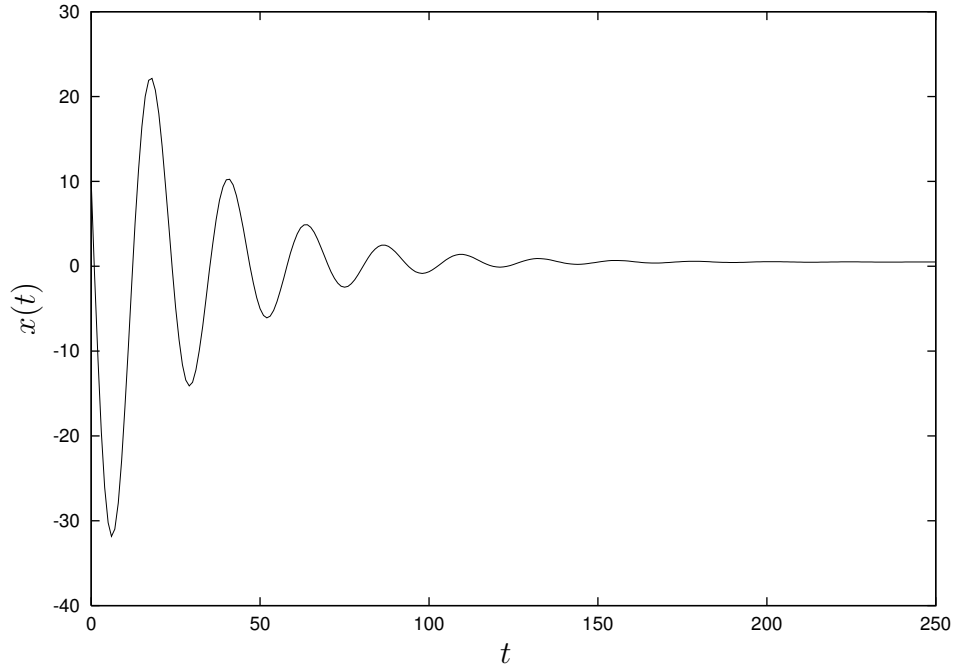
where $\theta_1 = \arg(\alpha)$ and $\theta_2 = \arg(\beta)$. When γ is complex, α and β will be complex conjugates. This leads to the simplified form

$$x(t) = k_1 + \|\alpha\|^t(k_2 + k_3)\cos(\theta t) + \hat{\imath}\|\alpha\|^t(k_2 - k_3)\sin(\theta t)$$

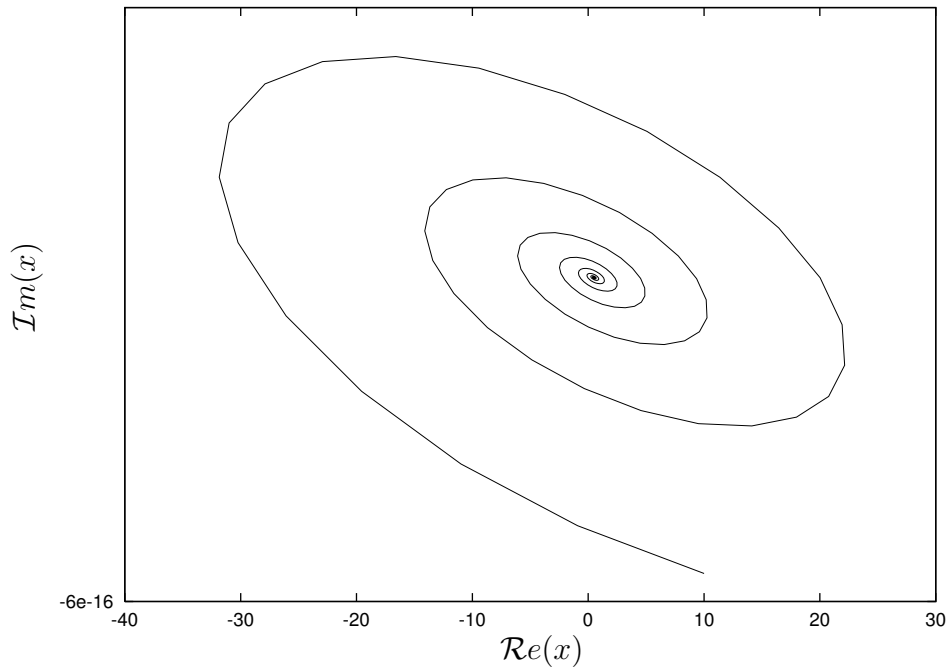
In this form it is immediately clear that the trajectory of a particle is analogous to the dampened vibrations observed in a spring-dashpot system [14]. The characteristic waveform associated with dampened vibrations is also clearly visible in Figure 3.5 below.

3.1.4 Example Trajectories

This section presents several plots of trajectories that have been obtained using equation (3.4). These trajectories were computed without any stochastic component; plots taking the stochastic component into account are presented in Section 3.1.5. Figures 3.5,

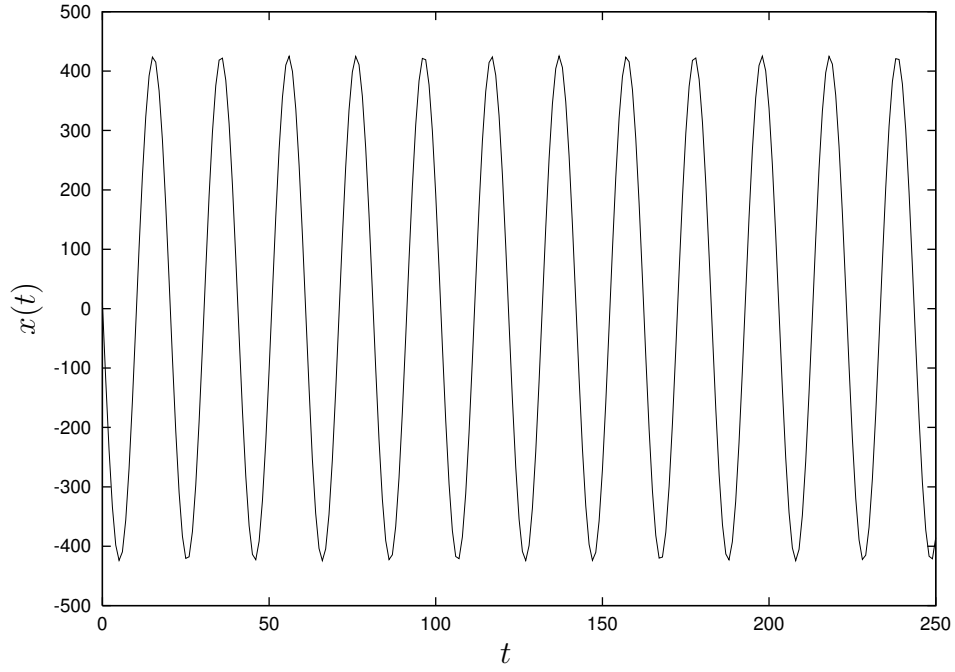


(a) Time domain

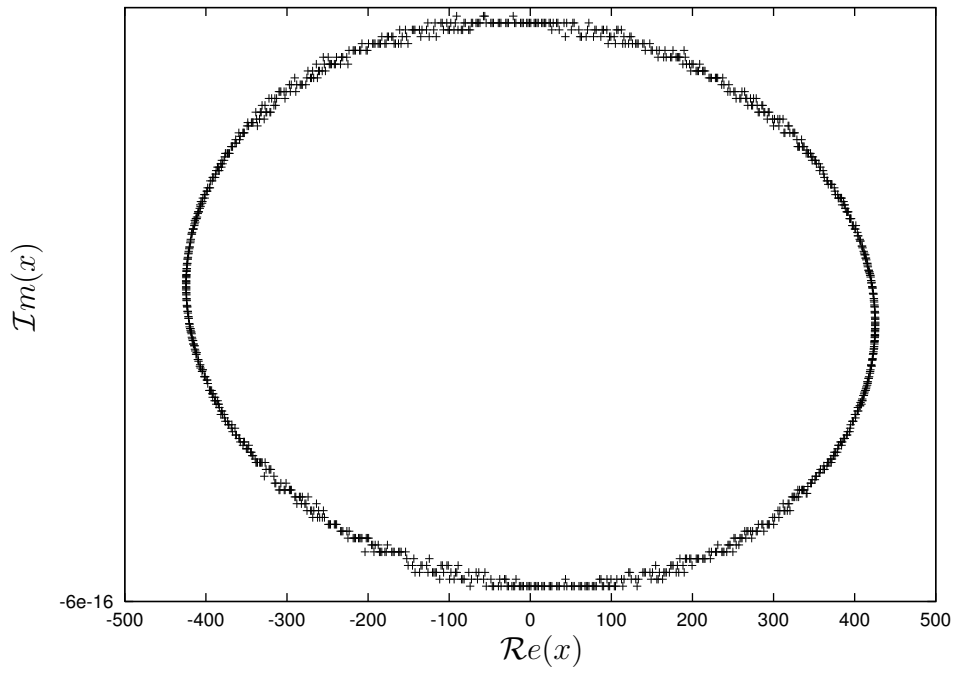


(b) Phase space

Figure 3.5: A convergent particle trajectory, obtained with the parameter settings $w = 0.5$ and $\phi_1 = \phi_2 = 1.4$. Figure (a) plots the particle position over time; Figure (b) shows the real and complex components of the particle trajectory over the same duration.

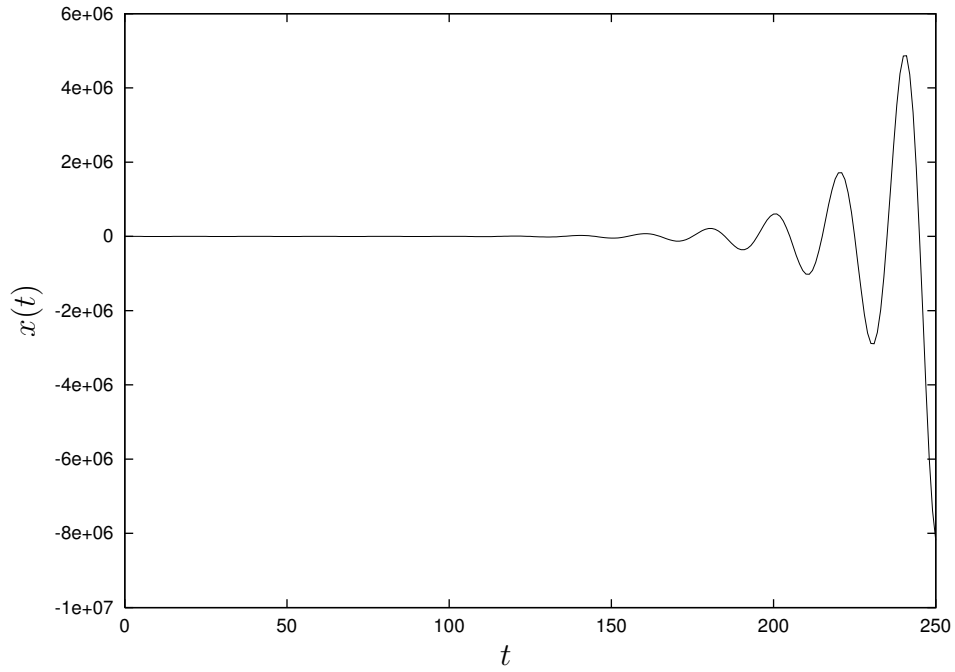


(a) Time domain

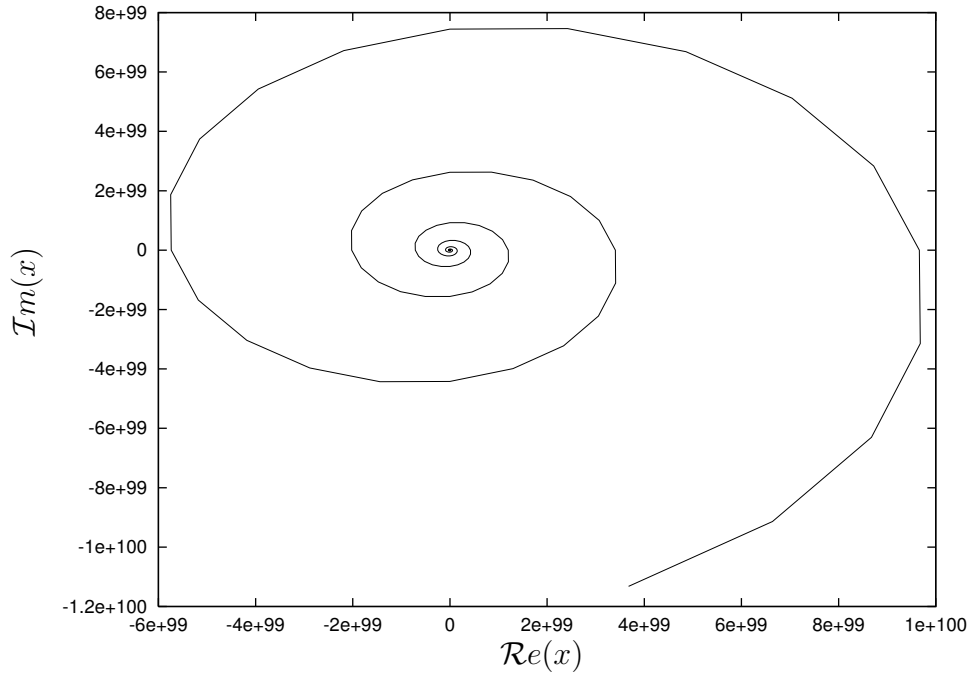


(b) Phase space

Figure 3.6: A cyclic particle trajectory, obtained with the parameter settings $w = 1.0$ and $\phi_1 = \phi_2 = 1.999$. Figure (a) plots the particle position over time; Figure (b) shows the real and complex components of the particle trajectory over the same duration.



(a) Time domain



(b) Phase space

Figure 3.7: A divergent particle trajectory, obtained with the parameter settings $w = 0.7$ and $\phi_1 = \phi_2 = 1.9$. Figure (a) plots the particle position over time; Figure (b) shows the real and complex components of the particle trajectory over the same duration.

3.6 and 3.7 show examples of the three types of behaviour that the non-stochastic PSO equations can exhibit: convergent, cyclic (a special form of divergent behaviour) and divergent. All these figures have been obtained experimentally, using 80-bit floating point numbers, with constant values $y = 1.0$ and $\hat{y} = 0$. The initial conditions were $x(0) = 10$, and $x(1) = 10 - 9\phi_1 - 10\phi_2$, with ϕ_1 and ϕ_2 as listed for each figure.

Figure 3.5 is a plot of a particle trajectory obtained with a set of parameters that leads to convergence. In Figure 3.5 it is clear that the amplitude of the oscillations decays over time. This represents the radius of the search pattern of a particle in search space. Initially the particle will explore a larger area, but the amplitude decreases rapidly until the particle searches a small neighbourhood surrounding $(c_1 y + c_2 \hat{y})/(c_1 + c_2)$. In the complex representation of $x(t)$, Figure 3.5(b), where t is any real number (instead of being restricted to integral values), the particle traces out a convergent spiral.

Figure 3.6 illustrates the second type of observed behaviour, namely that leading to cyclic trajectories. Some comments on Figure 3.6(b) are in order. Ideally, the figure would be a perfect ellipse since the superimposed sine waves should trace out a smooth curve. Because the figure was obtained experimentally, however, the points are somewhat "noisy" due to numerical inaccuracies. Instead of connecting the successive points using line segments (as was done for the other two figures), it was decided, for the sake of clarity, to plot only the points. Figure 3.6(a) clearly shows the non-convergent sinusoidal waveform of the particle trajectory.

Figure 3.7 exhibits the classic notion of divergence: as time passes, the particle moves (or more accurately, oscillates) further and further from $\hat{y}$, the global best position of the swarm. The spiral in Figure 3.7(b) is divergent, and although it looks similar to the one in Figure 3.5(b), the scale of the axes clearly show the divergent behaviour. For a search algorithm this type of trajectory is not generally desirable, since the trajectory will rapidly exceed the numerical range of the machine.

3.1.5 Trajectories under Stochastic Influences

In the previous section the stochastic component was treated as a constant by fixing the values of ϕ_1 and ϕ_2. Although it was shown that some characteristics of the trajectory can be applied to whole ranges of ϕ values, it is still not clear what the influence of

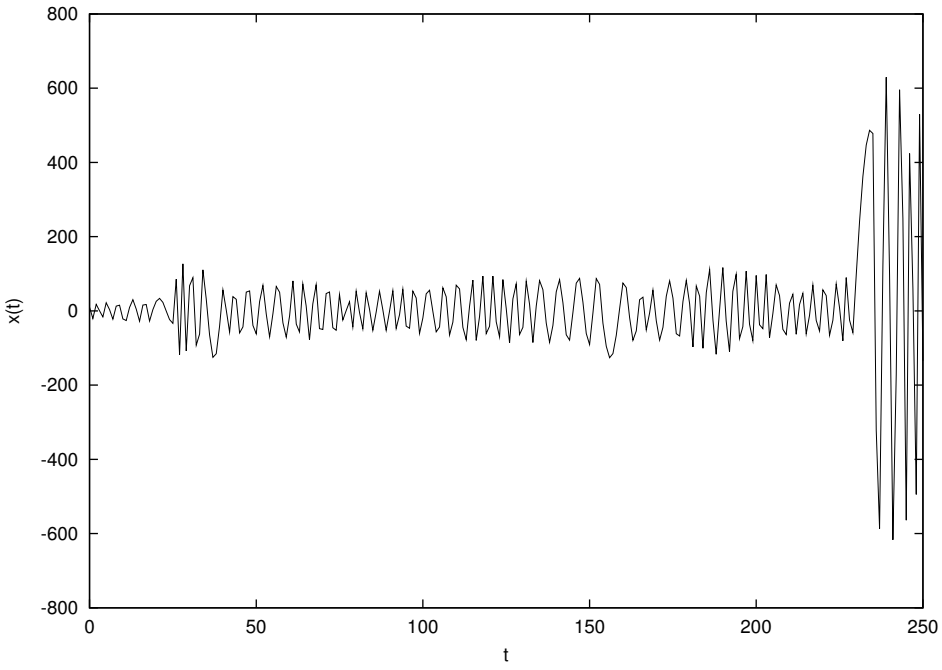


Figure 3.8: Stochastic particle trajectory, obtained using $w = 1.0$ and $c_1 = c_2 = 2.0$. Note that the y-axis scale of this figure is on the order of 10^2.

randomness will be on the trajectory. To investigate these phenomena, several sample trajectories are presented. The following parameters were used for all the experiments: $y = 1.0$, $\hat{y} = 0$, $x(0) = 10$ and $x(1) = 10 - 9\phi_1 - 10\phi_2$, with ϕ_1, ϕ_2 and w set to the values indicated for each experiment. The stochastic values for ϕ_1 and ϕ_2 were sampled so that $0 \leq \phi_1 \leq c_1$ and $0 \leq \phi_2 \leq c_2$. Because of the stochastic component, the plots presented in this section were obtained using PSO update equations (3.1) and (3.2), instead of the closed form solution offered by equation (3.4). This implies that discrete time was used, so that no phase plots were drawn. The notation x_t in this section therefore refers to the value of x at (discrete) time step t.

Figure 3.8 is a plot of the trajectory of a particle using the original PSO parameters, *i.e.* $w = 1.0$ and $c_1 = c_2 = 2.0$. Notice how the amplitude of the oscillations increases towards the right of the graph, a clear indication of the divergent behaviour of this configuration. This is in agreement with observations in the previous section, where these parameter settings led to cyclic (*i.e.* divergent) behaviour. The observed increase

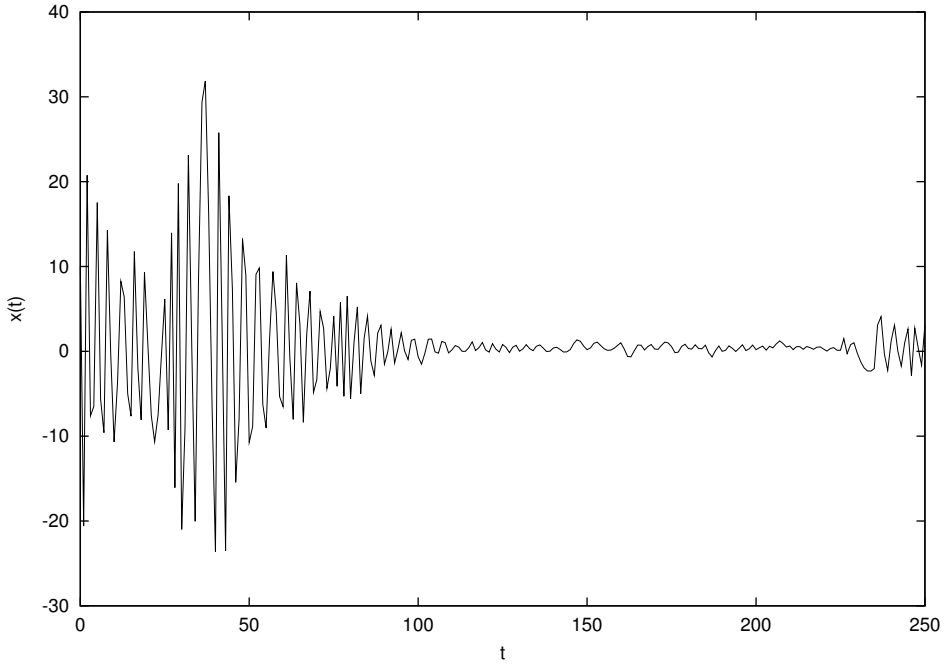


Figure 3.9: Stochastic particle trajectory, obtained using $w = 0.9$ and $c_1 = c_2 = 2.0$

in amplitude is caused by the randomness in the values of ϕ_1 and ϕ_2. A simple example will illustrate the nature of the problem. Assume the following (quite arbitrary) values for the parameters: $x_t = 10$, $x_{t-1} = 11$, $\phi_1 = c_1 r_1(t) = 1.9$, and $\phi_2 = c_2 r_2(t) = 1.8$. The new position of the particle can be calculated (using equation 3.3) as

$$x_{t+1} = -1.7x_t - x_{t-1} + 1.9y + 1.8\hat{y}$$

thus $x_{t+1} = -26.1$. If another iteration is executed, then

$$x_{t+2} = -1.7x_{t+1} - x_t + 1.9y + 1.8\hat{y}$$

which yields $x_{t+2} = 35.27$. If, however, different stochastic values are used so that $\phi_1 = 0.1$ and $\phi_2 = 0.2$, then

$$x_{t+2} = 1.7x_{t+1} - x_t + 0.1y + 0.2\hat{y}$$

resulting in $x_{t+2} = -55.27$. What this example illustrates is that alternating between large and small values for ϕ_1 and ϕ_2 may increase the distance between x_t and $(1-a)y+a\hat{y}$,

instead of decreasing it (or oscillating around it). This is caused in part by the negative sign associated with the x_{t-1} term, which changes the direction that the particle moves in at every alternate time step. A large distance at time step t may result in an almost doubling of the distance in the next time step. If ϕ_1 and ϕ_2 remain constant, then the particle is able to return to its previous position, since the step size is bounded.

Figure 3.9 was plotted using $w = 0.9$ and $c_1 = c_2 = 2.0$, again using stochastic values for ϕ_1 and ϕ_2. Notice how the oscillations first appear to increase in amplitude, but then gradually decrease. Near the end of the sample the amplitude increases again, but it eventually decreases at time $t = 300$ (not shown in the figure). Applying relation (3.21) to the parameters yields $0.5(2 + 2) - 1 = 2 > 0.9$, which implies that the trajectory will diverge when the upper bounds of ϕ_1 and ϕ_2 are considered. Convergent behaviour emerges when $\phi_1 + \phi_2 < 3.8$. This happens with a probability of $3.8/4 = 0.95$, since $0 < \phi_1 + \phi_2 < 4$ under a uniform distribution. In short, this implies that the trajectory of the particle will converge *most of the time*, occasionally taking divergent steps. The relative magnitude of the divergent steps versus the convergent steps must be taken into account to predict correctly whether the system will converge. Since this information is not available (because of the randomness) it is not possible to make this prediction accurately.

Figure 3.10 represents the trajectory of a particle using the parameter settings $w = 0.7$ and $c_1 = c_2 = 1.4$. Applying relation (3.21) shows that $0.5(1.4+1.4)-1 = 0.4 < 0.7$, so that the trajectory is expected to converge. This is clearly visible in the figure since the initial oscillations decay very rapidly. Minor oscillations, caused by the stochastic influence, remain present though. The parameter settings are now changed so that $w = 0.7$ and $\phi_1 = \phi_2 = 2.0$, as reflected in Figure 3.11. Note that relation (3.21) dictates that the upper bound for this trajectory is divergent, since $0.5(2 + 2) - 1 = 1 > 0.7$. When $\phi_1 + \phi_2 < 3.4$, however, convergent behaviour surfaces again. This happens with probability $3.4/4 = 0.85$, a lower figure than that obtained above with parameter settings $w = 0.9$ and $\phi_1 = \phi_2 = 2$. The trajectory in Figure 3.11 appears to have a faster rate of convergence than the one in Figure 3.9, though. This is offset by the fact that there are more large "bumps" in Figure 3.11, indicating that divergent steps occur more frequently.

These results indicate that it is not strictly necessary to choose the values of c_1 and

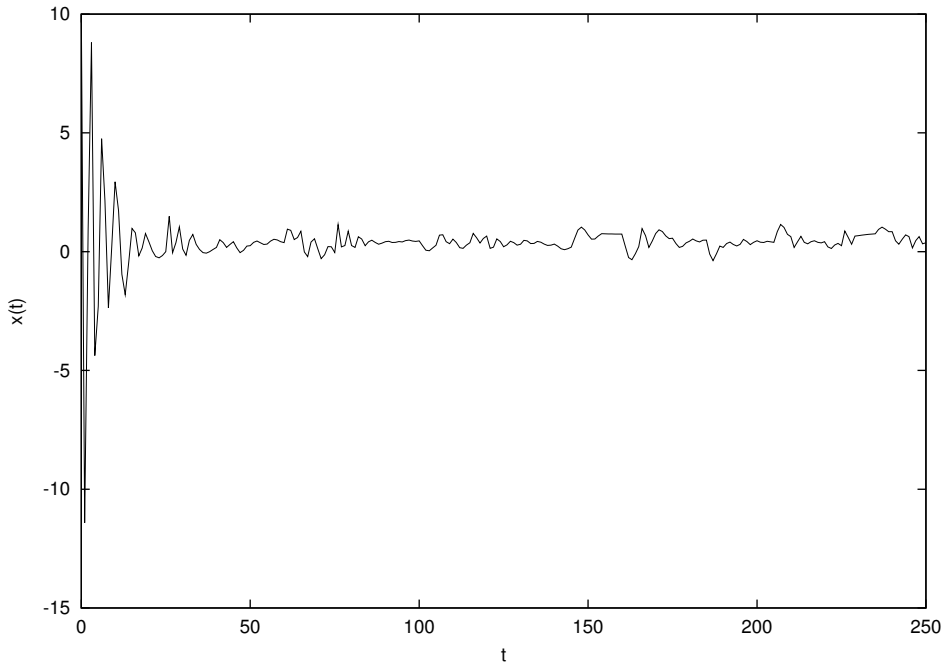


Figure 3.10: Stochastic particle trajectory, obtained using $w = 0.7$ and $c_1 = c_2 = 1.4$

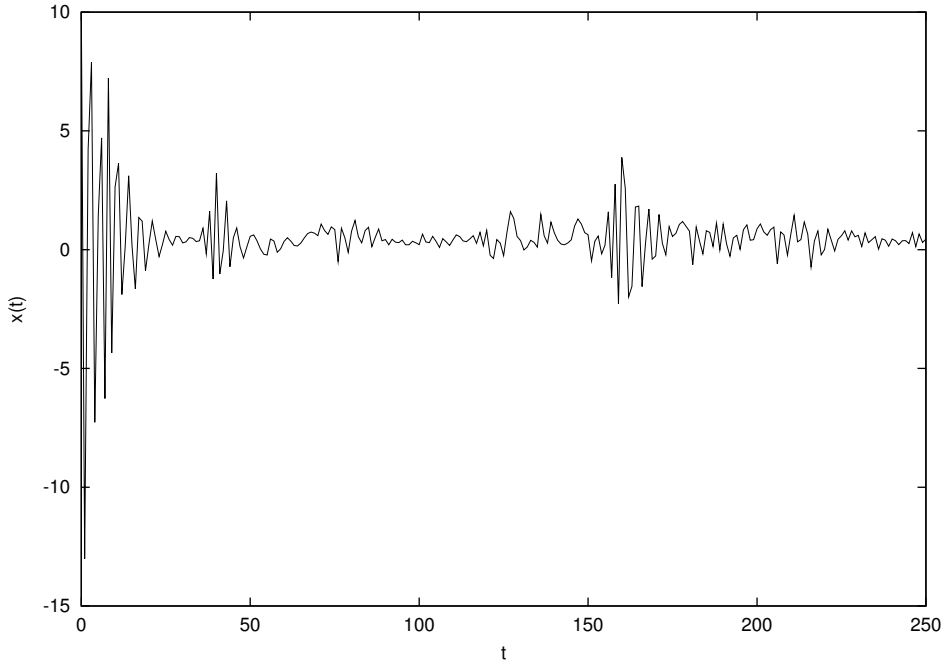


Figure 3.11: Stochastic particle trajectory, obtained using $w = 0.7$ and $c_1 = c_2 = 2.0$

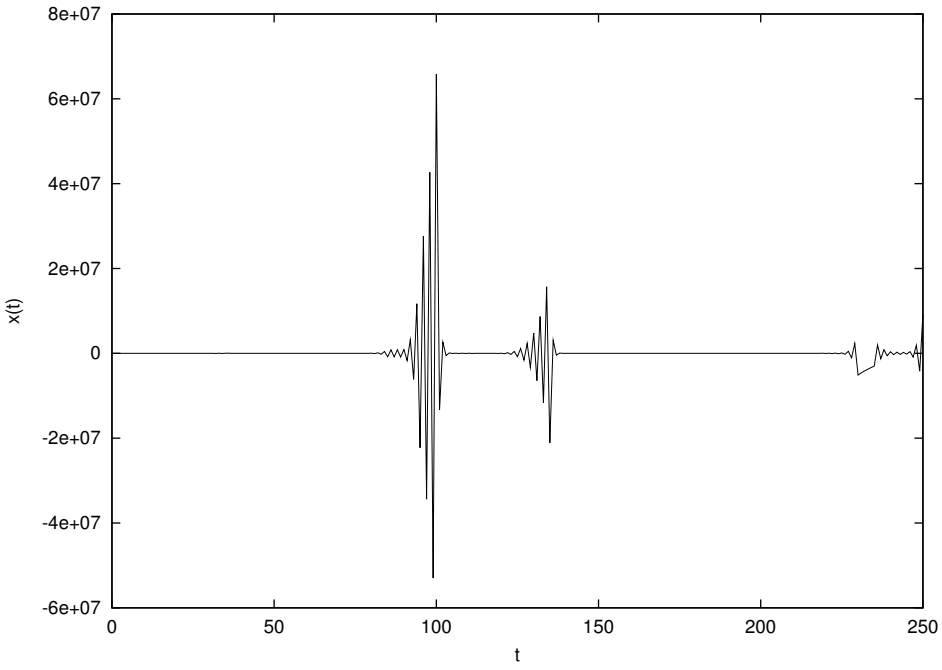


Figure 3.12: Stochastic particle trajectory, obtained using $w = 0.001$ and $c_1 = c_2 = 2.0$. Note that the y-axis scale of this figure is on the order of 10^7.

c_2 so that relation (3.21) is satisfied for all values of $0 < \phi_1 + \phi_2 < c_1 + c_2$, for a given w value. Let ϕ_{crit} denote the largest value of $\phi_1 + \phi_2$ for which relation (3.21) holds. Then

$$\phi_{crit} = \sup \phi \mid 0.5\phi - 1 < w, \quad \phi \in (0, c_1 + c_2] \tag{3.24}$$

All values of $\phi_1 + \phi_2 \leq \phi_{crit}$ then satisfy relation (3.21). As long as the ratio

$$\phi_{ratio} = \frac{\phi_{crit}}{c_1 + c_2} \tag{3.25}$$

is close to 1.0, the trajectory will converge without too many disruptions. As shown above, even a ϕ_{ratio} of 0.85 resulted in a system that converges without excessively large oscillations.

Extreme cases, like $w = 0.001$ and $c_1 = c_2 = 2.0$, results in $\phi_{ratio} \approx 1/2$. This system will take divergent steps 50% of the time, but as Figure 3.12 shows, the system always "recovers" after taking large divergent steps. The recovery is caused by the fact that roughly 50% of the time the particle will take a step along a *convergent* trajectory. The

probabilistically divergent behaviour can have a positive influence on the diversity of the solutions that the particle will examine, thereby improving its exploration capabilities. This property is especially valuable when optimising functions that contain many local minima.

Holland discussed the balance between *exploration* and *exploitation* that an algorithm must maintain [63]. Exploration ability is related to the algorithm's tendency to explore new regions of the search space, while exploitation is the tendency to search a smaller region more thoroughly. By choosing the PSO parameters carefully, a configuration can be found that maintains the balance reasonably well.

The rule of thumb for choosing the parameters w, ϕ_1 and ϕ_2 is that smaller w values result in faster rates of convergence. This is offset by how frequently a divergent step will be taken, as measured by the value ϕ_{ratio}, which is influenced by c_1 and c_2. If a w value is selected, then a truly convergent system can be constructed by choosing c_1 and c_2 so that $\phi_{ratio} = 1$. This results in a system with rapid convergence and little or no "exploration" behaviour. Choosing slightly larger c_1 and c_2 values (and keeping w fixed) results in a smaller ϕ_{ratio}. Such a system will have more "exploration" behaviour, but it will have more trouble with the "exploitation" phase of the search, *i.e.* it will have more disruptions to its trajectory.

3.1.6 Convergence and the PSO

It is important to note at this stage that if the trajectory of the particle converges, then it will do so towards a value derived from the line between its personal best position and the global best particle's position (see equation 3.20). Due to update equation (2.5), the personal best position of the particle will gradually move closer to the global best position, so that the particle will eventually converge on the position of the global best particle. At this point, the algorithm will not be able to improve its solution, since the particle will stop moving. This has no bearing on whether the algorithm has actually discovered the minimum of the function f — in fact, there's no guarantee that the position on which the particle has converged is even a local minimum. The next section deals with a modified PSO algorithm that addresses this problem.

3.2 Modified Particle Swarm Optimiser (GCPSO)

The different versions of the PSO algorithm introduced in Chapter 2, including the inertia weight and constriction factor versions, all have a potentially dangerous property: if $\mathbf{x}_i = \mathbf{y}_i = \hat{\mathbf{y}}$, then the velocity update will depend only on the value of $wv_{i,j}(t)$. In other words, if a particle's current position coincides with the global best position/particle, the particle will only move away from this point if its previous velocity and w are non-zero. If their previous velocities are very close to zero, then all the particles will stop moving once they catch up with the global best particle, which may lead to premature convergence of the algorithm. In fact, this does not even guarantee that the algorithm has converged on a local minimum — it merely means that all the particles have converged on the best position discovered so far by the swarm.

To address this issue a new parameter is introduced to the PSO algorithm. Let τ be the index of the global best particle, so that

$$\mathbf{y}_\tau = \hat{\mathbf{y}}$$

For reasons that will become clear in Section 3.3, a new velocity update equation for the global best particle (*i.e.* particle τ) is suggested, so that

$$v_{\tau,j}(t+1) = -x_{\tau,j}(t) + \hat{y}_j(t) + wv_{\tau,j}(t) + \rho(t)(1 - 2r_{2,j}(t)) \tag{3.26}$$

where ρ is a scaling factor defined below. The other particles in the swarm continue using the usual velocity update equation (*e.g.* equation 3.1). Briefly, the $-x_{\tau,j}(t)$ term "resets" the particle's position to the position $\hat{y}_j$. To this position a vector representing the current search direction, represented by the term $wv_{\tau,j}(t)$, is added. The $\rho(t)(1 - 2r_{2,j}(t))$ term generates a random sample from a sample space with side lengths $2\rho(t)$.

Combining the position update step (equation 3.2) and the new velocity update step (equation 3.26) for the global best particle τ results in the new position update equation

$$x_{\tau,j}(t+1) = \hat{y}_j(t) + wv_{\tau,j}(t) + \rho(t)(1 - 2r_2(t)) \tag{3.27}$$

The addition of the ρ term causes the PSO to perform a random search in an area surrounding the global best position $\hat{\mathbf{y}}$. The diameter of this search area is controlled by

the parameter ρ. The value of $\rho(t)$ is adapted after each time step, using

$$\rho(t+1) = \begin{cases} 2\rho(t) & \text{if } \#successes > s_c \\ 0.5\rho(t) & \text{if } \#failures > f_c \\ \rho(t) & \text{otherwise} \end{cases} \tag{3.28}$$

where the terms $\#failures$ and $\#successes$ denote the number of consecutive failures or successes, respectively, where a failure is defined as $f(\hat{\mathbf{y}}(t)) = f(\hat{\mathbf{y}}(t-1))$. A default initial value of $\rho(0) = 1.0$ was found empirically to produce acceptable results. The values s_c and f_c are threshold parameters, discussed in more detail below. The following additional rules must also be implemented to ensure that equation (3.28) is well-defined:

$$\#successes(t+1) > \#successes(t) \Rightarrow \#failures(t+1) = 0$$

and

$$\#failures(t+1) > \#failures(t) \Rightarrow \#successes(t+1) = 0$$

Thus, on a success the failure count is set to zero, and likewise the success count is reset when a failure occurs.

The optimal choice of values for the parameters f_c and s_c depend on the objective function. In high-dimensional search spaces it is difficult to obtain better values using a random search in only a few iterations, so it is recommended to set $f_c = 5$, $s_c = 15$. These settings imply that the algorithm is quicker to punish a poor ρ setting than it is to reward a successful ρ value; a strategy found empirically (in Section 5.3) to produce acceptable results.

Alternatively, the optimal values for f_c and s_c can be learnt dynamically. For example, the value of s_c can be increased every time that $\#failures > f_c$, in other words, it becomes more difficult to reach the success state if failures occur frequently. This prevents the value of ρ from oscillating rapidly. Using this scheme the parameters can adapt to the local conditions of the error surface, with the ability to learn new settings when the error surface changes. A similar strategy can be designed for f_c.

The value of ρ is adapted in an attempt to learn the optimal size of the sampling volume given the current state of the algorithm. When a specific ρ value repeatedly results in a success, a larger sampling volume is selected to increase the maximum distance

traveled in one step. Conversely, if ρ produces f_c consecutive failures, then the sampling volume is too large and must be reduced.

When ρ becomes sufficiently small (compared to the machine's precision, for example) the algorithm can either halt or keep ρ fixed at this lower bound until some other stopping criterion is met. Note that halting may not be the best option, as information regarding the position of the other particles must also be taken into account. A typical example might be where some of the particles are still exploring a distant region of the search space, while the global best particle has already converged on the local minimum closest to it. In this case the distant particles may still be able to discover a better minimum, so the algorithm should continue until the maximum allowed number of iterations have been reached.

The PSO algorithm using equation (3.26) to update the position of its global best particle is called the Guaranteed Convergence Particle Swarm Optimiser (GCPSO), for reasons that will become clear in Section 3.3.

3.3 Convergence Proof for the PSO Algorithm

This section presents a proof showing that the Guaranteed Convergence Particle Swarm Optimiser (GCPSO), introduced in Section 3.2, is a local search algorithm that is guaranteed to converge on a local minimiser. Before the proof is presented, though, some convergence criteria are defined. A *local search* algorithm is only guaranteed to find a local minimiser of the objective function. In contrast, a *global search* algorithm is one that is guaranteed to find the global minimiser of the objective function.

3.3.1 Convergence Criteria

The stochastic nature of the particle swarm optimiser makes it more difficult to prove (or disprove) properties like global convergence. Solis and Wets have studied the convergence of stochastic search algorithms, most notably that of pure random search algorithms, providing criteria under which algorithms can be considered to be global search algorithms, or merely local search algorithms [128]. Solis and Wets's definitions are used extensively in the study of the convergence characteristics of the PSO presented below.

For convenience, the relevant definitions from [128] have been reproduced below.

Global Convergence Criteria

Proposition 1 *Given a function f from $\mathbb{R}^n$ to $\mathbb{R}$ and S a subset of $\mathbb{R}^n$. We seek a point z in S which minimizes f on S or at least which yields an acceptable approximation of the infimum of f on S.*

This proposition provides a definition of what a global optimiser must produce as output, given the function f and the search space S. The simplest stochastic algorithm designed to perform this task is the basic random search algorithm. In the k^{th} iteration, this algorithm requires a probability space $(\mathbb{R}^n, \mathcal{B}, \mu_k)$, where μ_k is a probability measure [134] (corresponding to a distribution function on $\mathbb{R}^n$) on $\mathcal{B}$, and $\mathcal{B}$ is the σ-algebra of subsets of $\mathbb{R}^n$. The support of the probability measure μ_k will be denoted M_k. That is, M_k is the smallest closed subset of $\mathbb{R}^n$ of measure 1 under μ_k. The algorithm also needs a random initial starting point in S, called $\mathbf{z}_0$.

Step 0: Find $\mathbf{z}_0$ in S and set $k = 0$.

Step 1: Generate a vector ξ_k from the sample space $(\mathbb{R}^n, \mathcal{B}, \mu_k)$.

Step 2: Set $\mathbf{z}_{k+1} = D(\mathbf{z}_k, \xi_k)$, choose μ_{k+1}, set $k := k + 1$ and return to step 1.

Figure 3.13: The Basic Random Search Algorithm

The basic random search algorithm is outlined in Figure 3.13, where D is a function that constructs a solution to the problem. The solution suggested by D carries the guarantee that the newly constructed solution will be no worse than the current solution. Therefore it satisfies the condition:

H 1 $f(D(\mathbf{z}, \xi)) \leq f(\mathbf{z})$ and if $\xi \in S$, then $f(D(\mathbf{z}, \xi)) \leq f(\xi)$

Different D functions lead to different algorithms, but condition (H1) must be satisfied for an optimisation algorithm to work correctly.

Global convergence of any algorithm means that the sequence $\{f(\mathbf{z}_k)\}_{k=1}^{\infty}$ converges onto the infimum of f on S. A pathological case would be a function f that has a minimum consisting of a single discontinuous point on an otherwise smooth function, for example,

$$f = \begin{cases} x^2 & \forall x \neq 1 \\ -10 & \text{if } x = 1 \end{cases}$$

To compensate for such cases, instead of searching for the infimum, a search is undertaken for the essential infimum ψ, defined as

$$\psi = \inf(t : v[\mathbf{z} \in S | f(\mathbf{z}) < t] > 0) \tag{3.29}$$

where $v[A]$ is the Lebesgue measure [134] on the set A. Equation (3.29) means that there must be more than one point in a subset of search space yielding function values arbitrarily close to ψ, so that ψ is the infimum of the function values from this nonzero v-measurable set. Typically, $v[A]$ is the n-dimensional volume of the set A. This definition of ψ avoids the problem with the pathological case mentioned above by defining a new infimum so that there is always a non-empty volume surrounding it containing points of S, the search space. This way it is possible to approach the infimum without having to sample every point in S.

Now, an optimality region can be defined as

$$R_\epsilon = \{\mathbf{z} \in S | f(\mathbf{z}) < \psi + \epsilon\} \tag{3.30}$$

where $\epsilon > 0$. If the algorithm finds a point in the optimality region, then it has found an acceptable approximation to the global minimum of the function.

It is now possible to consider whether an algorithm can in fact reach the optimality region of the search space. A local search method has μ_k with bounded support M_k so that $v[S \cap M_k] < v[S]$ for all k, except for possibly a finite number of k values. Thus not all the M_k span all of the search space — this implies that a region of search space may never be visited. On the other hand, a true global search algorithm satisfies the following assumption:

H 2 *For any (Borel) subset A of S with $v[A] > 0$, we have that*

$$\prod_{k=0}^{\infty} (1 - \mu_k[A]) = 0 \tag{3.31}$$

where $\mu_k[A]$ is the probability of A being generated by μ_k.

This means that for any subset A of S with positive measure v, the probability of repeatedly missing the set A using random samples (*e.g.* the ξ_k above), must be zero. Since $R_\epsilon \subset S$, this implies that the probability of sampling a point in the optimality region must be nonzero.

The following theorem is due to Solis and Wets [128].

Theorem 1 (Global Search) *Suppose that f is a measurable function, S is a measurable subset of $\mathbb{R}^n$ and (H1) and (H2) are satisfied. Let $\{z_k\}_{k=0}^{+\infty}$ be a sequence generated by the algorithm. Then*

$$\lim_{k\to+\infty} P[z_k \in R_\epsilon] = 1$$

where $P[z_k \in R_\epsilon]$ is the probability that at step k, the point z_k generated by the algorithm is in R_ϵ.

Proof: From (H1) it follows that z_k or ξ_k in R_ϵ implies that $x_{k'} \in R_\epsilon$ for all $k' > k$. Thus

$$P[z_k \in R_\epsilon] = 1 - P[z_k \in S \setminus R_\epsilon] \geq 1 - \prod_{l=0}^{k}(1 - \mu_l[R_\epsilon])$$

(where $S \setminus R_\epsilon$ denotes the set S with R_ϵ removed) and hence

$$1 \geq \lim_{k\to+\infty} P[z_k \in R_\epsilon] \geq 1 - \lim_{k\to+\infty} \prod_{l=0}^{k-1}(1 - \mu_l[R_\epsilon])$$

By (H2) we have that $\prod_{k=0}^{\infty}(1 - \mu_k[A]) = 0$, thus

$$1 \geq \lim_{k\to+\infty} P[z_k \in R_\epsilon] \geq 1 - 0 = 1$$

This completes the proof. ■

By the Global Search theorem, we thus have that an algorithm satisfying (H1) and (H2) is a global optimisation algorithm.

Local Convergence Criteria

The previous section described the conditions under which an algorithm can be said to be a stochastic global optimiser. Although the basic random search algorithm described there satisfies these conditions, it is far too slow to use as a practical algorithm. Local stochastic optimisers have better rates of convergence, at the expense of no longer guaranteeing that the global optimum will be found. This section describes the criteria required for local convergence, with the aim of proving in the next section that the GCPSO is a local optimisation algorithm with sure convergence.

For an algorithm that fails to satisfy the global search condition (H2), a local search condition can be defined [128]:

H 3 *To any* $\mathbf{z}_0 \in S$*, there corresponds a* $\gamma > 0$ *and an* $0 < \eta \leq 1$ *such that:*

$$\mu_k[(\text{dist}(D(\mathbf{z},\xi), R_\epsilon) < \text{dist}(\mathbf{z}, R_\epsilon) - \gamma) \quad \text{or} \quad (D(\mathbf{z},\xi) \in R_\epsilon)] \geq \eta \tag{3.32}$$

for all k *and all* $\mathbf{z}$ *in the compact set* $L_0 = \{\mathbf{z} \in S | f(\mathbf{z}) \leq f(\mathbf{z}_0)\}$.

In equation (3.32), $\text{dist}(\mathbf{z}, A)$ denotes the distance between a point $\mathbf{z}$ and a set A, being defined as

$$\text{dist}(\mathbf{z}, A) = \inf_{\mathbf{b} \in A} \text{dist}(\mathbf{z}, \mathbf{b})$$

Therefore, an algorithm is a local optimisation algorithm if a nonzero η can be found such that at every step the algorithm can move $\mathbf{z}$ closer to the optimality region by at least distance γ, or $\mathbf{z}$ is already in the optimality region, with a probability greater or equal to η.

Combining conditions (H1) and (H3) leads to the desired local convergence theorem [128].

Theorem 2 (Local Search) *Suppose that* f *is a measurable function,* S *is a measurable subset of* $\mathbb{R}^n$ *and (H1) and (H3) are satisfied. Let* $\{\mathbf{z}_k\}_{k=0}^{\infty}$ *be a sequence generated by the algorithm. Then,*

$$\lim_{k \to \infty} P[\mathbf{z}_k \in R_\epsilon] = 1$$

where $P[\mathbf{z}_k \in R_\epsilon]$ *is the probability that at step* k*, the point* $\mathbf{z}_k$ *generated by the algorithm is in the optimality region,* R_ϵ.

Proof: Let $\mathbf{z}_0$ be the point generated in Step 0 of the search algorithm. Since L_0 is compact, there always exists an integer p such that (by assumption H3)

$$\gamma p > \text{dist}(\mathbf{a}, \mathbf{b}) \quad \forall \mathbf{a}, \mathbf{b} \in L_0.$$

By (H3) it follows that

$$P[\mathbf{z}_1 \in R_\epsilon] \geq \eta$$

and

$$P[\mathbf{z}_2 \in R_\epsilon] \geq \eta \times P[\mathbf{z}_1 \in R_\epsilon] \geq \eta^2.$$

These probabilities are disjoint, so repeated application (p times) of (H3) yields

$$P[\mathbf{z}_p \in R_\epsilon] \geq \eta^p$$

Applying (H3) another p times results in

$$P[\mathbf{z}_{2\times p} \in R_\epsilon] \geq \eta^{2\times p}$$

Hence, for $k = 1, 2, \ldots$

$$P[\mathbf{z}_{kp} \in R_\epsilon] = 1 - P[\mathbf{z}_{kp} \notin R_\epsilon] \geq 1 - (1 - \eta^p)^k.$$

Now (H1) implies that $\mathbf{z}_1, \ldots, \mathbf{z}_{p-1}$ all belong to L_0 and by the above it then follows that

$$P[\mathbf{z}_{kp+l} \in R_\epsilon] \geq 1 - (1 - \eta^p)^k \quad \text{for} \quad l = 0, 1, \ldots, p-1.$$

This shows that all steps between kp and $(k+1)p$ satisfy (H3). This completes the proof, since $(1 - \eta^p)^k$ tends to 0 as k goes to $+\infty$. ■

3.3.2 Local Convergence Proof for the PSO Algorithm

The proof presented here casts the PSO into the framework of a local stochastic search algorithm, thus allowing the use of Theorem 2 to prove convergence. Thus it remains to show that the PSO satisfies both (H1) and (H3). The proof will first be presented for unimodal optimisation problems, after which the multi-modal case will be discussed.

Unimodal functions

The proof for the PSO starts by choosing the initial value

$$\mathbf{x}_0 = \arg\max_{\mathbf{x}_i}\{f(\mathbf{x}_i)\},\ i \in 1..s,$$

where $\mathbf{x}_i$ represents the position of particle i. That is, $\mathbf{x}_0$ represents the worst particle, yielding the largest f value, of all particles in the swarm. Now define $L_0 = \{\mathbf{x} \in S | f(\mathbf{x}) \leq f(\mathbf{x}_0)\}$, the set of all points with f values smaller than that of the worst particle $\mathbf{x}_0$. It is assumed that all the particles lie in the same 'basin' of the function, so that $\mathbf{x}_i, \mathbf{y}_i \in L_0$, with L_0 compact. In the next section this will be extended to the general case with multiple basins.

From equations (2.7)–(2.8), function D (as introduced in assumption H1) is defined for the PSO as

$$D(\hat{\mathbf{y}}_k,\ \mathbf{x}_{i,k}) = \begin{cases} \hat{\mathbf{y}}_k & \text{if } f(\mathbf{g}(\mathbf{x}_{i,k})) \geq f(\hat{\mathbf{y}}_k) \\ \mathbf{g}(\mathbf{x}_{i,k}) & \text{if } f(\mathbf{g}(\mathbf{x}_{i,k})) < f(\hat{\mathbf{y}}_k) \end{cases} \tag{3.33}$$

The notation $\mathbf{g}(\mathbf{x}_{i,k})$ denotes the application of $\mathbf{g}$, the function performing the PSO updates, the definition of which follows below. Further, the symbol $\mathbf{x}_{i,k}$ will be used instead of the usual notation $\mathbf{x}_i(t)$, to stress the discrete nature of the time step. The definition of D above clearly complies with (H1), since the sequence $\hat{\mathbf{y}}$ is monotonic by definition.

Note that the dependence on the time step, k, has been made explicit above. The sequence $\{\hat{\mathbf{y}}_l\}_{l=0}^{k}$ is a sequence of the best positions amongst all the positions that all the particles visited up to and including step k.

It is permissible to view the computation of the value of $\mathbf{x}_{i,k+1}$ as the successive application of three functions $\mathbf{g}_1$, $\mathbf{g}_2$ and $\mathbf{g}_3$, each function adding a term to the previous result. Thus:

$$\mathbf{x}_{i,k+1} = \mathbf{g}(\mathbf{x}_{i,k}) = \mathbf{g}_1(\mathbf{x}_{i,k}) + \mathbf{g}_2(\mathbf{x}_{i,k}) + \mathbf{g}_3(\mathbf{x}_{i,k}) \tag{3.34}$$

where

$$\mathbf{g}_1(\mathbf{x}_{i,k}) \ = \ \mathbf{x}_{i,k} + w\mathbf{v}_{i,k} \tag{3.35}$$

Since $\mathbf{g}$ is a vector function, the next two equations will be specified component-wise to simplify the notation, thus $g_2(\mathbf{x}_{i,k})_j$ denotes the j^{th} dimension of the function $\mathbf{g}_2$. Two

uniform pseudo-random number sequences, $r_1(t)$ and $r_2(t)$, are used below, as defined for equations (2.7) and (2.8).

$$g_2(\mathbf{x}_{i,k})_j = c_1 r_{1,j}(t)[\mathbf{y}_{i,j,k} - \mathbf{x}_{i,j,k}] \quad \forall j \in 1..n \tag{3.36}$$

$$g_3(\mathbf{x}_{i,k})_j = c_2 r_{2,j}(t)[\hat{\mathbf{y}}_{j,k} - \mathbf{x}_{i,j,k}] \quad \forall j \in 1..n \tag{3.37}$$

By the definition of L_0, and the assumption that all the particles are initially in L_0, we have that:

$$\mathbf{y}_{i,0}, \mathbf{x}_{i,0} \in L_0$$

Let $\mathbf{g}^N(\mathbf{x}_{i,k})$ denote N successive applications of $\mathbf{g}$ on $\mathbf{x}_{i,k}$. Then,

Lemma 1 *There exists a value* $1 \leq N \leq +\infty$ *so that* $||\mathbf{g}^n(\mathbf{x}_{i,k}) - \mathbf{g}^{n+1}(\mathbf{x}_{i,k})|| < \epsilon, \forall n \geq N, \epsilon > 0$, *subject to choices of* w, ϕ_1 *and* ϕ_2 *so that* $\max(||\alpha||, ||\beta||) < 1$.

Proof: In Section 3.1 it was shown that

$$\lim_{t\to+\infty} x(t) = \lim_{t\to+\infty} k_1 + k_2\alpha^t + k_3\beta^t = \frac{\phi_1 y + \phi_2 \hat{y}}{\phi_1 + \phi_2} \tag{3.38}$$

and from equation (2.8) it follows that $\mathbf{x}(t+1) - \mathbf{x}(t) = \mathbf{v}(t+1)$. Therefore

$$\begin{aligned} \lim_{t\to+\infty} v(t+1) &= \lim_{t\to+\infty} x(t+1) - x(t) \\ &= \lim_{t\to+\infty} k_2\alpha^t(\alpha - 1) + k_3\beta^t(\beta - 1) \\ &= 0 \end{aligned} \tag{3.39}$$

when $||\alpha||, ||\beta|| < 1$. Now it follows, from equations (2.7) and (2.8), that

$$\begin{aligned} x(t+1) &= x(t) + v(t+1) \\ &= x(t) + wv(t) - x(t)(\phi_1 + \phi_2) + y\phi_1 + \hat{y}\phi_2 \end{aligned}$$

But by equation (3.38), we see that $x(t+1) = x(t)$ in the limit of t. Further, by equation (3.39), $v(t) = 0$ as $t \to +\infty$. Thus, in the limit

$$\begin{aligned} x(t) &= x(t) - x(t)(\phi_1 + \phi_2) + y\phi_1 + \hat{y}\phi_2 \\ \Rightarrow\ & -x(t)(\phi_1 + \phi_2) + y\phi_1 + \hat{y}\phi_2 = 0 \end{aligned}$$

This is certainly true when $x(t) = y = \hat{y}$. Thus the system will converge (but not necessarily on the local minimiser) when both y and $x(t)$ coincide with $\hat{y}$. ■

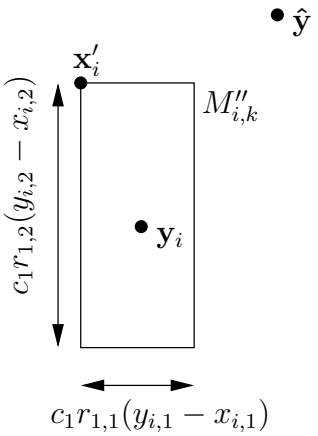


Figure 3.14: A depiction of the hyper-rectangular support of the space sampled by the function $\mathbf{g}_2$. Note that $\mathbf{y}_i$ need not be included in $M''_{i,k}$ — this depends on the value of c_1.

In short, once the PSO algorithm reaches the state where all $\mathbf{x}_i = \mathbf{y}_i = \hat{\mathbf{y}}$, $\forall i \in 1..s$, then the algorithm must stop, since no further progress can be made. The danger exists that the algorithm may reach this state before $\hat{\mathbf{y}}$ reaches the minimiser (local or global) of the function.

To better understand how the PSO generates new solutions, the components of $\mathbf{g}$ can be regarded separately. The application of function $\mathbf{g}_2$ can be seen as the action of sampling a point from a distribution with hyper-rectangular support $M''_{i,k}$, as shown in Figure 3.14. The side lengths of the hyper-rectangle $M''_{i,k}$ are dependent on the distance that $\mathbf{x}_{i,k}$ is from $\mathbf{y}_i$. Lemma 1 shows that this distance tends to zero, since in the limit $\mathbf{x}_{i,k} = \mathbf{y}_{i,k} = \hat{\mathbf{y}}_{i,k}$. This clearly violates assumption (H3), since the probability of sampling a point closer to the optimality region R_ϵ becomes zero before $\hat{\mathbf{y}}$ necessarily reaches R_ϵ. Note that the same argument applies to $\mathbf{g}_3$ and $\hat{\mathbf{y}}$, and is commonly referred to as *premature convergence*.

Examples of states that converge prematurely can easily be constructed. Consider a two-dimensional search space and a swarm with only two particles. One of the particles, say particle 0, will be the global best particle. Let the symbols $a_1..a_5$ and $p_1..p_2$ denote

arbitrary constants. Then the swarm will stagnate whenever it reaches the state

$$\begin{aligned}
\mathbf{v}_0 &= a_1(0,1) \\
\mathbf{v}_1 &= a_2(0,1) \\
\mathbf{x}_0 &= (p_1,p_2) \\
\mathbf{x}_1 &= (p_1,p_2) + a_3(0,1) \\
\mathbf{y}_0 &= (p_1,p_2) + a_4(0,1) \\
\mathbf{y}_1 &= (p_1,p_2) + a_5(0,1)
\end{aligned}$$

This is the state where all the particles are constrained to move only along one of the dimensions of the search space. There is a non-zero probability that the swarm could reach this state, or it could even have been initialized to this state. If the minimiser of the function is not of the form (p_1, p_3), where p_3 is an arbitrary value, then the swarm will not be able to reach it. The fundamental problem here is that all movement in the swarm is relative to other particles in the swarm — there is no "external force" to "bump" the particles to break free of the state described above.

In summary, there exists initial states in which the original PSO algorithm can start that will lead to a stagnant state in a finite number of steps. By using a large number of particles, the probability of becoming trapped in such a stagnant state is reduced dramatically. It is shown experimentally in Section 5.4, however, that with only 2 particles the swarm can rapidly stagnate. This implies that the basic PSO algorithm is not a local search algorithm, since it is not guaranteed to locate a minimiser from an arbitrary initial state.

One solution to the problem of stagnation is to use the modified velocity update step for the global best particle as presented in equation (3.26). The resulting algorithm is called the GCPSO, or Guaranteed Convergence PSO. Consider the new position update step derived by substituting (3.26) into (2.8):

$$x_{\tau,j}(t+1) = \hat{y}_j(t) + wv_{\tau,j}(t) + \rho(t)(1 - 2r_2(t))$$

This represents the action of sampling a point from a hypercube with side lengths 2ρ centered around $\hat{\mathbf{y}} + w\mathbf{v}$. Let M_k denote this hypercube, and let μ_k denote the uniform

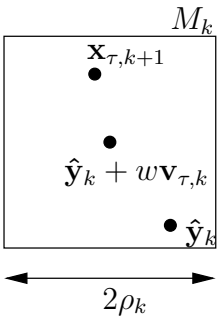


Figure 3.15: The hyper-cubic support M_k of the sample space μ_k, centered around the point $\hat{\mathbf{y}}_k + w\mathbf{v}_k$, as defined by the position update step of the GCPSO. The point $\mathbf{x}_{k+1}$ is an example of a point sampled by the new velocity update step.

probability measure defined on the hypercube M_k. Figure 3.15 illustrates how a new sample $\mathbf{x}_{\tau,k+1}$ is generated. Stagnation is prevented by ensuring that $\rho > 0$ for all time steps. Before proving that the GCPSO is a local search algorithm, a few details regarding $\mathbf{x}_{\tau,k}$ must first be discussed.

Note that $\hat{\mathbf{y}}$ is always in L_0. It is possible, however, that $\mathbf{x}_{\tau,k} \notin L_0$, due to the cumulative effect of a growing $\mathbf{v}$ vector, so that $\hat{\mathbf{y}} + w\mathbf{v}_k \notin L_0$. One of two scenarios now unfolds: $\hat{y} \in M_k$ or $\hat{y} \notin M_k$. In the first case, this means that a point arbitrarily close to $\hat{y}$ may be sampled, including $\hat{y}$ itself. Since $\hat{y} \in L_0$, this means that $v[M_k \cap L_0] > 0$, that is, the intersection of M_k and L_0 is not empty. The second case implies that ρ is such that M_k does not include $\hat{y}$. This happens when $\mathbf{v}_\tau(t)$ points outwards from L_0. Since $\hat{\mathbf{y}}$ is only updated when a better solution is found, and from the definition of L_0, it is clear that none of the points outside of L_0 will be selected to replace $\hat{\mathbf{y}}$. On the other hand, $\mathbf{x}_\tau(t)$ is able to move outside of L_0 because of the residual velocity $\mathbf{v}_\tau(t)$. Assume for the moment that ρ is insignificantly small. Equation (3.26) shows that, if $w < 1$, then clearly $||\mathbf{x}_\tau(t+1) - \hat{\mathbf{y}}|| < ||\mathbf{x}_\tau(t) - \hat{\mathbf{y}}||$, assuming that $\mathbf{x}_\tau(t) \neq \hat{\mathbf{y}}$. After a finite number of time steps l, $\mathbf{x}_\tau(t+l)$ will be in L_0 once more. This implies that $v[M_{k+l} \cap L_0] > 0$, so that a point arbitrarily close to $\hat{\mathbf{y}}$ can be sampled once more.

Both cases imply that a new sampled point arbitrarily close to $\hat{\mathbf{y}}$, and thus in L_0, can be generated. Note that the second case only comes into play when $\hat{\mathbf{y}}$ is close to the boundary of L_0. The first case, where $M_k \subset L_0$, can be considered the norm.

The existence of a non-degenerate sampling volume μ_k with support M_k has thus been shown for the GCPSO algorithm. Using this fact, it is now possible to consider the local convergence property of the GCPSO algorithm.

If we assume that S is compact and has a non-empty interior, then L_0 will also be compact with a non-empty interior. Further, L_0 will include the essential infimum, contained in the optimality region R_ϵ, by definition. Now R_ϵ is compact with a non-empty interior, thus we can define a ball B' centered at $\mathbf{c}'$ contained in R_ϵ, as shown in Figure 3.16. Now pick the point $\mathbf{x}' \in \arg\max_{\mathbf{x}}\{\text{dist}(\mathbf{c}', \mathbf{x}) | \mathbf{x} \in L_0\}$, as illustrated in Figure 3.16. Let B be the hypercube centered at $\mathbf{c}'$, with sides of length $2(\text{dist}(\mathbf{c}', \mathbf{x}') - 0.5\rho)$.

Let C be the convex hull of $\mathbf{x}'$ and B' (see Figure 3.16). Consider a line tangent to B', passing through $\mathbf{x}'$ (*i.e.* one of the edges of C). This line is the longest such line, for $\mathbf{x}'$ is the point furthest from B'. This implies that the angle subtended by $\mathbf{x}'$ is the smallest such angle of any point in L_0. In turn, this implies that the volume $C \cap B$ is smaller than that of $C' \cap B$ for any other convex hull C' defined by any arbitrary point $\mathbf{x} \in L_0$.

Then for all $\mathbf{x}$ in L_0

$$\mu_k[\text{dist}(D(\hat{\mathbf{y}}, \mathbf{x}_\tau), R_\epsilon) < \text{dist}(\mathbf{x}, R_\epsilon) - 0.5\rho] \geq \eta = \mu[C \cap B] > 0 \tag{3.40}$$

where μ_k is the uniform distribution on the hypercube centered at $\mathbf{x}$, with side length 2ρ. It was shown above that the modified PSO can provide such a hypercube.

Since $\mu[C \cap B] > 0$, the probability of selecting a new point $\mathbf{x}$ so that it is closer to the optimality region R_ϵ is always nonzero.

This is sufficient to show that the Guaranteed Convergence Particle Swarm Optimiser (GCPSO) complies with (H3), because

1. The GCPSO can always generate a sample around a point in L_0, assuming $\hat{\mathbf{y}}, \mathbf{y}_i \in L_0$;

2. Given any starting point in L_0, the GCPSO algorithm guarantees a non-degenerate sampling volume with a nonzero probability of sampling a point closer to the optimality region, R_ϵ.

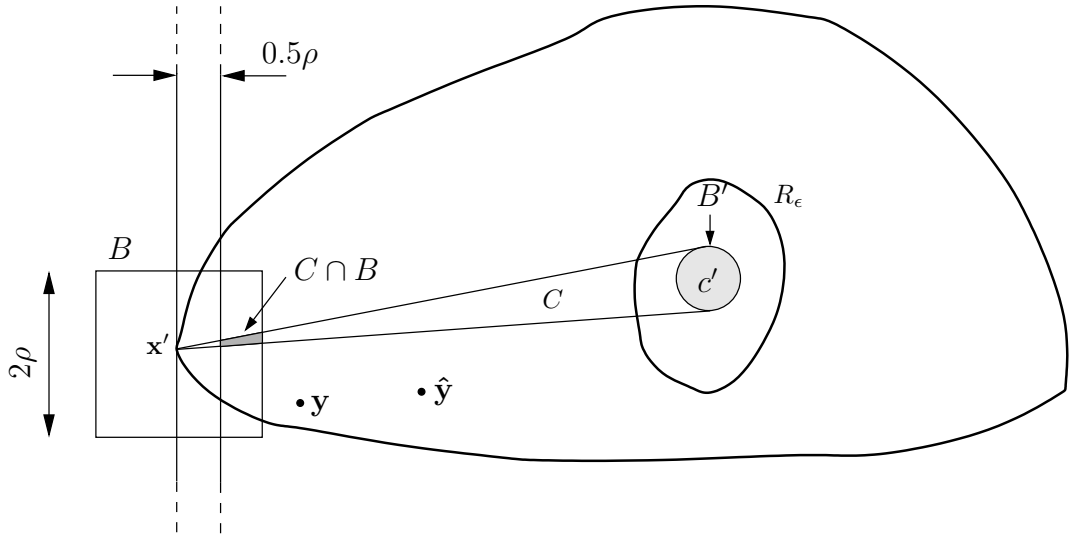


Figure 3.16: The intersection $C \cap B$

The GCPSO algorithm thus satisfies both (H1) and (H3).

This completes the proof that the sequence of values $\{\hat{\mathbf{y}}_k\}_{k=0}^{\infty}$ generated by the GCPSO algorithm will converge to the optimality region, under the constraints of a local optimisation algorithm, regardless of the initial state of the swarm.

Functions with multiple minima

It was assumed above that L_0 was convex-compact. A non-unimodal function, with S including multiple minima, will result in a non-convex set L_0. Even if all the particles are contained in the same convex subset, the modified PSO is not guaranteed to yield a point in the same convex subset as it started from, especially not during the earlier iterations. This is because the velocity update can yield a value larger than the diameter of the basin in which the particle currently resides. If the point found in a different convex subset yields a function value smaller than the current global best value, then the algorithm will move its global best position to this new convex subset of L_0. By Lemma 1, all the particles will eventually move into the new convex subset. This process could continue until the algorithm converges onto the essential infimum contained in its convex subset, at which point it will no longer be able to 'jump' out of the convex subset

if the diameter of the subset is greater than 2ρ, since the velocity update (*i.e.* the step size) will be smaller than ρ. The process of jumping between basins cannot continue *ad infinitum*, since all the particles will converge onto the position of the global best particle, as illustrated in Section 3.1.

What this implies is that a non-convex set L_0 does not preclude the algorithm from converging, but it illustrates that the algorithm will converge onto some local minimum, instead of the global minimum.

3.4 Stochastic Global PSOs

In the previous section it was proved that the GCPSO algorithm converges on a local minimiser with probability 1, as the number of iterations approaches infinity. It is possible to extend the GCPSO to become a stochastic global search algorithm, so that it will locate the global minimiser of the objective function. Unfortunately, the requisite number of iterations (and thus the time) to reach this global minimiser is unbounded when dealing with search spaces containing an infinite number of points. All computer implementations of search algorithms will work in a finite-precision approximation of $\mathbb{R}^n$, however, so it is possible to say that the upper bound is equal to the number of iterations required to examine every location in the search space. A technique that examines every location in search space is highly impractical — unfortunately this is the only way to obtain 100% certainty that the global minimiser has been found. This implies that algorithms that don't examine every point have at best only an asymptotic probability of 1 of locating the global minimiser as the number of iterations approaches infinity (or the number of points in search space). Two algorithms with this property are introduced below, but first it is shown that the GCPSO, as well as the original PSO, are not global search algorithms.

3.4.1 Non-Global PSOs

Theorem 1, defined in Section 3.3, specifies under which conditions an algorithm can be considered to be a global method. More importantly, from this theorem one can conclude that an algorithm that fails to satisfy (H2) is not a global search method. Recall the

definition of (H2):

For any (Borel) subset A of S with $v[A] > 0$, we have that

$$\prod_{k=0}^{\infty}(1-\mu_k[A]) = 0$$

where $\mu_k[A]$ is the probability of A being generated by μ_k.

This means that any arbitrary set $A \subset S$ must have a non-zero probability of being examined by the algorithm *infinitely often*. In other words, the condition must hold for all time steps t, with only a finite number of exceptions. Since $R_\epsilon \subset S$ this clearly implies that an algorithm that has a non-zero probability of generating samples from an arbitrary subset of S will thus (eventually) generate a sample $\xi \in R_\epsilon$. Conversely, any algorithm that consistently ignores any region of search space is not guaranteed to generate a sample $\xi \in R_\epsilon$, unless the algorithm had *a priori* information regarding the location of the global minimiser. This implies that a general algorithm, without *a priori* knowledge, must be able to generate an infinite number of samples distributed throughout the whole of S in order to guarantee that it will find the global minimiser with asymptotic probability 1.

Lemma 2 *The PSO algorithm does not satisfy (H2).*

Proof: Consider the mechanism that the original PSO uses to generate new samples. To satisfy (H2), the union of the sample spaces of the particles must cover S, so that

$$S \subseteq \cup_{i=1}^{s} M_{i,k}$$

at time step k, where $M_{i,k}$ denotes the support of the sample space of particle i. The shape of $M_{i,k}$ is defined as follows (derived from equation 3.3):

$$\begin{aligned} M_{i,k} &= (1+w-\phi_1-\phi_2)x_{i,j,k-1} - wx_{i,j,k-2} + \phi_1 y + \phi_2 \hat{y} \\ &= x_{i,j,k-1} + w(x_{i,j,k-1} - x_{i,j,k-2}) + \phi_1(y - x_{i,j,k-1}) + \phi_2(\hat{y} - x_{i,j,k-1}) \end{aligned}$$

where $0 \leq \phi_1 \leq c_1$ and $0 \leq \phi_2 \leq c_2$, and $x_{i,j,k}$ denotes the value of dimension j of the position of particle i at time step k. $M_{i,k}$ is a hyper-rectangle parameterised by ϕ_1 and

ϕ_2, with one corner specified by $\phi_1 = \phi_2 = 0$ and the other by $\phi_1 = c_1$ and $\phi_2 = c_2$. Regardless of the location of these corners it is clear that $v[M_{i,k} \cap S] < v[S]$ whenever

$$\max(c_1|y - x_{i,j,k-1}|, c_2|\hat{y} - x_{i,j,k-1}|) < 0.5 \times \text{diam}_j(S)$$

where $\text{diam}_j(S)$ denotes the length of S along dimension j. In Section 3.1 it was shown that $\mathbf{x}_i$ converges on $(c_1\mathbf{y}_i + c_2\hat{\mathbf{y}})/(c_1 + c_2)$, so that the side lengths of $M_{i,k}$ tend to 0 as k tends to infinity. Since the volume of each individual $M_{i,k}$ becomes smaller with increasing k values, it is clear that the volume of their union, $v[\cup_{i=1}^{s} M_{i,k}]$, must also decrease. This shows that, except for $k < k'$, with k' finite,

$$v[\cup_{i=1}^{s} M_{i,k} \cap S] < v[S]$$

so that the $M_{i,k}$ cannot cover S. Note that $k' = 0$ is also a valid state. Therefore there exists a finite k' so that for all $k \geq k'$ there will be a (possibly disjoint) set $A \subset S$ with $\sum_{i=0}^{s} \mu_{i,k}[A] = 0$, which implies that the PSO fails to satisfy (H2). ■

Since the original PSO fails, by Lemma 2, to satisfy (H2) it thus fails to satisfy Theorem 1 and is therefore not a global search algorithm.

Lemma 3 *The GCPSO algorithm does not satisfy (H2).*

Proof: It was shown in Lemma 2 that the update equations used by the original PSO fail to satisfy (H2). These equations are used in the GCPSO without modification to update all the particles except the global best particle, thus this update equation must now be examined. The support $M_{\tau,k}$ of the global best particle is described by

$$x_{\tau,j,k} = \hat{y}_{j,k} + wv_{\tau,j,k} + \rho_k(1 - 2r_k))$$

where $0 \leq r_k \leq 1$. The side length of the sampling volume is thus determined by ρ_k, which implies that $v[M_{\tau,k} \cap S] < v[S]$ whenever

$$\rho_k < \text{diam}(S)$$

The parameter ρ is updated according to the rules presented in Section 3.2. These rules cause ρ to decrease whenever the PSO consistently fails to improve the best solution discovered so far. This will happen once the GCPSO converges on a local minimum, so

$$\lim_{k \to +\infty} \rho_k = \rho_{min}$$

where ρ_{min} is an arbitrarily small value, with ρ_{min} typically several orders of magnitude smaller than the diameter of S. Keeping in mind that $\tau \in 1..s$, this implies that

$$v[\cup_{i=1}^{s} M_{i,k} \cap S] < v[S]$$

The GCPSO consequently fails to satisfy (H2). ■

By Theorem 1 we thus see that the GCPSO is not a global search algorithm, since it fails to satisfy (H2). The following section will propose some methods for extending the GCPSO to become a global search algorithm.

3.4.2 Random Particle Approach (RPSO)

The simplest way to construct a PSO-based global search algorithm is to directly address (H2). This can be achieved by adding *randomised particles* to the swarm. Particle i can be made a randomised particle by simply resetting its position to a random position in search space periodically.

Any number of particles in the swarm can be made random particles, but the optimal ratio of random versus normal particles depends on the swarm size. Let s_{rand} denote the number of random particles in the swarm. One possible implementation of the random particle approach is outlined in Figure 3.17. This implementation resets a specific particle's position only every s_{rand} iterations, allowing the particle to explore the region in which it was initialised before resetting it again. The resulting algorithm is called the Randomised Particle Swarm Optimiser, or RPSO.

It is trivial to show that the new RPSO algorithm is a global search algorithm. The personal best position update equations are unaltered, thus the algorithm clearly satisfies (H1), as was shown for the original PSO in Section 3.3. During each iteration one particle assumes a random position in the search space. The sample space from which this sample is drawn has support $M_k = S$, so that $v[M_k] = v[S]$. This satisfies (H2), so by Theorem 1 this is a global search algorithm.

3.4.3 Multi-start Approach (MPSO)

A different method of extending the GCPSO algorithm to become a global search algorithm can be constructed as follows:

Create and initialise an n-dimensional PSO : P
$s_{idx} \leftarrow 0$
repeat:
  if $s_{idx} \neq \tau$
    then $P.\mathbf{x}_{s_{idx}} = \text{random\_vector}()$
  $s_{idx} = (s_{idx} + 1)$ modulo s_{rand}
  for each particle $i \in [1..s]$ **:**
    if $f(P.\mathbf{x}_i) < f(P.\mathbf{y}_i)$
      then $P.\mathbf{y}_i = P.\mathbf{x}_i$
    if $f(P.\mathbf{y}_i) < f(P.\hat{\mathbf{y}})$
      then $P.\hat{\mathbf{y}} = P.\mathbf{y}_i$
  endfor
  Perform PSO updates on P using equations (2.7–2.8)
until stopping condition is true

Figure 3.17: Pseudo code for the RPSO algorithm

1. Initialise all the particles to random positions in the search space
2. Run the GCPSO algorithm until it converges on a local minimiser. Record the position of this local minimiser, and return to step 1.

There exist several criteria that can be used to determine whether the GCPSO algorithm has converged.

Maximum Swarm Radius: The maximum swarm radius can be computed directly using

$$r = \|x_m - x_\tau\|, \ m \in 1..s$$

where

$$\|x_m - x_\tau\| \geq \|x_i - x_\tau\|, \ \forall i \in 1..s$$

The normalised radius

$$r_{norm} = \frac{r}{\text{diam(S)}}$$

can then be used to decide how close the particles are to the global best particle. The analysis in Section 3.1 suggests that the swarm stagnates when all the particles coincide with the global best particle in search space. When r_{norm} is close to zero, the swarm has little potential for improvement, unless the global best particle is still moving. Alternatively, a single particle may still be wandering around while all the other particles have already coincided with the global best particle. Therefore this method is not reliable enough, especially since it does not take the objective function values into account.

It was found empirically that re-starting the swarm when $r_{norm} < 10^{-6}$ produced acceptable results on a test set of benchmark functions. Clearly smaller thresholds will increase the sensitivity of the algorithm, but note that re-starting the swarm too frequently will prevent it from performing a fine-grained search.

Cluster Analysis: A more aggressive variant of the Maximum Swarm Radius method can be constructed by clustering the particles in search space. The clustering algorithm works as follows:

1. Initialise the cluster C to contain only the global best position
2. All particles such that $\text{dist}(\mathbf{x}_i, C) < r_{thresh}$ are added to the cluster
3. Repeat steps 2–3 about 5 times.

The ratio $|C|/s$ is then computed, where $|C|$ denotes the number of particles in the cluster — note that only a single cluster is grown. If the ratio is greater than some threshold, say 0.6, then the swarm is considered to have converged. Note that this method has the same flaws as the Maximum Swarm Radius technique, except that it will more readily decide that the swarm has converged.

Empirical results obtained on a small set of test functions indicated that a value of $r_{thresh} = 10^{-6}$ produced acceptable results; the swarm was declared to have converged when more than 60% of the particles were clustered around the global best particle. Using a ratio of more than 60% decreases the sensitivity of the algorithm, bringing with it the possibility of failing to detect stagnation. Smaller

r_{thresh} values will increase the sensitivity, similarly to the corresponding threshold value in the Maximum Swarm Radius technique.

Objective Function Slope: This approach does not take into account the relative positions of the particles in search space; instead it bases its decision solely on the rate of change in the objective function. A normalised objective function value is obtained by using

$$f_{ratio} = \frac{f(\hat{\mathbf{y}}(t)) - f(\hat{\mathbf{y}}(t-1))}{f(\hat{\mathbf{y}}(t))}$$

If this normalised value is smaller than some threshold, a counter is incremented. Once the counter reaches a certain threshold, the swarm is assumed to have converged. This approach is superior to the other two methods mentioned first in that it actually determines whether the swarm is still making progress, instead of trying to infer it from the positions of the particles. There is one remaining flaw with this approach, though. If half of the particles (including the global best particle) are trapped in the basin of some minimum, the other half may yet discover a better minimum in the next few iterations. This possibility can be countered for by using one of the first two methods to check for this scenario.

The optimal threshold for the f_{ratio} value depends on the range of the objective function values, as well as the machine precision of the platform on which the algorithm is implemented. Empirical results indicated that a value of 10^{-10} works well. Smaller thresholds increases the sensitivity of the algorithm, possibly causing the algorithm to mistake a period of slow progress for stagnation.

Figure 3.18 is the outline of an algorithm making use of the multi-start (or restart) approach. Any of the convergence criteria mentioned above can be used. This type of algorithm is called the Multi-start Particle Swarm Optimiser (MPSO).

The MPSO algorithm is a global search algorithm, a property that will now be proved using Theorem 1. The MPSO satisfies (H1), similarly to the RPSO in the previous section. To satisfy (H2), the MPSO must be able to restart an infinite number of times. This requires that the GCPSO algorithm converges onto a local minimum, which was proved in Section 3.3, and that the convergence-detection mechanism subsequently detects this. The Maximum Swarm Radius and Cluster Analysis methods indicate that

Create and initialise an n-dimensional PSO : P
repeat:
  if $f(P.\hat{\mathbf{y}}) < f(\mathbf{z})$
    then $\mathbf{z} = P.\hat{\mathbf{y}}$
  if P has converged
    then re-initialise all particles
  for each particle $i \in [1..s]$ **:**
    if $f(P.\mathbf{x}_i) < f(P.\mathbf{y}_i)$
      then $P.\mathbf{y}_i = P.\mathbf{x}_i$
    if $f(P.\mathbf{y}_i) < f(P.\hat{\mathbf{y}})$
      then $P.\hat{\mathbf{y}} = P.\mathbf{y}_i$
  endfor
  Perform PSO updates on P using equations (2.7–2.8)
until stopping condition is true

Figure 3.18: Pseudo code for the MPSO algorithm

the swarm has converged when the particles are arbitrarily close to the global best position, $\hat{\mathbf{y}}$. In Section 3.3 it was shown that the particles tend to the state $\mathbf{x}_i = \mathbf{y}_i = \hat{\mathbf{y}}$, where this property was called stagnation. Since the swarm is guaranteed to reach this state, these two convergence criteria will always detect convergence and trigger a restart. The Objective Function Slope criterion will detect convergence whenever the value of $f(\hat{\mathbf{y}})$ stops changing. This state is guaranteed, upon discovery of a local minimum, by the local convergence property of the GCPSO. This implies that, regardless of the convergence-detection criterion used, the MPSO algorithm will be able to re-initialise the positions of all the particles an infinite number of times, if it is allowed to run for an infinite number of iterations.

The re-initialisation process assigns to each particle a position sampled from the whole search space S. Since the support of this sample space, M_k, is equal to the search space, we have that $v[M_k] = v[S]$. This satisfies (H2), which means that, by Theorem 1, the MPSO is a global search algorithm.

It is interesting to note that global convergence can be proved without requiring the local part to be a guaranteed local search algorithm. All that is required is that the local part of the algorithm must be able to satisfy some termination criterion — not necessarily convergence onto a local minimiser. As long as the algorithm re-initialises the population an infinite number of times the algorithm will satisfy (H2). This means that the GCPSO component in the MPSO algorithm can be replaced with the original PSO, and it will still be a global search algorithm. Note that the rate of convergence will be affected by this substitution, though.

3.4.4 Rate of Convergence

The rate of convergence of a stochastic global method like the MPSO is directly dependent on the volume of the sample space, since the number of points in the sample space grows exponentially with the dimension of the search space. This implies that the MPSO algorithm will easily find the global minimiser of a 2-dimensional objective function in a relatively small number of iterations. If a function of comparable complexity in 20 dimensions is considered, the algorithm will take significantly longer to find the global minimiser. An indication of the severity of the problem can be obtained by considering the following simple example. Let d be the side length of a hypercube defining the optimality region R_ϵ. The volume of the optimality region is then d^n, where n is the number of dimensions. If the search space S is a hypercube with side lengths l, then its volume will be l^n. The probability of generating a sample in the optimality region in the first iteration of the algorithm, assuming a uniform distribution function on S, is then

$$\frac{d^n}{l^n} = \left(\frac{d}{l}\right)^n$$

Since the optimality region is certainly smaller than the search space itself, this implies that $d/l < 1$, so that

$$\lim_{n\to+\infty} \left(\frac{d}{l}\right)^n = 0$$

If a pseudo-random number algorithm is used to generate the samples used by the search algorithm, and the period of the generator is sufficiently large, then the sampling process will be equivalent to sampling without replacement. The probability of hitting the

optimality region will thus increase slightly on successive samples, but this will not have a significant impact since the number of points in the search space still grows exponentially with the number of dimensions. This implies that the ability of the MPSO to find the global minimiser of a function in a finite number of iterations deteriorates very rapidly as the number of dimensions is increased.

The stretching technique proposed by Parsopoulos *et al.*, described in Section 2.5, uses the equivalent of a MPSO using the original PSO instead of the GCPSO for the local search component of their algorithm. They do not provide any proofs of global convergence, but they present empirical results to claim that their method is global. Their algorithm modifies the objective function, and re-initialises the particles once a local minimiser is discovered. It is important to realise, though, that their algorithm's ability to locate the global minimiser comes from the periodic re-initialisation, and not from the transform that they apply to the objective function. This means that their algorithm is subject to the same limitation that the MPSO suffers: the curse of dimensionality. The examples they present are all restricted to two dimensions, which means that the re-initialisation method will have a good chance of finding the global minimiser within a small number of iterations. These results are misleading, since the algorithm will perform significantly worse on say 100-dimensional problems.

3.4.5 Stopping Criteria

While it's relatively simple to design a stopping criterion for a local search algorithm, it is rather hard to do so for a global search algorithm, unless the value of the objective function in the global minimiser is known in advance. To illustrate: when the GCPSO algorithm fails to improve the value of $f(\hat{\mathbf{y}})$ over many consecutive iterations, it is relatively safe to assume that it has found a local minimum. When the MPSO exhibits the same behaviour, that is, it fails to improve on the best solution discovered so far after, say, 100 restarts, no such conclusion can be drawn. As illustrated above, the probability of hitting the optimality region decreases exponentially as the number of dimensions increases. This means that the number of restarts required before giving up must grow accordingly, as will now be shown.

Solis and Wets provided some guidelines for choosing the correct number of iterations

required for a stochastic search algorithm to discover the global minimiser [128]. For example, they defined the number of iterations required to reach the optimality region, N_λ, with at least probability $1-\lambda$ as follows:

$$P[\hat{\mathbf{y}}_k \notin R_\epsilon] \leq \lambda, \quad \forall k > N_\lambda$$

Let k denote the number of restarts in the MPSO algorithm. Then, if a value m is known so that $0 < m \leq \mu_k[R_\epsilon]$ for all k, then

$$P[\hat{\mathbf{y}}_k \notin R_\epsilon] \leq (1-m)^k$$

Choosing an integer

$$N_\lambda \geq \left\lceil \frac{\ln \lambda}{\ln(1-m)} \right\rceil$$

yields the required property, since when $k \geq N_\lambda$, then $\ln\lambda/\ln(1-m) \leq k$, because $(1-m)^k \geq \lambda$. Note that this calculation requires that a lower bound m is known for $\mu_k[R_\epsilon]$, which implies that the size and shape of R_ϵ is known in advance.

If the example from Section 3.4.4 is continued, we can use the value $m = (d/l)^n$ as such a lower bound. This implies that the number of iterations required to reach R_ϵ with probability $1-\lambda$ is

$$N_\lambda \geq \frac{\ln \lambda}{\ln\left(1-\left(\frac{d}{l}\right)^n\right)}$$

However, $\ln\left(1-\left(\frac{d}{l}\right)^n\right) \to 0$ as $n \to +\infty$, which means that $N_\lambda \to +\infty$, confirming our earlier suspicions.

Some methods have been proposed to approximate the distribution of the function $f(\hat{\mathbf{y}}_k)$ in an attempt to model $\mu_k[R_\epsilon]$. This approach requires that a sufficient number of samples in the neighbourhood of R_ϵ are available, with ϵ sufficiently small [4]. Unfortunately, this precludes the design of a stopping criterion based on this approximation, since we don't know where R_ϵ is when trying to minimise f.

The only viable stopping criterion appears to be one that specifies some arbitrary, problem specific, fixed number of iterations. Although this does not guarantee that the global minimiser will be discovered, it is easy to implement and corresponds to the real-world requirement that the algorithm must terminate after a specified duration of time.

3.5 Conclusion

This chapter discussed various issues related to the convergence behaviour of the Particle Swarm Optimiser. Section 3.1 presented some arguments related to the convergence of the trajectory of a PSO system. A trajectory that converges can be seen as a type of termination criterion for the PSO, but it does not help to determine whether the PSO has converged onto a global or local minimiser. That being said, it is still desirable to choose the PSO parameters so that the trajectory is convergent, since this is an indication of the rate of convergence of the algorithm. The analysis presented in Section 3.1 yields a useful rule for choosing these crucial PSO parameters.

A modified PSO algorithm, called the Guaranteed Convergence Particle Swarm Optimiser (GCPSO), was introduced in Section 3.2. This algorithm incorporates some features of a pure random search algorithm into the PSO, giving it the ability to break free from so called stagnant states. The importance of this property was illustrated in Section 3.3, where a formal proof was presented, showing that the GCPSO is a local search algorithm. This result means that the GCPSO is guaranteed to be able to find a local minimiser. Some other useful definitions regarding local and global search algorithms were also discussed in Section 3.3.1.

Although it has often been assumed to be true in the literature, the PSO algorithm is not in fact a global search algorithm. Section 3.4 proved that the PSO does not satisfy the requirements of a global search algorithm. Two methods of extending the PSO algorithm so that it does satisfy these requirements were presented, along with the relevant proofs. It was also shown that although these algorithms now have the ability to discover the global minimiser of any objective function, the number of iterations required to do so becomes infinite as the number of dimensions approach infinity. This implies that the performance of these techniques degrades considerably as the number of dimensions in the objective function increases.

Chapter 4

Models for Cooperative PSOs

This chapter introduces two models for constructing a cooperative Particle Swarm Optimiser. The first model, called CPSO-S_K, is similar to Potter's CCGA cooperative Genetic Algorithm; the second further adds "blackboard" style cooperation, and is called CPSO-H_K. Several algorithms based on these models are then presented, with their associated convergence proofs, where possible.

4.1 Models for Cooperation

This section briefly reviews some of the cooperative models introduced in Section 2.8. The focus of this section is on population-based cooperative algorithms. Other methods, involving solitary agents, exist, but fall outside the scope of this thesis [140].

A popular model for cooperation is the *island model*, due to Grosso[59]. This model is used to partition a large population into several smaller subpopulations, so that each subpopulation can be evolved on a separate machine. Periodically some individuals "migrate" from one subpopulation to another, allowing the subpopulations to share information regarding the solutions discovered so far. This model can also be simulated on a single processor; the aim is then to preserve diversity by maintaining several subpopulations with little communication between them, effectively preventing premature convergence.

If the island model is modified to allow some degree of overlap between the subpop-

ulations, the resulting model is called the *neighbourhood model* [88, 80]. This model increases the rate of information exchange between the subpopulations, effectively increasing the rate of convergence of the algorithm. Note that the *lbest* PSO algorithm makes use of this model.

Both the island and the neighbourhood models are merely increasing the effective size of the population, with some restrictions on which individuals can communicate directly with one another. This means that any individual, taken from any of the subpopulations, contains enough information to be used as a potential solution to the problem being solved.

The Cooperative Coevolutionary Genetic Algorithm (CCGA), due to Potter [106, 105], takes a different approach to the decomposition problem. Instead of distributing complete individual solutions over a set of subpopulations, the CCGA algorithm distributes the subcomponents of individual solutions over a set of subpopulations. For example, a function optimisation problem with n parameters is partitioned into n subpopulations, each subpopulation corresponding to one of the parameters. Potential solutions to the optimisation problem are constructed by taking one representative from each of the n subpopulations to build an n-dimensional vector. The different subpopulations are thus cooperating to find the solution, since no single subpopulation has the necessary information to solve the problem by itself.

Clearwater *et al.* [19] investigated the behaviour of cooperating agents in the context of constraint-satisfaction problems. The model they proposed is that of a "blackboard"; a shared memory to which each individual can post hints to or read hints from. Note that the blackboard is simply a metaphor for inter-agent communication — in this case, global communication. Clearwater *et al.* observed a super-linear speedup when their agents were cooperating via the blackboard, *versus* a linear speedup when multiple non-interacting agents were used. They also found that a group of non-cooperating agents, each searching in a different non-overlapping region of search space, resulted only in a linear speedup, similar to the overlapping non-cooperating agents. In the discrete constraint-satisfaction problem they investigated, the inter-agent cooperation had the effect of dynamically pruning the search space, which effectively decreased the time needed for the first agent to find the correct solution.

The model proposed by Clearwater *et al.* was later used to show that GAs can be seen as cooperative algorithms [22]. Note that the island model GA can easily be interpreted as a group of cooperating agents. Each subpopulation represents an agent, and the periodic migration of individuals between subpopulations acts as a "shared memory" for the whole population.

At first it appears that Potter's CCGA algorithm cannot be described by the model of Clearwater *et al.* The CCGA algorithm partitions the search space into disjoint subspaces, and each subpopulation (agent) is constrained to one such subspace. The local solution found in one subspace is of little direct use to an individual in another subpopulation, so there is no direct communication between the subpopulations. One interpretation is that the CCGA algorithm bypasses the inter-agent communication process by directly constraining the search spaces of all the other subpopulations. Consider again the function optimisation problem in n dimensions. A vector $\mathbf{x}$ is constructed by taking a representative from each of the n subpopulations. The algorithm now attempts to find a better solution for component j of this vector by using subpopulation j as an optimiser in a search space constrained by the other $n-1$ components of the vector $\mathbf{x}$. Subpopulation j will observe different fitness landscapes when different $\mathbf{x}$ vectors are used. The net result is that the subpopulations do exert an influence on one another by constraining the search space, although they do not share information directly.

Another CCGA-style algorithm, based on *stochastic generational search methods*, has been introduced by Subbu and Sanderson [131]. The stochastic generational search method constructs a distribution using randomly sampled points from the search space. This distribution is then used to focus the sample space from which new random samples are drawn. This method has provable stochastic global convergence behaviour. Subbu and Sanderson partition the search space into disjoint components, just like the CCGA algorithm. They modeled their algorithm in a network environment, where each subpopulation represents a node in the network. Each node has access to local information, and indirectly (at increased delay) to the information in other nodes. An experiment was constructed to compare a centralised algorithm to a distributed CCGA-style algorithm. In the centralised algorithm, one designated node requests information from all the other nodes in the network. This node then performs traditional, *i.e.* non-cooperative, optimi-

sation. In their distributed algorithm, each node periodically collects information from the other nodes, followed by CCGA-style cooperative optimisation of the subspace represented by that node. Their results show that the cooperative algorithm locates the minimum of the test functions in their experiment significantly faster than the centralised algorithm.

The rest of this chapter deals with the adaption of some of the above models for cooperation for use in a Particle Swarm Optimisation algorithm.

4.2 Cooperative Particle Swarm Optimisers

This section introduces a CCGA-style cooperative Particle Swarm Optimiser. The original CCGA model is generalised to allow more flexible partitioning of the search space.

4.2.1 Two Steps Forward, One Step Back

Before looking at cooperative swarms in depth, let us first consider one of the weaknesses of the standard PSO optimiser.

In the standard PSO algorithm, each particle represents a complete vector that can be used as a potential solution. Each update step is also performed on a full n-dimensional vector. This allows for the possibility that some components in the vector moved closer to the solution, while others actually moved *away* from the solution. As long as the effect of the improvement outweighs the effect of the components that deteriorated, the standard PSO will consider the new vector an overall improvement, even though some components of the vector may have moved further from the solution.

A simple example to illustrate this concept follows: Consider a 3-dimensional vector $\mathbf{x}$, and the error function $f(\mathbf{x}) = \|\mathbf{x} - \mathbf{a}\|^2$, where $\mathbf{a} = (20, 20, 20)$. This implies that the global minimiser of the function $\mathbf{x}^*$, is equal to $\mathbf{a}$.

Now consider a particle swarm containing, amongst others, a vector $\mathbf{x}_2$, and the global best position, $\hat{\mathbf{y}}$. If t represents the current time step, then, with a high probability,

$$\|\mathbf{x}_2(t+1) - \hat{\mathbf{y}}(t+1)\| < \|\mathbf{x}_2(t) - \hat{\mathbf{y}}(t)\|$$

if it is assumed that $\hat{\mathbf{y}}$ does not change during this specific iteration. That is, in the next

time step, $t+1$, particle 2 (represented by $\mathbf{x}_2$) will be drawn closer to $\hat{\mathbf{y}}$, as stipulated by the PSO update equations.

Assume that the following holds:

$$\begin{aligned} \hat{\mathbf{y}}(t) &= (17, 2, 17) \\ \mathbf{x}_2(t) &= (5, 20, 5) \end{aligned}$$

Application of the function f to these points shows that $f(\hat{\mathbf{y}}(t)) = 342$ and $f(\mathbf{x}_2(t)) = 450$. In the next epoch, the vector $\mathbf{x}_2$ will be drawn closer to $\hat{\mathbf{y}}$, so that the following configuration may result:

$$\begin{aligned} \hat{\mathbf{y}}(t+1) &= (17, 2, 17) \\ \mathbf{x}_2(t+1) &= (15, 5, 15) \end{aligned}$$

Note that the actual values of the components of $\mathbf{x}_2(t+1)$ depend on the stochastic influence present in the PSO update equations. The configuration above is certainly one possibility. This implies that $f(\mathbf{x}_2(t+1)) = 275$, even better than the function value of the global best position. Although the fitness of the particle improved considerably, note that the second component of the vector has changed from the correct value of 20, to the rather poor value of 5; valuable information has thus been lost unknowingly. This example can clearly be extended to a general case involving an arbitrary number of components.

This undesirable behaviour is a case of taking two steps forward, and one step back. It is caused by the fact that the error function is computed only after all the components in the vector have been updated to their new values. This means an improvement in two components (two steps forward) will overrule a potentially good value for a single component (one step back). Note that this is reminiscent of a rule of thumb when repairing computer hardware: never change two components at once, because then you won't know which component was responsible for solving/causing the problem.

One way to overcome this behaviour is to evaluate the error function more frequently, for example, once for every time a component in the vector has been updated, resulting in much quicker feedback. This increase in the number of times that the objective function must be evaluated can clearly have a negative impact on the running time

of the algorithm. The potential improved rate of convergence of the algorithm must be greater than the associated increase in processor usage if the proposed scheme is to be an overall improvement. Section 4.2.2 presents an algorithm that implements a variation of this scheme. Before this algorithm will be presented, though, a brief discussion of *deceptive functions* is in order.

Deceptive Functions

Most optimisation algorithms have specific biases, that is, certain assumptions that they make about the functions they are expected to minimise. This is why certain algorithms appear to defy the No Free Lunch theorem: they are biased towards the suite of test functions on which they outperform all other existing optimisation methods. To obtain the best possible performance in a specific problem (*e.g.* Neural Network training), an algorithm must be selected so that its biases coincide with the properties of the function being optimised.

On the other hand, some optimisation methods have specific biases that can be exploited during comparisons to other methods. One example of this phenomenon is that of *deceptive functions*. These are functions that exhibit *deceptive* behaviour [54], where good solutions, or even good directions of search, must be abandoned since they lead to suboptimal solutions.

In the GA context, this means a function that exploits, for example, the weaknesses of the crossover operator. The unitation of a bit-string is the number of "one"s in the string, expressed as a ratio with respect to the length of the string. An example deceptive function could have the global maximum of the function at minimum unitation; the sub-optimal solution is found at maximum unitation. For example, the function

$$f(\mathbf{b}) = \begin{cases} 0.5 \times \text{unitation}(\mathbf{b}) - 0.1 & \text{if unitation}(\mathbf{b}) > 0.2 \\ -5 \times \text{unitation}(\mathbf{b}) + 1.0 & \text{if unitation}(\mathbf{b}) \leq 0.2 \end{cases}$$

where $\mathbf{b}$ is a bit-string, can be considered deceptive. This function is shown in Figure 4.1. If two bit-strings $\mathbf{b}_1$ and $\mathbf{b}_2$ have unitation 0.6 and 0.8 respectively, then their offspring will have a unitation value of at least 0.4; the expected value is 0.7. The crossover operator may even produce offspring of unitation 1.0. This implies that the offspring

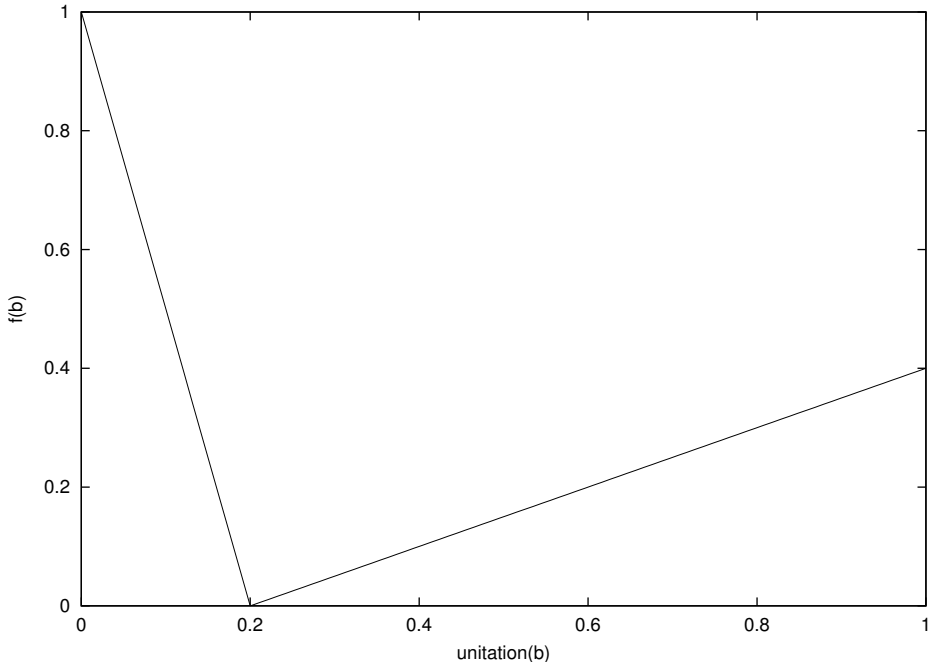


Figure 4.1: A deceptive function

will tend to have larger unitation, eventually (mis)leading the population to the sub-optimal solution.

Grefenstette contends that the concern regarding deception is of little worth, as a simple modification to the GA can make deceptive functions easy to solve [57]. For example, if the GA converged on the solution $\mathbf{b}$, then simply evaluate $f(\mathbf{b})$ and $f(\overline{\mathbf{b}})$, where $\overline{\mathbf{b}}$ is the bitwise complement of $\mathbf{b}$, and choose the better solution. This approach allows the GA to solve all deceptive functions similar to the one discussed above. Grefenstette further describes a more complicated solution that is guaranteed to solve any function that is deceptive in its unitation space.

Note that deception only affects local search algorithms, since true global search algorithms cannot be mislead. One of the more elegant solutions to the problem of multi-modal (including deceptive) functions is the one presented by Beasley *et al.* [10], where all extrema (local and global) are identified and then suppressed to allow the identification of other extrema. Clearly this approach can avoid any number of deceptive features in the objective function.

Although most of the research regarding deception focused on Genetic Algorithms (and their susceptibility to deception), it is certainly possible to construct deceptive functions to fool most non-global algorithms. A normal PSO algorithm can also be mislead, since it is a local minimisation algorithm, as shown in Section 3.4.1. The cooperative PSO algorithm introduced in the next section is more susceptible to deception than the original PSO, since it makes more assumptions regarding the nature of the objective function. Even though it is possible to show that many algorithms can easily be tricked by a deceptive function, it is still not easy to determine whether such deceptive functions are commonly found in real-world applications.

4.2.2 CPSO-S$_K$ Algorithm

The original PSO uses a population of n-dimensional vectors. These vectors can be partitioned into n swarms of one-dimensional vectors, each swarm representing a dimension of the original problem. Each swarm attempts to optimise a single component of the solution vector, essentially a one-dimensional optimisation problem. This decomposition is analogous to the decomposition used in the relaxation method [129, 48].

One complication to this configuration is the fact that the function to be minimised, f, requires an n-dimensional vector as input. If each swarm represents only a single dimension of the search space, it is clearly not possible to directly compute the fitness of the individuals of a single population considered in isolation. A *context vector* is required to provide a suitable context in which the individuals of a population can be evaluated. The simplest scheme for constructing such a context vector is to take the global best particle from each of the n swarms and concatenating them to form such an n-dimensional vector. To calculate the fitness for all particles in swarm j, the other $n-1$ components in the context vector are kept constant (with their values set to the global best particles from the other $n-1$ swarms) while the j^{th} component of the context vector is replaced in turn by each particle from the j^{th} swarm.

Figure 4.2 presents the Cooperative PSO-Split (CPSO-S) algorithm, first introduced by Van den Bergh and Engelbrecht in [136], a PSO that splits the search space into exactly n subspaces. Extending the convention introduced in Figure 2.5 (Page 24), $P_j.\mathbf{x}_i$ now refers to the position of particle i of swarm j, which can therefore be substituted

define $\mathbf{b}(j,z) \equiv (P_1.\hat{\mathbf{y}}, P_2.\hat{\mathbf{y}}, \ldots, P_{j-1}.\hat{\mathbf{y}}, z, P_{j+1}.\hat{\mathbf{y}}, \ldots, P_n.\hat{\mathbf{y}})$
Create and initialise n one-dimensional PSOs : $P_j,\ j \in [1..n]$
repeat:
  for each swarm $j \in [1..n]$ **:**
    for each particle $i \in [1..s]$ **:**
      if $f(\mathbf{b}(j, P_j.\mathbf{x}_i)) < f(\mathbf{b}(j, P_j.\mathbf{y}_i))$
        then $P_j.\mathbf{y}_i = P_j.\mathbf{x}_i$
      if $f(\mathbf{b}(j, P_j.\mathbf{y}_i)) < f(\mathbf{b}(j, P_j.\hat{\mathbf{y}}))$
        then $P_j.\hat{\mathbf{y}} = P_j.\mathbf{y}_i$
    endfor
    Perform PSO updates on P_j using equations (3.1–3.2, 3.26)
  endfor
until stopping condition is true

Figure 4.2: Pseudo code for the CPSO-S algorithm

into the j^{th} component of the context vector when needed. Each of the n swarms now has a global best particle $P_j.\hat{\mathbf{y}}$. The function $\mathbf{b}(j,z)$ returns an n-dimensional vector formed by concatenating all the global best vectors across all swarms, except for the j^{th} component, which is replaced with z, where z represents the position of any particle from swarm P_j.

This algorithm has the advantage that the error function f is evaluated after each component in the vector is updated, resulting in much finer-grained credit assignment. The current "best" context vector will be denoted $\mathbf{b}(1, P_1.\hat{\mathbf{y}})$. Note that $f(\mathbf{b}(1, P_1.\hat{\mathbf{y}}))$ is a strictly non-increasing function, since it is composed of the global best particles $P_j.\hat{\mathbf{y}}$ of each of the swarms, which themselves are only updated when their fitness improves.

Each swarm in the group only has information regarding a specific component of the solution vector; the rest of the vector is provided by the other $n-1$ swarms. This promotes cooperation between the different swarms, since they all contribute to $\mathbf{b}$, the context vector. Another interpretation of the cooperative mechanism is possible: Each particle i of swarm j represents a different context in which the vector $\mathbf{b}(j,\cdot)$ is evaluated, so that the fitness of the context vector itself is measured in different contexts. The

define $\mathbf{b}(j, \mathbf{z}) \equiv (P_1.\hat{\mathbf{y}}, \ldots, P_{j-1}.\hat{\mathbf{y}}, \mathbf{z}, P_{j+1}.\hat{\mathbf{y}}, \ldots, P_K.\hat{\mathbf{y}})$
$K_1 = n \bmod K$
$K_2 = K - (n \bmod K)$
Initialise K_1 $\lceil n/K \rceil$-dimensional PSOs : $P_j,\ j \in [1..K_1]$
Initialise K_2 $\lfloor n/K \rfloor$-dimensional PSOs : $P_j,\ j \in [(K_1+1)..K]$
repeat:
  for each swarm $j \in [1..K]$ **:**
    for each particle $i \in [1..s]$ **:**
      if $f(\mathbf{b}(j, P_j.\mathbf{x}_i)) < f(\mathbf{b}(j, P_j.\mathbf{y}_i))$
        then $P_j.\mathbf{y}_i = P_j.\mathbf{x}_i$
      if $f(\mathbf{b}(j, P_j.\mathbf{y}_i)) < f(\mathbf{b}(j, P_j.\hat{\mathbf{y}}))$
        then $P_j.\hat{\mathbf{y}} = P_j.\mathbf{y}_i$
    endfor
    Perform PSO updates on P_j using equations (3.1–3.2, 3.26)
  endfor
until stopping condition is true

Figure 4.3: Pseudo code for the generic CPSO-S$_K$ Swarm Algorithm

most successful context, corresponding to the particle i yielding the highest fitness, is retained for future use. For example, a 30-dimensional search space results in a CPSO-S algorithm with 30 one-dimensional swarms. During one iteration of the algorithm, $30 \times 30 = 900$ different combinations are formed, compared to only 30 variations produced by the original PSO. The advantage of the CPSO-S approach is that only one component is modified at a time, yielding the desired fine-grained search, effectively preventing the "two steps forward, one step back" scenario. There is also a significant increase in the amount of diversity in the CPSO-S algorithm, because of the many combinations that are formed using different members from different swarms.

Note that, should some of the components in the vector be correlated, they should be grouped in the same swarm, since the independent changes made by the different swarms will have a detrimental effect on correlated variables. This results in some swarms having one-dimensional vectors and others having c-dimensional vectors ($c < n$), something

which is easily allowed in the framework presented above. Unfortunately, it is not always known in advance how the components will be related. A simple approximation would be to blindly take the variables c at a time, hoping that some correlated variables will end up in the same swarm. Figure 4.3 presents the CPSO-S$_K$ algorithm, where a vector is split into K parts. Note that the CPSO-S algorithm presented in Figure 4.2 is really a special case of the CPSO-S$_K$ algorithm with $K = n$. The number of parts, K, is also called the *split factor*.

There is no explicit restriction on the type of PSO algorithm that should be used in the CPSO-S$_K$ algorithm. Because it has been shown that the GCPSO algorithm has guaranteed convergence on local minimisers, it will be assumed that the CPSO-S$_K$ algorithm consists of K cooperating GCPSO swarms, unless otherwise noted.

4.2.3 Convergence Behaviour of the CPSO-S$_K$ Algorithm

Before discussing the local convergence behaviour for the CPSO-S$_K$ algorithm, some examples will be presented to illustrate the concepts that will be used below.

Consider a function f, defined as

$$f(\mathbf{x}) = 5\tanh(x_1 + x_2) + 0.05(x_1 + 2)^2 \tag{4.1}$$

This function has the property that for a constant x_1, chosen from the range $x_1 \in [5, 10]$, the value of $f(\mathbf{x})$ is approximately equal to 5 for any $x_2 \in [-2, 10]$, since the function is reduced to $5\tanh(5 + x_2) \approx 5$, for $x_2 \geq -2$. This property is visible in Figure 4.4.

Assume that a CPSO-S$_2$ swarm (*i.e.* two cooperating swarms) will be used to minimise f, constrained to the region $x_1 \in [-10, 10]$, and $x_2 \in [-2, 10]$. Consider now that the global best particles of the two cooperating swarms, P_1 and P_2, are at positions $P_1.\hat{y} = 7.5$ and $P_2.\hat{y} = 0$. Note that this implies that $\mathbf{b}(1, P_1.\hat{y}) = (7.5, 0)$; this point belongs to the region yielding approximately constant f-values, as described in the previous paragraph. The value of $P_2.\hat{y}$ has no influence on the value of $f(\mathbf{b})$, since all points in this region have approximately the same f-value if the x_1 variable remains constant. This implies that it is not possible for the P_2 swarm to improve its position, in fact, all possible positions in the range $x_2 \in [-2, 10]$ are equally optimal.

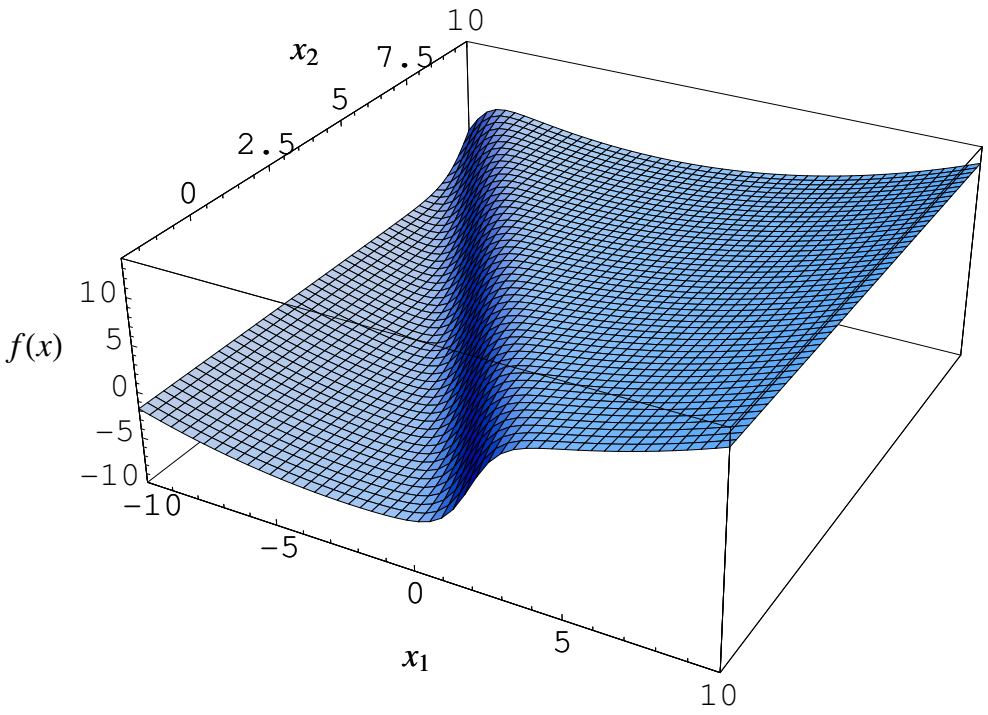


Figure 4.4: A plot of the function $f(\mathbf{x}) = 5\tanh(x_1 + x_2) + 0.05(x_1 + 2)^2$

The same is not true for the x_1 axis, however. Clearly a value of $x_1 < 0$ will allow the algorithm to "drop down" into the lower region containing the minimum. This means that the swarm P_1 has the opportunity to improve the position of its global best particle, $P_1.\hat{y}$. After "dropping down" into the region $x_1 < 0$, it may become possible for the swarm P_2 to find a better solution in the range $x_2 \in [-2, 0]$, ultimately leading to the discovery of the minimiser of the function, in this case, the position $(-2.2356, -2)$.

What this example illustrates is that it is quite possible that the context vector $\mathbf{b}(\mathbf{j}, \cdot)$ may constrain swarm j to a subspace where it is not possible for that swarm to make any significant improvement. This effect could be temporary, so that a change in the context vector (due to another swarm) may allow another swarm to discover further improvements. It is possible, however, for the algorithm to become trapped in a state where all the swarms are unable to discover better solutions, but the algorithm has not yet reached the local minimum. This is another example of *stagnation*, caused by the restriction that only one swarm is updated at a time, *i.e.* only one subspace is searched

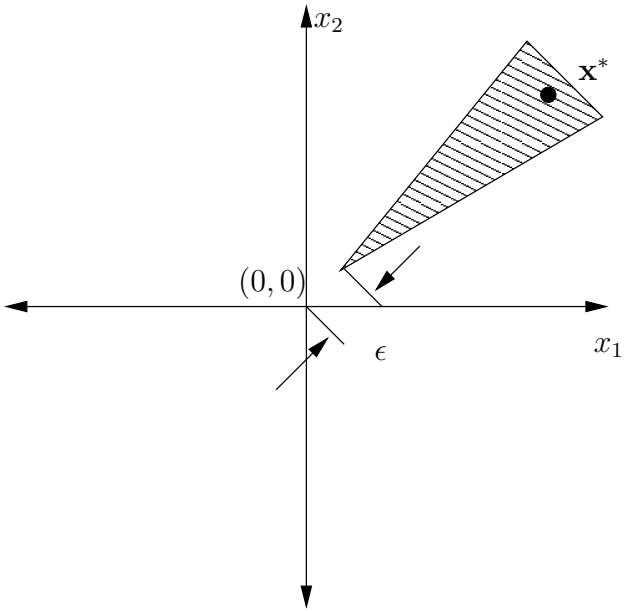


Figure 4.5: A diagram illustrating the constrained sub-optimality problem

at a time.

An example function will now be presented to show a scenario in which the CPSO-S_K algorithm stagnates. The example will again assume that a CPSO-S_2 algorithm is used to minimise the function. Figure 4.5 illustrates in two dimensions the nature of the problem. The figure is a top-down view of the search space, with the shaded triangular area representing a region that contains f-values that are smaller than any other values in the search space. This region has a slope that runs downward from the point $(0, 0)$ to the point $\mathbf{x}^*$, the global minimiser. The symbol ϵ denotes the distance from the origin to the tip of the triangular region; ϵ can be made arbitrarily small so that the triangle touches the origin in the limit. To simplify the discussion, assume that the function has the form $f(\mathbf{x}) = \|\mathbf{x}\|^2$, except for the shaded triangular region, which contains points yielding negative f-values.

If the swarm P_1 (constrained to the subspace $x_1 \in \mathbb{R}$) reaches the state where $P_1.\hat{y} = 0$, the context vector $\mathbf{b}$ will be of the form $\mathbf{b}(2, P_2.x_i) = (0, P_2.x_i)$, so that $f(\mathbf{b}) = \|(0, P_2.x_i)\|^2 = (P_2.x_i)^2$. This function can easily be minimised by the second swarm P_2, which is constrained to the subspace $x_2 \in \mathbb{R}$. The second swarm will find the minimum

located at $x_2 = 0$, so that the algorithm will terminate with a proposed solution of $(0, 0)$, which is clearly not the correct answer, since $\mathbf{x}^* \neq (0, 0)$. Both P_1 and P_2 have guaranteed convergence onto the local minimum of their respective subspaces, since they are using the GCPSO algorithm internally. When the algorithm reaches the step where $P_1.\hat{y} = 0$ and $P_2.\hat{y} = 0$, the GCPSO algorithm will examine a non-degenerate sample space around each of these global best particles. The problem is that the algorithm will find that 0 is in fact the local minimiser when only one dimension is considered at a time. The sequential nature of the algorithm, coupled with the property that $f(\mathbf{b}(1, P_1.\hat{y})$ is a strictly non-increasing sequence, prevents the algorithm from temporarily taking an "uphill" step, which is required to solve this particular problem. Even if ϵ is made arbitrarily small, the algorithm will not be able to sample a point inside the shaded area, since that would require the other swarm to have a global best position (*i.e.* $P_j.\hat{y}$) other than zero, which would require a step that would increase $f(\mathbf{b}(1, P_1.\hat{y}))$. What has happened here is that a local optimisation problem in $\mathbb{R}^2$ has become a global optimisation problem when considering the two subspaces x_1 and x_2 one at a time.

Note that the point $(0, 0)$ is not a local minimiser of the search space, although it is the concatenation of the individual minimisers of the subspaces x_1 and x_2. The fact that $(0, 0)$ is not a local minimiser can easily be verified by examining a small region around the point $(0, 0)$, which clearly contains points belonging to the shaded region as ϵ approaches zero. The term *pseudo-minimiser* will be used to describe a point in search space that is a local minimiser in all the pre-defined subspaces of $\mathbb{R}^n$, but not a local minimiser in $\mathbb{R}^n$ considered as a whole. This shows that the CPSO-S$_K$ algorithm is not guaranteed to converge on the local minimiser, because there exists states from which it can become trapped in the pseudo-minimiser located at $(0, 0)$. Due to the stochastic components in the PSO algorithm, it is unlikely that the CPSO-S$_K$ algorithm will become trapped in the pseudo-minimiser every time. The existence of a state that prevents the algorithm from reaching the minimiser destroys the guaranteed convergence property, though.

In contrast, the normal GCPSO would not have the same problem. If the global best particle of the GCPSO algorithm is located at this pseudo-minimum position, *i.e.* $\hat{\mathbf{y}} = (0, 0)$, then the sample space from which the global best particle could choose its

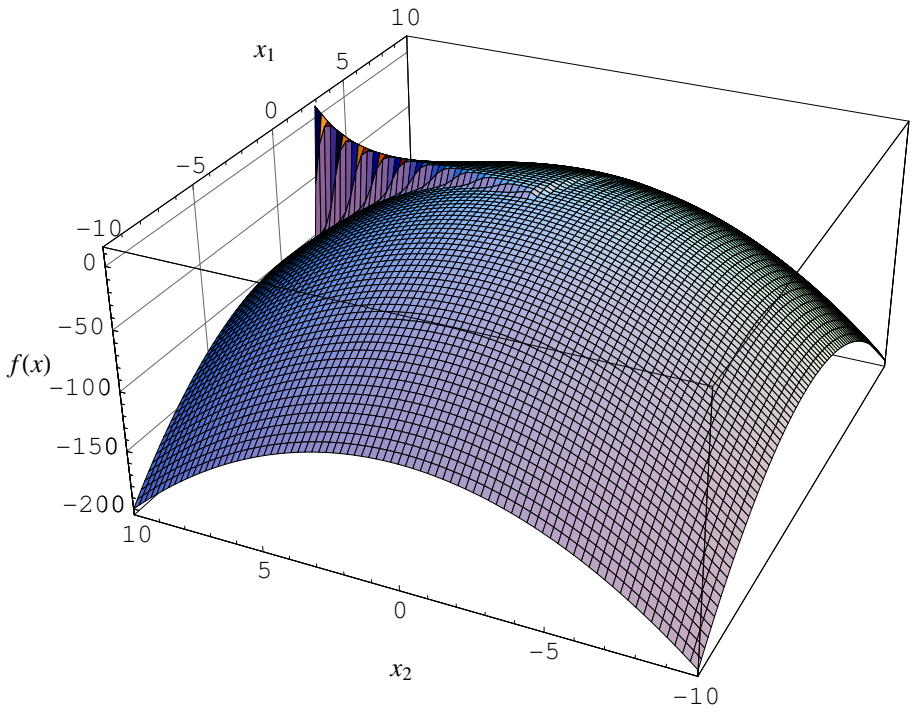


Figure 4.6: An inverted plot of the function $f(\mathbf{x})$, as defined in equation (4.2), rotated through a 75-degree angle. Note that the raised ridge is not parallel to either of the coordinate axes, causing the CPSO-S$_K$ algorithm to become trapped in a pseudo-minimum.

next position would be a square with side lengths ρ, centered at $(0, 0)$. Since $\rho > 0$ per definition, this square would always include points from the triangular shaded region in Figure 4.5. This implies that the GCPSO will be able to move away from the point $(0, 0)$ toward the local minimiser in $\mathbb{R}^2$ located at $\mathbf{x}^*$.

Equation (4.2) defines an example function that can trap the CPSO-S$_K$ algorithm in a suboptimal position.

$$\begin{aligned} f(\mathbf{h}(\mathbf{x},\theta)) &= (x_1^2 + x_2^2) - \\ &\quad (\tanh(10x_1) + 1)(\tanh(-10x_2) + 1) \exp\left(\frac{(x_1 + x_2)}{3}\right) \end{aligned} \tag{4.2}$$

where $\mathbf{h}$ represents the rotation operator

$$\mathbf{h}(\mathbf{x},\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

The search space is artificially restricted to the range $S = [-10, 10]^2$ so that there is a unique minimum at the edge of the search space. This function is graphically represented in Figure 4.6, with a rotation angle $\theta = -75$ degrees. Note that this function exhibits deceptive behaviour when the CPSO-S$_K$ algorithm is applied to it, since it misleads the algorithm to converge on a pseudo-minimum. When the GCPSO algorithm is used to minimise this function, it is not mislead by the deceptive nature of the function. This illustrates that deception usually targets one of the assumptions that the algorithms make with regard to the functions they attempt to optimise. The CPSO-S$_K$ algorithm assumes that it is able to find the minimum by searching in disjoint subspaces, which turns out to be a flawed strategy for some functions. A typical example of such a function is the one in Figure 4.6, where the raised ridge is not parallel to the coordinate axes, causing the CPSO-S$_K$ algorithm to become trapped in a pseudo-minimum.

There are several ways to augment the CPSO-S$_K$ algorithm to prevent it from becoming trapped in such pseudo-minima. The original CCGA-1 algorithm, due to Potter [106, 105], suffers from the same problem, although Potter did not identify the problem as such. Potter suggested that each element of the population P_j should be evaluated in two contexts. He called this approach the CCGA-2 algorithm. One context is constructed using the best element from the other populations, similar to the CCGA-1 and CPSO-S$_K$ algorithms. The second context is constructed using a randomly chosen element from each of the other populations. The individual under consideration receives the better of the two fitness values obtained in the two contexts. This approach is a compromise between the CCGA-1 approach and an exhaustive evaluation, where each element is evaluated against all other possible contexts that can be constructed from the current collection of populations. The exhaustive approach would require s^{K-1} function evaluations to determine the fitness of a single individual, where s is the population size, and K the number of populations. This rather large increase in the number of function evaluations would outweigh the advantage of using a cooperative approach.

The CCGA-2 approach has the disadvantage that the fitness of an individual is still only evaluated against a sample of possible values obtained from a search restricted to a subspace of the complete search space. In other words, it could still become trapped in a pseudo-minimiser, although this event is significantly less likely than for the CCGA-1

algorithm. The next section introduces a different solution that allows the CPSO-S_K algorithm to escape from pseudo-minima.

4.3 Hybrid Cooperative Particle Swarm Optimisers

In the previous section it was shown that the CPSO-S_K algorithm can become trapped in sub-optimal locations in search space. This section introduces an algorithm that combines the CPSO-S_K algorithm with the PSO in an attempt to retain the best properties of both algorithms. The term "hybrid" has been used to describe at least three different PSO-based algorithms [2, 136, 79]. In [136] the algorithm presented in Figure 4.7 was called the Hybrid PSO. This algorithm will now be called the CPSO-H_K algorithm in this thesis to resolve the ambiguity.

4.3.1 The CPSO-H_K Algorithm

Given that the GCPSO has certain desirable properties, *e.g.* the ability to guarantee convergence on a local minimiser, and that the CPSO-S_K algorithm has faster convergence on certain functions (see Section 5.5), it would be ideal to have an algorithm that could exploit both of these properties. In principle one could construct an algorithm that attempts to use a CPSO-S_K algorithm, but switches over to a GCPSO algorithm when it appears that the CPSO-S_K algorithm has become trapped. While this approach is a sound idea, it is difficult to design robust, general heuristics to decide when to switch between algorithms.

An alternative is to interleave the two algorithms, so that the CPSO-S_K algorithm is executed for one iteration, followed by one iteration of the GCPSO algorithm. Even more powerful algorithms can be constructed by exchanging information regarding the best solutions discovered so far by either component at the end of each iteration. This information exchange is then a form of cooperation between the CPSO-S_K component and the GCPSO component. Note that this is a form of "blackboard" cooperation, similar to the type described by Clearwater *et al.* [19].

A simple mechanism for implementing this information exchange is to replace some of the particles in one half of the algorithm with the best solution discovered so far by

define $\mathbf{b}(j, \mathbf{z}) \equiv (P_1.\hat{\mathbf{y}}, \ldots, P_{j-1}.\hat{\mathbf{y}}, \mathbf{z}, P_{j+1}.\hat{\mathbf{y}}, \ldots, P_K.\hat{\mathbf{y}})$
$K_1 = n \bmod K$
$K_2 = K - (n \bmod K)$
Initialise K_1 $\lceil n/K \rceil$-dimensional PSOs : $P_j,\ j \in [1..K_1]$
Initialise K_2 $\lfloor n/K \rfloor$-dimensional PSOs : $P_j,\ j \in [(K_1+1)..K]$
Initialise an n-dimensional PSO : Q
repeat:
  for each swarm $j \in [1..K]$ **:**
    for each particle $i \in [1..s]$ **:**
      if $f(\mathbf{b}(j, P_j.\mathbf{x}_i)) < f(\mathbf{b}(j, P_j.\mathbf{y}_i))$
        then $P_j.\mathbf{y}_i = P_j.\mathbf{x}_i$
      if $f(\mathbf{b}(j, P_j.\mathbf{y}_i)) < f(\mathbf{b}(j, P_j.\hat{\mathbf{y}}))$
        then $P_j.\hat{\mathbf{y}} = P_j.\mathbf{y}_i$
    endfor
    Perform PSO updates on P_j using equations (3.1–3.2, 3.26)
  endfor
  Select random $k \sim U(1, s/2) \mid Q.\mathbf{y}_k \neq Q.\hat{\mathbf{y}}$
  $Q.\mathbf{x}_k = \mathbf{b}(1, P_1.\hat{\mathbf{y}})$
  for each particle $j \in [1..s]$ **:**
    if $f(Q.\mathbf{x}_j) < f(Q.\mathbf{y}_j)$
      then $Q.\mathbf{y}_j = Q.\mathbf{x}_j$
    if $f(Q.\mathbf{y}_j) < f(Q.\hat{\mathbf{y}})$
      then $Q.\hat{\mathbf{y}} = Q.\mathbf{y}_j$
  endfor
  Perform PSO updates on Q using equations (3.1–3.2, 3.26)
  for swarm $j \in [1..K]$ **:**
    Select random $k \sim U(1, s/2) \mid P_j.\mathbf{y}_k \neq P_j.\hat{\mathbf{y}}$
    $P_j.\mathbf{x}_k = Q.\hat{\mathbf{y}}_j$
  endfor
until stopping condition is true

Figure 4.7: Pseudo code for the generic CPSO-H$_K$ algorithm

the other half of the algorithm. Specifically, after one iteration of the CPSO-S$_K$ half (the P_j swarms in Figure 4.7) of the algorithm, the context vector $\mathbf{b}(1, P_1.\hat{\mathbf{y}})$ is used to overwrite a randomly chosen particle in the GCPSO half (the Q swarm in Figure 4.7) of the algorithm. This is followed by one iteration of the Q swarm component of the algorithm, which yields a new global best particle, $Q.\hat{\mathbf{y}}$. This vector is then split into sub-vectors of the right dimensions and used to overwrite the positions of randomly chosen particles in the P_j swarms.

Although the particles that are overwritten during the information exchange process are randomly chosen, the algorithm does not overwrite the position of the global best position of any of the swarms, since this could potentially have a detrimental effect on the performance of the affected swarm. Empirical studies also indicated that too much information exchange using this mechanism can actually impede the progress of the algorithm. By selecting a particle (targeted for replacement) using a uniform random distribution it is highly likely that a swarm of s particles will have had all its particles overwritten in, say $2s$, information exchange events, except for the global best particle, which is explicitly protected. If the P_j swarms are lagging behind the Q swarm in terms of performance, this means that the P_j swarms could overwrite all the particles in the Q swarm with inferior solutions in only a few iterations. On the other hand, the Q swarm would overwrite particles in the P_j swarms at the same rate, so the overall best solution in the algorithm will always be preserved. The diversity of the particles will decrease significantly because of too-frequent information exchange, though. A simple mechanism to prevent the swarms from accidentally reducing the diversity is implemented by limiting the number of particles that can actively participate in the information exchange. For example, if only half of the particles are possible targets for being overwritten, then at most half of the diversity of the swarm can be jeopardised. This does not significantly affect the positive influence of the information exchange process. For example, if the Q swarm overwrites an inferior particle $P_j.\mathbf{x}_i$ with a superior value (from Q), then that particle i will become the global best particle of swarm j. During subsequent iterations more particles will be drawn to this new global best particle, possibly discovering better solutions along the way, *i.e.* the normal operation of the swarm is not disturbed.

4.3.2 Convergence Proof for the CPSO-H$_K$ Algorithm

The following trivial lemma follows directly from the definition of the CPSO-H$_K$ algorithm.

Lemma 4 *The CPSO-H$_K$ algorithm produces a sequence of points $\{\hat{\mathbf{y}}\}_{k=1}^{+\infty}$ that converges on the local minimiser of the search space S, with asymptotic probability one.*

Proof: The information exchange process between the P_j swarms and the Q does not affect the value of $f(Q.\hat{\mathbf{y}})$ negatively, since the exchange cannot overwrite the value of $Q.\hat{\mathbf{y}}$. This implies that the swarm Q still satisfies (H1) and (H3), so that by Theorem 2 the swarm Q is guaranteed to converge on a local minimiser in the search space S. Thus, regardless of whether the P_j swarms converge or not, the Q swarm is still guaranteed to converge, so that overall the algorithm is also guaranteed to converge. ■

Since the Q swarm is guaranteed to converge, *i.e.* it is able to avoid pseudo-minima like those encountered in Section 4.2.3, the algorithm as a whole can clearly avoid such pitfalls. Because the Q swarm shares its best solution with the P_j swarms, they will have knowledge of locations in search space that they could not reliably reach on their own. This means that the CPSO-S$_K$ component of the swarm is now less likely to become trapped in pseudo-minima.

Further, the CPSO-S$_K$ algorithm is by design better able to maintain diversity, resulting in a more thorough search than what the GCPSO algorithm is capable of on its own. This information is shared with the Q swarm, although only in a restricted fashion, since only the best context vector is ever visible to the Q swarm.

4.4 Conclusion

In Section 4.1 a brief overview of some models for cooperation that have been used previously was presented. These cooperation techniques have been applied to Evolutionary Algorithms (*e.g.* GAs), as well as more deterministic methods for solving constraint-satisfaction problems. The property that all cooperative algorithms have in common is that they use the information gathered by the other agents to restrict their own search

to a subspace of the whole search space, thereby reducing the complexity of the problem as observed by each agent.

Section 4.2 presented a cooperative PSO algorithm, called the CPSO-S$_K$, that is based on the CCGA model. This section also discussed the problems that deceptive functions present to search algorithms, culminating in an example that shows that the CPSO-S$_K$ algorithm does not have the desired guaranteed convergence property that the GCPSO algorithm has. The CPSO-S$_K$ makes stronger assumptions about the nature of the functions it is expected to minimise, thus it is more susceptible to deception.

An extension of the CPSO-S$_K$ algorithm, called the CPSO-H$_K$ algorithm, was introduced in Section 4.3. It was shown that this algorithm has the ability to escape from the pseudo-minima that can trap the CPSO-S$_K$ algorithm, so that the CPSO-H$_K$ algorithm recovers the desired guaranteed convergence property of the GCPSO algorithm, while retaining the improved diversity offered by the CPSO-S$_K$ algorithm. The CPSO-H$_K$ algorithm is also more interesting to study, since it now adds an additional level of cooperation to the existing CPSO-S$_K$ structure.

The various cooperative PSO algorithms introduced in this section are studied empirically in Section 5.5, where they are compared to the various other PSO-based algorithms encountered in this thesis.

Chapter 5

Empirical Analysis of PSO Characteristics

This chapter investigates several key properties of the original PSO algorithm, as well as the properties of many PSO-based algorithms. The effects of different parameter settings are investigated to determine their effect on the quality of solutions and the rate of convergence. A comparison between the original PSO algorithm and the GCPSO is presented, followed by an investigation of the characteristics of various global PSO algorithms. An in-depth investigation of the properties of various cooperative PSO algorithms studies the efficacy of these algorithms on several benchmark functions.

5.1 Methodology

This chapter investigates the performance of various PSO-based algorithms using several benchmark functions. Although these functions may not necessarily give an accurate indication of the performance of an algorithm on real-world problems, they can be used to investigate certain aspects of the algorithms under consideration. Chapter 6 presents the results of applying the same algorithms tested in this chapter on several neural network training tasks to verify the results of the experiments presented here.

The following functions were used to test the algorithms (3D-plots of the various functions are provided in Appendix D):

Spherical: A very simple, unimodal function with its global minimum located at $\mathbf{x}^* = \mathbf{0}$, with $f(\mathbf{x}^*) = 0$. This function has no interaction between its variables.

$$f_1(\mathbf{x}) = \sum_{i=1}^{n} x_i^2 \tag{5.1}$$

Rosenbrock: A unimodal function, with significant interaction between some of the variables. Its global minimum of $f(\mathbf{x}^*) = 0$ is located at $\mathbf{x}^* = (1, 1, \ldots, 1)$.

$$f_2(\mathbf{x}) = \sum_{i=1}^{n/2} \left(100(x_{2i} - x_{2i-1}^2)^2 + (1 - x_{2i-1})^2\right) \tag{5.2}$$

Ackley: A multi-modal function with deep local minima. The global minimiser is $\mathbf{x}^* = \mathbf{0}$, with $f(\mathbf{x}^*) = 0$. Note that the variables are independent.

$$f_3(\mathbf{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right) + 20 + e \tag{5.3}$$

Rastrigin: A multi-modal version of the Spherical function, characterised by deep local minima arranged as sinusoidal bumps. The global minimum is $f(\mathbf{x}^*) = 0$, where $\mathbf{x}^* = \mathbf{0}$. The variables of this function are independent.

$$f_4(\mathbf{x}) = \sum_{i=1}^{n} \left(x_i^2 - 10\cos(2\pi x_i) + 10\right) \tag{5.4}$$

Griewank: A multi-modal function with significant interaction between its variables, caused by the product term. The global minimiser, $\mathbf{x}^* = \mathbf{0}$, yields a function value of $f(\mathbf{x}^*) = 0$.

$$f_5(\mathbf{x}) = \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n}\cos\left(\frac{x_i}{\sqrt{i}}\right) + 1 \tag{5.5}$$

Schwefel: A multi-modal function with very deep sinusoidal indentations. The global minimiser is located near the one corner of the search space, so that

$$\mathbf{x}^* = (-420.9687, -420.9687, \ldots, -420.9687)$$

with a function value $f(\mathbf{x}^*) = 0$.

$$f_6(\mathbf{x}) = 418.9829n + \sum_{i=1}^{n} x_i \sin\left(\sqrt{|x_i|}\right) \tag{5.6}$$

Function	n	domain
Spherical	30	100
Rosenbrock	30	2.048
Ackley	30	30
Rastrigin	30	5.12
Griewank	30	600
Schwefel	30	500
Quadric	30	100

Table 5.1: Function parameters

Quadric: Another variation of the Spherical function, but with significant interaction between its variables. The global minimiser is located at $\mathbf{x}^* = \mathbf{0}$, so that $f(\mathbf{x}^*) = 0$.

$$f_7(\mathbf{x}) = \sum_{i=0}^{n} \left(\sum_{j=0}^{i} x_j \right)^2 \tag{5.7}$$

Table 5.1 lists the parameter settings for the functions in the benchmark suite. Note that all functions were tested using 30-dimensional search spaces. The PSO algorithms were initialised so that the initial particles in the swarm were distributed throughout the search space using a uniform random number generator. The value of each dimension j of particle i was thus sampled from a distribution of the form

$$x_{i,j} \sim U(-d, d)$$

where d is the appropriate domain value for the function under consideration.

The simulation-quality mt-19937 "Mersenne Twister" random number generator [82] was used to generate the uniform random numbers. This generator has a period of around 10^{6000} and is equi-distributed in 623 dimensions if successive numbers are used as vector components. The same random seed was used for all experiments.

All the results presented in this chapter are the mean values computed over 50 runs. Unless otherwise noted, the inertia weight of the PSO algorithm was set to the value $w = 0.72$; the acceleration coefficients were set so that $c_1 = c_2 = 1.49$. These values lead to convergent trajectories, as can be seen by applying relation (3.21); they further

correspond to popular values used in other publications [39, 36]. All experiments were run for 2×10^5 objective function evaluations, which is equivalent to 10000 iterations of a standard PSO algorithm using a swarm size of 20. Unless specified otherwise, all algorithms used a default swarm size of 20.

5.2 Convergence Speed *versus* Optimality

In Chapter 3 it was shown that certain choices of c_1, c_2 and w leads to a system with a convergent trajectory. The analysis indicated that the trajectory is convergent if $\max(\|\alpha\|, \|\beta\|) < 1$, with the obvious implication that smaller values will lead to more rapid convergence. What was not clear from the analysis presented in Chapter 3 was how the rate of convergence of an individual particle's trajectory influences the searching ability of the swarm as a whole. This section will investigate the behaviour of the PSO algorithm by methodically manipulating the parameters that influence the rate of convergence.

The two types of experiment that were performed used the following labels:

fixed: The values of w, c_1 and c_2 were kept constant throughout the simulation run.

linear: Both c_1 and c_2 were held constant, but the value of w was decreased linearly from 1.0 down to the value specified for the experiment (specified in Tables 5.2 and 5.9 for each experiment), over the duration of the first 1000 iterations of the algorithm. All the algorithms were run for 10000 iterations.

The motivation for using the linearly decreasing inertia weight is that a larger inertia weight during the earlier iterations may allow the particles to explore a larger region before they start to converge. The validity of this conjecture was tested below.

5.2.1 Convergent Parameters

It was shown in Chapter 3 that the PSO algorithm is guaranteed to terminate if a set of convergent parameters were selected, since the algorithm will reach a state in which no further improvement can be achieved. This section investigates the searching ability

Parameter	**Label**									
	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9
w	1.0	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1
c_1, c_2	2.0	1.9	1.8	1.7	1.6	1.5	1.4	1.3	1.2	1.1

Table 5.2: Convergent parameter configurations A_0–A_9.

Config	**Fixed**	**Linear**
A_0	6.95e+02 ± 2.60e+01	6.98e+02 ± 2.84e+01
A_1	3.29e+02 ± 1.19e+01	3.31e+02 ± 1.54e+01
A_2	1.76e+01 ± 1.16e+00	2.22e+01 ± 1.60e+00
A_3	2.02e−01 ± 3.02e−02	2.45e−01 ± 4.13e−02
A_4	8.03e+00 ± 1.17e+00	6.61e+00 ± 1.03e+00
A_5	2.25e+01 ± 1.70e+00	1.39e+01 ± 1.46e+00
A_6	5.79e+01 ± 9.01e+00	1.63e+01 ± 1.38e+00
A_7	2.07e+02 ± 1.96e+01	1.71e+01 ± 1.34e+00
A_8	3.38e+02 ± 2.96e+01	1.91e+01 ± 1.18e+00
A_9	3.82e+02 ± 2.99e+01	1.78e+01 ± 1.28e+00

Table 5.3: PSO performance on Rosenbrock's Function using convergent parameter configurations

of the swarm using different values of c_1, c_2 and w, subject to the constraint that the choice of parameters leads to convergent behaviour. Relation (3.21) (page 85) presents a convenient way to generate parameters that lead to a convergent trajectory. First, a w value is chosen arbitrarily. Then, by applying relation (3.21), a value for $c_1 + c_2$ is determined to ensure convergence. For the experiments conducted in this section, 10 different configurations were tested. These configurations are listed in Table 5.2.

Tables 5.3 and 5.4 show that the PSO performs best on unimodal functions when using parameter configuration A_3, regardless of whether a fixed or linearly decreasing w parameter was used. On some functions the result is dramatic, as can be observed in Table 5.4, where the difference in performance between the A_3 configuration and the next closest competitor was about 100 orders of magnitude.

Config	Fixed	Linear
A_0	6.32e+06 ± 1.39e+05	6.16e+06 ± 1.47e+05
A_1	2.89e+06 ± 8.97e+04	2.94e+06 ± 9.52e+04
A_2	3.33e+02 ± 1.41e+02	4.14e+02 ± 1.25e+02
A_3	1.61e−101 ± 1.04e−98	1.29e−89 ± 1.17e−90
A_4	7.44e+02 ± 2.84e+03	1.50e−02 ± 9.55e−01
A_5	2.99e+05 ± 3.58e+04	2.50e+00 ± 2.80e+02
A_6	1.73e+06 ± 9.43e+04	7.18e+01 ± 3.27e+02
A_7	2.77e+06 ± 1.70e+05	8.94e+02 ± 1.33e+03
A_8	4.24e+06 ± 2.06e+05	3.02e+03 ± 3.19e+03
A_9	3.90e+06 ± 2.01e+05	1.10e+04 ± 4.52e+03

Table 5.4: PSO performance on the Quadric Function using convergent parameter configurations

Config	Fixed	Linear
A_0	1.86e+01 ± 7.24e−02	1.85e+01 ± 7.69e−02
A_1	1.63e+01 ± 9.65e−02	1.62e+01 ± 8.41e−02
A_2	1.42e+00 ± 1.92e−01	1.95e+00 ± 1.61e−01
A_3	2.01e+00 ± 1.98e−01	2.12e+00 ± 1.76e−01
A_4	5.86e+00 ± 3.71e−01	1.90e+00 ± 1.47e−01
A_5	1.08e+01 ± 3.90e−01	2.45e+00 ± 1.83e−01
A_6	1.38e+01 ± 2.57e−01	3.03e+00 ± 2.02e−01
A_7	1.56e+01 ± 1.91e−01	3.62e+00 ± 2.18e−01
A_8	1.66e+01 ± 1.57e−01	4.40e+00 ± 2.29e−01
A_9	1.67e+01 ± 1.67e−01	4.99e+00 ± 2.98e−01

Table 5.5: PSO performance on Ackley's Function using convergent parameter configurations

Config	Fixed	Linear
A_0	6.49e+02 ± 3.52e+01	4.98e+02 ± 4.26e+01
A_1	1.19e+02 ± 2.59e+01	3.28e+02 ± 2.86e+01
A_2	1.74e+01 ± 1.35e+00	2.20e+01 ± 1.51e+00
A_3	1.84e−01 ± 3.22e−02	1.30e−01 ± 1.01e−01
A_4	3.89e−02 ± 5.68e−03	1.94e+00 ± 9.89e−01
A_5	8.24e−02 ± 2.74e−02	9.49e+00 ± 1.61e+00
A_6	2.54e−01 ± 1.23e−02	1.36e+01 ± 1.70e+00
A_7	8.08e−01 ± 2.75e−02	1.42e+01 ± 1.67e+00
A_8	1.61e+00 ± 2.02e−01	1.13e+01 ± 1.83e+00
A_9	2.38e+00 ± 3.94e−01	1.40e+01 ± 1.68e+00

Table 5.6: GCPSO performance on Rosenbrock's Function using convergent parameter configurations

Functions containing multiple minima require that the algorithm examines the search space more thoroughly, so that exploration behaviour becomes more important with respect to exploitation behaviour. Table 5.4 shows that configuration A_2 performed better than A_3 on Ackley's function. Note that configuration A_2 has slower theoretical convergence, since its $\max(\|\alpha\|, \|\beta\|)$ value would be greater than that of configuration A_3. The slower convergence allows it to explore a bit more before converging, which improves its performance on functions containing multiple minima. For the linearly decreasing inertia weight there was no significant difference in performance between the A_2, A_3 and A_4 configurations, since the linearly decreasing inertia weight promotes exploration during the earlier iterations. Note, however, that the fixed A_2 configuration still performed significantly better than the linear A_2, A_3 or A_4 configurations.

The experiments were repeated using the GCPSO algorithm, mainly to see if it responds similarly when different parameter configurations were used. A more detailed comparison between the PSO and the GCPSO is presented in Section 5.3.

Tables 5.6 and 5.7 show that the GCPSO also generally performed better on unimodal functions with parameter configuration A_3, except for the fixed w configuration on Rosenbrock's function, where the A_4 configuration performed significantly better.

Config	Fixed	Linear
A_0	5.23e+06 ± 2.90e+05	5.28e+06 ± 3.38e+05
A_1	3.13e+06 ± 2.11e+05	2.83e+06 ± 1.86e+05
A_2	4.52e+00 ± 1.03e+01	7.94e+01 ± 7.14e+01
A_3	1.37e−133 ± 5.38e−135	1.85e−129 ± 1.92e−87
A_4	1.53e−133 ± 9.10e−124	1.20e−102 ± 1.98e+01
A_5	1.48e−91 ± 1.37e−91	2.13e−75 ± 6.99e+01
A_6	1.50e−75 ± 1.73e−72	1.36e+02 ± 1.49e+03
A_7	1.55e−52 ± 2.59e−41	2.10e+02 ± 1.40e+03
A_8	5.40e−07 ± 1.21e+00	8.11e+02 ± 3.32e+03
A_9	3.81e+01 ± 1.58e+01	1.62e+03 ± 5.06e+03

Table 5.7: GCPSO performance on the Quadric Function using convergent parameter configurations

Config	Fixed	Linear
A_0	1.84e+01 ± 1.27e−01	1.83e+01 ± 1.06e−01
A_1	1.60e+01 ± 1.66e−01	1.63e+01 ± 1.67e−01
A_2	2.55e+00 ± 2.35e−01	2.12e+00 ± 2.05e−01
A_3	2.41e+00 ± 1.74e−01	2.12e+00 ± 1.52e−01
A_4	2.32e+00 ± 5.69e−01	2.81e+00 ± 1.62e−01
A_5	8.18e+00 ± 5.64e−01	3.30e+00 ± 2.07e−01
A_6	1.30e+01 ± 2.90e−01	4.31e+00 ± 2.49e−01
A_7	1.48e+01 ± 2.23e−01	4.97e+00 ± 2.67e−01
A_8	1.58e+01 ± 1.74e−01	6.01e+00 ± 2.59e−01
A_9	1.57e+01 ± 1.58e−01	7.27e+00 ± 3.64e−01

Table 5.8: GCPSO performance on Ackley's Function using convergent parameter configurations

Parameter	**Label**						
	B_0	B_1	B_2	B_3	B_4	B_5	B_6
w	0.7	0.7	0.7	0.7	0.7	0.65	0.75
c_1, c_2	1.7	1.6	1.5	1.71	1.789	1.7	1.7

Table 5.9: Miscellaneous parameter configurations B_0–B_6.

Table 5.8 presents the results of minimising Ackley's function using the GCPSO algorithm. There was no statistically significant difference in performance between the A_2, A_3 and A_4 with fixed w configurations; for the linearly decreasing inertia weight, the A_2 and A_3 configurations also performed similarly. On this function there was no significant difference overall between the fixed and linear algorithms using their respective best configurations, indicating that the linear algorithm provided no real advantage on this function.

The results presented so far show that an inertia weight of about 0.7 usually resulted in the best performance. The next section will investigate some more configurations in this region of parameter space.

5.2.2 Miscellaneous Parameters

Table 5.9 lists the parameter values for some arbitrary configurations with w values in the vicinity of 0.7. Configuration B_0–B_2 and B_6 all satisfy relation (3.21), so that these configurations all yield convergent trajectories. It is expected that B_2 will be more prone to premature convergence than B_0, owing to its smaller $\max(\|\alpha\|, \|\beta\|)$ value. The B_6 configuration, on the other hand, will have a slower rate of convergence than any of the B_0–B_2 configurations. Configuration B_3 has a ϕ_{ratio} of 0.99 (see equation 3.25, page 98), so that it is expected to take a step along a divergent trajectory with probability 0.01. Similarly, configuration B_4 has a ϕ_{ratio} of 0.95, so that it has a 5% chance of taking divergent steps at every iteration. Lastly, configuration B_5 has a ϕ_{ratio} of 0.97, but with a w value of 0.65, compared to the larger w value of 0.7 used in configurations B_3 and B_4. It is therefore expected that the B_5 configuration will have slightly faster convergence.

The miscellaneous parameter configurations resulted in widely differing results. Table 5.10 indicates that the faster rate of convergence of the B_2 configuration allowed

Config	Fixed	Linear
B_0	1.90e−01 ± 2.26e−02	2.73e−01 ± 4.42e−02
B_1	1.96e−02 ± 5.47e−03	2.61e−02 ± 3.73e−02
B_2	6.79e−03 ± 7.83e−03	2.59e−03 ± 8.20e−03
B_3	2.64e−01 ± 2.67e−02	3.28e−01 ± 4.72e−02
B_4	7.54e−01 ± 2.02e−01	7.65e−01 ± 3.85e−01
B_5	1.74e−02 ± 6.80e−03	2.73e−02 ± 1.38e−02
B_6	9.83e−01 ± 2.76e−01	1.26e+00 ± 5.05e−01

Table 5.10: PSO performance on Rosenbrock's Function using miscellaneous parameter configurations

Config	Fixed	Linear
B_0	3.31e−102 ± 3.64e−094	1.68e−091 ± 3.72e−090
B_1	1.60e−121 ± 8.08e−111	1.67e−107 ± 1.79e−104
B_2	1.64e−036 ± 5.83e−004	1.92e−019 ± 7.08e−011
B_3	1.47e−091 ± 3.94e−092	1.89e−087 ± 1.89e−086
B_4	1.66e−071 ± 6.70e−068	2.94e−064 ± 1.67e−063
B_5	1.89e−128 ± 4.30e−117	1.40e−116 ± 1.56e−108
B_6	1.93e−054 ± 1.40e−051	1.81e−049 ± 2.41e−047

Table 5.11: PSO performance on the Quadric Function using miscellaneous parameter configurations

Config	Fixed	Linear
B_0	2.49e+00 ± 2.24e−01	2.62e+00 ± 1.90e−01
B_1	2.85e+00 ± 2.74e−01	1.78e+00 ± 1.60e−01
B_2	4.12e+00 ± 3.12e−01	2.01e+00 ± 1.61e−01
B_3	2.07e+00 ± 1.75e−01	4.44e−15 ± 1.80e−01
B_4	6.66e−15 ± 8.03e−02	4.44e−15 ± 8.74e−02
B_5	2.41e+00 ± 2.71e−01	1.90e+00 ± 1.74e−01
B_6	6.66e−15 ± 3.23e−02	6.66e−15 ± 5.21e−02

Table 5.12: PSO performance on Ackley's Function using miscellaneous parameter configurations

it to achieve above average performance on Rosenbrock's function, although it was not significantly better than B_1 and B_5. The trend seems to be that smaller c_1 and c_2 values (chosen so that the trajectory is convergent), for a fixed w value, resulted in better performance. The linear versions of these configurations did not exhibit significantly different behaviour.

Table 5.11 presents a different picture: notice how large the variance of the B_2 configuration has become. The B_5 configuration still delivered good performance, significantly better than B_0 and B_1. Note that B_5 sometimes takes divergent steps, increasing the diversity of the solutions it examines.

Ackley's function, with multiple local minima, favours algorithms with a tendency to explore more widely (see Table 5.12). This explains why configuration B_2 performed worse than B_1, which in turn performed worse than B_0, since B_0 tends to converge more slowly. The B_4 and B_6 configurations produced much better results, but they had very large variances associated with their mean performance. This indicates that some of the runs discovered very good local minima (or even the global minimum), while other runs discovered only average solutions. The algorithms with a tendency to take divergent steps performed better on average (owing to their enhanced exploration ability), with the exception of B_6, which performed very well, despite the fact that it was a configuration with a convergent trajectory. The linear versions of the algorithm performed better on average, as expected for the Ackley function, since they favoured more exploration

Config	Fixed	Linear
B_0	2.46e−01 ± 3.49e−02	1.18e−01 ± 5.66e−02
B_1	3.57e−02 ± 1.39e−02	5.43e−03 ± 1.09e−02
B_2	3.68e−03 ± 4.10e−03	1.40e−01 ± 2.93e−01
B_3	2.31e−01 ± 4.56e−02	2.07e−01 ± 4.00e−02
B_4	7.90e−01 ± 2.16e−01	5.95e−01 ± 9.21e−02
B_5	2.86e−02 ± 1.41e−02	8.14e−03 ± 1.63e−02
B_6	9.97e−01 ± 2.73e−01	8.74e−01 ± 3.90e−01

Table 5.13: GCPSO performance on Rosenbrock's Function using miscellaneous parameter configurations

Config	Fixed	Linear
B_0	1.97e−136 ± 3.39e−137	1.71e−089 ± 1.61e−86
B_1	1.47e−154 ± 2.79e−151	1.34e−137 ± 4.93e−04
B_2	1.61e−146 ± 1.69e−140	1.26e−099 ± 8.31e−01
B_3	1.33e−132 ± 1.26e−128	1.92e−126 ± 1.03e−91
B_4	3.05e−108 ± 4.21e−103	1.59e−095 ± 3.93e−80
B_5	1.67e−155 ± 2.01e−154	1.56e−030 ± 1.94e−04
B_6	1.24e−087 ± 3.61e−087	1.78e−082 ± 5.10e−67

Table 5.14: GCPSO performance on the Quadric Function using miscellaneous parameter configurations

during earlier iterations.

The experiments were repeated for the GCPSO. Table 5.13 shows that there were no unexpected surprises; the different configurations responded similarly as they did when applied to the standard PSO.

Table 5.14 shows that the GCPSO tends to be more forgiving, since the B_2 configuration, using a fixed w value, improved relative to the other configurations when compared to the results obtained using a standard PSO. Configuration B_5 remained the top performer, although it was not significantly better than B_1. The B_3 and B_4 configurations still performed worse than the convergent ones, with the exception of B_6. Using linearly

Config	Fixed	Linear
B_0	2.36e+00 ± 2.00e−01	2.01e+00 ± 1.13e−01
B_1	2.41e+00 ± 3.03e−01	2.32e+00 ± 1.42e−01
B_2	2.41e+00 ± 3.84e−01	2.89e+00 ± 1.62e−01
B_3	2.41e+00 ± 2.14e−01	1.96e+00 ± 1.56e−01
B_4	2.01e+00 ± 1.64e−01	2.17e+00 ± 1.73e−01
B_5	2.41e+00 ± 2.46e−01	2.32e+00 ± 1.49e−01
B_6	2.01e+00 ± 1.74e−01	4.44e−15 ± 1.55e−01

Table 5.15: GCPSO performance on Ackley's Function using miscellaneous parameter configurations

decreasing inertia weights appears to degrade the performance of the GCPSO on this function, across all configurations.

Interestingly enough, the GCPSO appears to be relatively insensitive to its parameter settings when used to minimise Ackley's function. Table 5.15 indicates that all the fixed inertia weight configurations had approximately the same mean performance. The GCPSO's strong local convergence properties are responsible for this phenomenon — this point will be addressed in more detail in Section 5.3. With the exception of B_6, all linear configurations also had comparable performance.

5.2.3 Discussion of Results

The results presented above indicate that there was a mild trade-off between the performance of a specific configuration on a unimodal function, and the same configuration applied to a function with many local minima. This behaviour was especially noticeable in the results obtained with the original PSO. Configuration A_3, with an inertia weight of 0.7 and acceleration coefficients $c_1 = c_2 = 1.7$, produced very good results on the unimodal functions. In contrast, the best performance on Ackley's function was obtained with parameter settings B_4 and B_6, both of which performed average or below average on the unimodal functions. It appears that no parameter setting tested above could achieve the best of both worlds, suggesting that some alternate mechanism must be introduced to improve the performance of the PSO algorithm on multi-modal functions. Section 5.4

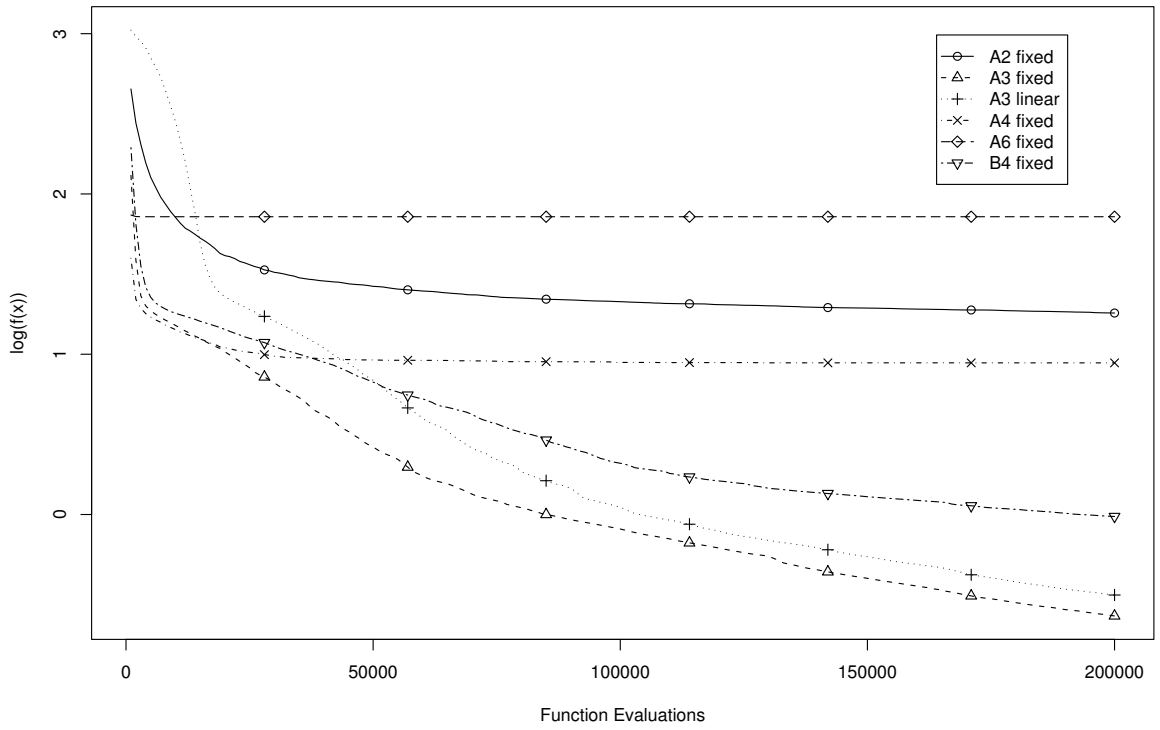


Figure 5.1: The Rosenbrock function error profile, obtained using the original PSO algorithm.

investigates the performance of the global PSO algorithms, MPSO and RPSO, on the some non-unimodal functions.

The linearly decreasing inertia weight helped the PSO to find better solutions on Ackley's function, but generally it had a negative impact on the performance of the algorithm when applied to unimodal functions. The linearly decreasing weight will thus be excluded from further experiments, unless explicitly noted otherwise.

Overall, the performance of the GCPSO was more consistent, indicating that it was less sensitive to the choice of parameters (within limits), but that this could affect the performance of the GCPSO on multi-modal functions negatively. From now on, it will be assumed that the local convergence behaviour of the PSO and GCPSO are similarly affected by the choice of parameters.

The effect of different parameter settings are clearly visible in a plot of the function

value over time. Figure 5.1 shows the error profile of the original PSO algorithm, obtained by minimising the Rosenbrock function. Notice how the A_4 and A_6 configurations had relatively steep slopes during the first few iterations, but then they rapidly leveled off without making any further progress during the rest of the simulation run. Clearly, these configurations suffered from *premature convergence*. The A_2 configuration had a much slower rate of convergence, as can clearly be seen in the plot. Although its rate of progress gradually slowed down, it did not completely level off, indicating that if it was allowed to run for much longer it may have discovered better solutions. The A_3 configuration started with a rapid descent that slowed down after about 1/3 of the time had passed. Note that the slope did not level off towards the end, so that some improvement was still possible. The linearly decreasing inertia weight version of the algorithm had a slower start, mostly because the inertia weight was close to 1.0 during the earlier iterations, effectively preventing the trajectories of the particles from converging. After the inertia weight stabilised, the curve looks similar to that of the fixed inertia weight, but shifted slightly to the right and upwards. The B_4 parameter configuration resulted in performance that was similar to A_3, but exhibiting somewhat slower convergence owing to the disruptive influence of the divergent steps.

Figure 5.2 is the error profile of Ackley's function, obtained using the original PSO algorithm with different parameter configurations. The A_4 and A_6 parameter configurations both caused the PSO algorithm to converge prematurely. The slower rate of convergence offered by the A_2 configuration allowed the PSO to keep on improving its solution, although the rate of improvement was rather slow. Note that it almost reached the same solution discovered by the A_3 fixed inertia weight configuration. Note that both the fixed A_3 configuration and the A_3 linearly decreasing inertia weight configuration stagnated after the first 50000 function evaluations. This is a clear indication that they had become trapped in local minima, and could not escape.

The usefulness of the linearly decreasing inertia weight can clearly be observed in the figure: although it started slower, it usually picked a better local minimum before it became trapped. The B_4 algorithm produced better results than any other configuration, but it still became trapped in a local minimum. Note that the rate of improvement of the B_4 configuration during the earlier training stages was similar to that of the

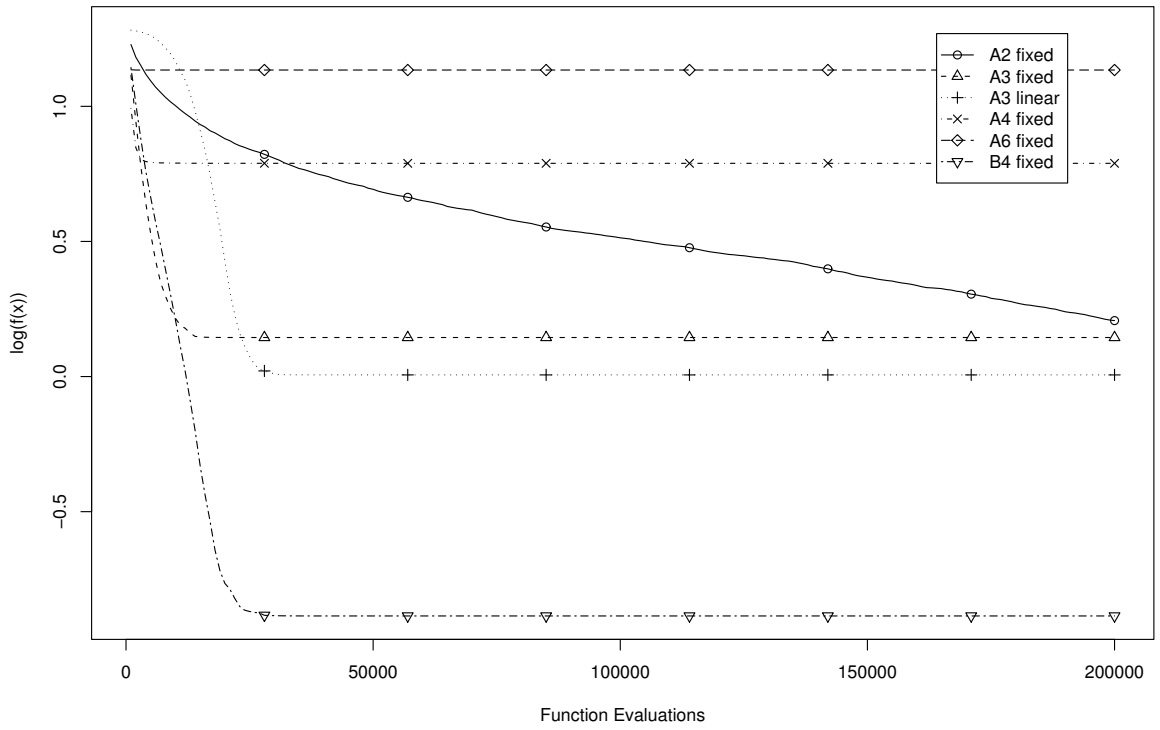


Figure 5.2: The Ackley function error profile, obtained using the original PSO algorithm.

A_3 configuration because they have very similar parameter settings. The occasional divergent step taken by the B_4 configuration allowed it to avoid becoming trapped in a local minimum during the earlier iterations. Later, however, as the particles in the swarm moved closer to one another, the magnitude of these divergent steps began to decrease until they were too small to allow the swarm to escape from a local minimum, thus the swarm became trapped just like the A_3 configuration.

These plots reinforce the statements made above: a faster rate of convergence (*i.e.* A_3) works well on unimodal functions, but the swarm could become trapped in sub-optimal minima on multi-modal functions. A slower rate of convergence, like that offered by configuration A_2, allowed the swarm to avoid becoming trapped in a local minimum, but the rate of convergence was so slow that the algorithm was not competitive over shorter simulation runs. The occasional divergent step taken by the B_4 algorithm allowed it to avoid local minima for a while, thus improving its performance on functions similar to

Ackley's. This benefit came at the cost of slower convergence on unimodal functions. In short, there appears to be no single parameter setting that resulted in the best overall performance. The user must know in advance whether the function will contain many local minima, and choose the appropriate parameter configuration. That being said, the A_3 parameter configuration did consistently perform well, so further experiments made use of similar settings.

5.3 GCPSO Performance

In the previous section the PSO and GCPSO were tested using various parameter configurations. This section presents a side-by-side comparison of the GCPSO and PSO algorithms. All experiments were conducted with the parameter settings of $c_1 = c_2 = 1.49$ and $w = 0.72$, a popular choice in recent publications [39, 36]. Note that these values correspond approximately to parameter configuration B_2 of the previous section.

Table 5.16 presents the results of minimising various functions using both the PSO and GCPSO algorithms. The column labeled "s" lists the number of particles in the swarm for each row. When comparing the results in the table, keep in mind that both the Ackley and Rastrigin functions contain many local minima, and that both the GCPSO and PSO algorithms are not explicitly designed to deal with this type of function without some mechanism to help them to locate the global minimiser. This implies that the quality of the solutions discovered by the algorithms are highly dependent on the initial positions of the particles, which were randomly chosen.

On the Ackley function, the GCPSO performed significantly better than the PSO when using only two particles, but there was no significant difference between the performance of the algorithms when 20 particles were used. The Rastrigin function produced similar results in the two-particle case, but the original PSO performed significantly better on the 20-particle experiment. Keep in mind that the results for both the Ackley and Rastrigin functions are presented only for completeness, since no valid conclusions can be drawn from these results.

The two unimodal functions in Table 5.16 clearly show the stronger local convergence property of the GCPSO. The Spherical function is exceedingly simple, having no inter-

Function	s	**GCPSO**	**PSO**
Ackley	2	1.85e+01 ± 1.99e−01	1.93e+01 ± 2.03e−01
	20	2.70e+00 ± 6.36e−01	3.40e+00 ± 4.58e−01
Rastrigin	2	1.81e+02 ± 2.13e+01	2.93e+02 ± 1.46e+01
	20	7.61e+01 ± 5.07e+00	6.52e+01 ± 4.84e+00
Spherical	2	6.54e−084 ± 1.44e−007	4.03e+004 ± 2.80e+003
	20	2.09e−201 ± 0.00e+00	1.15e−110 ± 1.37e−097
Quadric	2	2.87e+003 ± 1.90e+003	1.14e+007 ± 1.14e+006
	20	9.45e−152 ± 3.87e−146	3.65e−107 ± 4.20e−098

Table 5.16: Comparing GCPSO and PSO on various functions

action between the variables and only a single (thus global) minimum. The GCPSO is able to minimise this function to a very high degree when using only two particles; in contrast, the original PSO struggles with premature convergence if only two particles are used. This is a clear example of the stagnation mentioned in Section 3.3. Note that adding more particles allowed the standard PSO to perform significantly better; even the GCPSO benefitted from the increased diversity offered by the larger number of particles.

The last unimodal function, Quadric, illustrates the same concept, but to a less striking degree. This function has significant interaction between its variables, making this problem harder to solve than the Spherical function. Note that there was a large jump in performance between using 2 particles and 20 particles, even when using the GCPSO. This implies that the greater diversity provided by the additional particles helps the PSO to solve the problem more quickly.

Note that the GCPSO has some additional parameters that can be fine-tuned. The default parameters, specified in Section 3.2, were used throughout. Although these parameters may not be optimal, they have been found to produce acceptable results on a small set of test functions.

To summarise, the GCPSO algorithm has significantly faster convergence on unimodal functions, especially when smaller swarm sizes are used. This improved performance is not visible on multi-modal functions, because the GCPSO can still become trapped in local minima, just like the original PSO. In fact, the GCPSO may be slightly

more prone to becoming trapped because of its faster rate of convergence. The next section studies the efficacy of several mechanisms designed to help the GCPSO (and PSO) to escape from such local minima.

5.4 Global PSO Performance

This section compares the two different strategies for introducing global search behaviour into the PSO algorithm. These two strategies were implemented as the MPSO and RPSO algorithms, introduced in Section 3.4. The MPSO algorithm has three different mechanisms (see Section 3.4.3) that can be used to detect convergence, signalling that the algorithm must re-start. These three convergence-detection methods are also compared.

The four stochastic global PSO algorithms used in the experiments below are:

MPSO$_{radius}$: The MPSO algorithm, using the *maximum swarm radius* convergence detection technique described in Section 3.4.3. The algorithm was configured so that the swarm was declared to have stagnated when $r_{norm} < 10^{-6}$. This value was found (informally) to produce good results on a small set of test functions.

MPSO$_{cluster}$: The MPSO algorithm, using the *cluster analysis* convergence detection technique described in Section 3.4.3. The value of r_{thresh} was set to 10^{-6}; the swarm was re-initialised whenever more than 60% of the particles were clustered around the global best particle. Again, these values were found empirically to result in acceptable performance on a small set of test functions.

MPSO$_{slope}$: The MPSO algorithm, using the *objective function slope* convergence detection technique described in Section 3.4.3. The algorithm re-initialised the swarm whenever $f_{ratio} < 10^{-10}$ for more than 500 consecutive iterations. These values were chosen based on previous experience with the algorithm, where it was found that they result in acceptable performance.

RPSO: The RPSO algorithm, described in Section 3.4.2. No extra particles were added to act as randomised particles, so that three of the 20 normal particles in the swarm were converted into randomised particles, leaving only 17 normal particles. This

Algorithm	**Mean** $f(\mathbf{x})$	**Median** $f(\mathbf{x})$
GCPSO	2.96e+00 ± 6.36e−01	2.70e+00
MPSO_{radius}	7.51e−01 ± 1.35e−01	9.31e−01
$\text{MPSO}_{cluster}$	1.65e+00 ± 1.46e−01	9.31e−01
MPSO_{slope}	1.65e+00 ± 1.70e−01	1.34e+00
RPSO	2.96e+00 ± 3.53e−01	2.96e+00

Table 5.17: Comparing various global algorithms on Ackley's function

value was chosen to limit the possibly disruptive influence that the randomised particles could have on the overall behaviour of the swarm. Note that a single randomised particle is sufficient to achieve the theoretical global convergence property of this algorithm.

All experiments were performed using a swarm size of 20.

These algorithms are all stochastic global optimisers, thus they are not guaranteed to find the global (or even a good local) minimum on every run. Since only one poor solution, say on the order of 10^{-3}, can skew the mean of a population otherwise consisting of values on the order of 10^{-19}, the median of each simulation run is also provided along with the mean.

Table 5.17 presents the results of applying the four global PSO-based algorithms to the task of minimising Ackley's function. The MPSO_{radius} algorithm performed significantly better than any other algorithm. Note that all three the MPSO algorithm performed significantly better than the GCPSO and RPSO algorithms. On this function it does not appear that the RPSO algorithm had any positive influence on the performance of the algorithm, since it had the same mean performance as the GCPSO algorithm.

The results in Table 5.18 were obtained by minimising Rastrigin's function using the various algorithms. Note that there was once again no significant difference between the performance of the RPSO algorithm and that of the GCPSO algorithm. All three the MPSO algorithms performed significantly better than the GCPSO. Amongst themselves the MPSO_{radius} and $\text{MPSO}_{cluster}$ algorithms performed significantly better than the MPSO_{slope} algorithm, although there was no significant difference between the

Algorithm	**Mean** $f(\mathbf{x})$	**Median** $f(\mathbf{x})$
GCPSO	7.61e+01 ± 5.07e+00	7.61e+01
MPSO_{radius}	4.58e+01 ± 1.45e+00	4.58e+01
$\text{MPSO}_{cluster}$	4.48e+01 ± 1.41e+00	4.48e+01
MPSO_{slope}	4.97e+01 ± 1.59e+00	4.97e+01
RPSO	7.41e+01 ± 3.49e+00	7.41e+01

Table 5.18: Comparing various global algorithms on Rastrigin's function

Algorithm	**Mean** $f(\mathbf{x})$	**Median** $f(\mathbf{x})$
GCPSO	2.21e−02 ± 4.84e−03	1.23e−02
MPSO_{radius}	1.99e−09 ± 1.87e−10	1.52e−09
$\text{MPSO}_{cluster}$	3.69e−02 ± 8.19e−03	1.48e−02
MPSO_{slope}	1.68e−03 ± 7.39e−04	2.17e−19
RPSO	4.53e−02 ± 7.72e−03	1.23e−02

Table 5.19: Comparing various global algorithms on Griewank's function

performance of the MPSO_{radius} and $\text{MPSO}_{cluster}$ algorithms.

Table 5.19 presents the results of minimising Griewank's function using the various algorithms. Both the $\text{MPSO}_{cluster}$ and RPSO algorithms performed significantly *worse* than the GCPSO. Clearly, the $\text{MPSO}_{cluster}$ algorithm had no beneficial effect on this function, most likely because it failed to trigger, *i.e.* it could not detect that the swarm has stagnated. This phenomenon can be attributed to the fact that the adjustable parameter (r_{thresh}) in the $\text{MPSO}_{cluster}$ algorithm was set to a value that was too small for Griewank's function. If the function contains many local minima very close to one another, the swarm could stagnate with a few particles scattered over several neighbouring minima. In this case, the cluster detection algorithm could fail, since there are too few particles close to the global best particle. The RPSO algorithm basically prevents some of the particles from performing their usual role in the minimisation process, so it is quite probable that the RPSO may perform worse than the GCPSO on some functions.

Note the large difference between the mean and the median values of the MPSO_{slope} algorithm results. Based on the median, it is clear that the MPSO_{slope} algorithm per-

Algorithm	**Mean** $f(\mathbf{x})$	**Median** $f(\mathbf{x})$
GCPSO	4.87e+03 ± 1.31e+02	4.87e+03
MPSO_{radius}	3.71e+03 ± 8.07e+01	3.71e+03
$\text{MPSO}_{cluster}$	4.85e+03 ± 1.32e+02	4.85e+03
MPSO_{slope}	4.08e+03 ± 8.25e+01	4.08e+03
RPSO	4.73e+03 ± 1.38e+02	4.73e+03

Table 5.20: Comparing various global algorithms on Schwefel's function

formed better than the MPSO_{radius} algorithm on most of the runs, although the mean does not reflect this. This is an indication that the MPSO_{slope} algorithm did not trigger often enough to detect stagnation during all the simulation runs. This suggests that future research should investigate automated methods for adjusting the sensitivity of the MPSO_{slope} algorithm.

Applying the various algorithms to the task of minimising Schwefel's function yielded the results presented in Table 5.20. The RPSO and $\text{MPSO}_{cluster}$ algorithms did not perform significantly better than the baseline GCPSO algorithm. The other two MPSO algorithms, MPSO_{slope} and MPSO_{radius}, did show a significant improvement over the GCPSO algorithm. The MPSO_{radius} algorithm further showed a significant improvement in performance over the MPSO_{slope} algorithm, making it the overall best algorithm on this function.

Figure 5.3 further elucidates the characteristics of the various algorithms when applied to Griewank's function. The GCPSO, $\text{MPSO}_{cluster}$ and RPSO algorithm all behave similarly, clearly failing to make any further improvements after about 10000 function evaluations. This behaviour is normal for the GCPSO, since it is not explicitly designed to deal with multiple local minima. For the RPSO, this implies that the randomised particles have no effect. In fact, on none of the problems did the RPSO offer any improvement in performance over the GCPSO.

The $\text{MPSO}_{cluster}$ algorithm fails to trigger, thus it remains trapped in a local minimum. The other two MPSO algorithms performed significantly better, as can clearly be seen in Figure 5.3. Note that the MPSO_{radius} algorithm was able to detect stagnation more quickly than the MPSO_{slope} algorithm. While this appears to work well

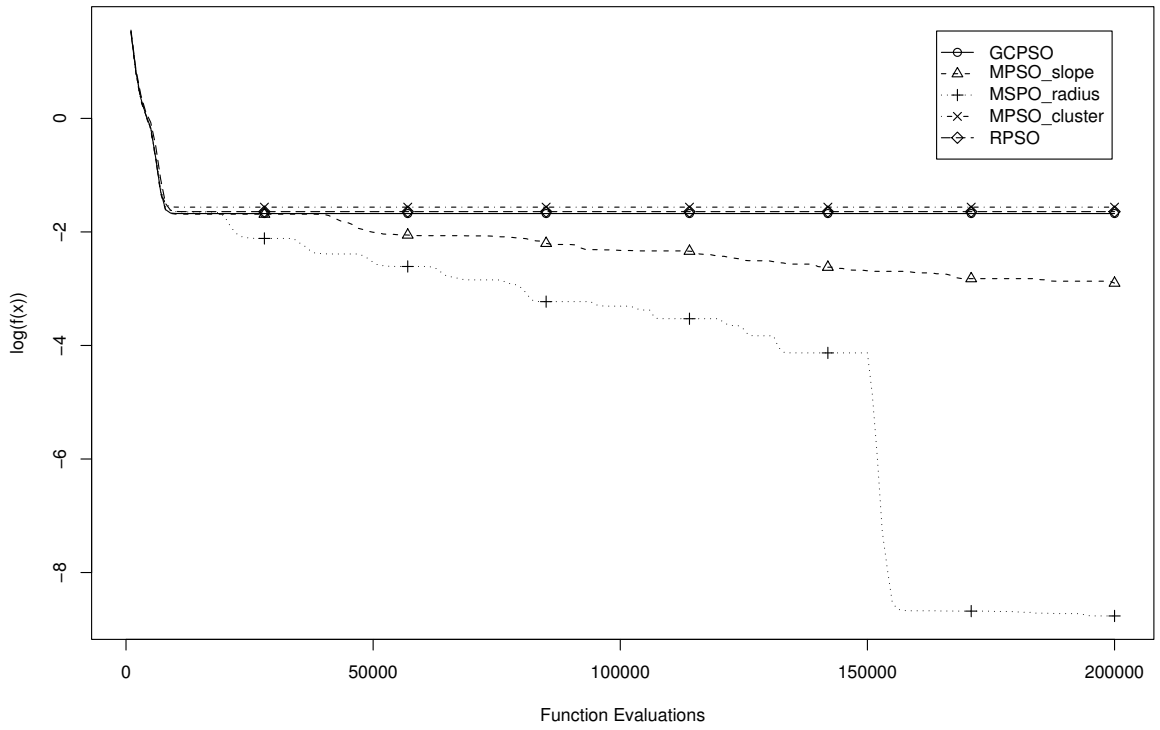


Figure 5.3: The Griewank function error profile, obtained using the various global PSO algorithms.

on Griewank's function, MPSO$_{radius}$ may be too hasty in declaring stagnation on other functions. Keep in mind that these curves represent the mean value of the objective function at each time step, thus the MPSO$_{slope}$ algorithm's curve does not accurately portray its true behaviour. Recall that there is a large difference between the MPSO$_{slope}$ algorithm's mean and median values for this particular function. If the graph was drawn using the median values, instead of the mean values, then the MPSO$_{slope}$ algorithm would have been correctly been portrayed as the better of the MPSO$_{slope}$ and MPSO$_{radius}$ algorithms. The reason for choosing to display this particular plot was that it clearly shows how some algorithms (*e.g.* MPSO$_{cluster}$) fail to detect stagnation on some functions.

5.4.1 Discussion of Results

The results presented in this section indicate that the MPSO family of algorithms have the ability to improve the performance of the GCPSO on functions containing multiple minima. The effectiveness of the MPSO algorithm depends on the algorithm used to determine when to re-initialise the swarm. From the results it is clear that the MPSO_{slope} and MPSO_{radius} algorithms were able to improve on the performance of the GCPSO consistently, while the $\text{MPSO}_{cluster}$ algorithm sometimes failed to deliver improved performance.

The next section investigates the performance of the cooperative PSO algorithms. Because of the improved diversity present in these algorithms, they are expected to perform well on multi-modal functions.

5.5 Cooperative PSO Performance

This section compares the performance of the various CPSO-S and CPSO-H algorithms. Since the CPSO-S and CPSO-H algorithms make strong assumptions about the decomposability of the functions they minimise, several test functions will be evaluated in both their original and their rotated versions. Rotating the coordinate axes of the search space through random angles induces a high degree of correlation between the dimensions, making the problems considerably harder for CCGA-style algorithms (which includes the CPSO-S and CPSO-H algorithms) to solve [115, 105].

All the algorithms were tested using both the standard PSO update equations, as well as the GCPSO update equations. In the tables below these two versions will simply be indicated with the column labels "GCPSO" and "PSO". The normal non-CPSO algorithms will be identified with the label "Standard", so that, for example, an entry with a row label of "Standard" and a column label of "GCPSO" will refer to the GCPSO algorithm as introduced in Section 3.2.

It is expected that the performance of a CPSO algorithm using the standard PSO update equations will be different from that of the same CPSO algorithm using the GCPSO update equations, since the former has slower convergence, thus implying greater diversity.

5.5.1 Experimental Design

The standard PSO algorithms perform significantly less work during a single iteration of the algorithm's outer loop, compared to the amount of work done during one iteration of the outer loop of a CPSO algorithm. Some other measure of the amount of work done must therefore be considered. Instead of counting the number of iterations that the outer loop is executed, the number of times that the objective function is evaluated is counted. In practice, a maximum number of function evaluations is specified for a simulation run. This value is then used to compute the requisite number of iterations that the specific algorithm must use. First, observe that, for a standard PSO,

$$F = s \times I$$

where F is the number of times the objective function is evaluated, s is the swarm size and I the number of iterations that the algorithm is run. To ensure a fair comparison between two non-cooperative algorithms the value F must be held constant by computing the value of I so that $I = F/s$.

This approach can be used to compare the influence of the swarm size on the performance of the PSO algorithm. For example, if a budget of 10000 function evaluations is available, then a swarm with 10 particles must run for $10000/10 = 1000$ iterations. Using 20 particles decreases the number of iterations, so that the larger swarm may only run for $10000/20 = 500$ iterations.

When comparing the performance of a CPSO algorithm to that of a standard PSO algorithm, the same principle must be observed. If a CPSO algorithm splits the search space into K parts, then

$$F = s \times K \times I$$

This implies that the number of iterations available to the CPSO algorithm is reduced by a factor K — a significant reduction if $K = n$, where n is the number of dimensions of the search space. The number of allowed iterations for a CPSO-H$_K$ algorithm is computed similarly by using a factor of $K + 1$ rather than K, to compensate for the extra swarm that is used in the algorithm (see Section 4.3.1 for details).

The practical implication of this approach is that each individual particle in a PSO swarm has the opportunity to update its position K times more frequently than a particle

in a CPSO-S_K algorithm, if both algorithms use the same swarm size. The only way to increase the number of opportunities each particle in the CPSO swarm receives to update its position is to decrease the swarm size. For example, if a split factor of $K = 2$ is selected, then the CPSO-S_2 algorithm can run for $10000/(2 \times 20) = 250$ iterations using a swarm size of 20. By reducing the swarm size to 10, the algorithm is allowed to run for $10000/(2 \times 10) = 500$ iterations, the same number as a standard PSO using a swarm size of 20. This means that the CPSO-S_K algorithm usually performs better when using smaller swarm sizes than the corresponding standard PSO.

All the experiments conducted in this thesis used this method for determining the correct number of iterations for each type of algorithm to ensure that the PSO and CPSO algorithms were compared on equal footing.

The following algorithms were compared in the results presented below:

Standard: A standard PSO algorithm, using a fixed inertia weight of $w = 0.72$ and acceleration coefficients $c_1 = c_2 = 1.49$. These values were selected to correspond with settings often encountered in the literature [39, 36].

CPSO-S: The CPSO-S algorithm introduced in Section 4.2.2, with acceleration coefficients $c_1 = c_2 = 1.49$. The inertia weight was configured to decrease linearly from 1.0 down to 0.72 during the first 1000 iterations. Note that this algorithm is equivalent to a CPSO-S_K algorithm with $K = n$, where $n = 30$ for all the functions tested here.

CPSO-H: The CPSO-H algorithm introduced in Section 4.3.1. The same parameter settings as the CPSO-S algorithm were used.

CPSO-S_6: A CPSO-S_K algorithm with $K = 6$. The value 6 was chosen so that the number of swarms and the number of variables searched by each swarm was approximately the same. Previous experience with the CPSO-S_K algorithm indicated that this split factor produces acceptable results [138]. This swarm also used the same parameter settings as the CPSO-S algorithm.

CPSO-H_6: A CPSO-H_K algorithm with $K = 6$. Again, the same parameter settings as the CPSO-S algorithm were used.

GA: A binary-coded Genetic Algorithm with a population size of 100, using 16 bits to encode each variable. A two-point bitwise crossover algorithm, together with a single-bit mutation operator, was used. Selection pressure was applied using a fitness-proportionate model, with a one-element elitist strategy. The crossover rate was set to 0.6 and the mutation rate to 0.02. This algorithm was configured to correspond to the GA used by Potter during his CCGA experiments [106].

CCGA: A CCGA-1 algorithm [106], where each of the functions was split into 30 components, resulting in 30 cooperating swarms. The same settings as used for the GA algorithm were implemented. This configuration was again chosen to correspond to Potter's experiments.

Note that all the PSO algorithms were tested using both the standard PSO update equation for the global best particle, as well as the GCPSO update equation for the global best particle.

5.5.2 Unrotated Functions

Tables 5.21 and 5.22 present the results of minimising two unimodal functions using the different algorithms. On Rosenbrock's function the standard GCPSO was less sensitive to the swarm size than the standard PSO. These two standard algorithms also produced better results than any of the other algorithms on this problem, with the standard PSO taking the lead. Both the GA-based algorithms performed significantly worse than the PSO-based algorithms, despite their larger population sizes (which correspond to the population size used by Potter [106, 105]). All the CPSO algorithms exhibited a general tendency to produce better results with smaller swarm sizes, except the CPSO-H_6 algorithm, which seemed to favour larger swarm sizes.

The Quadric function was considerably easier to solve than Rosenbrock's function, as can clearly be seen in Table 5.22. The standard GCPSO performed significantly better than any other algorithm by a rather large margin, followed by the PSO-version of the CPSO-S algorithm. Note that the PSO-version of the CPSO-S algorithm (using a swarm size of 10) performed better than the Standard PSO algorithm. This was unexpected, since this function has significant correlation between its variables. It is

Algorithm	s	**GCPSO**	**PSO**
Standard	10	4.16e−02 ± 6.43e−03	1.30e−01 ± 1.45e−01
	15	2.55e−02 ± 7.77e−03	5.53e−03 ± 6.19e−03
	20	2.73e−02 ± 1.04e−02	9.65e−03 ± 7.28e−03
CPSO-S	10	6.89e−01 ± 3.58e−02	7.58e−01 ± 1.16e−01
	15	6.73e−01 ± 3.23e−02	7.36e−01 ± 3.04e−02
	20	7.70e−01 ± 1.29e−01	9.06e−01 ± 3.56e−02
CPSO-H	10	2.56e−01 ± 2.27e−02	2.92e−01 ± 2.19e−02
	15	2.93e−01 ± 2.08e−02	3.14e−01 ± 1.74e−02
	20	4.12e−01 ± 3.13e−02	4.35e−01 ± 2.48e−02
CPSO-S_6	10	2.05e+00 ± 6.88e−01	1.41e+00 ± 4.73e−01
	15	2.03e+00 ± 6.20e−01	2.47e+00 ± 7.00e−01
	20	1.80e+00 ± 5.92e−01	1.59e+00 ± 5.03e−01
CPSO-H_6	10	7.62e−02 ± 1.32e−02	1.94e−01 ± 2.63e−01
	15	5.83e−02 ± 5.87e−03	2.59e−01 ± 2.47e−01
	20	1.02e−01 ± 1.76e−02	4.21e−01 ± 3.21e−01
GA	100	6.32e+01 ± 1.19e+01	
CCGA	100	3.80e+00 ± 1.93e−01	

Table 5.21: Unrotated Rosenbrock's function: mean value after 2×10^5 function evaluations

Algorithm	s	**GCPSO**	**PSO**
Standard	10	6.40e−159 ± 1.25e−158	1.08e+00 ± 1.41e+00
	15	4.59e−166 ± 1.03e−167	2.85e−72 ± 5.41e−72
	20	2.13e−146 ± 3.87e−146	2.17e−98 ± 4.20e−98
CPSO-S	10	2.99e−75 ± 5.84e−75	2.55e−128 ± 4.98e−128
	15	8.42e−61 ± 1.07e−60	7.26e−89 ± 1.14e−88
	20	6.57e−50 ± 1.24e−49	3.17e−67 ± 2.21e−67
CPSO-H	10	3.61e−61 ± 7.08e−61	5.41e−95 ± 1.05e−94
	15	1.91e−58 ± 3.20e−58	6.74e−81 ± 8.92e−81
	20	3.14e−48 ± 4.68e−48	1.45e−63 ± 1.98e−63
CPSO-S_6	10	2.18e−09 ± 3.84e−09	4.63e−07 ± 6.14e−07
	15	6.39e−07 ± 8.80e−07	1.36e−05 ± 1.76e−05
	20	3.26e−06 ± 4.01e−06	1.20e−04 ± 8.99e−05
CPSO-H_6	10	6.92e−38 ± 1.35e−37	2.63e−66 ± 5.08e−66
	15	1.01e−32 ± 1.99e−32	9.00e−46 ± 1.09e−45
	20	8.36e−33 ± 1.61e−32	1.40e−29 ± 1.15e−29
GA	100	1.68e+06 ± 2.56e+05	
CCGA	100	1.38e+02 ± 9.20e+01	

Table 5.22: Unrotated Quadric function: mean value after 2×10^5 function evaluations

Algorithm	s	**GCPSO**	**PSO**
Standard	10	7.90e+00 ± 8.86e−01	7.33e+00 ± 6.23e−01
	15	4.06e+00 ± 5.26e−01	4.92e+00 ± 5.81e−01
	20	3.27e+00 ± 6.36e−01	3.57e+00 ± 4.58e−01
CPSO-S	10	2.91e−14 ± 1.65e−15	2.90e−14 ± 1.60e−15
	15	2.93e−14 ± 1.43e−15	3.01e−14 ± 1.42e−15
	20	2.96e−14 ± 1.69e−15	3.05e−14 ± 1.84e−15
CPSO-H	10	2.85e−14 ± 1.41e−15	2.78e−14 ± 1.71e−15
	15	3.10e−14 ± 1.66e−15	2.92e−14 ± 1.67e−15
	20	3.13e−14 ± 1.88e−15	2.98e−14 ± 1.56e−15
CPSO-S_6	10	6.94e−08 ± 5.21e−08	1.12e−06 ± 4.01e−07
	15	1.35e−06 ± 6.53e−07	1.11e−05 ± 4.35e−06
	20	3.15e−06 ± 1.44e−06	5.42e−05 ± 1.66e−05
CPSO-H_6	10	2.21e−10 ± 3.24e−10	9.42e−11 ± 7.58e−11
	15	1.29e−11 ± 1.41e−11	9.57e−12 ± 7.96e−12
	20	2.08e−12 ± 2.15e−12	2.73e−12 ± 2.03e−12
GA	100	1.38e+01 ± 4.04e−01	
CCGA	100	9.51e−02 ± 3.39e−02	

Table 5.23: Unrotated Ackley's function: mean value after 2×10^5 function evaluations

Algorithm	s	**GCPSO**	**PSO**
Standard	10	8.76e+01 ± 5.60e+00	8.27e+01 ± 5.64e+00
	15	7.64e+01 ± 5.29e+00	7.44e+01 ± 5.66e+00
	20	7.42e+01 ± 5.07e+00	6.79e+01 ± 4.84e+00
CPSO-S	10	0.00e+00 ± 0.00e+00	0.00e+00 ± 0.00e+00
	15	0.00e+00 ± 0.00e+00	0.00e+00 ± 0.00e+00
	20	0.00e+00 ± 0.00e+00	0.00e+00 ± 0.00e+00
CPSO-H	10	0.00e+00 ± 0.00e+00	0.00e+00 ± 0.00e+00
	15	0.00e+00 ± 0.00e+00	0.00e+00 ± 0.00e+00
	20	0.00e+00 ± 0.00e+00	0.00e+00 ± 0.00e+00
CPSO-S_6	10	3.81e−01 ± 1.92e−01	1.39e−01 ± 1.12e−01
	15	2.60e−01 ± 1.66e−01	6.00e−02 ± 6.62e−02
	20	1.79e−01 ± 1.28e−01	1.46e−01 ± 1.03e−01
CPSO-H_6	10	2.37e+00 ± 5.22e−01	1.47e+00 ± 3.16e−01
	15	1.18e+00 ± 2.80e−01	8.77e−01 ± 2.20e−01
	20	9.40e−01 ± 2.59e−01	7.78e−01 ± 1.87e−01
GA	100	1.29e+02 ± 7.00e+00	
CCGA	100	1.22e+00 ± 2.35e−01	

Table 5.24: Unrotated Rastrigin's function: mean value after 2×10^5 function evaluations

also interesting to note that the GCPSO-version of the CPSO-S algorithm performed significantly worse than the PSO-version. This is because the GCPSO-version had a faster rate of convergence, so that the diversity of each of the constituent swarms was lower than that of the PSO-version, leading to a decrease in performance.

The CPSO-S_6 algorithm exhibited exactly the opposite behaviour: the GCPSO-version performed better than the PSO-version. Note that the CPSO-S_6 algorithm used six swarms with 5-dimensional search spaces each. Since the rate of convergence of a PSO algorithm in a 5-dimensional search space is clearly slower than its rate of convergence of in a 1-dimensional search space, diversity was better preserved. This meant that the CPSO-S_6 algorithm did not suffer from the same premature convergence problem that the GCPSO-version of the CPSO-S algorithm had.

Observe that the CPSO-H_6 algorithm performed significantly better than the CPSO-S_6 algorithm, but still significantly worse than the non-CPSO algorithms. There are two possible interpretations of its intermediate level of performance: either the CPSO-S_K component had a negative influence on the performance of the non-CPSO component of the algorithm, or the non-CPSO component improved the performance of the CPSO-S_K component. The performance of this algorithm on the non-unimodal functions (discussed below) provides the necessary clues for deciding which interpretation is more likely.

Overall, it appears that the standard (*i.e.* non-CPSO) algorithms had better performance on unimodal functions. When considering some functions with multiple local minima, like Ackley and Rastrigin's functions, the roles were reversed. Table 5.23 presents results obtained by minimising Ackley's function using the various algorithms. It is immediately clear that the CPSO algorithms performed significantly better than the standard ones on this function. One possible explanation is the increased diversity offered by the group of cooperating swarms, thus reducing the probability of falling into an inferior local minimum during the earlier iterations of the algorithm. Note that there was not much of a difference between the performance of the PSO and GCPSO versions of the CPSO-S and CPSO-H algorithms, unlike the behaviour observed on the unimodal functions.

The GCPSO-version of the CPSO-S_6 algorithm again performed better than the PSO-version; for the CPSO-H_6 algorithm, the performance of the the two variations was

not significantly different. The CCGA algorithm performed significantly better than the standard PSO-based algorithms, even though the standard GA algorithm had the worst performance overall.

Another trend can be observed on Ackley's function: Larger swarm sizes (thus more diversity) improves the performance of the standard PSO and GCPSO algorithms significantly. The opposite trend can be observed in the behaviour of the CPSO-S, CPSO-H and CPSO-S_6 algorithms. The explanation for this behaviour is that the CPSO algorithms (with the exception of the CPSO-H_6 algorithm) generate sufficient diversity through their cooperative structure, and therefore they need the extra iterations[1] more than they need the extra diversity.

Table 5.24 shows the results of minimising Rastrigin's function, another non-unimodal function. A dramatic difference was observed between the performance of the CPSO-S (and CPSO-H) algorithms, and that of the standard PSO-based algorithms. A likely explanation for the superior performance of the CPSO-S algorithm is that an n-dimensional Rastrigin function can be decomposed into n components of the form

$$f(x) = x^2 - 10\cos(2\pi x) + 10$$

This function is depicted in Figure 5.4. Note how the distance between successive local minima remains constant. If the swarm starts with some particles located near different local minima, some of the particles may assume velocities that are approximately a multiple of the distance between two local minima. The quadratic term in Rastrigin's function ensures that the particles move in the direction of the global minimum. Under these circumstances a particle will quickly be able to find the interval $[-0.5, 0.5]$ in which the global minimum is located. This explains why the CPSO-S and CPSO-H algorithms are able to locate the global minimum so easily.

It is far less likely that the CPSO-S_6 algorithm will find a 5-dimensional velocity vector with the "optimal step size" property. Therefore, neither the CPSO-S_6 nor the CPSO-H_6 algorithms exhibited the same level of performance as the CPSO-S and CPSO-H swarms.

Figure 5.5 plots the performance of some of the algorithms on Ackley's function over time. All algorithms shown in the plot were the GCPSO-versions. Note how the standard

[1] recall the description of the experimental set-up in Section 5.5.1

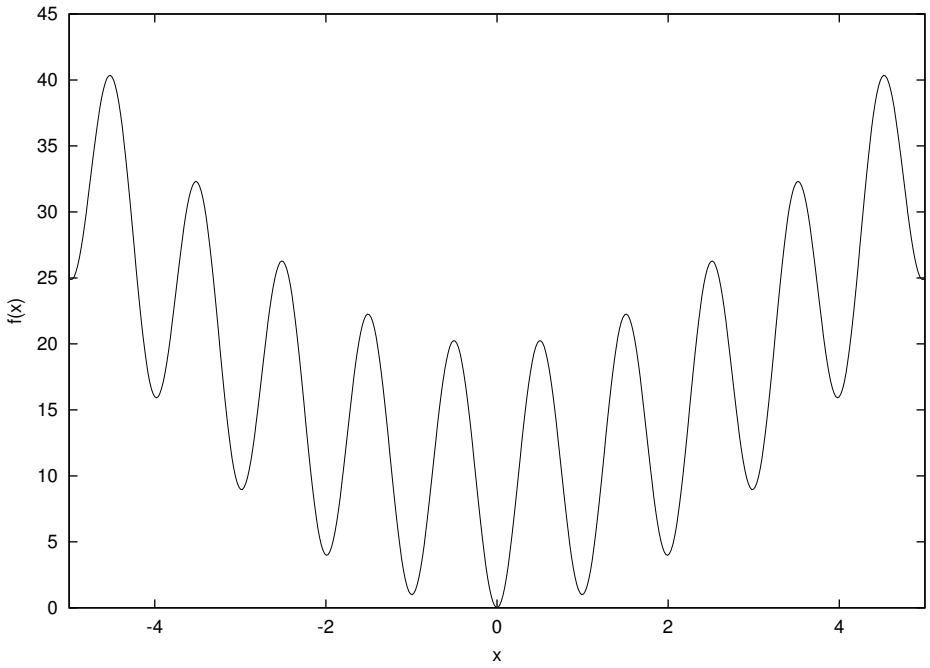


Figure 5.4: A plot of Rastrigin's function in one dimension.

GCPSO quickly becomes trapped in a local minimum, just as observed previously in Section 5.4. The CPSO algorithms were clearly able to avoid becoming trapped in local minima for longer than the standard PSO.

Figure 5.6 shows how the CPSO-S and CPSO-S_6 algorithms perform compared to the GA-based algorithms, as well as the MPSO algorithm (using the *maximum swarm radius* convergence detection algorithm). Although the MPSO algorithm is guaranteed to find the global minimiser eventually, the CPSO algorithms are clearly more efficient. The performance plots for other non-unimodal functions exhibited similar properties.

5.5.3 Rotated Functions

All the CPSO algorithms make the assumption that a function can be minimised by decomposing it into disjoint subspaces, and minimising each subspace separately. This means that they assume there is little or no correlation between the different components which they decomposed the search space into. By testing these algorithms on functions that have highly correlated variables it is possible to see how robust the CPSO algorithms

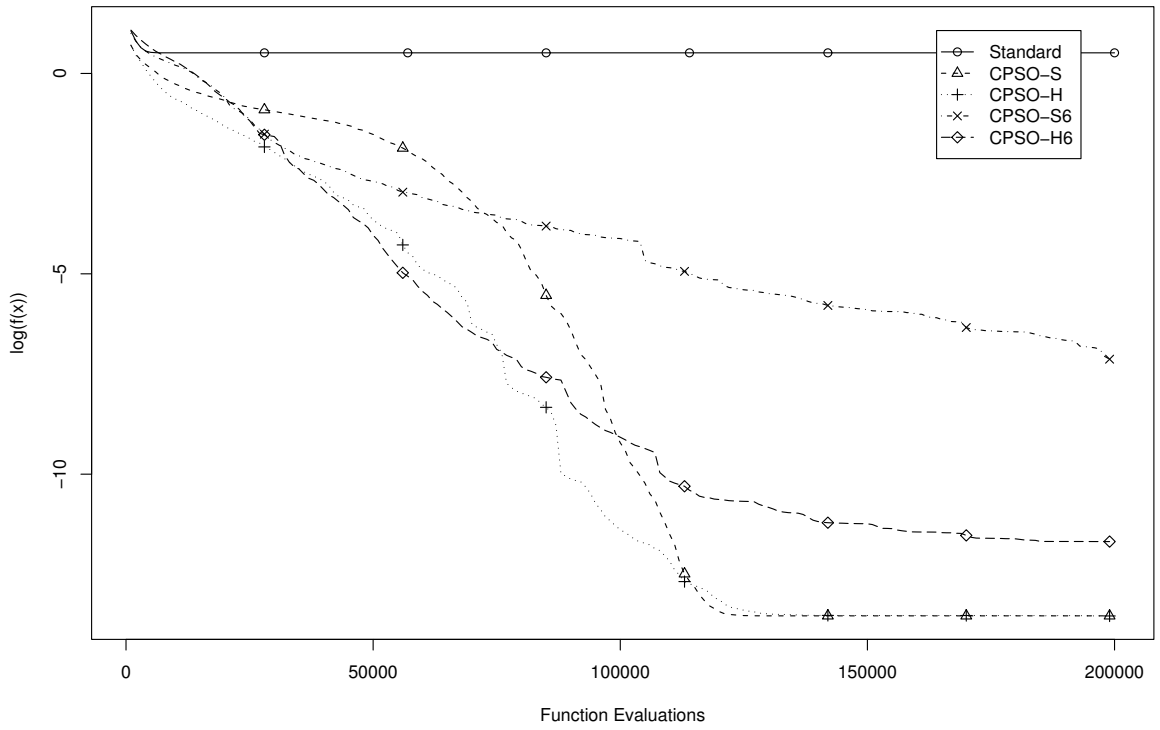


Figure 5.5: The unrotated Ackley Function error profile, obtained using the various CPSO algorithms.

are when they are faced with functions that target their main weakness. If a function with uncorrelated variables is rotated through arbitrary angles, its variables become highly correlated [115]. This method was used to obtain the results presented in this section.

The results of minimising the rotated version of Rosenbrock's function are presented in Table 5.25. All the algorithms performed worse than they did on the unrotated version of this function. The standard PSO and both variations of the CPSO-S algorithms experienced the most severe degradation of performance, though. This is an indication of the fact that the CPSO-S algorithm suffers from the increased correlation between the variables. Even though the CPSO-S algorithm did suffer a significant decrease in performance, it was still able to solve the problem more effectively than either of the two GA-based algorithms.

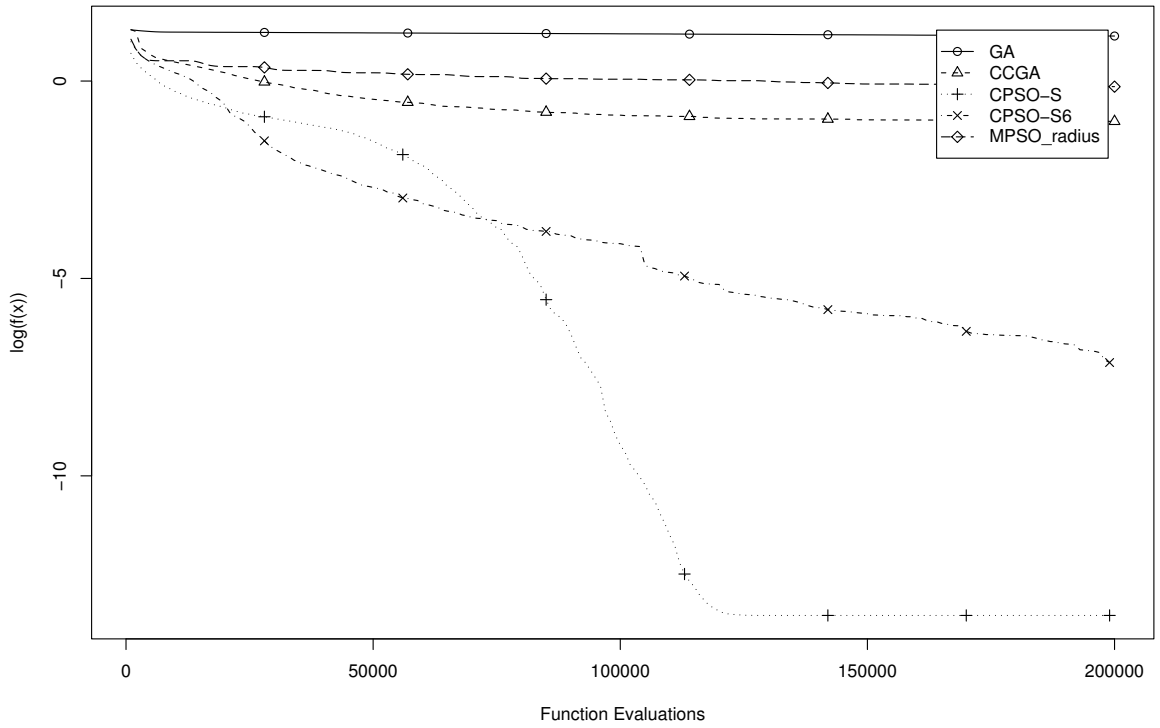


Figure 5.6: The unrotated Ackley Function error profile, obtained using some of the CPSO algorithms, with the profile of the MPSO and GA-based algorithms included for comparison.

Algorithm	s	**GCPSO**	**PSO**
Standard	10	1.03e−01 ± 3.46e−02	3.32e−01 ± 9.50e−02
	15	9.59e−02 ± 7.56e−03	2.84e−01 ± 5.17e−02
	20	1.50e−01 ± 1.21e−02	3.16e−01 ± 3.41e−02
CPSO-S	10	3.70e+00 ± 8.98e−01	3.23e+00 ± 7.78e−01
	15	3.32e+00 ± 8.00e−01	2.58e+00 ± 5.36e−01
	20	4.24e+00 ± 9.02e−01	4.37e+00 ± 8.51e−01
CPSO-H	10	5.50e−01 ± 1.71e−01	4.26e−01 ± 3.83e−02
	15	5.80e−01 ± 1.49e−01	4.96e−01 ± 4.53e−02
	20	7.31e−01 ± 1.20e−01	1.06e+00 ± 2.96e−01
CPSO-S_6	10	3.14e+00 ± 7.51e−01	2.65e+00 ± 6.69e−01
	15	2.47e+00 ± 5.51e−01	3.84e+00 ± 9.81e−01
	20	3.94e+00 ± 8.21e−01	4.27e+00 ± 7.73e−01
CPSO-H_6	10	2.43e−01 ± 5.63e−02	1.77e−01 ± 3.62e−02
	15	2.28e−01 ± 3.06e−02	3.73e−01 ± 2.07e−01
	20	2.89e−01 ± 3.78e−02	4.73e−01 ± 1.35e−01
GA	100	6.15e+01 ± 1.42e+01	
CCGA	100	1.32e+01 ± 2.19e+00	

Table 5.25: Rotated Rosenbrock's function: mean value after 2×10^5 function evaluations

Algorithm	s	**GCPSO**	**PSO**
Standard	10	2.43e+00 ± 2.40e+00	6.02e+03 ± 2.17e+03
	15	1.35e+00 ± 9.90e−01	3.35e+02 ± 1.35e+02
	20	1.36e+00 ± 8.95e−01	1.12e+02 ± 4.91e+01
CPSO-S	10	1.73e+03 ± 5.62e+02	1.47e+03 ± 4.77e+02
	15	1.50e+03 ± 4.65e+02	1.28e+03 ± 3.88e+02
	20	1.86e+03 ± 5.09e+02	1.72e+03 ± 5.91e+02
CPSO-H	10	2.86e+02 ± 1.55e+02	2.15e+02 ± 8.75e+01
	15	2.16e+02 ± 9.16e+01	3.45e+02 ± 9.92e+01
	20	3.42e+02 ± 1.12e+02	4.10e+02 ± 1.32e+02
CPSO-S_6	10	1.61e+03 ± 4.59e+02	2.89e+03 ± 1.07e+03
	15	2.52e+03 ± 9.36e+02	2.99e+03 ± 1.07e+03
	20	2.44e+03 ± 8.49e+02	4.64e+03 ± 1.55e+03
CPSO-H_6	10	1.29e+02 ± 7.08e+01	2.40e+02 ± 1.04e+02
	15	1.70e+02 ± 7.33e+01	7.06e+02 ± 3.24e+02
	20	3.46e+02 ± 1.22e+02	1.03e+03 ± 5.24e+02
GA	100	1.07e+06 ± 2.09e+05	
CCGA	100	6.53e+03 ± 2.38e+03	

Table 5.26: Rotated Quadric function: mean value after 2×10^5 function evaluations

Algorithm	s	**GCPSO**	**PSO**
Standard	10	8.08e+00 ± 8.20e−01	7.54e+00 ± 5.82e−01
	15	4.61e+00 ± 6.85e−01	5.09e+00 ± 5.11e−01
	20	3.42e+00 ± 5.17e−01	3.42e+00 ± 3.74e−01
CPSO-S	10	1.92e+01 ± 9.86e−02	1.73e+01 ± 1.45e+00
	15	1.85e+01 ± 8.18e−01	1.81e+01 ± 1.09e+00
	20	1.87e+01 ± 7.58e−01	1.85e+01 ± 7.76e−01
CPSO-H	10	1.53e+01 ± 1.05e+00	1.43e+01 ± 1.57e+00
	15	1.56e+01 ± 1.48e+00	1.43e+01 ± 1.48e+00
	20	1.61e+01 ± 1.20e+00	1.60e+01 ± 1.42e+00
CPSO-S_6	10	1.95e+00 ± 1.55e+00	7.98e−01 ± 1.06e+00
	15	1.91e+00 ± 1.60e+00	1.14e+00 ± 1.26e+00
	20	4.17e−01 ± 7.40e−01	1.54e+00 ± 1.46e+00
CPSO-H_6	10	8.70e−01 ± 9.95e−01	8.23e−01 ± 1.04e+00
	15	1.20e+00 ± 1.27e+00	8.12e−01 ± 1.05e+00
	20	1.11e+00 ± 1.23e+00	8.51e−01 ± 1.83e+00
GA	100	1.27e+01 ± 1.55e+00	
CCGA	100	1.57e+01 ± 1.87e+00	

Table 5.27: Rotated Ackley's function: mean value after 2×10^5 function evaluations

Algorithm	s	**GCPSO**	**PSO**
Standard	10	9.68e+01 ± 7.95e+00	9.76e+01 ± 5.90e+00
	15	8.89e+01 ± 6.24e+00	8.48e+01 ± 5.42e+00
	20	8.36e+01 ± 5.44e+00	7.87e+01 ± 6.79e+00
CPSO-S	10	8.59e+01 ± 7.49e+00	7.55e+01 ± 7.53e+00
	15	8.56e+01 ± 8.73e+00	8.15e+01 ± 6.26e+00
	20	8.02e+01 ± 6.47e+00	7.89e+01 ± 6.41e+00
CPSO-H	10	8.16e+01 ± 7.25e+00	7.91e+01 ± 6.97e+00
	15	8.49e+01 ± 6.77e+00	8.21e+01 ± 6.49e+00
	20	8.47e+01 ± 8.18e+00	8.12e+01 ± 5.92e+00
CPSO-S_6	10	4.84e+01 ± 4.42e+00	5.41e+01 ± 5.18e+00
	15	5.06e+01 ± 3.79e+00	4.66e+01 ± 3.84e+00
	20	4.91e+01 ± 4.71e+00	5.04e+01 ± 5.50e+00
CPSO-H_6	10	6.20e+01 ± 5.83e+00	6.16e+01 ± 5.08e+00
	15	5.62e+01 ± 4.40e+00	5.94e+01 ± 5.04e+00
	20	5.86e+01 ± 5.69e+00	5.41e+01 ± 4.63e+00
GA	100	1.37e+02 ± 1.78e+01	
CCGA	100	6.93e+01 ± 1.02e+01	

Table 5.28: Rotated Rastrigin's function: mean value after 2×10^5 function evaluations

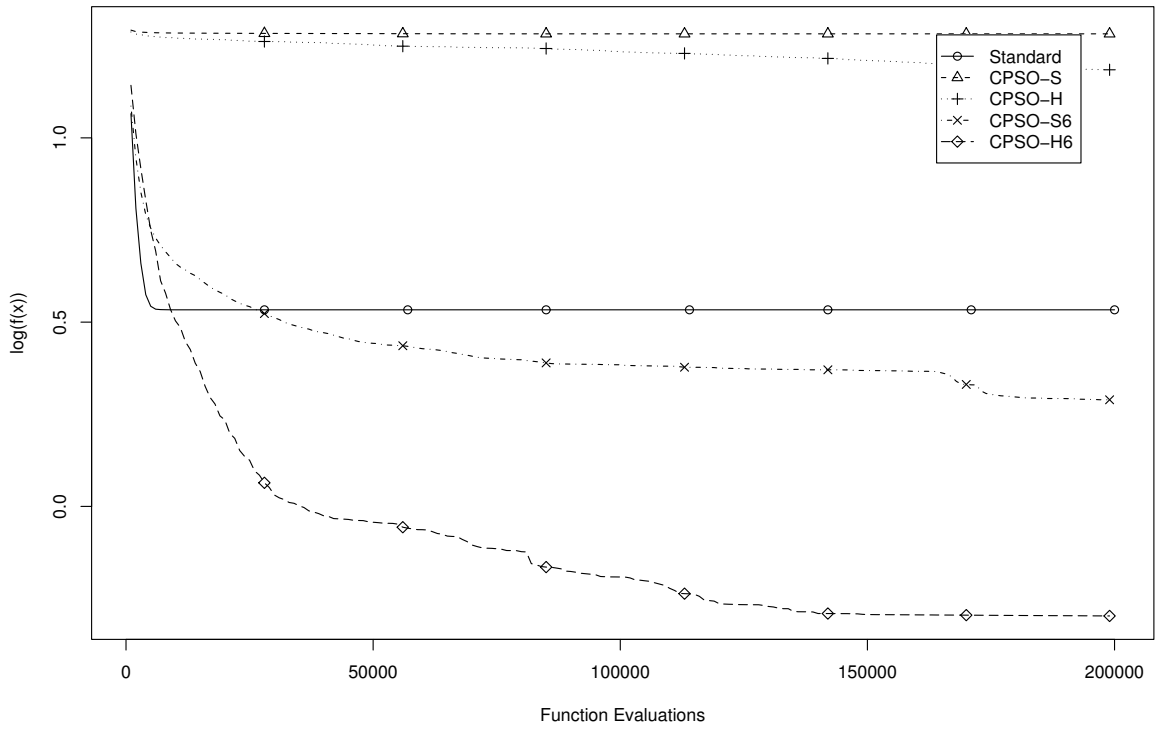


Figure 5.7: The rotated Ackley Function error profile, obtained using the various CPSO algorithms.

Table 5.26 shows an even more dramatic degradation in performance for the Quadric function. The best-performing algorithm remains to be the standard GCPSO, although its performance was significantly worse (by about 160 orders of magnitude) than what it achieved on the unrotated version of this function. The relative ranking of the different algorithms on this function remained roughly the same as observed on the unrotated function.

Overall, the CPSO algorithms exhibited a severe degradation in performance, compared to their previous performance on the unrotated versions of the unimodal functions. On the other hand, the standard algorithms experienced a similar decrease in performance — but they still retained the lead over the CPSO algorithms. The GCPSO versions of all the algorithms showed a slightly smaller performance decrease, especially the standard GCPSO. This phenomenon is caused by the stronger local convergence

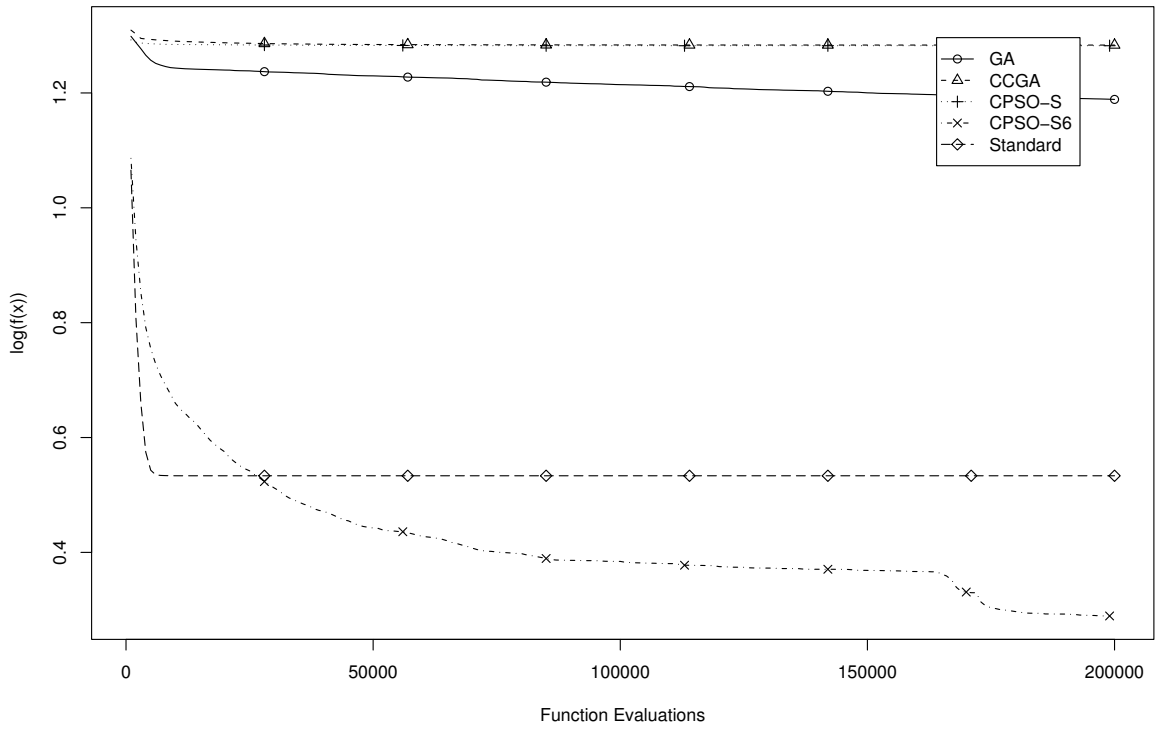


Figure 5.8: The rotated Ackley Function error profile, obtained using the various PSO and GA algorithms.

ability of the GCPSO.

Repeating the experiment on the non-unimodal functions produced some interesting results. Table 5.27, obtained by minimising the rotated Ackley function, shows that the CPSO-S_6 and CPSO-H_6 algorithms took over the lead from the CPSO-S and CPSO-H algorithms. The performance of the standard GCPSO and PSO algorithms was not significantly affected by the rotation, unlike the case was with the unimodal functions.

Table 5.28 presents the results of applying the different algorithms to the rotated version of Rastrigin's function. Clearly the performance of all the CPSO algorithms degraded significantly. The CPSO-S_6 remained the algorithm with the best performance on this function. The significant degradation in the performance of the CPSO-S and CPSO-H algorithms was a direct consequence of the high degree of inter-variable correlation caused by the rotation of the coordinate axes. Since synchronisation between

the different swarms in the CPSO-S and CPSO-H algorithms is limited, they lose the advantages offered by the smaller search spaces assigned to each swarm.

Figures 5.7 and 5.8 show how the different algorithms were affected by the coordinate rotation. Figure 5.7 clearly shows how the CPSO-S algorithm becomes trapped during the earlier iterations. The CPSO-H algorithm is able to improve its best solution somewhat, owing to the help of the standard PSO component integrated into it. Both the CPSO-S_6 and CPSO-H_6 algorithms were able to steadily improve their solutions.

Compared to the two GA-based algorithms, the standard GCPSO and CPSO-S_6 algorithms were clearly superior (see Figure 5.8). Note how the CCGA algorithm and the CPSO-S algorithms were trapped at approximately the same function value. The standard GA does not appear to have stagnated, showing a slow but steady improvement throughout the simulation run.

Based on the performance of the algorithms on both the rotated and unrotated functions, it appears as if the CPSO-S_6 and CPSO-H_6 algorithms are good alternatives to the standard GCPSO or PSO. These two CPSO algorithms suffered less degradation on the rotated functions than the CPSO-S and CPSO-H algorithms did, and they performed significantly better than the standard GCPSO and PSO algorithms on the non-unimodal functions, especially when the variables of the function were correlated.

5.5.4 Computational Complexity

The results presented so far in this section only considered the mean function value at the end of a fixed number of function evaluations. A different type of experiment can be performed to measure the convergence speed (complexity) and robustness of an algorithm, *i.e.* how often is it able to reach a specific performance level.

Table 5.29 lists the threshold values that were used for the various functions tested. These values correspond to the threshold values used by Eberhart *et al.* in [39], except the threshold for the Quadric function, which was selected to correspond to the threshold of the Spherical function used by Eberhart *et al.* The algorithm terminated as soon as the objective function value dropped below the threshold value specified in the table. Each algorithm was allowed a maximum of 2×10^5 function evaluations. The average number of function evaluations required by each algorithm was recorded, excluding the runs that

Function	Threshold
Rosenbrock	100
Ackley	5.00
Rastrigin	100
Quadric	0.01

Table 5.29: Parameters used during experiments

were not able to reach the threshold within 2×10^5 evaluations. Tables 5.30–5.33 report these results, where the N_s column indicates the number of successful runs (out of 50), and the F column lists the average number of function evaluations the algorithm needed to reach the threshold on each successful run.

Table 5.30 shows that none of the algorithms had any trouble reaching the threshold on either the normal or the rotated version of Rosenbrock's function, with the CPSO algorithms performing better than the standard PSO algorithms. In fact, these results show that Rosenbrock's function is relatively insensitive to rotation.

The results obtained by minimising the Quadric function, presented in Table 5.31, show a different trend. Although none of the algorithms (except the standard PSO with only 10 particles) had any difficulty in reaching the threshold on the unrotated version of the Quadric function, almost all of them failed on the rotated version. The obvious explanation for this phenomenon is that the threshold value was too small for the algorithms to reliably reach it on the rotated function. The standard GCPSO algorithm was the top performer on the rotated function, an indication of the strong local convergence abilities of the GCPSO extension.

Ackley's function produced some interesting results, as can be seen in Table 5.32. Note how the standard GCPSO and PSO algorithms had difficulty reaching the threshold even on the unrotated function; in contrast, all the CPSO algorithms could reliably reach it. When the function was rotated, though, the CPSO-S and CPSO-H algorithms failed completely; the standard GCPSO and PSO algorithms mostly maintained the same level of performance. One explanation for the poor performance of the CPSO-S and CPSO-H algorithms is that the rotation caused the different swarms to become unsynchronised, since the sinusoidal bumps no longer appeared in a regular grid with

		Unrotated				**Rotated**			
		GCPSO		**PSO**		**GCPSO**		**PSO**	
Algorithm	s	N_s	F	N_s	F	N_s	F	N_s	F
Standard	10	50	548	50	609	50	591	50	661
	15	50	751	50	820	50	776	50	790
	20	50	943	50	861	50	973	50	855
CPSO-S	10	50	316	50	320	50	408	50	420
	15	50	427	50	424	50	552	50	532
	20	50	558	50	562	50	670	50	672
CPSO-H	10	50	317	50	332	50	408	50	411
	15	50	424	50	426	50	558	50	525
	20	50	562	50	556	50	686	50	653
CPSO-S_6	10	50	526	50	436	50	524	50	516
	15	50	489	50	453	50	622	50	581
	20	50	506	50	521	50	746	50	660
CPSO-H_6	10	50	691	50	511	50	692	50	609
	15	50	659	50	661	50	765	50	723
	20	50	688	50	685	50	965	50	910

Table 5.30: Rosenbrock's function: Computational complexity

		Unrotated				Rotated			
		GCPSO		PSO		GCPSO		PSO	
Algorithm	s	N_s	F	N_s	F	N_s	F	N_s	F
Standard	10	50	9284	38	34838	12	167765	0	N/A
	15	50	9599	50	16735	12	162369	1	26161
	20	50	11347	50	14574	3	154956	2	175788
CPSO-S	10	50	65642	50	70215	0	N/A	0	N/A
	15	50	71594	50	77265	0	N/A	0	N/A
	20	50	77345	50	83168	0	N/A	0	N/A
CPSO-H	10	50	39827	50	40056	1	90211	0	N/A
	15	50	48691	50	53341	0	N/A	0	N/A
	20	50	55240	50	61430	0	N/A	0	N/A
CPSO-S_6	10	50	47540	50	77818	1	59911	0	N/A
	15	50	58125	50	101565	0	N/A	0	N/A
	20	50	74698	50	115687	1	91848	0	N/A
CPSO-H_6	10	50	15478	50	21693	0	N/A	0	N/A
	15	50	22059	50	31068	6	123157	0	N/A
	20	50	38121	50	44086	0	N/A	0	N/A

Table 5.31: Quadric function: Computational complexity

		Unrotated				**Rotated**			
		GCPSO		**PSO**		**GCPSO**		**PSO**	
Algorithm	s	N_s	F	N_s	F	N_s	F	N_s	F
Standard	10	12	1586	11	2099	11	1570	6	1988
	15	35	2018	32	3019	25	2040	32	3385
	20	46	2480	37	2986	40	2628	41	3200
CPSO-S	10	50	967	50	935	0	N/A	5	6240
	15	50	1078	50	1053	3	2468	2	11644
	20	50	1211	50	1227	2	4332	2	43314
CPSO-H	10	50	1089	50	1068	2	1976	4	24420
	15	50	1150	50	1154	1	2212	2	5836
	20	50	1246	50	1245	2	2407	2	2401
CPSO-S_6	10	50	2734	50	3264	43	12760	50	6670
	15	50	3508	50	4136	47	3859	47	4533
	20	50	4054	50	4994	47	5187	46	5686
CPSO-H_6	10	50	2540	50	3170	45	3199	45	3758
	15	50	3831	50	3706	44	4945	43	4568
	20	50	3936	50	4492	47	5028	41	5692

Table 5.32: Ackley function: Computational complexity

		Unrotated				**Rotated**			
		GCPSO		**PSO**		**GCPSO**		**PSO**	
Algorithm	s	N_s	F	N_s	F	N_s	F	N_s	F
Standard	10	36	1636	45	2112	29	2046	41	2403
	15	45	1985	43	2525	37	2432	39	2912
	20	45	2326	49	3341	39	3348	41	3142
CPSO-S	10	50	380	50	375	39	4815	40	3516
	15	50	430	50	436	39	5921	37	5187
	20	50	539	50	546	38	7428	41	4817
CPSO-H	10	50	389	50	388	39	3785	40	4484
	15	50	430	50	430	35	4186	39	5366
	20	50	542	50	545	37	4261	37	5658
CPSO-S_6	10	50	2060	50	2226	49	6458	48	7562
	15	50	2439	50	2750	49	8275	50	7517
	20	50	2846	50	3029	50	9782	50	9874
CPSO-H_6	10	50	2568	50	1966	47	21525	46	4613
	15	50	2594	50	2843	47	10467	44	9566
	20	50	2812	50	3456	48	4603	45	5639

Table 5.33: Rastrigin function: Computational complexity

respect to the coordinate axes (and thus the subspaces searched by the CPSO-S and CPSO-H algorithms). The CPSO-S_6 and CPSO-H_6 algorithms were able to reliably reach the threshold despite the rotation of the coordinate axes, without a significant increase in the required number of function evaluations in most cases.

Table 5.33 presents the results of a similar experiment conducted using Rastrigin's function. The general trend was the same as that observed on Ackley's function, but less dramatic. The CPSO-S_6 and CPSO-H_6 algorithms still produced the best results, both in terms of speed and robustness.

Overall, the standard GCPSO, CPSO-S_6 and CPSO-H_6 algorithms were the most robust algorithms on the set of test functions considered in this section.

5.6 Conclusion

Section 5.2 studied the influence of different parameter settings on the performance of the PSO and GCPSO algorithms. The results indicate that there is a trade-off between parameter settings that perform well on unimodal functions, and those that perform well on multi-modal functions. This is related to the amount of diversity in the swarm: high diversity is desirable when dealing with multi-modal functions, but not equally important when minimising unimodal functions. If a parameter configuration leads to rapid convergence on unimodal functions, it may not be able to maintain sufficient diversity for optimising multi-modal functions effectively. It was found that acceptable performance on both unimodal and multi-modal functions can be achieved by using an inertia weight of 0.7. Choosing different values for the acceleration coefficients so that $c_1 = c_2 = d$, where $d \in [1.5, 1.8]$, results in behavior ranging from rapid convergence on unimodal functions, to more robust behaviour on multi-modal functions.

Section 5.3 showed that the GCPSO algorithm has significantly better convergence properties on unimodal functions, compared to the original PSO algorithm. The differences between the GCPSO and PSO were especially pronounced when small swarm sizes were considered, implying that the original PSO algorithm is prone to premature convergence unless it has a sufficiently large swarm size. Both algorithms performed equally well on multi-modal functions.

Several techniques for extending the PSO (or GCPSO) to become a global search algorithm, with guaranteed convergence on the global minimiser, were investigated in Section 5.4. The results showed that the RPSO algorithm, which introduces randomised particles to the swarm, offers no discernible advantage over the standard PSO algorithm. In contrast, the MPSO algorithm, which re-initialises the swarm when it perceives that too little progress is being made, performed significantly better than the standard PSO algorithm. These results provide experimental support of the proven global convergence property of the MPSO algorithm.

Section 5.5 studied the performance of the various cooperative PSO algorithms. Initial results indicated that the CPSO-S and CPSO-H algorithms have superior performance on multi-modal functions with low inter-variable correlation, compared to the standard PSO algorithms. The CPSO-S_6 and CPSO-H_6 also had better performance than the standard PSO algorithms, but somewhat worse performance than the CPSO-S and CPSO-H algorithms.

The experiments were repeated using the same functions, but rotating them through arbitrary angles to increase the degree of inter-variable correlation. As could be expected, the CPSO-S and CPSO-H algorithms suffered a severe degradation in performance, compared to their performance on the unrotated functions. On the rotated functions they performed significantly worse than the standard PSO algorithms. In contrast, the CPSO-S_6 and CPSO-H_6 algorithms did not perform much worse on the rotated functions, so that they still produced better results than the standard PSO algorithms.

The robustness of the various CPSO algorithms was also investigated, with similar results: The CPSO-S_6 and CPSO-H_6 algorithms were able to consistently perform better than the standard PSO on multi-modal functions, regardless of the degree of inter-variable correlation.

Chapter 6

Neural Network Training

This chapter presents an application of the PSO-based optimisation techniques presented in Chapters 3 and 4. These algorithms are applied to the problem of training feedforward Neural Networks to solve classification problems, as well as training the networks to approximate certain example functions. The experiments are performed using both summation unit networks and product unit networks. The product unit networks experiments are included because these networks are typically more difficult to train than the summation unit networks.

6.1 Multi-layer Feedforward Neural Networks

Neural Networks (NNs), or more correctly, Artificial Neural Networks (ANNs), provide a general method for learning arbitrary mappings between two data sets. These data sets can be real-valued, discrete or vector valued, may contain incorrect examples or they may be noisy — the Neural Network is still able to learn the mapping. Past applications include, amongst others, the recognition of handwritten characters [77], learning to recognise spoken words [75] and recognising images of faces [25].

The term Neural Network describes a large family of general mapping techniques, and it is beyond the scope of this thesis to discuss the various details regarding the different types of neural networks in existence. Books by Mitchell [86] and Bishop [11] are recommended as starting points for further investigation.

The aspect of Neural Networks relevant to this thesis is the process of training the

network. A typical Neural Network contains a number of adjustable parameters called *weights*. In particular, *supervised learning* involves finding a set of weights that minimises the mapping error. The data set used to train the network contains a set of input vectors (called input patterns) and their corresponding output values. The mapping error is then the difference between the output value specified in the data set, and the output of the network when the corresponding input pattern is presented to it.

For most real-world problems it is not possible to obtain the complete set of input patterns and their respective output values, so that the data set used to train the network is only a sample taken from the true population. The objective of training the network is to minimise the mapping error of the network in real-world scenarios, which implies that the performance of the network on input patterns it has never seen before is more important than the error of the network on the patterns in the training data set. If a second sample of input patterns (and their respective output values) is available it is possible to obtain an approximation of the network's error with respect to unseen data. This second data set is called the test set; the mapping error of the network measured over this set is called the test set error, or more commonly, the generalisation error.

The number of weights in the network is one of the factors that determines its learning ability. Assuming a fixed type of network, *e.g.* a two-layer feed-forward summation unit network with sigmoidal activation functions (see Section 6.1.1 for a more detailed description of this type of network), the number of hidden units determines the number of weights in the network, since the number of input and output units are determined by the problem under consideration. This implies that a network with more hidden units will potentially be able to learn more complex mappings — if a suitable training algorithm is available. The difficulty with Neural Network training is that the data set may be such that a network trained to fit the training set may perform poorly on the test set, thus also performing poorly in the real-world application from which the data set was sampled. This phenomenon is called *overfitting*, which typically occurs when the network has too many adjustable weights. Several ways exist for preventing the network from overfitting the data in the training set. For example, if the true population (from which the training set was sampled) has a given level of complexity, then the network should preferably have the correct number of hidden units to match the complexity of

the true population. If the training set contains some noise, then the a network with too many hidden units will be able to learn the noise as well, reducing the generalisation ability of the network. By limiting the number of hidden units in the network so that the network is unable to learn the noise, generalisation performance will improve. This is the approach taken by the Optimal Brain Damage algorithm [78], which attempts to discard extraneous hidden units until only the minimum required number of hidden units remain. The network may also have extraneous input units that can have a detrimental effect on the learning ability of the network. A method for pruning irrelevant hidden *and* input units has been developed by Engelbrecht [40].

Other approaches include *regularisation* [11] (chapter 9), where the assumption is made that smoother network mappings have a smaller probability of overfitting the data. For summation unit networks this implies that the network weights (excluding the bias weights) should be small — this form of regularisation is often called *weight decay*. This topic will be revisited in the context of summation units in Section 6.1.1. Alternatively, the network can start with a very simple architecture, and add more hidden units until the error becomes sufficiently small. This approach is called *growing*; an example of an architecture that uses this approach is the *cascade-correlation* network [43].

The aim of this chapter is to show that the Particle Swarm Optimiser, and the various derived algorithms presented in this thesis, are suitable for training Neural Networks. The aim is not to find the optimal architecture for each of the test problems, nor is the aim to design and train the networks until the best generalisation errors possible are reached, but simply to show that the PSO-based algorithms can train the networks as well as other existing techniques are able to. Two types of Neural Networks, namely summation unit networks and product unit networks, are trained on a variety of classification and function approximation problems using various training algorithms.

6.1.1 Summation-unit Networks

A depiction of a two-layer summation unit network (also called a Multi-Layer Perceptron, or MLP) is presented in Figure 6.1. This network is called a two-layer network because of the two layers of weights found in the network: those running from the input units to the hidden units, and those running from the hidden units to the output units.

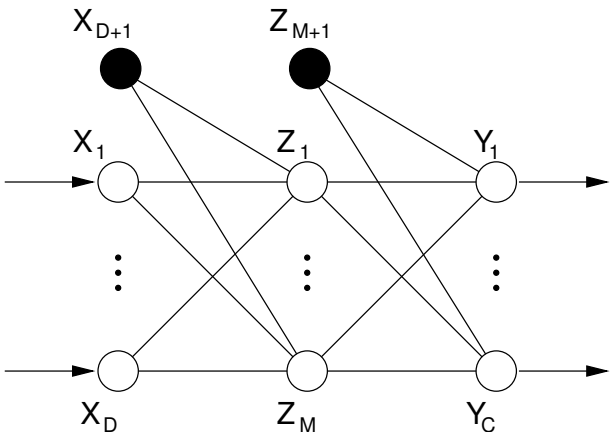


Figure 6.1: Summation unit network architecture

The network consists of D input units (plus a bias unit), M hidden units (plus a bias unit) and C output units.[1] Linear functions serve as the activation functions in the output layer, obviating the need for rescaling the output data. The hidden units make use of standard sigmoidal activation functions.

A forward propagation through the network is defined in equation (6.1):

$$y_k(\mathbf{x}) = \sum_{j=1}^{M+1} w_{kj}\ g\left(\sum_{i=1}^{D+1} w_{ji}\ x_i\right) \tag{6.1}$$

where

$$g(a) = \frac{1}{1+\exp(-a)} \tag{6.2}$$

The following convention is used for labeling the weights: w_{kj} is a weight in the second layer of weights, between the output unit k and the hidden layer unit j, and w_{ji} is a first-layer weight between hidden unit j and input unit i. Note that $1 \leq i \leq D+1$, $1 \leq j \leq M+1$ and $1 \leq k \leq C$.

The mapping error of the network is computed using the Mean Squared Error (MSE)

[1] Please note the new meanings associated with some symbols, *e.g.* $\mathbf{x}$. These new meanings are only applicable to this section (Section 6.1), and should not be confused with the regular meaning of the symbols as used throughout the rest of the thesis.

function,

$$E = \frac{1}{2n} \sum_{p=1}^{n} \sum_{k=0}^{C} \left(y_{k,p} \left(\mathbf{x}_p \right) - t_{k,p} \right)^2 . \tag{6.3}$$

The symbol $t_{k,p}$ refers to the *target* value for output unit k, belonging to pattern p. This implies that the network makes use of a supervised training algorithm. For a classification problem, the value of t_k must be either 0 or 1, since a 1-of-C coding scheme is used. This means that the target vector $\mathbf{t}$ is of the form $\mathbf{t} = (0, 0, \ldots, 1, 0, \ldots, 0)$.

Note that the sigmoid function, $g(a)$, saturates when $|a| > 10$, so that the output of the activation function becomes $g(a) \approx 0$ or $g(a) \approx 1$, almost like a step function. When the network weights in the input-to-hidden layer are small, so that $|a| < 2$, the response of the sigmoid function is approximately linear, resulting in a relatively smooth network mapping.

Summation units have the desirable property that they can approximate any continuous function to arbitrary accuracy if a large enough number of hidden units are used [66, 13].

6.1.2 Product-unit Networks

The product unit network was first introduced by Durbin and Rumelhart [33], and can be used in more or less any situation where the better known summation unit back-propagation networks have been used.

A product unit network with D inputs, M hidden units and C output units is shown in Figure 6.2, assuming that product units are used only in the hidden layer, followed by summation units in the output layer, with linear activation functions throughout. The value of an output unit y_k for pattern p is calculated using

$$y_k = \sum_{j=0}^{M+1} w_{kj} \prod_{i=1}^{D} x_{i,p}^{w_{ji}} \tag{6.4}$$

where w_{kj} is a weight from output unit y_k to hidden unit z_j, w_{ji} is a weight from hidden unit z_j to input unit x_i and $z_{M+1} \equiv 1.0$, the hidden-to-output layer bias unit. The mapping error of the network is computed in the same way as used for the summation unit networks, using equation (6.3).

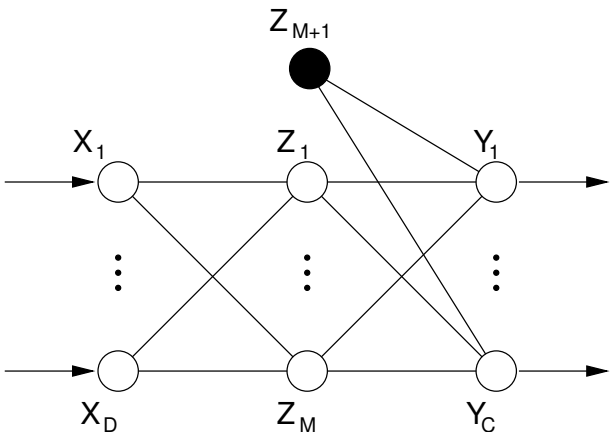


Figure 6.2: Product unit network architecture

Note that quadratic functions of the form $ax^2 + c$ can be represented by a network with only one input unit and one hidden unit, thus a 1-1-1 product unit network can be used, compared to a summation unit network requiring at least 2 hidden units [41]. This is an indication of the increased information storage capacity of the product unit neural network [33].

Unfortunately, the usual optimisation algorithms like gradient descent cannot train the product unit networks with the same efficacy that they exhibit on summation unit networks, due to the more turbulent error surface created by the product term in (6.4). Global-like optimisation algorithms like the Particle Swarm Optimiser, the Leapfrog algorithm and Genetic Algorithms are better suited to the task of training PUNNs [41, 64].

Product unit networks also have smoother network mappings when the weights are smaller. This is because an input attribute x_i is raised to the power w_{ji}, so that a larger weight value increases the sensitivity of the network to that specific attribute.

6.2 Methodology

This section briefly outlines the methods used to conduct the experiments presented in this chapter.

6.2.1 Measurement of Progress

For classification problems, four distinct measures were recorded during each experiment. These measures were:

ε_T: The training set classification error, computed as the number of misclassified patterns in the training set, divided by the total number of patterns in the training set, expressed as a percentage.

ε_G: The test set classification error (also called the generalisation error), computed as the number of misclassified patterns in the test set, divided by the total number of patterns in the test set, expressed as a percentage. Note that none of the patterns in the test set were accessible to the training algorithms during the training process.

$\mathbf{MSE}_T$: The training set mean squared error.

$\mathbf{MSE}_G$: The test set mean squared error, also called the generalisation mean squared error.

Only the last two measures can be computed for function approximation problems, since there are no output classes in a function approximation problem.

In Section 6.1 a brief discussion of overfitting was presented. Techniques like *early stopping*, *regularisation* and *architecture selection* can be used to help alleviate overfitting, but these techniques were developed using gradient-based training algorithms — other algorithms may require modifications to these techniques.

Lawrence and Giles have shown that the degree of overfitting is dependent on the type of algorithm that was used to train the network [76]. They used a function approximation problem to illustrate that the Conjugate Gradient (CG) algorithm was much more prone to overfitting, even when using smaller networks with only a few hidden units. The training sets they used in their experiments were generated using a sinusoidal function, perturbed with some additional noise. The underlying function was simple enough so that 4 hidden units were sufficient to learn it. They found that the GD algorithm could successfully learn the function when using only 4 hidden units, and that the generalisation ability of the trained network was very good. Increasing the number of hidden units to

100 did not seriously affect the generalisation ability of the network trained using the GD algorithm.

In contrast, they found that the Conjugate Gradient (CG) algorithm would overfit the network significantly, especially when a large number of hidden units was used. This implies that different training algorithms do not respond similarly to changes in the network architecture, implying that the optimal architecture for a given problem is dependent on the training algorithm. This observation further suggests that some mechanism must be implemented to prevent overfitting in addition to selecting the optimal architecture to obtain an algorithm that offers good generalisation performance on a large number of problems.

Since a study of overfitting falls outside the scope of this thesis, the experiments conducted in this chapter did not make use of any of these techniques, *i.e.* no attempt was made to prevent overfitting of the data, other than selecting a simple network architecture. A study of mechanisms that can be used to prevent overfitting when training a network using the PSO algorithm is left as a topic for future research.

To compensate for the lack of a mechanism to prevent overfitting, only the training errors are analysed in the results presented in this chapter. The corresponding generalisation error is provided along with each training error to give a rough indication of whether overfitting occurred. Using only the training errors to compare the performance may not make much sense in a real-world application, but it is the only fair method of comparing the algorithms in the absence of a mechanism that prevents overfitting. Therefore, instead of viewing the results below as an indication of the suitability of using PSO-based algorithms to train Neural Networks, the training process can be seen as just another function minimisation problem, where the training error is interpreted as the fitness of a potential solution.

How should one interpret the training error results? What does it mean if algorithm A produced a lower training error than algorithm B when both algorithms were allowed to run for the same duration of time? Two things can be inferred from this observation:

1. Algorithm A will require less time (or processing power) than algorithm B if the algorithms were to be stopped once a fixed training error value is reached;

2. Algorithm A will be able to train networks that are too complex for algorithm B

to train successfully in the same duration of time.

Even though the optimal number of hidden units for each problem may depend on the training algorithm, the Optimal Brain Damage (OBD) algorithm was used to determine a suitable number of hidden units for the summation unit networks used in this chapter. It was assumed that the same number of hidden units could be used in the product unit networks, since they have a learning capacity comparable to that of summation units. Determining the optimal number of hidden units for a product unit network for each problem studied below falls outside the scope of this thesis, since only the training errors are compared. A method for determining the optimal number of hidden units for a product unit network, using a standard PSO, is currently under development [64].

6.2.2 Normality Assumption

Many statistical tests, including the calculation of the mean of a set of observations, assume that the data[2] has a normal distribution. The normality of the results obtained in the experiments was tested, using the Shapiro-Wilk test [112], at a significance level of 0.01, so that a set of results with a p-value of less than 0.01 is flagged as having a non-normal distribution. For each experiment the algorithms with non-normally distributed results were identified.

Regardless of the fact that statistics like the mean of a set of results are possibly inaccurate when the data has a non-normal distribution, these values are still reported. These statistics should be regarded with care, though.

The ε_T and ε_G values are discrete values in the range $[0, 100]$, since they are computed as the fraction of misclassified patterns using

$$\varepsilon_T = 100 \times \frac{\#misclassified\ training\ patterns}{\#training\ patterns}$$

As a result, the resolution of the values of ε_T and ε_G is limited by the number of training and test patterns, respectively. This implies that if a set of results all have the same discrete value (*e.g.* all ε_T values are zero), then the data will have a non-normal

[2]The word *data* will be treated as a singular mass noun, similarly to *information*. According to [103], this practice is acceptable in modern English.

distribution. Clearly the mean is a valid statistic in this case, since the values are all equal.

Computing the mean of a discrete variable is a questionable practice. If it is assumed that the values ε_T and ε_G are actually continuous variables, it is possible to compute their mean, although this value may not be realisable in the experiment itself. If the size of the training set approaches infinity, then the classification errors will become better approximations of continuous values, so that the mean may become a realisable value. Therefore, even though the data sets used in the experiments have a finite number of instances, the mean is still computed as an estimator of the true population mean value.

All one-sided t-tests are performed on a 5% confidence level. If two values μ_1 and μ_2 are compared, where $\mu_1 < \mu_2$, then the hypotheses are as follows:

H_0: $\mu_1 < \mu_2$

H_1: $\mu_1 \geq \mu_2$

If the p-value of the t-test is then greater than 0.05, the null hypothesis is rejected, implying that values μ_1 and μ_2 do not differ significantly. The word "significantly" is used below to refer to this property in the strict sense.

6.2.3 Parameter Selection and Test Procedure

The behaviour of the CPSO-S_K algorithm varies with different K values, so each network configuration was trained with a few sample values for K. Previous experience with the algorithm has shown that there exists certain critical values for K which are of interest [138]. The simulations were performed with a bias towards K values in the ranges that were deemed interesting. Once a specific K value has been identified as producing the best results on a particular problem, the assumption was made that this value will result in the same behaviour when used in the CPSO-H_K algorithm.

In all the classification and function approximation problems studied in this chapter the data sets were partitioned into training sets and test sets. If a data set consisted of p patterns, then the training set was constructed to contain $2p/3$ patterns, with the remaining $1p/3$ patterns assigned to the test set. Before each simulation elements from

the test and training sets were randomly exchanged, so that no two training runs in the same experiment used exactly the same training and test sets. The same random seed and pseudo-random number generation algorithm was used for each training algorithm, so that all training algorithms used the exact same collection of training and test sets. Since there was no deterministic relationship between any two test sets (in the same experiment) generated this way, this method does not introduce any systematic bias that may affect the results.

Each training algorithm was used to train the network on 50 different training sets generated using the approach described in the previous paragraph. The results reported below are the mean values computed over all 50 runs.

The various algorithms used in the experiments below were:

GD: A Gradient Descent training algorithm, described in Appendix E.

SCG: A Scaled Conjugate Gradient Descent algorithm, described in Appendix E.

GA: A real-valued Genetic Algorithm, with a mutation rate of 0.95 and a crossover rate of 0.01, and a population size of 100. A Two-parent arithmetic crossover operator was used. Mutation was implemented using a uniform real-valued mutation operator with an interval width of 2. A Fitness-proportionate selection operator was implemented. These values were found experimentally to produce acceptable results.

CCGA: A CCGA-1 Genetic Algorithm (due to Potter [106, 105]) implementation, with each weight in the network assigned to its own subpopulation. This algorithm used the same parameter settings as those used in the standard GA.

GCPSO: A standard GCPSO algorithm (see Section 3.2), with an inertia weight $w = 0.72$ and acceleration coefficients $c_1 = c_2 = 1.49$. A swarm size of 10 particles was selected. These values have been found to produce acceptable results [138].

MPSO: A multi-start MPSO algorithm, as described in Section 3.4. The *maximum swarm radius* stopping criterion, re-starting the swarm when $r_{norm} < 10^{-3}$, was used. This setting was found to result in acceptable sensitivity for this convergence

detection mechanism. The other PSO parameters were the same as those used in the GCPSO.

CPSO-S$_K$: A CPSO-S$_K$ algorithm, as described in Section 4.2. The same parameter settings as those of the GCPSO were used, except that a linearly decreasing inertia weight was used instead of a fixed inertia weight. Previous experience has shown that this algorithm benefits from the increased diversity offered by a linearly decreasing inertia weight [138].

CPSO-H$_K$: A CPSO-H$_K$ algorithm, as described in Section 4.3. This algorithm used the same parameter settings as the CPSO-S$_K$ algorithm.

The running times for all the algorithms were calibrated so that they all used the same amount of processor time as the GCPSO needed to train for 4×10^4 forward propagations through the network. The GD and SCG algorithms were both highly efficient implementations, as detailed in Appendix E. The overheads of the GA, CCGA and various PSO-based algorithms were comparable, so that they all performed 4×10^4 forward propagations in approximately the same duration of time.

The Iris, Breast Cancer, Wine, Diabetes and Hepatitis classification problems used below can all be found in the UCI machine learning repository [12]. Specifically, the Breast Cancer problem used the Wisconsin Breast Cancer data, and the Diabetes problem made use of the Pima-Indian diabetes data.

6.3 Network Training Results

This section presents results obtained by training both summation and product unit networks using the various algorithms described in Section 6.2.3.

6.3.1 Iris

The Iris classification problem is the simplest classification problem studied in this chapter. The data set comprises 150 patterns with 4 attributes and 3 output classes. The optimal number hidden units, as determined by the OBD algorithm, results in a 4-4-3 architecture.

Algorithm	ε_T	ε_G
CPSO-S_2	0.71 ± 0.28	5.10 ± 0.93
CPSO-S_6	0.69 ± 0.24	5.30 ± 0.87
CPSO-S_7	0.62 ± 0.24	5.80 ± 0.91
CPSO-S_8	0.82 ± 0.28	5.87 ± 0.91

Table 6.1: Classification error for a summation unit network, applied to the Iris problem, obtained with the CPSO-S_K algorithm using different values for K.

Algorithm	ε_T	ε_G
GD	1.16 ± 0.32	5.10 ± 1.15
SCG	1.56 ± 0.34	4.80 ± 0.90
CCGA	21.53 ± 1.41	28.07 ± 1.75
GA	25.22 ± 1.78	30.17 ± 1.85
GCPSO	1.62 ± 0.38	5.63 ± 1.01
MPSO	1.47 ± 0.38	5.03 ± 0.81
CPSO-S_7	0.62 ± 0.24	5.80 ± 0.91
CPSO-H_7	1.73 ± 0.64	6.63 ± 1.51

Table 6.2: Comparing different training algorithms on the Iris classification problem using a summation unit network.

Summation Unit Network

All algorithms, except the GA and CCGA, produced a non-normal distribution of training and generalisation errors. Because the Iris problem has such a small network architecture, most of the algorithms succeeded in training the network almost to perfection. Most training algorithms were thus able to train the network until only 0, 1, or 2 training patterns were misclassified. The discrete nature of these values is thus responsible for the observed non-normal distributions.

The summation unit network had a total of $5 \times 4 + (4+1) \times 3 = 35$ weights. Table 6.1 shows that the best-performing split factor for the Iris problem was $K = 7$. The results in Table 6.2 were obtained using various training algorithms. Note that the GA and CCGA algorithms are lagging behind the others by a significant distance.

Algorithm	ε_T	ε_G
CPSO-S_2	21.67 ± 6.97	29.70 ± 7.13
CPSO-S_5	19.98 ± 5.99	31.17 ± 6.12
CPSO-S_6	14.16 ± 2.66	25.50 ± 3.33
CPSO-S_7	21.04 ± 7.00	29.90 ± 7.05

Table 6.3: Classification error for a product unit network, applied to the Iris problem, obtained with the CPSO-S_K algorithm using different values for K.

The t-test confirms that the training performance of both the SCG and GCPSO algorithms was significantly worse than that of the GD algorithm. Further, the CPSO-S_7 algorithm was significantly better than the GD algorithm. In contrast, the MPSO and GD algorithms had similar performance up to a 10% confidence level, implying that the transition from GCPSO to MPSO made the MPSO competitive with the GD algorithm on this particular problem.

Overall, the CPSO-S_7 algorithm was the best performer as far as training performance was concerned.

Product Unit Network

All algorithms produced non-normal ε_T distributions when using the product unit network architecture, for the same reasons as the summation unit network: all training runs for a specific algorithm resulted in a discrete ε_T value drawn from a very small set.

The product unit network had a total of $4 \times 4 + (4 + 1) \times 3 = 31$ weights. Note that this may not have been the optimal architecture. Table 6.3 shows that the optimal split factor for the product unit network is smaller than that of the summation unit network, since the optimal value is now $K = 6$.

Table 6.4 shows that none of the algorithms produced satisfactory results on the product unit network. This implies that the network architecture was not optimal; the network would probably perform better if a different number of hidden units was used. Even though the network architecture was sub-optimal, many interesting observations can be made based on the results in Table 6.4. Clearly the gradient-based algorithms perform significantly worse than any of the other algorithms. A detailed discussion of this

Algorithm	ε_T	ε_G
GD	90.44 ± 3.02	91.33 ± 3.01
SCG	90.93 ± 2.84	91.33 ± 2.63
CCGA	25.89 ± 4.07	32.10 ± 3.82
GA	37.29 ± 3.62	42.93 ± 3.78
GCPSO	17.16 ± 3.64	28.27 ± 4.51
MPSO	17.40 ± 3.39	28.23 ± 4.05
CPSO-S_6	14.16 ± 2.66	25.50 ± 3.33
CPSO-H_6	31.84 ± 6.75	40.17 ± 6.35

Table 6.4: Comparing different training algorithms on the Iris classification problem using a product unit network.

phenomenon is presented in Section 6.4; further comments will be deferred until then. The PSO-based algorithms were clearly more efficacious than the rest, with the exception of the CPSO-H_K algorithm. Note that the MPSO algorithm produced effectively the same result as the GCPSO algorithm; the t-test indicates that their results are similar with a p-value of 0.46. Comparing the CPSO-S_6 algorithm to the GCPSO yields a p-value of 0.09, meaning that the CPSO-S_6 algorithm does not perform significantly better than the GCPSO at a confidence level of 5%.

Based on the results of Table 6.4, the CPSO-S_K algorithm was the best performer on the Iris classification problem using a product unit network, although it was not significantly better than the next best algorithm, the GCPSO.

6.3.2 Breast Cancer

The breast cancer classification problem comprises 600 patterns with 9 attributes and one output class. The optimal number of hidden units for the summation unit network, as determined by the OBD algorithm, results in a 9-8-1 architecture.

Algorithm	ε_T	ε_G
CPSO-S_2	1.50 ± 0.17	4.75 ± 0.43
CPSO-S_5	1.26 ± 0.17	5.01 ± 0.58
CPSO-S_6	1.12 ± 0.11	4.93 ± 0.45
CPSO-S_9	1.18 ± 0.14	4.84 ± 0.33
CPSO-S_{14}	1.04 ± 0.13	5.17 ± 0.46
CPSO-S_{15}	1.12 ± 0.15	4.85 ± 0.41
CPSO-S_{16}	1.17 ± 0.13	4.69 ± 0.46

Table 6.5: Classification error for a summation unit network, applied to the Breast cancer problem, obtained with the CPSO-S_K algorithm using different values for K.

Algorithm	ε_T	ε_G
GD	2.29 ± 0.20	4.05 ± 0.35
SCG	2.78 ± 0.19	3.73 ± 0.33
CCGA	4.30 ± 0.18	4.64 ± 0.36
GA	4.18 ± 0.19	4.52 ± 0.37
GCPSO	2.08 ± 0.20	4.12 ± 0.37
MPSO	1.95 ± 0.18	4.09 ± 0.38
CPSO-S_{14}	1.04 ± 0.13	5.17 ± 0.46
CPSO-H_{14}	2.09 ± 0.17	4.06 ± 0.34

Table 6.6: Comparing different training algorithms on the Breast cancer classification problem using a summation unit network.

Summation Unit Network

All the algorithms, with the exception of the CPSO-S_6 algorithm, produced normally distributed ε_T values at a significance level of 1%. This enhances the credibility of statistics like the mean and t-test values computed on this set of results.

The summation unit network had a total of $10 \times 8 + (8 + 1) = 89$ weights. Table 6.5 shows that the optimal split factor for this problem is $K = 14$. Note that the performance of most of the CPSO-S_K algorithms are similar; in particular, note that the performance of the CPSO-S_9 and CPSO-S_{14} algorithms did not differ significantly.

Algorithm	ε_T	ε_G
CPSO-S_2	4.25 ± 0.47	11.17 ± 0.88
CPSO-S_5	3.96 ± 0.35	12.07 ± 0.88
CPSO-S_6	5.59 ± 1.87	12.87 ± 1.93
CPSO-S_9	8.38 ± 3.61	15.64 ± 3.12
CPSO-S_{14}	8.64 ± 2.85	16.56 ± 2.40
CPSO-S_{15}	12.81 ± 5.76	20.39 ± 5.13
CPSO-S_{16}	11.74 ± 4.40	19.45 ± 4.18

Table 6.7: Classification error for a product unit network, applied to the Breast cancer problem, obtained with the CPSO-S_K algorithm using different values for K.

Table 6.6 presents a comparison between the algorithms under consideration applied to the task of training a summation unit network to solve the Breast cancer classification problem. The PSO-based algorithms were able to reduce the training error to a value below that of any value achieved by the other algorithms. Compared to its poor performance on the Iris problem, the GA-based algorithm results are now much closer to those of the other algorithms, although it still differed from its closest competitor at a 5% confidence level.

The GD, GCPSO and MPSO algorithms all performed similarly on a 5% significance level, with the CPSO-S_{14} algorithm performing significantly better than any of these three. This also makes the CPSO-S_{14} algorithm the overall best performer on the summation unit Breast cancer problem.

Product Unit Network

Most algorithms, with the exception of CCGA, GA, CPSO-S_2 and CPSO-S_5, produced non-normal ε_T distributions on this problem.

Table 6.7 shows that the an acceptable split factor for the CPSO-S_K algorithm is $K = 5$. This problem used a network with $9 \times 8 + 9 = 81$ weights in total.

Table 6.8 compares the performance of the different algorithms on the task of training a product unit network to solve the Breast cancer classification problem. Based on the outcome of a t-test, the MPSO algorithm did not perform significantly better than the

Algorithm	ε_T	ε_G
GD	61.85 ± 9.07	63.48 ± 8.85
SCG	47.99 ± 9.47	49.37 ± 9.01
CCGA	5.00 ± 0.47	8.56 ± 0.73
GA	5.04 ± 0.38	7.76 ± 0.62
GCPSO	4.72 ± 0.59	10.87 ± 0.84
MPSO	4.22 ± 0.53	11.07 ± 0.90
CPSO-S_5	3.96 ± 0.35	12.07 ± 0.88
CPSO-H_5	13.55 ± 5.14	18.23 ± 5.08

Table 6.8: Comparing different training algorithms on the Breast cancer classification problem using a product unit network.

GCPSO algorithm. The CCGA performed similarly to the GCPSO algorithm at a 5% confidence level, but it performed significantly worse than the MPSO algorithm. The CPSO-S_5 algorithm performed significantly better than the GCPSO algorithm, but not statistically significantly better than the MPSO algorithm. This implies that the MPSO and CPSO-S_5 algorithms were tied for the first place on this problem.

6.3.3 Wine

The Wine classification problem contains 178 patterns with 13 input attributes and 3 output classes. The OBD algorithm found that the optimal number of hidden units for a summation unit network is 5, so that a 13-5-3 network architecture was employed.

Summation Unit Network

Only the CCGA, GA, SCG, GD and CPSO-S 16 algorithms had normal ε_T distributions at a 1% significance level. The Wine problem had a total of $14 \times 5 + (5 + 1) \times 3 = 88$ network weights. Table 6.9 shows that the CPSO-S_{16} algorithm had the most acceptable performance amongst the different CPSO-S_K algorithms.

Table 6.10 shows that the gradient and PSO algorithms all performed similarly, with only the GA-based algorithms lagging behind. The t-test confirms that none of the

Algorithm	ε_T	ε_G
CPSO-S_2	0.12 ± 0.11	7.70 ± 1.52
CPSO-S_3	0.05 ± 0.06	7.97 ± 1.40
CPSO-S_4	0.08 ± 0.07	8.40 ± 1.06
CPSO-S_5	0.05 ± 0.08	8.43 ± 1.26
CPSO-S_6	0.02 ± 0.03	7.43 ± 1.01
CPSO-S_7	0.02 ± 0.03	7.53 ± 1.15
CPSO-S_8	0.05 ± 0.06	7.77 ± 1.06
CPSO-S_9	0.03 ± 0.05	7.90 ± 1.33
CPSO-S_{16}	0.00 ± 0.00	6.63 ± 0.96
CPSO-S_{17}	0.02 ± 0.03	6.80 ± 1.10

Table 6.9: Classification error for a summation unit network, applied to the Wine problem, obtained with the CPSO-S_K algorithm using different values for K.

Algorithm	ε_T	ε_G
GD	0.00 ± 0.00	4.77 ± 0.71
SCG	0.00 ± 0.00	5.03 ± 0.73
CCGA	14.63 ± 1.71	23.73 ± 2.31
GA	35.98 ± 2.99	44.00 ± 3.29
GCPSO	0.17 ± 0.31	5.30 ± 0.96
MPSO	0.07 ± 0.07	5.80 ± 0.83
CPSO-S_{16}	0.00 ± 0.00	6.63 ± 0.96
CPSO-H_{16}	0.05 ± 0.08	6.53 ± 1.02

Table 6.10: Comparing different training algorithms on the Wine classification problem using a summation unit network.

Algorithm	ε_T	ε_G
CPSO-S_2	62.85 ± 3.61	90.07 ± 2.59
CPSO-S_3	67.08 ± 2.08	93.17 ± 1.55
CPSO-S_4	61.73 ± 2.89	90.23 ± 2.00
CPSO-S_5	66.00 ± 2.54	93.30 ± 1.66
CPSO-S_6	64.58 ± 2.81	92.27 ± 2.02
CPSO-S_7	64.12 ± 2.90	92.60 ± 1.50
CPSO-S_8	66.03 ± 2.74	92.50 ± 1.96
CPSO-S_9	66.03 ± 2.24	93.60 ± 1.23
CPSO-S_{16}	67.14 ± 2.67	93.27 ± 1.67
CPSO-S_{21}	69.90 ± 3.90	93.90 ± 1.83

Table 6.11: Classification error for a product unit network, applied to the Wine problem, obtained with the CPSO-S_K algorithm using different values for K.

algorithms performed significantly better than any other, except for the CCGA and GA algorithms that clearly performed significantly worse.

It appears that this problem was a relatively simple classification problem, based on the observation that most algorithms were able to reduce the training classification error to near zero, while the generalisation error remained acceptable ($\approx 6\%$).

Product Unit Network

The following algorithms did not have normally distributed ε_T values: GA, GD, SCG, CPSO-S_2, CPSO-S_3, CPSO-S_5 CPSO-S_6, CPSO-S_8, and CPSO-S_9

The product unit network had $13 \times 5 + (5+1) \times 3 = 83$ weights in total. Table 6.11 indicates that an acceptable split factor for the CPSO-S_K algorithm is $K = 4$.

Table 6.12 compares the performance of the algorithms under consideration tasked with training a product unit network to solve the Wine classification problem. It is interesting to note that both the GA-based algorithms had better performance than the other algorithms on this task. Amongst the PSO-based algorithms, note that the MPSO and the GCPSO had similar performance at a 5% confidence level. The t-test confirms that the CPSO-S_4 algorithm performed significantly better than the MPSO or GCPSO

Algorithm	ε_T	ε_G
GD	96.53 ± 1.24	97.57 ± 0.92
SCG	90.12 ± 7.68	89.80 ± 7.71
CCGA	35.64 ± 3.51	53.80 ± 3.91
GA	43.88 ± 3.62	57.83 ± 4.00
GCPSO	70.10 ± 3.98	90.97 ± 2.16
MPSO	70.69 ± 4.02	91.67 ± 1.74
CPSO-S_4	61.73 ± 2.89	90.23 ± 2.00
CPSO-H_4	81.86 ± 2.17	94.60 ± 1.02

Table 6.12: Comparing different training algorithms on the Wine classification problem using a product unit network.

algorithms, in contrast with the CPSO-H_4 that performed significantly worse.

The product unit network performed significantly worse than the summation unit network on this relatively simple classification task. This may be because the optimal architecture for a product unit network trained to solve this problem may differ significantly from the optimal architecture for the summation unit network.

6.3.4 Diabetes

The Diabetes classification problem comprises 700 patterns, each consisting of 8 attributes, with two output classes. The OBD algorithm determined that the optimal number of hidden units for a summation network was 16, so that a 8-16-1 network architecture was used.

Summation Unit Network

All algorithms had normally distributed ε_T values for this problem. Based on the results presented in Table 6.13, it appears that the most acceptable split factor for the CPSO-S_K algorithm was $K = 16$. This network had a total of $9 \times 16 + (16 + 1) = 161$ weights.

Table 6.14 presents a comparison of the performance of the algorithms under consideration, applied to the task of training a summation unit network to solve the Dia-

Algorithm	ε_T	ε_G
CPSO-S_2	21.15 ± 0.53	28.69 ± 0.83
CPSO-S_{14}	15.94 ± 0.46	30.99 ± 0.84
CPSO-S_{15}	16.36 ± 0.55	30.39 ± 0.71
CPSO-S_{16}	15.69 ± 0.55	30.65 ± 0.78
CPSO-S_{31}	16.27 ± 0.46	29.87 ± 0.87
CPSO-S_{32}	16.25 ± 0.49	31.11 ± 0.87
CPSO-S_{33}	16.11 ± 0.49	30.66 ± 0.87

Table 6.13: Classification error for a summation unit network, applied to the Diabetes problem, obtained with the CPSO-S_K algorithm using different values for K.

Algorithm	ε_T	ε_G
GD	23.56 ± 0.44	27.68 ± 0.69
SCG	25.73 ± 0.44	27.56 ± 0.72
CCGA	28.49 ± 0.43	30.03 ± 0.73
GA	28.70 ± 0.50	29.88 ± 0.79
GCPSO	22.60 ± 0.58	28.06 ± 0.70
MPSO	21.86 ± 0.49	28.71 ± 0.67
CPSO-S_{16}	15.69 ± 0.55	30.65 ± 0.78
CPSO-H_{16}	22.58 ± 0.57	27.78 ± 0.76

Table 6.14: Comparing different training algorithms on the Diabetes classification problem using a summation unit network.

Algorithm	ε_T	ε_G
CPSO-S_2	25.34 ± 1.03	35.56 ± 0.99
CPSO-S_{31}	46.24 ± 7.32	54.77 ± 5.95
CPSO-S_{32}	41.02 ± 5.20	50.68 ± 3.94
CPSO-S_{33}	39.48 ± 6.18	49.05 ± 5.05
CPSO-S_{14}	26.58 ± 2.95	38.94 ± 2.38
CPSO-S_{15}	29.33 ± 4.21	40.68 ± 3.35
CPSO-S_{16}	26.89 ± 3.24	39.02 ± 2.46

Table 6.15: Classification error for a product unit network, applied to the Diabetes problem, obtained with the CPSO-S_K algorithm using different values for K.

betes classification problem. A t-test confirms that the GCPSO, MPSO, CPSO-S_{16} and CPSO-H_{16} algorithms all performed significantly better than the GD algorithm. In turn, the MPSO and CPSO-S_{16} algorithms performed significantly better than the GCPSO algorithm, with the CPSO-S_{16} algorithm having been the best overall performer.

Product Unit Network

All of the algorithms, except for the MPSO and CPSO-S_2 algorithms, had non-normally distributed ε_T values. Table 6.15 indicates that the most acceptable split factor for the CPSO-S_K algorithm was $K = 2$. Note, however, that $K = 14$ was also a good choice. Note that this network had a total of $8 \times 16 + (16 + 1) = 145$ weights.

The results of using the different algorithms to train a product unit network to solve the Diabetes classification problem are presented in Table 6.16. A t-test indicates that the performance of the GA and CCGA algorithms did not differ significantly, however, they both performed significantly worse than the GCPSO algorithm. The GCPSO, MPSO and CPSO-S_2 algorithms all performed similarly, taking the lead over all the other algorithms.

Algorithm	ε_T	ε_G
GD	52.93 ± 7.76	58.33 ± 6.72
SCG	53.15 ± 7.21	59.16 ± 6.28
CCGA	31.52 ± 1.57	35.99 ± 1.61
GA	30.72 ± 1.36	34.24 ± 1.34
GCPSO	26.18 ± 1.27	35.04 ± 1.05
MPSO	25.32 ± 0.98	34.72 ± 0.99
CPSO-S_2	25.34 ± 1.03	35.56 ± 0.99
CPSO-H_2	32.36 ± 1.62	37.98 ± 1.41

Table 6.16: Comparing different training algorithms on the Diabetes classification problem using a product unit network.

6.3.5 Hepatitis

The Hepatitis classification problem consists of 154 patterns, each made up of 19 attributes, with 2 output classes. The optimal number of hidden units, as determined by the OBD algorithm, was 9. A network with a 19-9-1 architecture was therefore used.

Summation Unit Network

All algorithms, with the exception of the GA and CCGA algorithms, had non-normal ε_T distributions. The results in Table 6.17 indicate that many of the CPSO-S_K algorithms were able to train the network until the classification error on the training set, ε_T, was zero. The results of the CPSO-S_{13} algorithm were selected arbitrarily to represent the family of CPSO-S_K algorithms in the comparisons below.

Table 6.20 presents a comparison between the different algorithms. The performance of the GA and CCGA algorithms clearly lagged behind the other algorithms by a significant distance. The t-test results indicate that the GCPSO and MPSO algorithms performed significantly worse than the GD and CPSO-S_{13} algorithms. The GCPSO also performed significantly worse than the SCG algorithm, but a t-test shows that the performance of the MPSO and SCG algorithms were similar.

Even though this problem had a total of $20 \times 9 + (9 + 1) = 190$ weights, it was

Algorithm	ε_T	ε_G
CPSO-S_2	0.42 ± 0.22	27.00 ± 1.96
CPSO-S_{12}	0.00 ± 0.00	28.64 ± 1.46
CPSO-S_{13}	0.00 ± 0.00	27.00 ± 1.96
CPSO-S_{14}	0.00 ± 0.00	28.04 ± 1.73
CPSO-S_{37}	0.02 ± 0.04	25.84 ± 1.69
CPSO-S_{38}	0.00 ± 0.00	26.20 ± 1.64
CPSO-S_{39}	0.00 ± 0.00	27.96 ± 1.34

Table 6.17: Classification error for a product unit network, applied to the Hepatitis problem, obtained with the CPSO-S_K algorithm using different values for K.

Algorithm	ε_T	ε_G
GD	0.00 ± 0.00	30.08 ± 1.82
SCG	0.13 ± 0.10	28.84 ± 1.66
CCGA	12.50 ± 0.64	20.24 ± 1.58
GA	18.35 ± 0.95	22.92 ± 1.48
GCPSO	0.46 ± 0.34	28.12 ± 1.56
MPSO	0.21 ± 0.14	29.64 ± 1.95
CPSO-S_{13}	0.00 ± 0.00	27.00 ± 1.96
CPSO-H_{13}	0.56 ± 0.18	24.72 ± 1.80

Table 6.18: Comparing different training algorithms on the Hepatitis classification problem using a summation unit network.

Algorithm	ε_T	ε_G
CPSO-S_2	6.38 ± 0.84	37.56 ± 2.64
CPSO-S_{12}	2.38 ± 0.70	43.80 ± 2.20
CPSO-S_{13}	1.50 ± 0.74	46.00 ± 2.18
CPSO-S_{14}	2.15 ± 1.01	48.36 ± 2.30
CPSO-S_{37}	5.19 ± 3.38	57.28 ± 3.86
CPSO-S_{38}	4.29 ± 3.26	54.84 ± 3.95
CPSO-S_{39}	5.52 ± 2.92	57.76 ± 4.52

Table 6.19: Classification error for a product unit network, applied to the Hepatitis problem, obtained with the CPSO-S_K algorithm using different values for K.

relatively easy for all the algorithms to find values for the weights that resulted in zero training errors. All these algorithms had significantly larger generalisation errors, a sure indication that the trained networks were overfitted.

Product Unit Network

A normality test indicates that only the GA, GCPSO and CPSO-S_2 algorithms had normally distributed training errors. The results in Table 6.19 suggest a split factor of $K = 13$ for the CPSO-S_K algorithm. The network had a total of $19 \times 9 + (9 + 1) = 181$ weights.

Table 6.20 compares the different algorithms on the task of training a product unit network to solve the Hepatitis classification problem. The CCGA algorithm performed very well on this test, significantly better than even the MPSO algorithm. The GA, on the other hand, performed significantly worse than the GCPSO algorithm, even though it still had acceptable performance. The results of the GCPSO and MPSO algorithms were comparable. The best algorithm in terms of training performance was the CPSO-S_{13} algorithm, with a t-test score indicating that it performed significantly better than the CCGA algorithm.

The most interesting feature of the results in Table 6.20 is that both the GA-based algorithms had significantly smaller generalisation errors than any of the other algorithms, even though these two algorithms had training errors comparable to the rest.

Algorithm	ε_T	ε_G
GD	48.69 ± 9.71	66.32 ± 6.00
SCG	40.13 ± 10.21	65.80 ± 6.01
CCGA	2.40 ± 0.41	24.68 ± 1.38
GA	7.73 ± 0.84	22.84 ± 1.58
GCPSO	5.37 ± 1.05	35.96 ± 2.15
MPSO	4.88 ± 1.39	35.72 ± 2.56
CPSO-S_{13}	1.50 ± 0.74	46.00 ± 2.18
CPSO-H_{13}	10.33 ± 2.16	41.12 ± 3.34

Table 6.20: Comparing different training algorithms on the Hepatitis classification problem using a product unit network.

One possible explanation for this behaviour is that the GA was more prone to keep the weights near the values to which they were initialised. Since the initialisation range for the GAs was $[-1, 1]$ for each weight, the tendency to stay in this range resulted in the GA producing smoother network mappings. The PSO-based algorithms used a more aggressive strategy, so that they were more likely to stray further from their initial weights. This implies that they were more likely to overfit the training set, a phenomenon that can clearly be seen in the results of the CPSO-S_{13} algorithm.

6.3.6 Henon Map

The Henon map is a curve generated by equation (6.5). Figure 6.3 illustrates this curve in three dimensions.

$$z_t = 1 + 0.3z_{t-2} - 1.4z_{t-1}^2 \tag{6.5}$$

with $z_1, z_2 \sim U(-1, 1)$.

For this problem, the neural networks were given a noise-free data set of 100 points generated with equation (6.5). Because the data set is a noise-free description of the function that generated them, the network cannot overfit the data points, assuming there's enough of them to describe the function accurately. A 2-5-1 network architecture was selected arbitrarily. Further, since there's no concept of a classification error associated

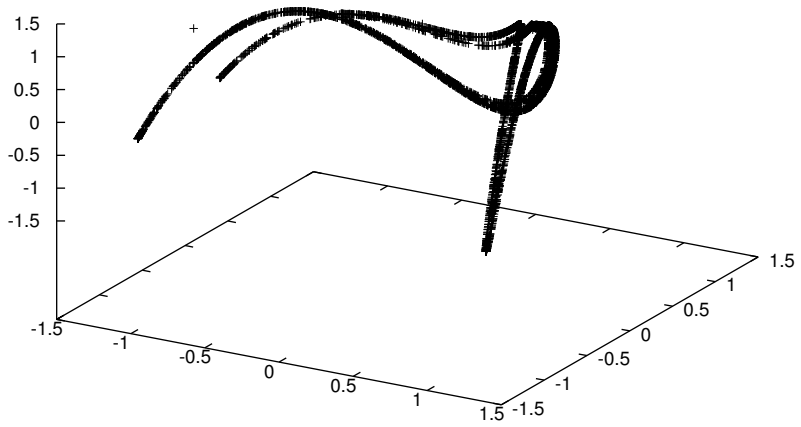


Figure 6.3: The Henon map

Algorithm	MSE_T	MSE_G
CPSO-S_2	3.02e-06 ± 7.00e-07	1.77e-05 ± 6.96e-06
CPSO-S_4	4.90e-06 ± 1.57e-06	4.24e-05 ± 3.65e-05
CPSO-S_8	7.48e-06 ± 2.47e-06	5.02e-05 ± 3.37e-05

Table 6.21: Mean Squared Error values for a summation unit network, applied to the Henon map problem, obtained with the CPSO-S_K algorithm using different values for K.

with this problem, only the MSE_T and MSE_G values will be considered.

Summation Unit Network

All the algorithms, except the CCGA, had non-normally distributed MSE_T values. The summation unit network had $3 \times 5 + (5 + 1) = 21$ weights in total, far fewer than any of the classification problems. Table 6.21 indicates that a split factor of $K = 2$ produced acceptable results for the CPSO-S_K algorithm.

Algorithm	MSE_T	MSE_G
GD	6.29e-07 ± 5.29e-08	1.83e-06 ± 3.51e-07
SCG	1.08e-06 ± 8.17e-08	3.11e-06 ± 4.74e-07
CCGA	6.12e-04 ± 9.52e-05	1.49e-03 ± 2.84e-04
GA	1.13e-04 ± 3.33e-05	2.56e-05 ± 8.34e-05
GCPSO	3.37e-06 ± 1.17e-06	1.03e-05 ± 3.85e-06
MPSO	3.10e-06 ± 7.36e-07	1.17e-05 ± 5.57e-06
CPSO-S_2	3.02e-06 ± 7.00e-07	1.77e-05 ± 6.96e-06
CPSO-H_2	7.93e-05 ± 3.58e-05	2.44e-04 ± 1.08e-04

Table 6.22: Comparing different training algorithms on the Henon function approximation problem using a summation unit network.

Algorithm	MSE_T	MSE_G
CPSO-S_2	4.16e-05 ± 3.11e-05	1.51e-03 ± 2.77e-03
CPSO-S_4	1.54e-04 ± 5.77e-05	3.64e-02 ± 7.14e-02
CPSO-S_8	1.57e-04 ± 6.02e-05	8.24e-04 ± 4.91e-04

Table 6.23: Mean Squared Error values for a product unit network, applied to the Henon map problem, obtained with the CPSO-S_K algorithm using different values for K.

The results in Table 6.22 show how well each algorithm was able to train the summation unit network to approximate the Henon map. The CCGA, GA, GCPSO, MPSO, CPSO-S_2 and CPSO-H_2 algorithms all performed significantly worse than the GD and SCG algorithms. Both the GA-based algorithms performed significantly worse than the PSO-based algorithms.

Product Unit Network

All of the algorithms produced MSE_T values that had a non-normal distribution. Table 6.23 shows that the an acceptable value for K is 2, in other words, the weight vector was split into only two parts. The network contained a total of $2 \times 5 + 6 = 16$ weights.

The results in Table 6.24 compares how well each algorithm was able to train the product unit to learn the Henon map. The two gradient-based algorithms performed

Algorithm	MSE_T	MSE_G
GD	2.41e-03 ± 9.26e-04	5.11e-02 ± 6.51e-02
SCG	1.01e-02 ± 9.86e-03	7.40e-03 ± 3.46e-03
CCGA	1.12e-03 ± 4.74e-04	2.66e-03 ± 1.06e-03
GA	1.00e-03 ± 2.77e-04	7.38e-02 ± 1.42e-01
GCPSO	7.19e-05 ± 4.07e-05	3.38e-04 ± 2.17e-04
MPSO	3.38e-05 ± 2.63e-05	7.06e-04 ± 1.16e-03
CPSO-S_2	4.16e-05 ± 3.11e-05	1.51e-03 ± 2.77e-03
CPSO-H_2	1.37e-04 ± 9.93e-05	8.42e-04 ± 7.77e-04

Table 6.24: Comparing different training algorithms on the Henon function approximation problem using a product unit network.

significantly worse than any other algorithm; the two GA-based algorithms performed equally well. With a p-value of 0.059, the MPSO did not perform significantly better than the GCPSO, at the usual 5% confidence level. Similarly, the performance of the CPSO-S_2 algorithm was not significantly different from the MPSO and GCPSO algorithms. Lastly, the CPSO-H_2 did not perform significantly worse than the GCPSO, but it did perform more poorly than the MPSO. Overall, the PSO-based algorithms produced the best results on this test problem, beating both the GA-based algorithms.

6.3.7 Cubic Function

The Cubic function is a very simple function, originally used by Engelbrecht and Ismail to test the abilities of a product unit network [41]. This function is defined as

$$f(z) = z^3 - 0.04z \tag{6.6}$$

where $z \in [-1, 1]$. A data set of 100 points was constructed, again without adding any noise. The optimal network architecture for a product unit network is 1-2-1, so the summation unit network was also tested in this configuration.

Algorithm	MSE_T	MSE_G
CPSO-S_2	9.45e-06 ± 7.56e-06	2.39e-05 ± 2.06e-05
CPSO-S_3	3.04e-05 ± 1.36e-05	5.90e-05 ± 2.66e-05
CPSO-S_5	1.60e-05 ± 9.77e-06	3.25e-05 ± 1.88e-05

Table 6.25: Mean Squared Error values for a summation unit network, applied to the Cubic function approximation problem, obtained with the CPSO-S_K algorithm using different values for K.

Algorithm	MSE_T	MSE_G
GD	1.00e-04 ± 3.61e-06	2.19e-04 ± 1.82e-05
SCG	1.06e-04 ± 1.57e-06	2.21e-04 ± 1.70e-05
CCGA	1.06e-04 ± 5.70e-06	2.29e-04 ± 2.51e-05
GA	1.17e-05 ± 5.66e-06	2.38e-05 ± 1.16e-05
GCPSO	6.84e-06 ± 6.56e-06	1.43e-05 ± 1.34e-05
MPSO	5.42e-06 ± 6.14e-06	9.04e-06 ± 9.36e-06
CPSO-S_2	9.45e-06 ± 7.56e-06	2.39e-05 ± 2.06e-05
CPSO-H_2	7.82e-05 ± 1.09e-05	1.66e-04 ± 2.70e-05

Table 6.26: Comparing different training algorithms on the Cubic function approximation problem using a summation unit network.

Summation Unit Network

The MSE_T distribution of all the algorithms, except SCG, were non-normal at a 1% confidence level. According to the values in Table 6.25, a split factor of $K = 2$ was acceptable. The network contained a total of $2 \times 2 + 3 = 7$ weights.

Table 6.26 presents the results of training a summation unit network to approximate the Cubic function. The GD algorithm performed significantly better than the SCG algorithm at a 5% confidence level. The CCGA algorithm performed significantly worse than the GA. There was no statistically significant difference between the GA and the GCPSO, MPSO and CPSO-S_2 algorithms; in contrast, the CPSO-H_2 did perform worse than the GA.

Overall, the MPSO algorithm had the best performance, and although it was not

Algorithm	MSE_T	MSE_G
CPSO-S_2	3.81e-03 ± 3.01e-03	9.61e-03 ± 7.87e-03
CPSO-S_3	3.35e-03 ± 3.04e-03	8.12e-03 ± 7.77e-03
CPSO-S_4	1.04e-02 ± 6.07e-03	1.98e-02 ± 1.31e-02

Table 6.27: Mean Squared Error values for a product unit network, applied to the Cubic function approximation problem, obtained with the CPSO-S_K algorithm using different values for K.

Algorithm	MSE_T	MSE_G
GD	2.74e-01 ± 4.05e-01	5.38e-01 ± 6.85e-01
SCG	1.69e-01 ± 1.77e-01	1.05e-01 ± 1.14e-01
CCGA	4.29e+140 ± 8.62e+140	1.84e+103 ± 3.71e+103
GA	5.99e-04 ± 2.77e-04	9.84e-04 ± 3.34e-04
GCPSO	1.13e-05 ± 5.16e-06	2.77e-05 ± 1.32e-05
MPSO	1.21e-05 ± 4.57e-06	3.04e-05 ± 1.19e-05
CPSO-S_3	3.35e-05 ± 3.04e-05	8.12e-05 ± 7.77e-05
CPSO-H_3	5.83e-05 ± 4.76e-05	9.98e-05 ± 8.35e-05

Table 6.28: Comparing different training algorithms on the Cubic function approximation problem using a product unit network.

significantly better than the GA, it did outperform the gradient-based algorithms.

Product Unit Network

None of the algorithms had normally distributed MSE_T values. The product unit network had $1 \times 2 + 3 = 5$ weights in total. Table 6.27 indicates that the CPSO-S_3 algorithm performed best amongst the different CPSO-S_K algorithms, suggesting that K should be 3.

Table 6.28 presents the results of training a product unit network to approximate the Cubic function. Note that the CCGA algorithm failed to train the network — one possible explanation for this phenomenon is that the CCGA algorithm could not learn the one weight value that was equal to 3.0, which was relatively far from the CCGA's

initialisation space of [-1,1]. The gradient-based GD and SCG algorithms, as usual, performed poorly on the product unit network training task.

The GA performed significantly worse than the GCPSO. There was no significant difference in performance amongst the GCPSO, MPSO and CPSO-S_3 algorithms. The CPSO-H_3 did perform significantly worse than any of the other PSO-based algorithms.

6.4 Discussion of Results

Although the tables in the previous section provided some insight into the behaviour of the different algorithms, more information can be gleaned from a plot of the network's MSE_T and MSE_G values over time. For example, if an algorithm has become trapped in a local minimum, then the MSE_T curve will become a flat line, clearly showing that the algorithm failed to make any further improvement.

There are several problems with plotting the MSE_T and MSE_G curves, though. A single run of the algorithm may not be representative of the average behaviour of the algorithm, so the information obtained over multiple runs must be considered. The stochastic algorithms, *e.g.* the GAs and PSOs, have significantly different behaviour at a given moment in time over many runs, especially earlier during the training process. An approximation of the true curve can be obtained by averaging the values of multiple runs (at the same time step) to produce an "average" curve. These curves may not be entirely accurate, because they assume that the distribution of the solutions discovered by the algorithm at time t is normal, which may not be true for every time step. Even though this method makes strong assumptions regarding the nature of the MSE_T curve, the resulting curves are usually accurate enough to see some interesting trends.

Figures 6.4 and 6.5 are plots of the summation unit network MSE_T values over time for the Diabetes and Hepatitis problems, respectively. Note how all the algorithms show rapid improvement during the first few iterations on the Diabetes problem (Figure 6.4). After this initial stage, the different algorithms exhibit significantly different behaviour. The GD algorithm had the worst performance of all the algorithms represented in the figure. The curves for the GCPSO and MPSO algorithms are identical up to around 100 seconds into the simulation run, where the MPSO curve gradually drops away from the

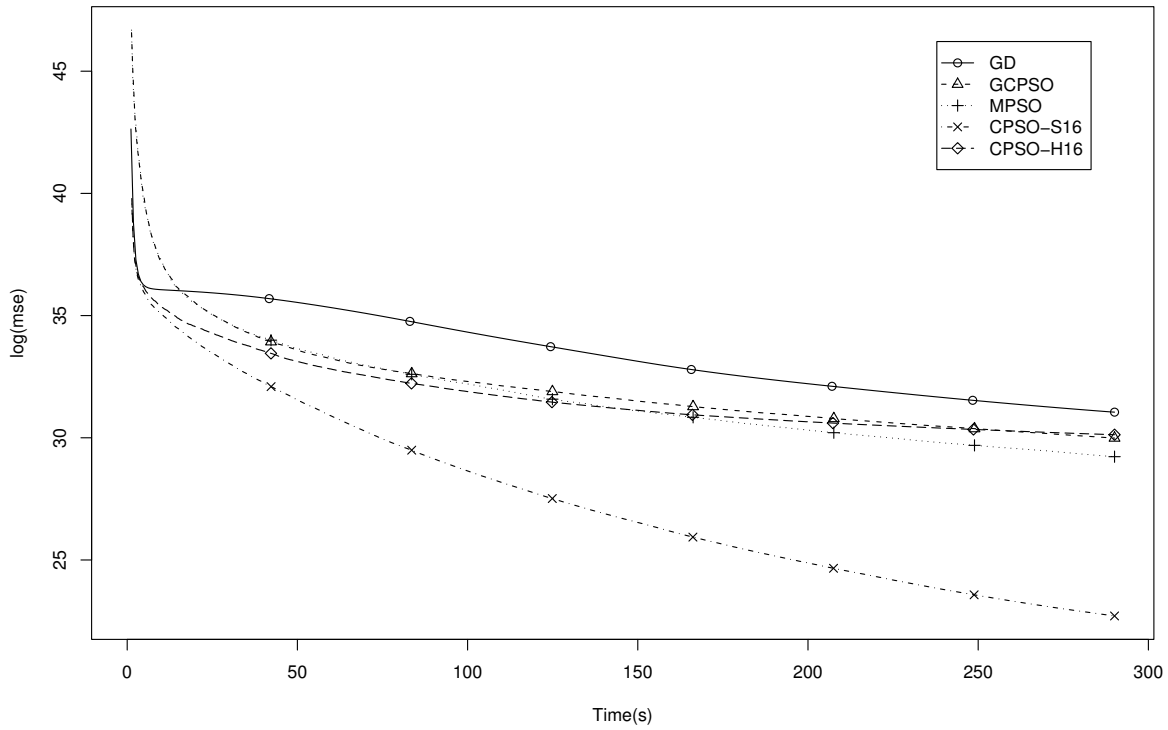


Figure 6.4: The MSE_T of the Diabetes problem, using a summation unit network, plotted over time

GCPSO curve. This clearly illustrates that the MPSO algorithm is detecting stagnation (or slow progress) and re-starting the algorithm, resulting in improved performance. The curve of the CPSO-S_{16} algorithm has a significantly better rate of improvement, and does not appear to slow down as it nears the end of the simulation run. This behaviour is typical for the CPSO-S_K algorithm, and was observed on most of the plots obtained from the summation unit network experiments.

Figure 6.5 presents a slightly different scenario. In this case, the GD algorithm had significantly better rate of improvement than the CPSO-S_{16} algorithm, until near the end of the simulation run. The first 0.1 seconds of the simulation was not included in the plot in order to improve the clarity of the diagram — this explains the neat ordering of the curves at the left edge of the plot. Keep in mind that most algorithms were able to reduce the training classification error (ε_T) to near zero on this particular problem. Other

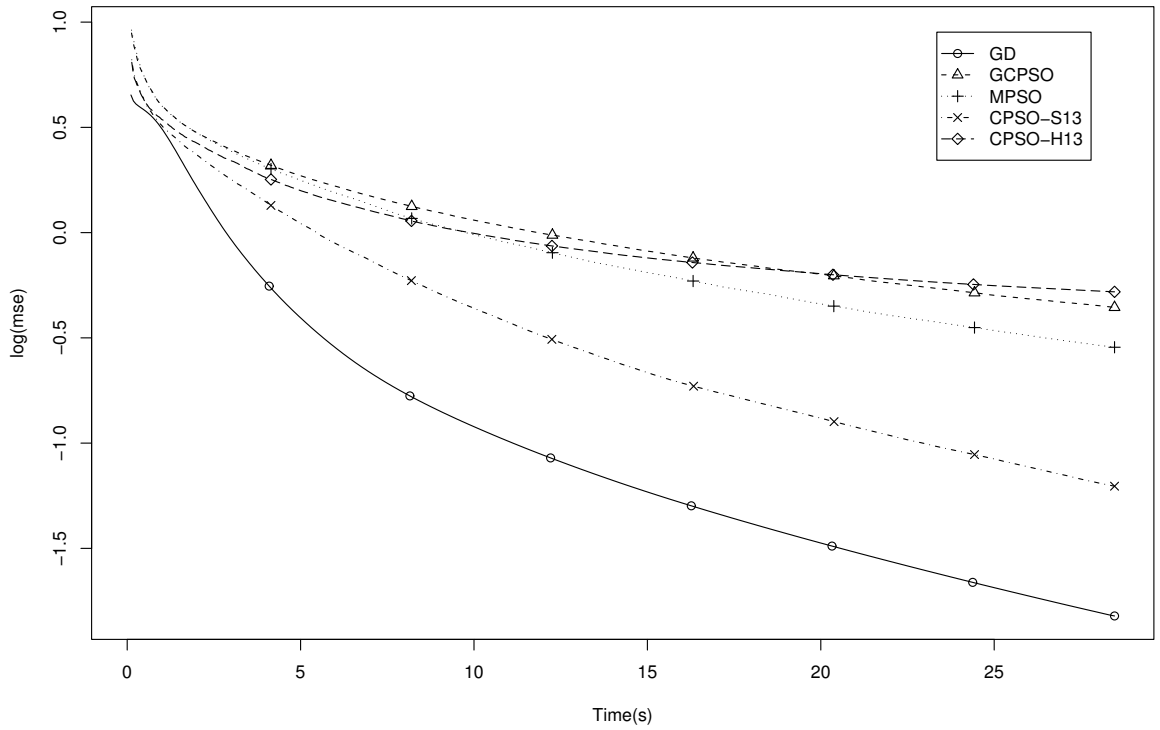


Figure 6.5: The MSE_T of the Hepatitis problem, using a summation unit network, plotted over time

than the fact that the GD algorithm performed better than the CPSO-S_{16} algorithm, the relative ranking and behaviour of the algorithms remain the same as observed in Figure 6.4.

Similar curves can be plotted for the MSE computed on the test set (*i.e.* the MSE_G values). Unfortunately, it is not possible to obtain such clean, noise-free curves from the test set as can be obtained from the training set. All the algorithms tested preserved a copy of the best solution discovered so far during each training run. This implies that the MSE_T curve, associated with the training error, is a monotonically decreasing function of time. Conversely, the generalisation error is expected to be able to rise over time if the network is overfitting the data, so that the MSE_G curve is not necessarily monotonic. This makes the MSE_G curve very noisy when the MSE_G values at each time step are averaged over a number of simulation runs. In general, the PSO algorithms compute the

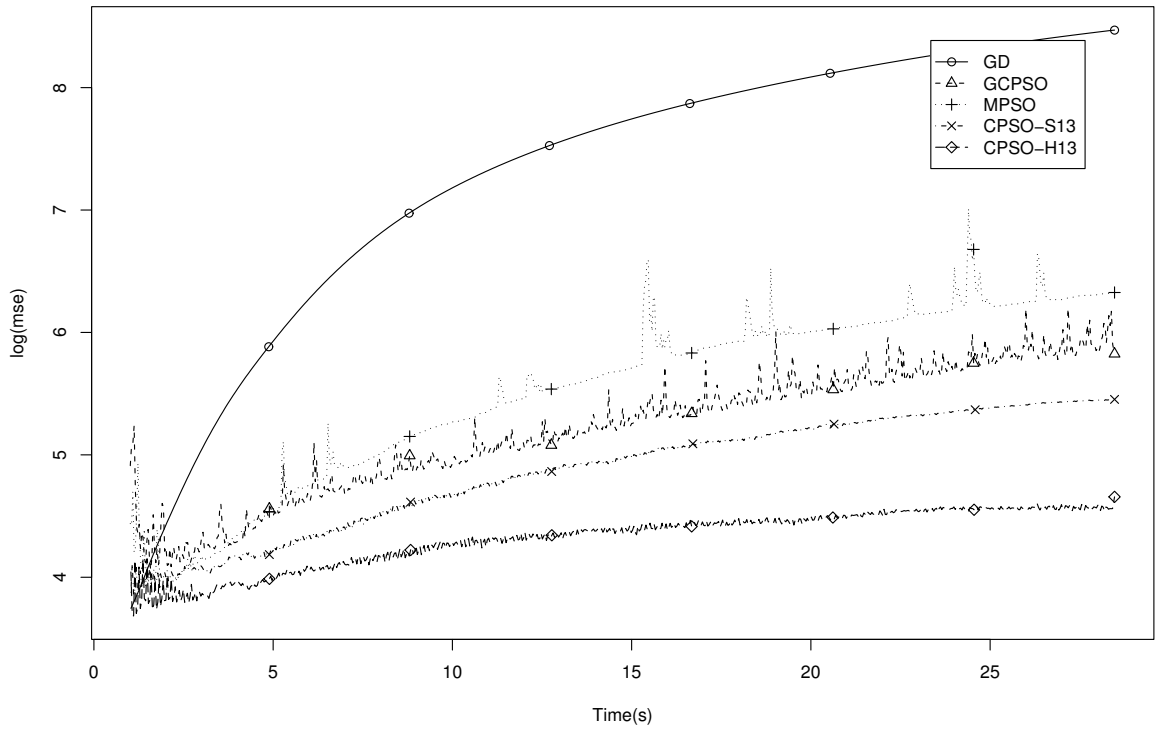


Figure 6.6: The MSE_G of the Hepatitis problem, using a summation unit network, plotted over time

MSE_G value for each particle in the swarm, since the fitness evaluation of each particle corresponds one forward propagation through the network. A diverse swarm will thus include particles with widely differing MSE_G values, making the curve noisy.

Figure 6.6 presents such a curve for a summation unit network applied to the Hepatitis problem. The GD algorithm produces a smooth curve because it only maintains a single solution. All the other population-based algorithms have the characteristic noise described in the previous paragraph. Note how the MPSO algorithm has some large spikes that occasionally appear. These are caused by the re-starts that the algorithm uses to prevent stagnation — each re-start increases the diversity of the population, which implies that some inferior solutions may temporarily appear in the population. Keep in mind that such an inferior solution will have a large training error as well as a large generalisation error, which explains why the re-start are marked by *increases* in

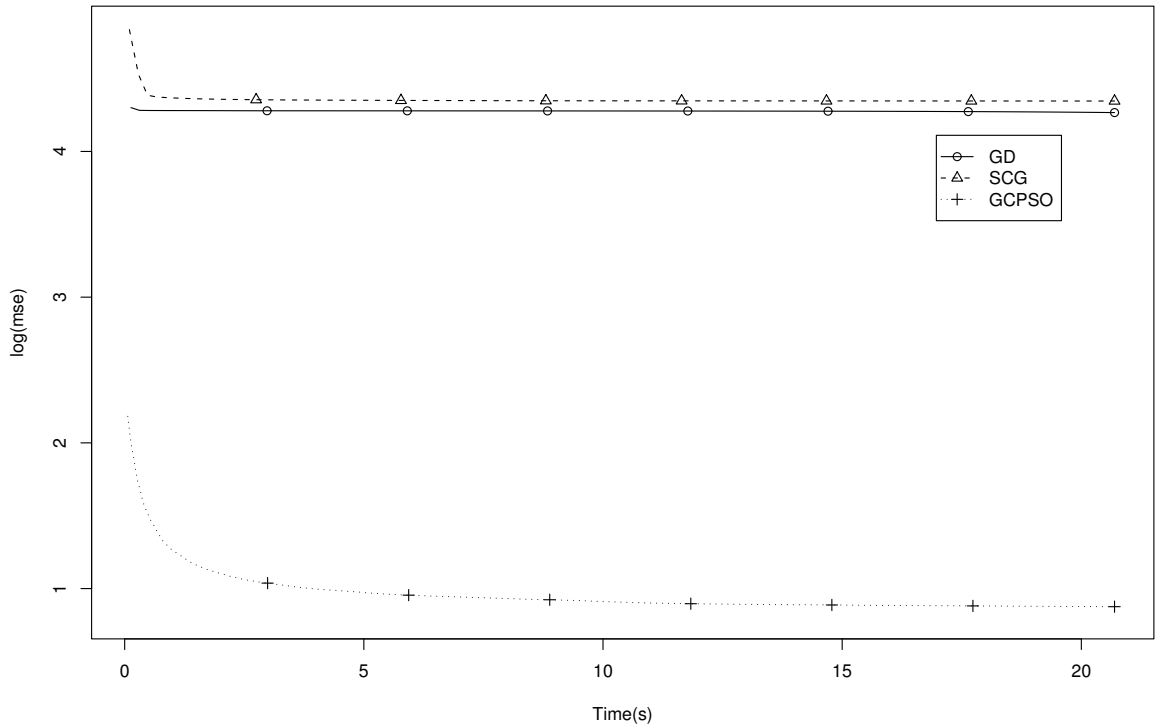


Figure 6.7: The MSE_T of the Iris problem, using a product unit network, plotted over time. This plot clearly shows how poorly the GD and SCG algorithms perform compared to the PSO-based algorithms.

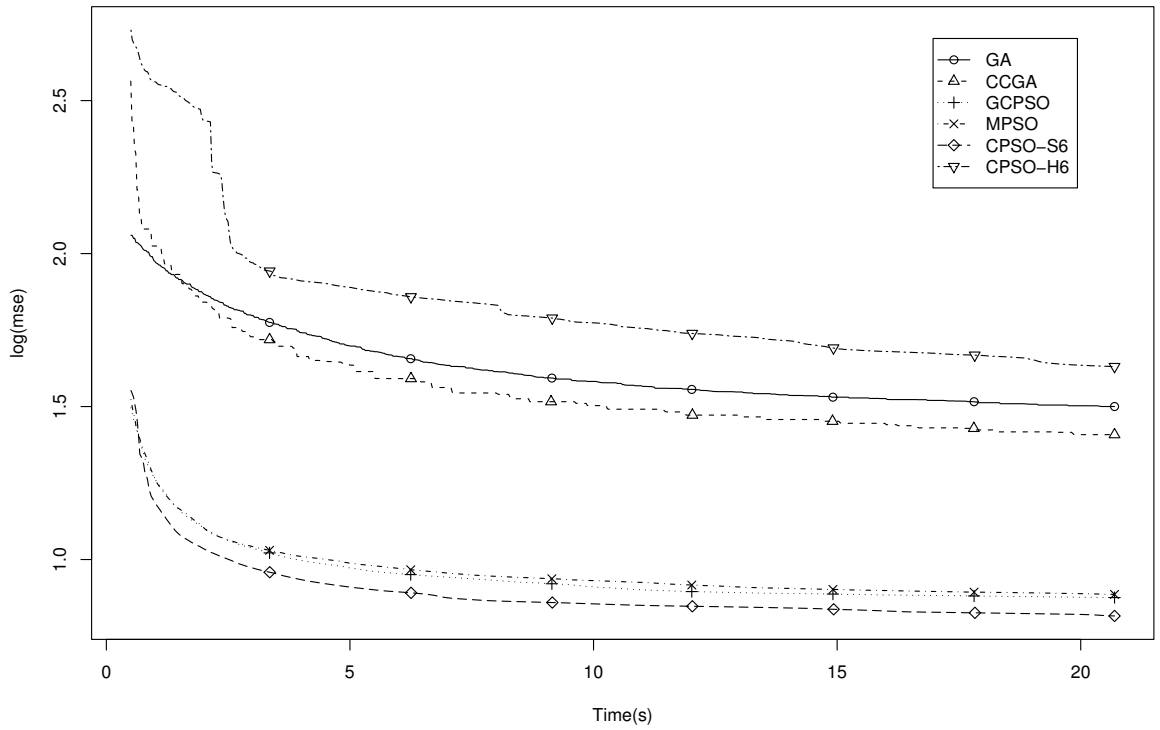


Figure 6.8: The MSE_T of the Iris problem, using a product unit network, plotted over time. This plot compares the various GA-based and PSO-based algorithms.

the generalisation error.

Clearly all the algorithms are overfitting the network on the training set, since they all have MSE_G curves that increase during the course of a simulation run. In particular, the GD algorithm has the largest MSE_G values, which corresponds directly to its ability to rapidly decrease the training error, as observed in Figure 6.5. Since none of the algorithms had any mechanism to prevent overfitting, it would not be fair to claim that the CPSO-S_{13} algorithm had better generalisation performance, even though the diagram seems to imply this. A further implication of the severe overfitting that all the training algorithms exhibited is that the number of hidden units recommended by the OBD algorithm was not optimal.

Similar plots can be obtained for the product unit networks. Figure 6.7 compares the MSE_T curves of the GCPSO algorithm with those of the gradient-based algorithms.

Clearly the GD and SCG algorithms fail to make any progress after the first second of the simulation run has passed. The strong local convergence property of the gradient-based algorithms causes them to become trapped in local minima shortly after the training session started. This phenomenon has also been observed by Engelbrecht and Ismail [41]. Because of the large difference in magnitude between the MSE_T values of the gradient-based algorithms and those of the other algorithms, the gradient-based algorithms will be excluded from further plots obtained from product unit networks.

Figure 6.8 is a plot of the MSE_T curves of the various algorithms (excluding the gradient-based ones) obtained by training a product unit network to solve the Iris classification problem. Note that the two GA-based algorithms and the CPSO-H_6 algorithm appear to belong to roughly the same performance category. The other algorithms, namely GCPSO, MPSO and CPSO-S_6, all belonged to another category that performed significantly better than the first one. In particular, note that the CPSO-S_6 algorithm had the best overall performance on this problem.

It appears that the CPSO-S_K algorithm consistently performed better than the standard GCPSO or even the MPSO, as long as the correct split factor (K values) was selected.

Table 6.29 is a summary of the number of network weights in each problem, and the split factor that produced the best results. The SUN column corresponds to the summation unit networks, and the PUN column to the product unit networks. In both cases W denotes the number of weights.

Although the number of experiments were far too few to form any reliable rules, it is interesting to note that the K-values (split factors) satisfied the relation

$$K = a\sqrt{W}$$

where a was approximately 1.3 for the summation unit networks, and 0.5 for the product unit networks. This appears to be a useful heuristic for choosing the appropriate split factor. Figures 6.9 and 6.10 show plots of the least-squares fits for the summation and product unit networks, respectively.

Note that the constant associated with the product unit networks is smaller than that of the summation unit networks, implying that the product unit networks favour smaller K values. This is also apparent from the values presented in Table 6.29. One possible

	SUN		PUN	
Problem	K	W	K	W
Iris	7	35	6	31
Cancer	14	89	5	81
Wine	16	88	4	83
Diabetes	16	161	2	145
Hepatitis	13	190	13	181
Henon	2	21	3	16
Cubic	2	7	3	5

Table 6.29: Summary of the split factors and the number of weights in each problem.

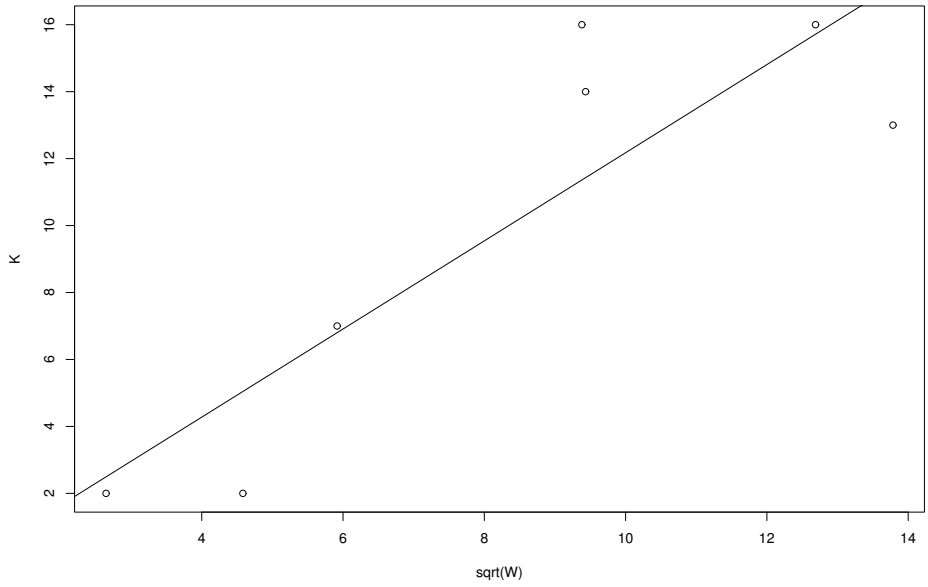


Figure 6.9: A plot of K against $\sqrt{W}$, for the summation unit networks

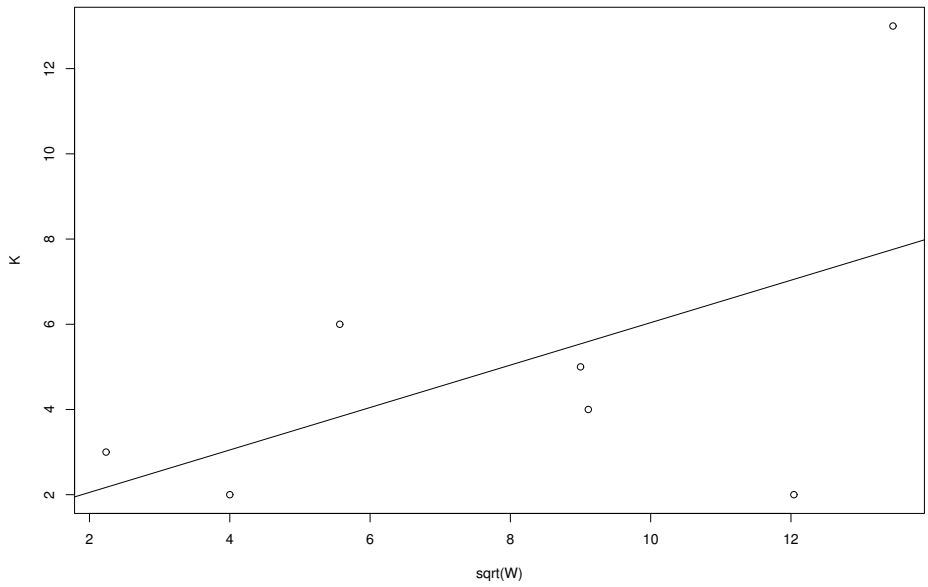


Figure 6.10: A plot of K against $\sqrt{W}$, for the product unit networks

explanation for this behaviour is that there's more interaction between the weights of a product unit network (compare equation 6.1 with equation 6.4), thus implying that large split factors will have a detrimental effect on the performance of the CPSO-S_K algorithm.

6.5 Conclusion

This chapter presented results obtained by training various summation and product unit networks using the different PSO-based algorithms introduced in Chapters 3 and 4. The same networks were also trained with gradient-based and GA-based algorithms to provide an indication of how the PSO-based algorithms perform compared to existing efficient techniques.

Based on these results, it is clear that the GCPSO, MPSO and CPSO-S_K algorithms are competitive in terms of performance with the GD and SCG algorithms on summation unit networks. With a suitable choice of K, the split factor, the CPSO-S_K algorithm performed significantly better than any of the other PSO-based algorithms.

The GD and SCG algorithms failed to train the product unit networks — this behaviour was expected, since other publications reported the same findings. The GA-based algorithms were able to train the product unit networks, but generally the performance of the PSO-based algorithms was superior. Amongst the PSO-based algorithms, the CPSO-S_K algorithm again produced the best results, as long as an appropriate value for K was selected.

Although it was not able to outperform the CPSO-S_K algorithm, the MPSO algorithm consistently performed better than the GCPSO algorithm on both types of network. Even though this algorithm has some parameters that can be fine-tuned for best performance, the same setting produced good results in all the experimental settings, making it somewhat easier to use than the CPSO-S_K algorithm, which has to be fine-tuned for optimal performance on each problem.

The CPSO-H_K algorithm did not appear to offer any advantage, often performing worse than even the GCPSO algorithm on both the summation and product unit networks.

Chapter 7

Conclusion

This chapter briefly summarises the findings and contributions of this thesis, followed by a discussion of numerous directions for future research.

7.1 Summary

This thesis investigated the behaviour of various Particle Swarm Optimiser algorithms. A significant portion of existing PSO-related publications relied almost exclusively on empirical research to draw conclusions. The theoretical analyses presented in Chapter 3 provides numerous new insights into the underlying mechanics of the PSO algorithm.

Chapter 3 analysed the convergence properties of the trajectory of a single particle. A set of equations was developed that can be used to determine whether a specific configuration of parameters, *i.e.* the values of the inertia weight and acceleration coefficients, leads to a convergent particle trajectory. This model correctly predicts the trajectory of a single particle in the absence of a stochastic component.

When the stochastic component is introduced, the model can no longer predict the exact trajectory of a particle, but it is still able to classify a parameter configuration as either divergent or convergent, so that the long term behaviour remains predictable.

The development of a model that could describe the behaviour of a single particle made it possible to investigate the local convergence properties of the Particle Swarm Optimiser. The PSO algorithm was viewed as a random search algorithm with self-

evolving strategy parameters, thus allowing the application of well-known random search convergence proofs to the PSO. This resulted in an unexpected discovery: The PSO algorithm is not able to guarantee that it will locate the minimiser of even a unimodal function. It was shown that many states exist in which the PSO becomes trapped in a stagnant state from which it is not possible to reach the minimiser of the objective function.

This discovery led to the development of a new update equation for the global best particle in the swarm. The new update equation is guaranteed to have a non-degenerate sampling volume at all times, so that it is no longer possible for the swarm to stagnate. A new algorithm making use of this improved update equation, called the Guaranteed Convergence PSO (GCPSO), was introduced, with its corresponding formal proof of guaranteed local convergence.

Since this algorithm only had guaranteed convergence on local minima, further methods for extending the GCPSO to become a global optimisation algorithm were investigated. Two algorithms with guaranteed global convergence were introduced. The Randomised PSO (RPSO) attempts to continually sample the whole search space by continually randomising a subset of the particles while the rest rest of the swarm explore the search space as usual. The second algorithm, called the Multi-start PSO (MPSO), attempts to detect when the PSO has found a local minimum. Once a local minimum is found, the algorithm re-starts the algorithm with new randomly chosen initial positions for the particles. Formal proofs of global convergence were presented for both algorithms.

Chapter 4 presented two types of cooperative PSO algorithm. The CPSO-S$_K$ algorithm was based on Potter's CCGA algorithm [106]. This algorithm partitions the search space into disjoint subspaces, assigning a swarm to each subspace. The swarms then cooperate to solve the problem by sharing the best solutions they have discovered in their respective subspaces. It was shown that this algorithm can become trapped in so-called pseudo-minima. These pseudo-minima are solutions that are constructed using local minimisers from the different subspaces, but they are not local minima in the original search space. It was shown that pseudo-minima can prevent the CPSO-S$_K$ algorithm from reaching the true local minimum. A second algorithm, CPSO-H$_K$, builds on the CPSO-S$_K$ algorithm by adding an additional swarm that attempts to search the

whole search space (*i.e.* the union of the disjoint subspaces). This extra swarm also exchanges information regarding the best solutions discovered so far with the group of subspace-swarms inherited from the CPSO-S$_K$ algorithm. The CPSO-H$_K$ algorithm can not become trapped in pseudo-minima, and was shown to possess guaranteed convergence on local minima using the GCPSO algorithm's local convergence proof.

Empirical results were presented in Chapter 5 to support the theoretical models developed in Chapters 3 and 4. It was shown that adjusting the inertia weight and acceleration coefficients of the PSO algorithm leads to a tradeoff between the performance of the algorithm on unimodal and multi-modal functions. Further results confirmed that the GCPSO algorithm is able to minimise unimodal functions significantly more effectively than the standard PSO algorithm, but that the performance of the two algorithms were merely comparable on multi-modal functions.

The MPSO algorithm was shown to have significantly better performance on such multi-modal functions, compared to the GCPSO or PSO. In contrast, the empirical results showed that the RPSO does not appear to offer any significant improvement over the standard GCPSO or PSO algorithms.

The cooperative PSO algorithms make stronger assumptions about the nature of the objective functions they optimise. The sensitivity of these algorithms to changes affecting those assumptions were investigated, where it was found that the cooperative algorithms were sensitive to the degree of inter-variable correlation. When the variables of a function were independent, the CPSO-S and CPSO-H algorithms showed a significant improvement over the standard GCPSO and PSO algorithms on multi-modal functions, however, when the variables were highly correlated, this advantage disappeared. The CPSO-S$_K$ and CPSO-H$_K$ offered more modest improvements over the standard GCPSO and PSO algorithms on multi-modal functions, but they were not significantly affected by the presence of inter-variable correlation.

Chapter 6 applied the various PSO-based algorithms to the task of training both summation and product unit networks. It was shown that the PSO-based algorithms are able to reduce the training error of a network as effectively as the GD or SCG algorithms on summation unit networks, and significantly better when applied to product unit networks. Amongst the PSO-based algorithms, it was shown that the MPSO and CPSO-

S_K algorithms performed significantly better than the standard GCPSO algorithm.

7.2 Future Research

Throughout this thesis several new directions for future research presented themselves. These ideas are briefly summarised below.

Adaptive GCPSO parameters

The GCPSO algorithm, introduced in Section 3.2, makes use of several new parameters to govern the behaviour of the global best particle. The GCPSO algorithm employed in Chapters 5 and Chapter 6 used parameters that were set to fixed values. These fixed values were found to produce acceptable performance on a small set of test functions.

These GCPSO parameters have a strong resemblance to the strategy parameters found in Evolution Strategies algorithms. ES algorithms evolve the appropriate values for the strategy parameters while they optimise the objective function. A possible direction for future research is thus to use the same approach to evolve appropriate values for f_c and s_c. Alternatively, an entirely different strategy can be used to dynamically evolve a suitable ρ value. Several algorithms evolving such strategy parameters are suggested by Bäck in [8] (Chapter 21).

Extending the GCPSO to the *lbest* model

The GCPSO algorithm described in Section 3.2 assumed that the *gbest* version of the PSO was used. The modified update equations for the global best particle can clearly be applied directly to the neighbourhood best particle in an *lbest* swarm. Since the global best particle in the GCPSO algorithm no longer acts like the global best particle in a standard PSO algorithm, it may not be desirable to let all the neighbourhood best particles use the modified update equations. For example, a single best particle can be identified amongst all the neighbourhood bests as a candidate for using the modified update equations.

The effect of using the modified update equations on the *lbest* swarm is therefore a potential topic for future research.

Improved Global PSO algorithms

The MPSO algorithm introduced in Section 3.4.3 was shown to offer a significant improvement in performance over the standard GCPSO algorithm because it was able to escape from local minima. The major problem with the MPSO is that the algorithm could repeatedly become trapped in exactly the same minimum, since no attempt is made to explicitly avoid minima that have been detected previously. Note that the MPSO does have guaranteed convergence on the global minimiser, but that it could require an unacceptable duration of time to reach the global minimiser. This implies that the algorithm should be augmented with additional mechanisms for avoiding previously visited minima — without affecting the guaranteed global convergence property.

The MPSO algorithm can thus be extended to avoid previously visited minima using the techniques of Parsopoulos *et al.* [101, 99], or those of Beasley *et al.* [10]. Both these techniques have some undesirable side-effects in the form of false minima that they introduce as part the process of removing the previously discovered minima. Future research topics thus includes further investigation of the application of these techniques to the MPSO, as well as designing methods that do not introduce false minima.

Extending the cooperative PSO algorithms to the *lbest* model

The cooperative PSO algorithms introduced in Section 4 made use of the *gbest* PSO model. Previous research indicates that the *lbest* PSO may be more robust than the *gbest* model on multi-modal functions [37], owing to its slower rate of convergence. This could be exploited in the cooperative PSO algorithms to increase the diversity of each of the separate swarms, further increasing the robustness of the CPSO-S_K and CPSO-H_K algorithms. The major difficulty with this approach is that the construction of the context vector currently makes use of the global best particle of each of the separate swarms. Since there is no global best particle in an *lbest* PSO by default, some alternative strategy must be used. One could easily search the *lbest* swarm to find such a global best particle, or some probabilistic method can be used to select one of the neighbourhood best particles.

Future research should therefore attempt to discover an appropriate mechanism for building a suitable context vector.

Global cooperative PSO algorithms

The cooperative algorithms implemented in this thesis did not use any of the convergence-detection mechanisms of the MPSO algorithm, which implies that they were merely local search algorithms. Future research should investigate the effect of introducing convergence detection mechanisms into the cooperative algorithms. Note that the semantics of some of the convergence detection mechanisms used in the MPSO may change when applied to a CPSO algorithm. For example, the *maximum swarm radius* and *cluster analysis* algorithms can be applied amongst the particles of each of the cooperating swarms, so that each swarm can be re-started individually. In contrast, the *objective function slope* technique can be applied to the whole context vector, which implies that all the swarms must be re-started. It should also be possible to apply the objective function slope metric to each individual swarm, so that each one can be re-started independently. These options should be investigated, since the CPSO algorithms have shown that they are proficient at minimising multi-modal function.

Parallel implementations of the cooperative PSO algorithms

The CCGA-style decomposition clearly lends itself to parallel implementation. Results have been published regarding the impact of environmental parameters like network delay on such cooperative algorithms [131]. These experiments must be repeated using cooperative PSO algorithms to see whether they are affected differently by network delays or the frequency of updates to the context vector.

Dynamic CPSO-H$_K$ algorithm

The CPSO-H$_K$ algorithm introduced in Section 4.3 apportions the available budget of function evaluations to the P_j swarms and the Q swarm in a fixed manner. This was achieved by allowing the P_js one iteration of its outer loop, followed by one iteration of Q's outer loop. Assuming that functions with low inter-variable correlation are better solved by the CPSO-S$_K$ part (the P_js), the Q swarm should receive fewer processor time by only allowing it one iteration of its outer loop for every l times that the P_j outer loop is executed, where $l \in \mathbb{N}$. By monitoring the relative performance of the P_j swarms

versus the Q swarm, the value of l could be increased or decreased.

Alternatively, two values l_p and l_q can be maintained so that the P_j swarms execute their outer loop l_p times, followed by l_q iterations of the Q swarm's outer loop. Starting with initial values of $l_p = l_q = 1$, the balance can be adjusted based on the relative performance of the two components P and Q. In theory this algorithm would dynamically adapt so that the most effective type of swarm for a specific problem is always used. Investigating the effectiveness of this technique is a possible topic for future research.

Optimal swarm size

The MPSO algorithm has been proven to be able to escape from local minima. This implies that the algorithm could in theory use a smaller number of particles, since it no longer needs that many particles to avoid becoming trapped prematurely. This, together with the stronger local convergence property of the GCPSO, implies that the optimal swarm size for the MPSO may differ from that of the original PSO.

The cooperative PSOs also have more complex behaviour with regard to swarm sizes. Although some initial investigations regarding the effects of swarm size have been undertaken [137], further research could reveal a much-needed theoretical model for choosing the correct swarm size.

Overfitting neural networks

Chapter 6 investigated the performance of the PSO-based algorithms on the task of training summation and product unit networks. These experiments were designed not to incorporate any mechanisms to prevent overfitting, since only the training performance was of interest. Giles has shown that overfitting behaviour is dependent on the type of training algorithm that is used [76]. The overfitting behaviour of the different PSO algorithms should be compared with the overfitting behaviour of other network training algorithms. If the overfitting behaviour of the PSO-based algorithms differs from that of the gradient-based algorithms, alternate mechanisms for preventing overfitting must be designed specifically for the PSO-based methods.

Bibliography

[1] P. J. Angeline. Evolutionary Optimization versus Particle Swarm Optimization: Philosophy and Performance Differences. In *Evolutionary Programming VII*, volume 1447, Lecture notes in Computer Science, pages 601–610. Springer, 1998. 55

[2] P. J. Angeline. Using Selection to Improve Particle Swarm Optimization. In *Proceedings of IJCNN'99*, pages 84–89, Washington, USA, July 1999. 26, 36, 37, 38, 58, 143

[3] P. J. Angeline and J. B. Pollack. Competitive Environments Evolve Better Solutions for Complex Tasks. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 264–270, Urbana-Champaign, IL, USA, 1993. 65

[4] F. Archetti, B. Bertrò, and S. Steffè. A Theoretical Framework for Global Optimization via Random Sampling. Technical Report A-25, Cauderni del Dipartimento di Ricerca Operative e Scienze Statistiche, Università di Pisa, 1975. 125

[5] R. Axelrod. *The Evolution of Cooperation*. Basic, New York, 1984. 66

[6] R. Axelrod. Evolution of strategies in the iterated prisoner's dillema. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 32–41. Morgan Kaufmann, 1987. 66

[7] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, USA, 1996. 13

[8] T. Bäck, D. B. Fogel, and T. Michalewicz, editors. *Advanced Algorithms and Operators*, volume 2 of *Evolutionary Computation*. Institute of Physics Publishing, Bristol and Philidelphia, 1999. 11, 243

[9] T. Bäck, D. B. Fogel, and T. Michalewicz, editors. *Basic Algorithms and Operators*, volume 1 of *Evolutionary Computation*. Institute of Physics Publishing, Bristol and Philidelphia, 1999. 11, 12, 13

[10] D. Beasley, D. R. Bull, and R. R. Martin. A Sequential Niche Technique for Multimodal Function Optimization. In *Evolutionary Computation*, volume 2, pages 101–125. MIT press, 1993. 45, 47, 68, 133, 244

[11] C. M. Bishop. *Neural Networks for Pattern Recognition*, chapter 7, pages 253–294. Oxford University Press, 1995. 8, 198, 200

[12] C. Blake, E. Keogh, and C.J. Merz. UCI repository of machine learning databases, 1998. University of California, Irvine, Dept. of Information and Computer Sciences, http://www.ics.uci.edu/∼mlearn/MLRepository.html. 55, 209

[13] E. K. Blum and L. K. Li. Approximation theory and feedforward networks. In *Neural Networks*, volume 4, pages 511–515, 1991. 202

[14] R. L. Burden and J. D. Faires. *Numerical Analysis*, chapter 5.11, pages 314–321. PWS Publishing Company, Boston, fifth edition, 1993. 89

[15] E. Cantú-Paz. A Summary of Research on Parallel Genetic Algorithms. IlliGAL Report No. 95007, Illinois Genetic Algorithms Laboratory (IlliGAL), Urbana-Champaign, IL, USA, July 1995. 68, 69

[16] A. Carlisle and G. Dozier. Adapting Particle Swarm Optimization to Dynamic Environments. In *Proceedings of the International Conference on Artificial Intelligence*, pages 429–434, Las Vegas, NV, USA, 2000. 52

[17] A. Carlisle and G. Dozier. Tracking Changing Extrema with Particle Swarm Optimizer. Technical Report CSSE01-08, Auburn University, Alabama, USA, 2001. 53

[18] S. Christensen and F. Oppacher. What can we learn from No Free Lunch? A First Attempt to Characterize the Concept of a Searchable Function. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1219–1226, San Franscisco, USA, July 2001. 11

[19] S. H. Clearwater, T. Hogg, and B. A. Huberman. Cooperative Problem Solving. In *Computation: The Micro and Macro View*, pages 33–70, Singapore: World Scientific, 1992. 26, 71, 72, 77, 128, 143

[20] M. Clerc. The Swarm and the Queen: Towards a Deterministic and Adaptive Particle Swarm Optimization. In *Proceedings of the Congress on Evolutionary Computation*, pages 1951–1957, Washington DC, USA, July 1999. IEEE Service Center, Piscataway, NJ. 35, 36

[21] M. Clerc and J. Kennedy. The Particle Swarm: Explosion, Stability and Convergence in a Multi-Dimensional Complex Space. *IEEE Transactions on Evolutionary Computation*, 2001. in press. 60, 61, 77

[22] H. G. Cobb. Is the Genetic Algorithm a Cooperative Learner? In *Foundations of Genetic Algorithms 2*, pages 277–296. Morgan Kaufmann Publishers, 1992. 72, 129

[23] D. Corne, M. Dorigo, and F. Glover, editors. *New Ideas in Optimizaton*, chapter 25, pages 379–387. McGraw Hill, 1999. 23, 35, 36

[24] D. Corne, M. Dorigo, and F. Glover, editors. *New Ideas in Optimizaton*, chapter 14, pages 217–279. McGraw Hill, 1999. 77

[25] G. W. Cottrell. Extracting features from faces using compression networks: Face, identity, emotion and gender recognition using holons. In D. Touretzky, editor, *Connection Models: Proceedings of the 1990 Summer School*. Morgan Kaufmann, San Mateo, CA, USA, 1990. 198

[26] C. R. Darwin. *On the Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life*. Murray, London, 1859. (New York: Modern Library, 1967). 11

[27] L. Davis. Applying Adaptive Algorithms to Epistatic Domains. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 162–164, 1985. 17

[28] K. A. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. Ph.D. thesis, University of Michigan, Ann Arbor, MI, 1975. 68

[29] K. Deb and R. Agrawal. Simulated Binary Crossover for Continuous Search Space. *Complex Systems*, 9:115–148, 1995. 17

[30] J. E. Dennis, Jr and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, chapter 6, pages 111–152. Prentice-Hall, 1983. 8, 10

[31] L. C. W Dixon and G. P. Szegø, editors. *Towards Global Optimization*. North-Holland, Amsterdam, 1975. 10

[32] L. C. W Dixon and G. P. Szegø, editors. *Towards Global Optimization*. North-Holland, Amsterdam, 1978. 10

[33] R. Durbin and D. Rumelhart. Product Units: A Computationally Powerful and Biologically Plausible Extension to Backpropagation Networks. *Neural Computation*, 1:133–142, 1989. 202, 203

[34] R. C. Eberhart and X. Hu. Human Tremor Analysis Using Particle Swarm Optimization. In *Proceedings of the Congress on Evolutionary Computation*, pages 1927–1930, Washington D.C, USA, July 1999. IEEE Service Center, Piscataway, NJ. 21, 55

[35] R. C. Eberhart and J. Kennedy. *Swarm Intelligence*. Morgan Kaufmann, 2001. 26, 31, 32, 56, 57

[36] R. C. Eberhart and Y. Shi. Tracking and Optimizing Dynamic Systems with Particle Swarms. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 94–100, Seoul, Korea, 2001. 54, 151, 164, 173

[37] R. C. Eberhart, P. Simpson, and R. Dobbins. *Computational Intelligence PC Tools*, chapter 6, pages 212–226. Academic Press Professional, 1996. 22, 27, 29, 30, 34, 244

[38] Russ C. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, Nagoya, Japan, 1995. IEEE Service Center, Piscataway, NJ. 2, 21

[39] Russ C. Eberhart and Y. Shi. Comparing Inertia Weights and Constriction Factors in Particle Swarm Optimization. In *Proceedings of the Congress on Evolutionary Computing*, pages 84–89, San Diego, USA, 2000. IEEE Service Center, Piscataway, NJ. 36, 54, 87, 151, 164, 173, 190

[40] A. P. Engelbrecht. A New Pruning Heuristic Based on Variance Analysis of Sensitivity Information. *IEEE Transactions on Neural Networks*, 12(6), November 2001. 200

[41] A. P. Engelbrecht and A. Ismail. Training product unit neural networks. *Stability and Control: Theory and Applications*, 2(1–2):59–74, 1999. 21, 56, 203, 227, 236

[42] L. J. Eshelman and J. D. Schaffer. Real-coded genetic algorithms and interval schemata. In D. Whitley, editor, *Foundations of Genetic Algorithms II*, pages 187–202. Morgan Kaufmann, San Mateo, CA, USA, 1993. 17

[43] S. E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 524–532. Morgan Kaufmann, San Mateo, CA, USA, 1990. 200

[44] R. A. Fisher. The use of Multiple Measurements in Taxonomic Problems. *Annals of Eugenics*, 7:179–188, 1936. 55

[45] L. J. Fogel. Autonomous Automata. *Industrial Research*, (4):14–19, 1962. 15

[46] L. J. Fogel. *On the Organization of Intellect*. PhD thesis, University of California at Los Angeles, 1964. 15

[47] L. J. Fogel and D. B. Fogel. Artificial Intelligence through Evolutionary Programming. Final Report Contract PO-9-X56-1102C-1, US Army Research Institute, 1986. 15

[48] M. Friedman and L. S. Savage. Planning experiments seeking minima. In C. Eisenhart, M. W. Hastay, and W. A. Wallis, editors, *Selected Techniques of Statistical Analysis for Scientific and Industrial Research, and Production and Management Engineering*, pages 363–372. McGraw-Hill, New York, 1947. 72, 134

[49] Y. Fukuyama et al. Practical Distribution State Estimation using Hybrid Particle Swarm Optimization. In *Proceedings of the IEEE Power Engineering Society Winter Meeting*, Columbus, 2001. 58

[50] Y. Fukuyama and H. Yoshida. A Particle Swarm Optimization for Reactive Power and Voltage Control in Electric Power Systems. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 87–93, Seoul, Korea, 2001. 32, 57

[51] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA, USA, 1989. 16

[52] D. E. Goldberg and R. L. Lingle. Alleles, Loci, and the Traveling Saleman Problem. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 154–159, 1985. 17

[53] D. E. Goldberg and J. Richardson. Genetic Algorithms with Sharing for Multimodal Function Optimization. In *Proceedings of the 2nd International Conference on Genetic Algorithms*, pages 41–49, 1987. 67

[54] D.E. Goldberg, K. Deb, and J. Horn. Massive Multimodality, Deception, and Genetic Algorithms. In *Parallel Problem Solving from Nature*, volume 2, pages 37–46, Amsterdam, North-Holland, 1992. 132

[55] V. S. Gordon and D. Whitley. Dataflow parallelism in Genetic Algorithms. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature II*, pages 533–542. North-Holland, Amsterdam, 1992. 69

[56] V. S. Gordon and D. Whitley. Serial and Parallel Genetic Algorithms and Function Optimizers. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 177–183, Urbana-Champaign, IL, USA, 1993. 68

[57] J. J. Grefenstette. Deception Considered Harmful. In *Foundations of Genetic Algorithms 2*, pages 75–91. Morgan Kaufmann, 1992. 133

[58] D. M. Greig. *Optimisation*, chapter 3–4. Longman Inc., New York, USA, 1980. 6

[59] P. Grosso. *Computer Simulations of Genetic Adaption: Parallel Subcomponent Interaction in a Multilocus Model*. PhD thesis, University of Michigan, 1985. 69, 127

[60] Z. He, C. Wei, L. Yang, X. Gao, S. Yao, R. C. Eberhart, and Y. Shi. Extracting Rules from Fuzzy Neural Network by Particle Swarm Optimization. In *Proceedings of the IEEE International Conference of Evolutionary Computation*, pages 74–77, Anchorage, Alaska, USA, 1998. 56

[61] W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, pages 313–324. Addison-Wesley, Redwood City, CA, USA, 1992. 66, 75

[62] J. Holland. Outline for a logical theory of adaptive systems. *Journal of the ACM*, 3:297–314, 1962. 16

[63] J. Holland. *Adaption in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975. 16, 99

[64] A. Ismail. Training and Optimisation of Product Unit Neural Networks. submitted M.Sc thesis, Department of Computer Science, University of Pretoria, South Africa, 2001. 203, 206

[65] P. Jinchun, C. Yaobin, and R. Eberhart. Battery pack state of charge estimator design using computational intelligence approach. In *The Fifteenth Annual Battery Conference on Applications and Advances*, pages 173–177, 2000. 56

[66] L. K. Jones. Constructive approximations for neural networks by sigmoidal functions. *Proceedings of the IEEE*, 78(10):1586–1589, 1990. 202

[67] J. Kennedy. The particle swarm: Social adaption of knowledge. In *Proceedings of the International Conference on Evolutionary Computation*, pages 303–308, Indianapolis, IN, USA, 1997. 25

[68] J. Kennedy. Small Worlds and Mega-Minds: Effects of Neighbourhood Topology on Particle Swarm Performance. In *Proceedings of the Congress on Evolutionary Computation*, pages 1931–1938, Washington DC, USA, July 1999. IEEE Service Center, Piscataway, NJ. 40

[69] J. Kennedy. Stereotyping: Improving Particle Swarm Performance With Cluster Analysis. In *Proceedings of the Congress on Evolutionary Computing*, pages 1507–1512, San Diego, USA, 2000. IEEE Service Center, Piscataway, NJ. 41

[70] J. Kennedy and R. C. Eberhart. Particle Swarm Optimization. In *Proceedings of IEEE International Conference on Neural Networks*, volume IV, pages 1942–1948, Perth, Australia, 1995. IEEE Service Center, Piscataway, NJ. 2, 21, 27, 55

[71] J. Kennedy and R. C. Eberhart. A Discrete Binary Version of the Particle Swarm Algorithm. In *Proceedings of the Conference on Systems, Man, and Cybernetics*, pages 4104–4109, 1997. 31

[72] J. Kennedy and W. M. Spears. Matching Algorithms to Problems: An Experimental Test of the Particle Swarm and Some Genetic Algorithms on the Multimodal Problem Generator. In *Proceedings of the International Conference on Evolutionary Computation*, pages 78–83, Anchorage, Alaska, 1998. 32

[73] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983. 33

[74] J. Koza. *Genetic Programming: on the Programming of Computers by Means of Natural Selection and Genetics*. MIT Press, Cambridge, MA, USA, 1992. 17

[75] K. J. Lang, A. H. Waibel, and G. E. Hinton. A time-delay neural network architecture for isolated word recognition. *Neural Networks*, (3):33–43, 1990. 198

[76] S. Lawrence and C. L. Giles. Overfitting and Neural Networks: Conjugate Gradient and Backpropagation. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, Como, Itally, July 2000. 204, 246

[77] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4), 1989. 198

[78] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 598–605. Morgan Kaufmann, San Mateo, CA, USA, 1990. 200

[79] M. Løvbjerg, T. K. Rasmussen, and T. Krink. Hybrid Particle Swarm Optimiser with Breeding and Subpopulations. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, San Francisco, USA, July 2001. 38, 39, 45, 143

[80] B. Manderick and P. Spiessens. Fine-grained parallel genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, Fairfax, WA, USA, 1989. 69, 128

[81] M. J. Matarić. Designing and Understanding Adaptive Group Behavior. *Adaptive Behavior*, 4:51–80, 1995. 28

[82] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998. 150

[83] E. Mayr. *Animal Species and Evolution*. Belknap, Cambridge, MA, 1963. 12

[84] S. Milgram. The small world problem. *Psychology Today*, 22:61–67, 1967. 40

[85] M. M. Millonas. Swarms, phase transitions, and collective intelligence. In *Artificial Life III*, pages 417–445. Addison-Wesley, Reading, MA, 1994. 28

[86] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Portland, Oregon, USA, 1997. 198

[87] M. F. Möller. A scaled conjugate gradient algorithm for fast supervised learning. In *Neural Networks*, volume 6, pages 525–533, 1993. 8, 279

[88] H. Mülenbein. Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, Fairfax, WA, USA, 1989. 69, 128

[89] R. H. J. M. Otten and L. P. P. P. van Ginneken. *The Annealing Algorithm*. Kluwer, Boston, USA, 1989. 33

[90] E. Ozcan and C. K. Mohan. Analysis of a Simple Particle Swarm Optimization System. In *Intelligent Engineering Systems Through Artificial Neural Networks*, volume 8, pages 253–258, 1998. 59

[91] E. Ozcan and C. K. Mohan. Particle Swarm Optimization : Surfing the Waves. In *Proceedings of the International Congress on Evolutionary Computation*, pages 1939–1944, Washington, USA, 1999. 59, 60, 77, 87

[92] J. Paredis. Coevolutionary constraint satisfaction. In *The 3rd Conference on Parallel Problem Solving from Nature*, Jerusalem, Israel, October 1994. 67

[93] J. Paredis. Steps towards co-evolutionary classification neural networks. In R. Brooks and P. Maes, editors, *Artificial Life IV*, pages 102–108. MIT Press, Cambridge, MA, USA, 1994. 66

[94] J. Paredis. The Symbiotic Evolution of Solutions and their Representations. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 359–365, Pittsburgh, PA, USA, 1995. 68

[95] J. Paredis. Coevolutionary evolution. *Artificial Life Journal*, 2:255–375, 1996. 66

[96] J. Paredis. Symbiotic evolution for epistatic functions. In *Proceedings of the 12th European Conference on Artificial Intelligence*, pages 228–232, Budapest, Hungary, August 1996. 68

[97] K. E. Parsopoulos, V. P. Plagianakos, G. D. Magoulas, and M. N. Vrahatis. Improving Particle Swarm Optimizer. In P. Pardalos, editor, *Advances in Convex Analysis and Global Optimization*. Kluwer Academic, 2001. to appear. 47

[98] K. E. Parsopoulos, V. P. Plagianakos, G. D. Magoulas, and M. N. Vrahatis. Objective function "Stretching" to Alleviate Convergence to Local Minima. *Nonlinear Analysis, Theory and Applications*, 2001. in press. 47

[99] K. E. Parsopoulos, V. P. Plagianakos, G. D. Magoulas, and M. N. Vrahatis. Stretching Technique for Obtaining Global Minimizers Through Particle Swarm Optimization. In *Proceedings of the Particle Swarm Optimization Workshop*, pages 22–29, Indianapolis, USA, 2001. 47, 244

[100] K. E. Parsopoulos and M. N. Vrahatis. Particle Swarm Optimizer in Noisy and Continuously Changing Environments. In M. H. Hamza, editor, *Artificial Intelligence and Soft Computing*, pages 289–294. IASTED/ACTA, Anaheim, CA, USA, 2001. 52

[101] K. E. Parsopoulos and M.N. Vrahatis. Modification of the Particle Swarm Optimizer for Locating all the Global Minima. In V. Kurkova, N. C. Steele, R. Neruda, and M. Karny, editors, *Artificial Neural Networks and Genetic Algorithms*, pages 324–327. Springer, 2001. 47, 244

[102] B. A. Pearlmutter. Fast exact multiplication by the Hessian. *Neural Computing*, 6(1):147–160, 1994. 280

[103] J. Pearsall, editor. *The Concise Oxford Dictionary, 10$^{\text{th}}$ edition*. Clays Ltd, Bungay, Suffolk, 1999. 206

[104] E. Polak. *Optimization: Algorithms and Consistent Approximations*. Springer-Verlag, New York, USA, 1997. 8

[105] M. A. Potter. *The Design and Analysis of a Computational Model of Cooperative Coevolution*. PhD thesis, George Mason University, Fairfax, Virginia, USA, 1997. 69, 71, 72, 128, 142, 171, 174, 208

[106] Mitchell A. Potter and Kenneth A. de Jong. A Cooperative Coevolutionary Approach to Function Optimization. In *The Third Parallel Problem Solving from Nature*, pages 249–257, Jerusalem, Israel, 1994. Springer-Verlag. 69, 70, 77, 128, 142, 174, 208, 241, 263

[107] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, second edition, 1992. 23

[108] I. Rechenberg. Cybernetic Solution Path of an Experimental Problem. Library translation No 1122, Royal Aircraft Establishment, Farnborough, UK, 1965. 15

[109] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der Biologischen Evolution*. Frommann-Holzboog, Stuttgart, 1973. 15

[110] W. T. Reeves. Particle Systems — A Technique for Modeling a Class of Fuzzy Objects. In *SIGGRAPH 83*, pages 359–376, 1983. 29

[111] C. W. Reynolds. Flocks, Herds and Schools: A Distributed Behavioral Model. In *Computer Graphics (Proceedings of the ACM SIGGRAPH Conference)*, number 21(4), pages 25–34, Anaheim, CA, USA, July 1987. 27

[112] P. Royston. An Extension of Shapiro and Wilk's W Test for Normality to Large Samples. *Applied Statistics*, (31):115–124, 1982. 206

[113] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, pages 318–362. MIT Press, 1986. 278

[114] J. Salerno. Using the particle swarm optimization technique to train a recurrent neural model. In *Proceedings of the Ninth IEEE International Conference on Tools with Artificial Intelligence*, pages 45–49, 1997. 55

[115] R. Salomon. Reevaluating genetic algorithm performance under coordinate rotation of benchmark functions. *BioSystems*, (39):263–278, 1996. 171, 182

[116] J. D. Schaffer, R. A. Caruana, L. J. Eshelman, and R. Das. A Study of Control Parameters Affecting Online Performance of Genetic Algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 51–60, Fairfax, WA, USA, 1989. 18

[117] C. Schumacher, M. D. Vose, and L. D. Whitley. The No Free Lunch and Problem Description Length. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 565–570, San Franscisco, USA, July 2001. 11

[118] H-P. Schwefel. *Kybernetische Evolution als Strategie der experimentellen Forschung in der Strømungstechnik*. Diplomarbeit, Techniche Universität, Berlin, 1965. 15

[119] H-P. Schwefel. Numerische Optimierung von Computer-Modellen mittels der Evolutionsstraregie. In *Interdisciplinary Systems Research*. Birkhäuser, Basel, 1977. 15

[120] Y. Shi and R. C. Eberhart. Parameter Selection in Particle Swarm Optimization. In *Evolutionary Programming VII: Proceedings of EP 98*, pages 591–600, 1998. 33

[121] Y. Shi and R. C. Eberhart. Empirical Study of Particle Swarm Optimization. In *Proceedings of the Congress on Evolutionary Computation*, pages 1945–1949, Washington D.C, USA, July 1999. IEEE Service Center, Piscataway, NJ. 21, 33

[122] Y. Shi and R. C. Eberhart. Fuzzy Adaptive Particle Swarm Optimization. In *Proceedings of the Congress on Evolutionary Computation*, Seoul, Korea, 2001. 34

[123] Y. Shi and R. C. Eberhart. Particle Swarm Optimization with Fuzzy Adaptive Inertia Weight. In *Proceedings of the Workshop on Particle Swarm Optimization*, Indianapolis, IN, USA, 2001. Purdue School of Engineering and Technology, IUPUI. 34

[124] Y. Shi and Russ C. Eberhart. A Modified Particle Swarm Optimizer. In *IEEE International Conference of Evolutionary Computation*, Anchorage, Alaska, May 1998. 21, 32, 33

[125] A. Slomson. *An Introduction to Combinatorics*, chapter 6.4. Chappman and Hall Mathematics, 1991. 79, 269, 270

[126] J. A. Snyman. A New and Dynamic Method for Unconstrained Minimization. *Applied Mathematical Modeling*, 6:449–462, 1982. 56

[127] J. A. Snyman. An Improved Version of the Original LeapFrog Dynamic Method for Unconstrained Minimization: LFOP1(b). *Applied Mathematical Modeling*, 7:216–218, June 1983. 56

[128] F. Solis and R. Wets. Minimization by Random Search Techniques. *Mathematics of Operations Research*, 6:19–30, 1981. 77, 102, 103, 105, 106, 125

[129] R. V. Southwell. *Relaxation Methods in Theoretical Physics*. Clarendon Press, Oxford, UK, 1946. 72, 134

[130] W. M. Spears. Simple Subpopulation Schemes. In *Proceedings of the Evolutionary Programming Conference*, pages 296–307, 1994. 44

[131] R Subbu and A. C. Sanderson. Modeling and Convergence Analysis of Distributed Coevolutionary Algorithms. In *Proceedings of the IEEE International Congress on Evolutionary Computation*, San Diego, CA, USA, July 2000. 77, 129, 245

[132] P. N. Suganthan. Particle Swarm Optimizer with Neighbourhood Operator. In *Proceedings of the Congress on Evolutionary Computation*, pages 1958–1961, Washington DC, USA, July 1999. IEEE Service Center, Piscataway, NJ. 39, 40

[133] V. Tandon. Closing the gap between CAD/CAM and optimized CNC end milling. Master's thesis, Purdue School of Engineering and Technology, Indianapolis, IN, USA, 2000. 56

[134] J. C. Taylor. *An Introduction to Measure and Probability*. Springer-Verlag New York, Inc., 175 Fith Avenue, New York, NY, 10010, USA, 1997. 103, 104

[135] F. van den Bergh. Particle Swarm Weight Initialization in Multi-layer Perceptron Artificial Neural Networks. In *Development and Practice of Artificial Intelligence Techniques*, pages 41–45, Durban, South Africa, September 1999. 21, 55

[136] F. van den Bergh and A. P. Engelbrecht. Cooperative Learning in Neural Networks using Particle Swarm Optimizers. *South African Computer Journal*, (26):84–90, November 2000. 21, 134, 143

[137] F. van den Bergh and A. P. Engelbrecht. Effects of Swarm Size on Cooperative Particle Swarm Optimisers. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 892–899, San Francisco, USA, July 2001. 246

[138] F. van den Bergh and A. P. Engelbrecht. Training Product Unit Networks using Cooperative Particle Swarm Optimisers. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, pages 126–132, Washington DC, USA, July 2001. 173, 207, 208, 209

[139] H. Viktor, A.P. Engelbrecht, and I. Cloete. Incorporating Rule Extraction from ANNs into a Cooperative Learning Environment. In *NEURAP 98: Neural Networks & Their Applications*, pages 385–391, Marseilles, France, 1998. 57

[140] H. L. Viktor. *Learning by Cooperation: An Approach to Rule Induction and Knowledge Fusion*. PhD thesis, Department of Computer Science, University of Stellenbosch, South Africa, 1999. 127

[141] T. P. Vogl, J. K. Mangis, A. K. Rigler, W. T. Zink, and D. L. Alkon. Accelerating the convergence of the back-propagation method. *Biological Cybernetics*, (59):257–263, 1988. 278

[142] Watts and Strogetz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998. 41

[143] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioural Sciences*. PhD thesis, Harvard University, Boston, MA, USA, 1974. 278

[144] D. H. Wolpert and W. G. Macready. No Free Lunch Theorems for Search. Technical Report SFI-TR-95-02-010, Santa Fe Institute, July 1995. 10

[145] D. H. Wolpert and W. G. Macready. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation*, (4):67–82, 1997. 10

[146] C. Zhang and H. Shao. An ANN's Evolved by a New Evolutionary System and Its Application. In *Proceedings of the 39th IEEE Conference on Decision and Control*, volume 4, pages 3562–3563, 2000. 56

Appendix A

Glossary

This appendix provides brief descriptions of the commonly used terms occurring in this thesis.

CCGA: The Cooperative Coevolutionary Genetic Algorithm, introduced by Potter [106]. This algorithm partitions the search space into disjoint subspaces, and then uses a collection of cooperating subpopulations to optimise each of the subspaces.

Convergence, converged: A term used loosely to indicate that an algorithm has reached the point where it does not appear to make any further progress. In the more formal sense, a sequence $\{z_k\}_{k=1}^{+\infty}$ is said to converge when the limit

$$\lim_{k \to +\infty} z_k$$

exists. If the limit does not exit, the sequence is said to be *divergent*. When a particle's trajectory is described as convergent, the term is used in the strict sense as applied to the position of the particle converging onto some limit.

Cooperative/coevolutionary algorithms: Algorithms, usually of the evolutionary family, that use a collection of interacting populations. The fitness of individuals in one population depend on the state of the other populations, coercing the populations to either cooperate or to compete, depending on the type of fitness function employed.

CPSO: A Cooperative Particle Swarm Optimiser. This family includes two variations of the PSO, based loosely on the CCGA cooperation model.

GA: A Genetic Algorithm. A type of Evolutionary Algorithm relying mostly on its recombination operator, regarding its mutation operator as a background operator. This algorithm is also characterised by weak selection pressure, usually implemented using a fitness-proportionate selection operator.

CGPSO: Guaranteed Convergence Particle Swarm Optimiser. This is a version of the PSO algorithm that is guaranteed to converge on a local minimum of the objective function, with asymptotic probability one.

Evolutionary algorithms: A family of algorithms that simulate natural evolution, most often on a computer. This includes population-based algorithms that use random variation and selection to create new solutions. Instances of this paradigm includes Evolution Strategies, Evolutionary Programming and Genetic Algorithms.

GD: Gradient Descent algorithm. A technique which finds a local minimum by moving over the function surface along the direction of steepest descent, as determined by the gradient of the function at that point.

Global minimiser, global minimum: The global minimum is the function value $f(\mathbf{x}^*)$, where $\mathbf{x}^*$ denotes the global minimiser, defined as

$$f(\mathbf{x}^*) \leq f(\mathbf{x}),\ \forall \mathbf{x} \in S$$

where S denotes the search space.

Global search algorithm: An optimisation algorithm that locates the global minimum (or maximum) of the objective function, in all of the search space.

Local minimiser, local minimum: A local minimiser, $\mathbf{x}_B^*$, of the region B, is defined so that

$$f(\mathbf{x}_B^*) \leq f(\mathbf{x}),\ \forall \mathbf{x} \in B$$

where $B \subset S$, and S denotes the search space.

Local search algorithm: An optimisation algorithm that locates the local minimum (or maximum) of a region $B \subset S$. The local minimum is not necessarily the minimum of the search space S; it is merely the minimum of the region B, where B is typically defined to contain only a single minimum.

MSE: Mean Squared Error. A metric used to compute, amongst other things, the difference between the output of a Neural Network and the desired output value specified in the data set. Mathematically, the MSE can be be computed between two sequences a_i and b_i using

$$MSE = \frac{1}{2n}\sum_{i=1}^{n}(a_i - b_i)^2$$

where n denotes the number of elements in the sequences. Note that the sum is normalised using a factor of $1/(2n)$ rather than $1/n$, by convention.

MPSO: Multi-start Particle Swarm Optimiser. This is a PSO algorithm with guaranteed convergence onto the global minimiser of the objective function, with asymptotic probability one. This is achieved by periodically re-initialising the positions of the particles.

Neural network: A configurable mapping between an input space and an output space. These networks can represent arbitrary mappings by suitable adjustment of their weights (configurable parameters). Typical uses include classification problems, and approximating arbitrary functions when only sampled data points are available.

No Free Lunch theorem: A theorem that contends that all optimisation algorithms perform equally well when their performance is amortised over the set of all possible functions. This implies that all algorithms perform on average as well as a random search (without replacement), or a straight enumeration of all possible solutions.

Objective function: The function that is optimised during an optimisation process, to compute either the set of parameters yielding the minimum or the maximum function value.

PSO: Particle Swarm Optimiser. A sociologically inspired, stochastic, population-based optimisation algorithm. The algorithm maintains a population of particles that interact with each other while exploring the search space.

RPSO: Randomised Particle Swarm Optimiser. Another theoretically global PSO algorithm. This algorithm contains several particles that continually sample the whole search space, in search of better solutions.

SCG: Scaled Conjugate Gradient descent algorithm. A specialisation of the Gradient Descent algorithm which constructs better directions of search, and performs line searches along those directions. This algorithm has a faster rate of convergence than the basic GD algorithm.

Appendix B

Definition of Symbols

This appendix lists the commonly-used symbols found throughout this thesis

The number of the page on which a symbol first appears can be found by consulting the index under the appropriate entry.

$\mathbf{x}_i$: Particle i's *current position*. index: position, current;

$\mathbf{y}_i$: Particle i's *personal best position*. index: position, personal best;

$\mathbf{v}_i$: Particle i's *current velocity*. index: velocity;

$\hat{\mathbf{y}}$: The swarm's *global best position*. index: position, global best;

$\hat{\mathbf{y}}_i$: The best position in the neighbourhood of particle i. index: position, local best;

r_1, r_2: Uniform pseudo-random numbers. index: random sequences;

s: The swarm size, or number of particles. index: swarm size;

c_1, c_2: The two acceleration coefficients. index: acceleration coefficients;

f: Denotes the function being minimised. Takes a vector input and returns a scalar value.

w: The *inertia weight*, affecting the PSO velocity update equation. index: inertia weight;

τ: The index of the global best particle, used in the GCPSO algorithm. index: GCPSO;

k, t: These variables denote time or time steps;

K: This variable denotes the *split factor* , used in the CPSO-S$_K$ and CPSO-H$_K$ algorithms. index: split factor;

Appendix C

Derivation of Explicit PSO Equations

This appendix provides a derivation of the closed-form PSO equations used in Chapter 3.

Consider the PSO update equations, repeated here for convenience:

$$\begin{aligned} v_{i,j}(t+1) &= wv_{i,j}(t) + c_1 r_{1,j}(t)[y_{i,j}(t) - x_{i,j}(t)] + \\ &\quad c_2 r_{2,j}(t)[\hat{y}_j(t) - x_{i,j}(t)] \end{aligned} \tag{C.1}$$

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1) \tag{C.2}$$

To simplify the notation the derivation will be performed in one dimension only, using a single particle. This allows us to drop both the i and j subscripts, further implying that x_t denotes the value of x at time step t, rather than the value x associated with particle number t. The stochastic component will also be removed temporarily with the substitutions $\phi_1 = c_1 r_1(t)$ and $\phi_2 = c_2 r_2(t)$. Further, the position of the particle will be considered in discrete time only, resulting in the following equations:

$$v_{t+1} = wv_t + \phi_1(y_t - x_t) + \phi_2(\hat{y}_t - x_t) \tag{C.3}$$

$$x_{t+1} = x_t + v_{t+1} \tag{C.4}$$

Substituting (C.3) into (C.4) results in

$$x_{t+1} = x_t + wv_t + \phi_1(y_t - x_t) + \phi_2(\hat{y}_t - x_t) \tag{C.5}$$

Grouping the related terms yields

$$x_{t+1} = (1 - \phi_1 - \phi_2)x_t + \phi_1 y_t + \phi_2 \hat{y}_t + w v_t \tag{C.6}$$

But, from equation (C.5) it is known that

$$x_t = x_{t-1} + v_t$$

Thus,

$$v_t = x_t - x_{t-1} \tag{C.7}$$

Substituting (C.7) into (C.6) and grouping the terms yield

$$x_{t+1} = (1 + w - \phi_1 - \phi_2)x_t - w x_{t-1} + \phi_1 y_t + \phi_2 \hat{y}_t \tag{C.8}$$

which is a non-homogeneous recurrence relation that can be solved using standard techniques [125]. This recurrence relation can be written as a matrix-vector product, so that

$$\begin{bmatrix} x_{t+1} \\ x_t \\ 1 \end{bmatrix} = \begin{bmatrix} 1 + w - \phi_1 - \phi_2 & -w & \phi_1 y + \phi_2 \hat{y} \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_t \\ x_{t-1} \\ 1 \end{bmatrix} \tag{C.9}$$

The characteristic polynomial of the matrix in (C.9) is

$$(1 - \lambda)(w - \lambda(1 + w - \phi_1 - \phi_2) + \lambda^2) \tag{C.10}$$

which has a trivial root of $\lambda = 1.0$, and two other solutions

$$\alpha = \frac{1 + w - \phi_1 - \phi_2 + \gamma}{2} \tag{C.11}$$

$$\beta = \frac{1 + w - \phi_1 - \phi_2 - \gamma}{2} \tag{C.12}$$

where

$$\gamma = \sqrt{(1 + w - \phi_1 - \phi_2)^2 - 4w} \tag{C.13}$$

Note that α and β are both eigenvalues of the matrix in equation (C.9). The explicit form of the recurrence relation (C.8) is then given by

$$x_t = k_1 + k_2 \alpha^t + k_3 \beta^t \tag{C.14}$$

where k_1, k_2 and k_3 are constants determined by the initial conditions of the system [125]. Since there are three unknowns, a system of three equations must be constructed to find their values. The initial conditions of the PSO provides two such conditions, x_0 and x_1, corresponding to the position of the particle at time steps 0 and 1. Note that this is equivalent to specifying the initial position x_0 and the initial velocity v_0. The third constraint of the system can be calculated using the recurrence relation to find the value of x_2, thus

$$x_2 = (1 + w - \phi_1 - \phi_2)x_1 - wx_0 + \phi_1 y_t + \phi_2 \hat{y}_t$$

From these three initial conditions, the system

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & \alpha & \beta \\ 1 & \alpha^2 & \beta^2 \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \\ k_3 \end{bmatrix} \tag{C.15}$$

is derived, which can be solved using Gauss-elimination, yielding

$$\begin{aligned} k_1 &= \frac{\alpha\beta x_0 - x_1(\alpha + \beta) + x_2}{(\alpha - 1)(\beta - 1)} \\ k_2 &= \frac{\beta(x_0 - x_1) - x_1 + x_2}{(\alpha - \beta)(\alpha - 1)} \\ k_3 &= \frac{\alpha(x_1 - x_0) + x_1 - x_2}{(\alpha - \beta)(\beta - 1)} \end{aligned}$$

Using the property $\alpha - \beta = \gamma$, these equations can be further simplified to yield

$$k_1 = \frac{\phi_1 y_t + \phi_2 \hat{y}_t}{\phi_1 + \phi_2} \tag{C.16}$$

$$k_2 = \frac{\beta(x_0 - x_1) - x_1 + x_2}{\gamma(\alpha - 1)} \tag{C.17}$$

$$k_3 = \frac{\alpha(x_1 - x_0) + x_1 - x_2}{\gamma(\beta - 1)} \tag{C.18}$$

Note that both y_t and $\hat{y}_t$ are dependent on the time step t. These values may change with every time step, or they may remain constant for long durations, depending on the objective function and the position of the other particles. Whenever either y_t or $\hat{y}_t$ changes the values of k_1, k_2 and k_3 must be recomputed. It is possible to extrapolate the

trajectory of a particle by holding both y and $\hat{y}$ constant. This implies that the positions of the other particles remain fixed, and that the particle does not discover any better solutions itself.

Although these equations were derived under the assumption of discrete time, no such restriction is necessary. Equation (C.14) can be expressed in continuous time as well, resulting in

$$x(t) = k_1 + k_2\alpha^t + k_3\beta^t \tag{C.19}$$

using the same values for k_1, k_2 and k_3 as derived above for the discrete case.

Appendix D

Function Landscapes

This appendix presents three-dimensional plots of the various synthetic benchmark functions used in Chapter 5.

All the functions are drawn inverted, so that the minimum of the function appears as a maximum. Owing to the shape of these functions, this representation make visible more clearly the nature of the function.

All the functions are plotted as they are defined in Section 5.1 (page 148), using the domains specified in Table 5.1 (page 150). The only exception is Griewank's function, which is plotted in the domain $[-30, 30]^2$, since the fine detail is not visible in the plot when viewed using the full domain of $[-600, 600]^2$.

The unimodal nature of the Spherical, Rosenbrock and Quadric functions is clearly visible in Figures D.1, D.2 and D.7. The massively multi-modal nature of the Ackley, Rastrigin, Griewank and Schwefel functions can be observed in Figures D.3, D.4, D.5 and D.6.

Please turn page ...

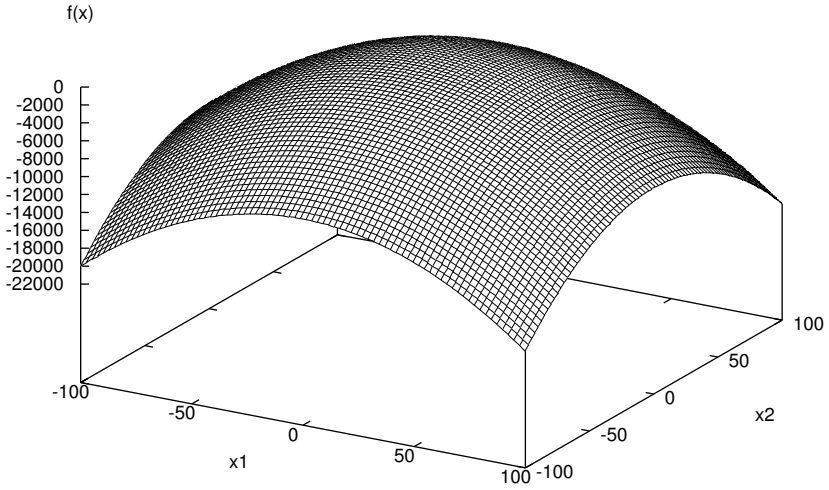


Figure D.1: The Spherical function

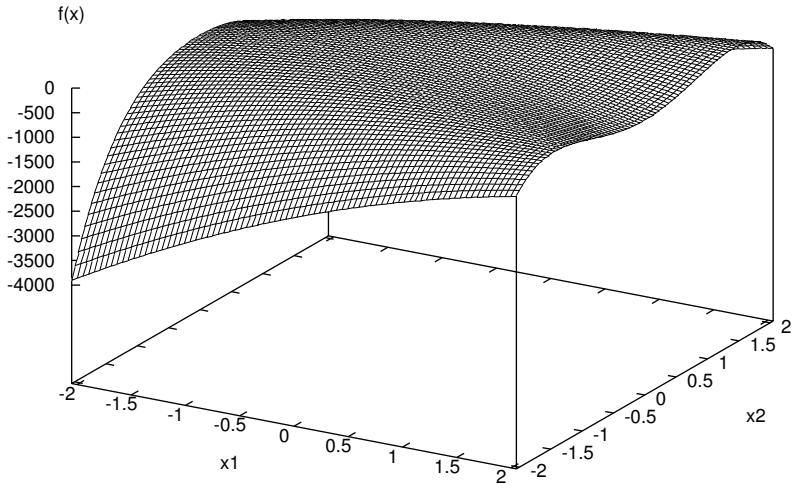


Figure D.2: Rosenbrock's function

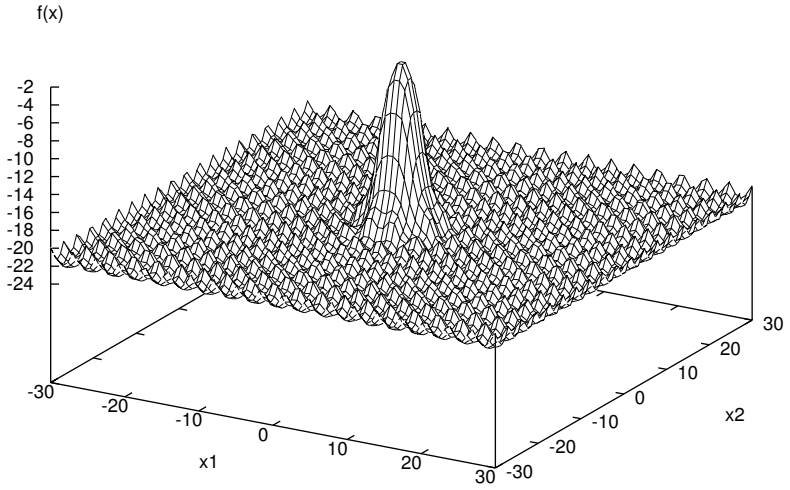


Figure D.3: Ackley's function

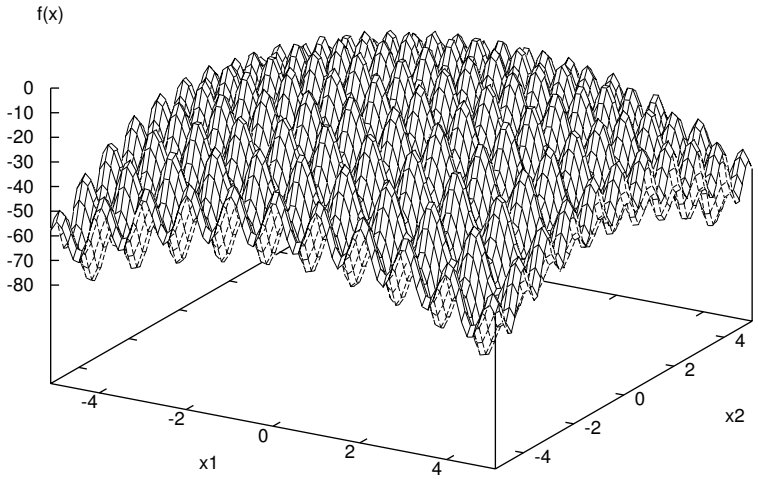


Figure D.4: Rastrigin's function

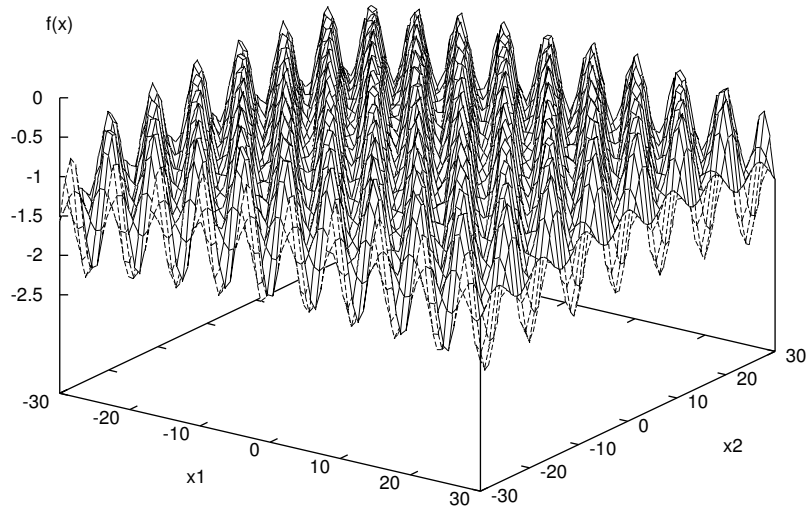


Figure D.5: Griewank's function

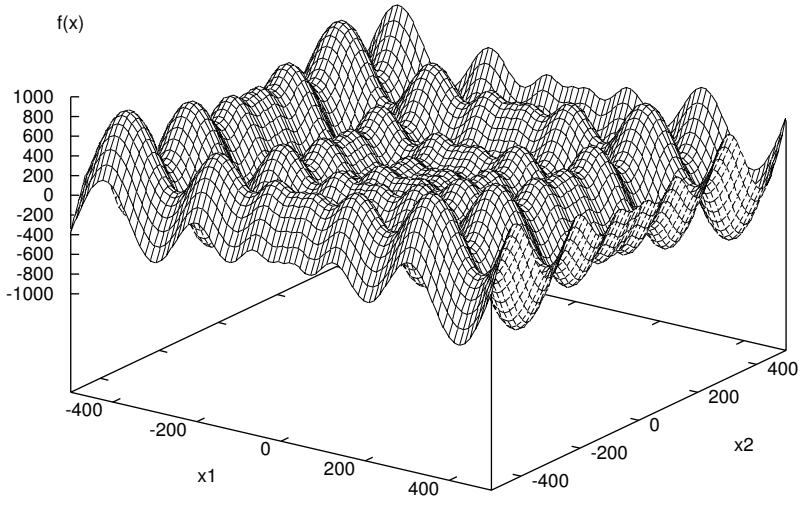


Figure D.6: Schwefel's function

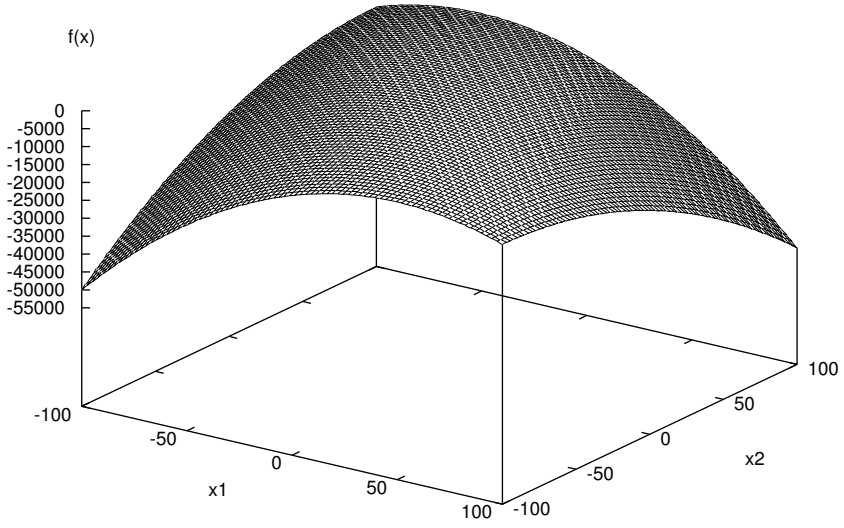


Figure D.7: The Quadric function

Appendix E

Gradient-based Search Algorithms

This appendix briefly describes two efficient gradient-based optimisation algorithms commonly used to train summation unit networks.

An important part of gradient-based training algorithms is the choice of initial values for the weight vector $\mathbf{w}$. Usually, random values from the distribution

$$w_i \sim U\left(-\frac{1}{\sqrt{fan\_in}}, \frac{1}{\sqrt{fan\_in}}\right)$$

are used, where fan_in is the in-degree of the unit, *i.e.* the number of weights entering the unit.

The next step in the training process is to apply an algorithm that will find a weight vector $\mathbf{w}$ that minimizes the error function E. Since the weight vector changes during the training process, the vector $\mathbf{w}_t$ will be used to indicate the value of the vector $\mathbf{w}$ at time step t.

E.1 Gradient Descent Algorithm

One of the simplest summation unit network training algorithms is known as the Gradient Descent (GD) algorithm, also referred to as the *steepest descent* algorithm.

The algorithm computes the gradient of the error surface at the current location in search space, starting with an initial weight vector that is randomly chosen, as described

above. The gradient at the current location will be denoted $\mathbf{g}$, so that

$$\mathbf{g} \equiv \nabla E(\mathbf{w}_t) \tag{E.1}$$

where $E(\mathbf{w}_t)$ is the error function evaluated as position $\mathbf{w}_t$ in the weight space. The error back-propagation technique, originally due to Werbos [143], its use further advocated by Rumelhart *et al.* [113], is used to calculate $\mathbf{g}$. This technique only requires $\mathcal{O}(W)$ operations, where W is the number of dimensions in $\mathbf{w}$. This is significantly faster (and more accurate) than the finite-difference approach, which requires $\mathcal{O}(W^2)$ operations.

Geometrically, the vector $\mathbf{g}$ points in the direction in which the slope of the error surface is the steepest, away from the minimum. If an algorithm is to take a sufficiently small step in the direction of $-\mathbf{g}$, then the error at this new point will be smaller. The value of the weight vector is thus updated at each iteration using the rule

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \mathbf{g} \tag{E.2}$$

where η_t is called the *learning rate*. It is customary to use a sequence of η_t values that decreases with increasing t values, since this guarantees that the $\mathbf{w}_t$ sequence is convergent. The larger the learning rate, the further the algorithm will move in one step, with the obvious danger of stepping over a minimum so that the error value actually increases if η_t is too large.

The GD algorithm used in this thesis implemented a variation known as the *bold driver* technique [141]. This technique learns the appropriate learning rate while it is minimising the error function. Specifically, the implementation used in Chapter 6 used the following rule:

$$\eta_{t+1} = \begin{cases} 1.1\eta_t & \text{if } \Delta E \leq 0 \\ 0.5\eta_t & \text{if } \Delta E > 0 \end{cases}$$

where $\Delta E \equiv E(\mathbf{w}_t) - E(\mathbf{w}_{t-1})$ represents the change in the value of the error function between steps $t-1$ and t. The algorithm keeps a copy of $\mathbf{w}_{t-1}$, the weight vector at time $t-1$. If $\Delta E > 0$ it restores the old weight vector by setting $\mathbf{w}_t = \mathbf{w}_{t-1}$, and halves the learning rate. This ensures that the algorithm never takes an uphill step. It will also continually increase the learning rate as long as it manages to decrease the error.

Several problems remain with the GD algorithm, including that the direction of steepest descent, $-\mathbf{g}$, may not be the optimal search direction. This issue is addressed

by a significantly more powerful algorithm known as the Scaled Conjugate Gradient (SCG) algorithm, presented next.

E.2 Scaled Conjugate Gradient Descent Algorithm

The scaled conjugate gradient algorithm, introduced by Möller [87], does not search along the direction of steepest descent. Instead, it constructs a *conjugate direction*, $\mathbf{d}$, using

$$\mathbf{d}_{t+1} = -\mathbf{g}_{t+1} + \beta_t \mathbf{d}_t \tag{E.3}$$

where β_t is computed using the *Polak-Ribiere* formula

$$\beta_t = \frac{\mathbf{g}_{t+1}^T(\mathbf{g}_{t+1} - \mathbf{g}_t)}{\mathbf{g}_t^T \mathbf{g}_t} \tag{E.4}$$

Once the direction of search has been determined, the algorithm attempts to take a step along this direction that will minimise the error along this line. In other words, it minimises the function $E(\mathbf{w} + \alpha\mathbf{d})$ by computing the appropriate value for α.

The scaled conjugate training algorithm achieves this by using a local quadratic approximation of the error surface, which is obtained by performing an n-dimensional Taylor expansion around the current position in weight space. This approximation, with explicit reference to the time step t omitted, is as follows:

$$E(\mathbf{w}) \approx E_0 + \mathbf{b}^T\mathbf{w} + \frac{1}{2}\mathbf{w}^T\mathbf{H}\mathbf{w}$$

The derivative of the above approximation, with respect to $\mathbf{w}$, can be found by evaluating

$$\mathbf{g}(\mathbf{w}) = \mathbf{b} + \mathbf{H}\mathbf{w}.$$

Note that $\mathbf{g}$ is the first derivative of E with respect to the weight vector $\mathbf{w}$, as defined in equation (E.1); $\mathbf{H}$ is the *Hessian matrix* — the second order derivative of E with respect to $\mathbf{w}$, so that $\mathbf{H} \equiv \nabla^2 E(\mathbf{w})$.

The error function E is minimized along the direction $\mathbf{d}_t$ by finding α_t using

$$\alpha_t = -\frac{\mathbf{d}_t^T \mathbf{g}_t}{\mathbf{d}_t^T \mathbf{H}\mathbf{d}_t + \lambda_t ||\mathbf{d}_t||^2} \tag{E.5}$$

The minimum of the error function along the direction $\mathbf{d}_t$ is then

$$E(\mathbf{w} + \alpha_t \mathbf{d}_t) \tag{E.6}$$

The λ term in equation (E.5) controls the *region of trust* radius. The region of trust parameter is adjusted according to how well the local quadratic approximation describes the error surface. This is measured using the decision variable

$$\Delta_t = \frac{2E(\mathbf{w}_t) - E(\mathbf{w}_t + \alpha_t \mathbf{d}_t)}{\alpha_t \mathbf{d}_t^T \mathbf{g}_t}$$

The value of λ_t is then adjusted using

$$\lambda_{t+1} = \begin{cases} \lambda_t/2 & \text{if } \Delta_t > 0.75 \\ 4\lambda_t & \text{if } \Delta_t < 0.25 \\ \lambda_t & \text{otherwise} \end{cases}$$

This process represents a single step of the training algorithm. After the weight vector $\mathbf{w}$ has been updated using

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \mathbf{d}_t$$

the process is repeated by re-evaluating equations (E.3)–(E.6). The algorithm is guaranteed to reduce the error at each step.

Note that the above algorithm requires the calculation of the first and second order derivatives of the error function. If the weight vector has W dimensions, then the finite difference (perturbation) approach to calculating the derivative will require W^2 forward propagations through the network. The Hessian will require W^3 steps, which quickly becomes computationally intractable with larger W values.

The expensive Hessian evaluation is replaced by the fast Hessian-vector product technique proposed by Pearlmutter [102], which is used to calculate $\mathbf{H}\mathbf{d}_t$. Note that the GD algorithm still takes significantly less time to perform one iteration compared to SCG, since the SCG algorithm requires one extra forward and backward propagation through the network to compute the value of $\mathbf{H}\mathbf{d}_t$.

Appendix F

Derived Publications

This appendix lists all the papers that have been published, or are currently being reviewed, that were derived from the work leading to this thesis.

1. F. van den Bergh. Particle Swarm Weight Initialization in Multi-layer Perceptron Artificial Neural Networks. In *Development and Practice of Artificial Intelligence Techniques*, pages 41–45, Durban, South Africa, September 1999.

2. F. van den Bergh and A. P. Engelbrecht. Cooperative Learning in Neural Networks using Particle Swarm Optimizers. *South African Computer Journal*, (26):84–90, November 2000.

3. F. van den Bergh and A. P. Engelbrecht. Effects of Swarm Size on Cooperative Particle Swarm Optimisers. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 892–899, San Francisco, USA, July 2001.

4. F. van den Bergh and A. P. Engelbrecht. Training Product Unit Networks using Cooperative Particle Swarm Optimisers. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, pages 126–132, Washington DC, USA, July 2001.

5. F. van den Bergh and A. P. Engelbrecht. A cooperative approach to particle swarm optimisation. *IEEE Transactions on Evolutionary Computation*. Submitted December 2000.

6. F. van den Bergh, A. P. Engelbrecht, and D. G. Kourie. A convergence proof for the particle swarm optimiser. *IEEE Transactions on Evolutionary Computation*. Submitted September 2001.

Index