Particle Swarm Optimizer for Finding Robust Optima

J.K. Vis
jvis@liacs.nl

June 23, 2009

ABSTRACT

Many real-world processes are subject to uncertainties and noise.
Robust Optimization methods seek to obtain optima that are robust to noise and it can deal with uncertainties in the problem definition. In this thesis we will investigate if Particle Swarm Optimizers (PSO) are suited to solve problems for robust optima. A set of standard benchmark functions will we used to test two PSO algorithms – Canonical PSO and Fully Informed Particle Swarm – against other optimization approaches. Based on observations of the behaviour of these PSOs our aim is to develop improvements for solving for robust optima. The emphasis lies on finding appropriate neighbourhood topologies.

CONTENTS

1. INTRODUCTION

In real-world applications of optimization techniques it is often the case that a optimum needs to be robust, that is, even in the presence of noise on the input and output variables the solution should be guaranteed to have a high quality. Robust Design Optimization deals with finding such robust optima.

The Particle Swarm Optimization algorithm (PSO) is a recently proposed metaheuristic for optimization in vector spaces. It works with a swarm of simple agents related to search points that collectively search for the optimum, thereby simulating the behavior of natural swarms searching for food.

This Bachelor Thesis seeks to investigate whether PSO has inherent advantages when searching for robust optima and how it compares to Evolutionary Algorithms that are also frequently used in that field. Moreover it has to be found whether PSO can be modified in a way that makes it better suitable for finding robust optima.

1.1 Objectives

The primary objectives of this thesis are summarized as follows:

- Implementation of two common PSO variants (Canonical PSO and Fully Informed Particle Swarm) in MATLAB using a standard optimization template. These algorithms are intended to be part of a optimization package for MATLAB.
- Comparison of these variants against Evolutionary Algorithms on standard and robust design benchmarks.
- Modifications to these variants for finding robust optima.

- Comparison of the modifications against the standard algorithms on the same benchmarks.

1.2 Thesis Outline

Chapter 2 starts with an introduction to the mathematical principles of optimization. From there we define a simple continuous optimization problem. The last section of chapter 2 is dedicated to robust optimization. We introduce some mathematical definitions and conclude with defining measurements for robustness.
In chapter 3, we will discuss the development of the Particle Swarm Optimizer briefly before presenting two common algorithms (Canonical PSO and Fully Informed Particle Swarm).
Chapter 4 focusses on the two sets of benchmark problems that are used in the empirical performance study in chapter 5. Each test problem is briefly described and its domain and dimensionality are defined.
In chapter 5, we will present the results of the empirical performance study on both sets of benchmark problems.
Chapter 6 presents our conclusions.
In the appendices, we present the MATLAB code for the PSO algorithms used in this thesis as well as a overview of a comparison of solution quality between different PSOs found in literature.

2. OPTIMIZATION

Optimization, in mathematics, deals with searching for the optimal solution(s) for a given problem. The optimal solution(s) must be chosen from a set of possible solutions, the so-called search space. In general, this set of possible solutions is very large. In this thesis, we will focus on a particular subset of optimization problems. We will only consider optimization problems with exactly one objective function. This is called single-objective optimization as opposed to multi-objective optimization. Furthermore, we will not consider any constraints on either input or output parameters. We assume that the uncontrollable parameters are modeled within the system and are therefore not explicitly present. The search space is confined to real-valued parameters which are often limited by box-constraints (lower and upper bound on parameter values).
Our mathematical definition of a simple optimization problem:

Definition 1. *A simple continuous optimization problem is defined as a pair* (S, f) *where:*

- S *the set of all possible solutions:* $S = \mathbb{R}^N$ *where* N *is the number of controllable parameters.*
- f *a single objective function:* $f : S \to \mathbb{R}$ *that needs to be optimized.*

Where a solution vector $\vec{x}$ is often limited between a lower and upper bound: $\vec{x}_{lb} \leq \vec{x} \leq \vec{x}_{ub}$.

In optimization, we distinguish two types: maximization and minimization. In maximization, we search for a solution that is greater or equal than all other solutions. In minimization, we search for the opposite: a solution that is smaller or equal than all other solutions.

Definition 2. *The set of maximal solutions $S_{max} \subseteq S$ of a function $f : S \to \mathbb{R}$ is defined as:*

$$\vec{x}_{max} \in S_{max} \Leftrightarrow \forall \vec{x} \in S : f(\vec{x}_{max}) \geq f(\vec{x})$$

Definition 3. *The set of minimal solutions $S_{min} \subseteq S$ of a function $f : S \to \mathbb{R}$ is defined as:*

$$\vec{x}_{min} \in S_{min} \Leftrightarrow \forall \vec{x} \in S : f(\vec{x}_{min}) \leq f(\vec{x})$$

Although both problems (maximization and minimization) occur it is clear from the definitions 2 and 3 that we can easily transform a maximization problem into a minimization problem and vice versa.
In this thesis, we choose to solve minimization problems. Therefore, we define a conversion function to transform a maximization problem to a minimization problem. Assume f is a function to be maximized then we can minimize the function $\alpha \cdot f$ where $\alpha \in \mathbb{R}$ and $\alpha < 0$. We choose α to be -1.

2.1 Robust Optimization

In real-world optimization, problems there are a number of other factors that influence the optimization process. Most notably these factors are uncertainties and noise. For example in practical applications, it may be very difficult to control the parameters $\vec{x}$ with an arbitrary precision. Another example: assume that the system uses sensors to determine the quality of the solution. These sensors are likely to be limited in their precision and produce, therefore, inaccurate results.
We give the following definition for robust optimization:

Definition 4. *[Kruisselbrink (2009)] Robust optimization deals with the optimization on models or systems in which uncertainty and/or noise is present and where the optimization process needs to account for such uncertainties in order to obtain optimal solutions that are also meaningful in practice.*

As there are many types of noise and uncertainties we will only consider robustness against input parameters.

Definition 5. *[Paenke et al. (2006)] The expected quality of a solution $\vec{x}$ is defined as:*

$$f_{exp} = \int_{-\infty}^{\infty} f\left(\vec{x} + \vec{\delta}\right) \cdot pdf\left(\vec{\delta}\right) \, d\vec{\delta}$$

Where $\vec{\delta}$ is distributed according to the probability density function pdf $\left(\vec{\delta}\right)$: $\vec{\delta} \sim$ pdf.

Definition 6. *[Paenke et al. (2006)] The quality variance of a solution $\vec{x}$ is defined as:*

$$f_{var} = \int_{-\infty}^{\infty} \left(f\left(\vec{x} + \vec{\delta}\right) - f_{exp}(\vec{x}) \right)^2 \cdot pdf\left(\vec{\delta}\right)\, d\delta$$

It is impossible to compute the functions 5 and 6 analytically for complex problems. Therefore, we use Monte Carlo integration by sampling over a number of instances of $\vec{\delta}$. This method has as major drawback that, as each sample corresponds to a objective function evaluation, the number of objective function evaluations dramatically increases. When objective function evaluations are expensive this approach is not viable. However, as this thesis focusses on theoretical problems, we do not care about this increase in objective function evaluations. So we use Monte Carlo integration as measure for robustness:

$$f_{\text{effective}} = \frac{1}{m} \sum_{i=1}^{m} f\left(\vec{x} + \vec{\delta_i}\right) \tag{2.1}$$

Where m denotes the number of samples.

As opposed to the Monte Carlo integration measure 2.1 we can also use worst-case robustness as a measure:

$$f_{\text{worst case}} = \min_{i=1}^{m} \left(f\left(\vec{x} + \vec{\delta_i}\right) \right) \tag{2.2}$$

For all empirical experiments presented in chapter 5 we use: $m = 10,000$.

3. PARTICLE SWARM OPTIMIZER

A Particle Swarm Optimizer (PSO) is a nature-inspired swarm intelligence algorithm. Swarm intelligence is a problem-solving technique that relies on interactions of simple processing units (also known in artificial intelligence as agents). The notion of a swarm implies multiple units that are capable of interacting with each other resulting in a complex behavior. The notion of intelligence suggests that this approach is successful. In the following section, we will describe the general characteristics of PSOs as well as the canonical form of the algorithm and a common variant: the Fully Informed Particle Swarm.

The initial ideas of Kennedy and Eberhart [Eberhart and Kennedy (1995)] were to combine cognitive abilities with social interaction. They were inspired by the work of Heppner and Grenander [Heppner and Grenander (1990)] in which they studied natures flocks of birds, schools of fish and swarms of insects. In 1995, these ideas were developed into the Particle Swarm Optimizer. Since then many different applications and variants have been developed [Poli et al. (2007)].

Every PSO uses a population of particles. The number of particle in a swarm is typically far less than the number of individuals in an evolutionary algorithm. A particle in this population is interconnected to other particles. This interconnection is called the neighborhood topology. Neighborhood refers to a communication structure rather than a geographical neighborhood. To use these particles to explore the search space we need a so-called change rule. This rule moves the particles through the search space at a given moment t in time depending on its position at moment $t-1$ as well as the position of its previous best location. This is the cognitive aspect of the PSO. The social aspect is introduced by an interaction rule. A particles position is not only dependent on its own best position in history, but also on the best position in history of its neighbors.

3.1 Canonical Particle Swarm Optimizer

In algorithm 1 each particle is composed of three N-dimensional vectors, where N is the dimensionality of the search space and a real-value:

- $\vec{x}_i$ the current position in the search space of particle i,
- $\vec{p}_i$ the best position in history of particle i,
- $\vec{v}_i$ the velocity of particle i,
- $pbest_i$ the quality of the solution of the best position of particle i,

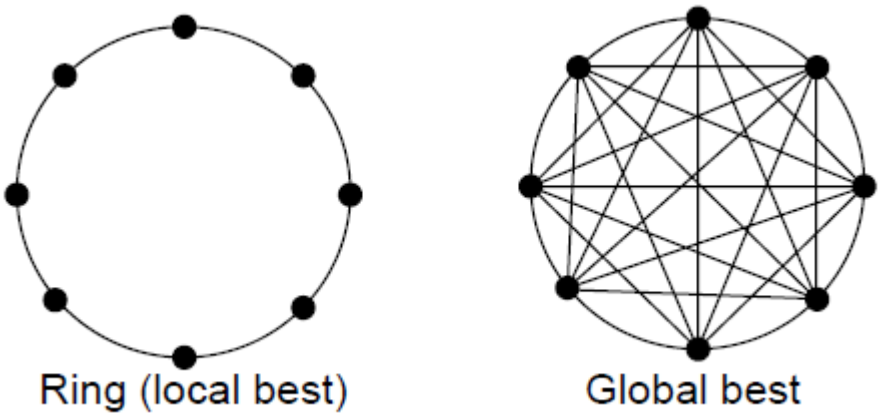

Fig. 3.1: Two different neighborhood topologies

3.1.1 Parameters

This algorithm uses a few parameters. First, the designer has to specify the number of particles in the swarm: λ. Typical values range from $\lambda = 20$ to $\lambda = 50$. As is stated by Carlisle and Dozier [Carlise and Dozier (2001)] $\lambda = 30$ seems to be a good choice for problems with dimensionality $N = 30$. This number is small enough to be efficient and produces reliable good quality results, but it may be adjusted for the perceived difficulty of the problem at hand. As a guide rule choose the number of particles proportional to the

problems dimensionality.
Two parameters, φ_1 and φ_2, responsible for the behavior of the swarm. These parameters are often called acceleration coefficients, because they control the magnitude of the adjustments towards the particles personal best and its global best. φ_1 controls the cognitive aspect (adjustment towards the personal best), while φ_2 controls the social aspect (adjustment towards the global best). For the canonical parameter settings, we use: $\varphi_1 = 2.05$ and $\varphi_2 = 2.05$ as is most commonly found in literature. The constant χ is called the constriction coefficient (introduced by Clerc and Kennedy [Clerc and Kennedy (2002)]). A constriction coefficient is necessary to keep velocities under control, as they would quickly increase to unacceptable levels within a few iterations. χ is dependent on φ_1 and φ_2, which eliminates the need to set another parameter [Clerc and Kennedy (2002)]:

$$\chi = \frac{2}{\varphi - 2 + \sqrt{\varphi^2 - 4\varphi}} \tag{3.1}$$

Where $\varphi = \varphi_1 + \varphi_2 > 4$.

Another major influence of the behavior of a PSO is its neighborhood topology. A particle in the population is interconnected to other particles. This interconnection is called the neighborhood topology. Note that this neighborhood is not defined in the Euclidian search space. Instead it is preset connectivity relation between the particles (see figure 3.1). Two topologies are commonly used from the development of PSOs called: global best and local best. Where the global best topology is a fully interconnected population in which every particle can be influenced by every other particle in the swarm. The local best topology is a ring lattice. Every particle is connected to two immediate neighboring particles in the swarm. The advantage of this structure is that parts of the population can converge at different optima. This, while it is mostly slower to converge, is less likely to converge at local sub-optima. Typically, topologies are symmetric and reflexive.

3.1.2 Initialization

As there exists no better way to position the particle in the search space they are most commonly initialized uniformly random within the search space. If one chooses to initialize the velocities to a vector of zeroes then $\vec{p}_i$ should be different from $\vec{x}_i$ to enable the particles to start moving, but commonly

$\vec{x}_i$ and $\vec{v}_i$ are initialized randomly while $\vec{p}_i$ is initialized as $\vec{x}_i$ for the first iteration. The nonzero velocities move the particles through the search space in a randomly chosen direction and magnitude.
$pbest_i$ contains the objective function value of the best position of a particle. At initialization, it is set to infinity to always allow an improvement in the first iteration.

3.1.3 Loop

The first phase is to evaluate all particles and update their personal bests as required. The variable g denotes the particles global best (in the defined neighborhood topology). The second phase is to adjust the positions of the particles. First the velocities are updated by the so-called change rule:

$$\vec{v}_i = \chi \left(\vec{v}_i + \vec{U}(0, \varphi_1) \otimes (\vec{p}_i - \vec{x}_i) + \vec{U}(0, \varphi_2) \otimes (\vec{p}_{g_{nbh(i)}} - \vec{x}_i) \right) \tag{3.2}$$

Where:

- $\vec{U}(0, \varphi_i)$ represents a vector (of size N) of uniformly distributed random values between 0 and φ_i.
- $\otimes$ denotes component-wise multiplication, e.g.
$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \otimes \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} x_1 \cdot y_1 \\ x_2 \cdot y_2 \\ x_3 \cdot y_3 \end{pmatrix}$$
- $g_{nbh(i)}$ is a reference to the best particle in the neighborhood of particle i.

Then the positions are updated using the following rule:

$$\vec{x}_i = \vec{x}_i + \vec{v}_i \tag{3.3}$$

In algorithm 1 all particle's personal bests (and global bests within their neighborhood) are updated first. Once all updates are performed the particles are moved. These are called synchronous updates as opposed to asynchronous updates, where once the personal best is updated the particle is immediately moved. All algorithms used in this thesis use synchronous updates as this strategy is generally better in finding global optima. A stop

criterion is used to end the iterations of the loop. Ideally we stop when the particle swam has converged to the global minimum, but in general, it is hard to tell when this criterion is met. So in practice a fixed number of iterations or a fixed number of objective function evaluations is commonly used. Alternatively, the algorithm could stop when a sufficiently good solution is found.

Algorithm 1 Canonical Particle Swarm Optimizer

{Initialization}
for $i = 1$ to λ **do**
  $\vec{x}_i \sim \vec{U}(\vec{x}_{lb}, \vec{x}_{ub})$
  $\vec{v}_i \sim \vec{U}(\vec{x}_{lb}, \vec{x}_{ub})$
  $\vec{p}_i = \vec{x}_i$
  $pbest_i = \infty$
end for

{Loop}
while termination condition not true **do**
  for $i = 1$ to λ **do**
    {Update personal best positions}
    if $f(\vec{x}_i) < pbest_i$ **then**
      $\vec{p}_i = \vec{x}_i$
      $pbest_i = f(\vec{x}_i)$
    end if
    {Update best particle in each neighborhood}
    if $f(\vec{x}_i) < pbest_{g_{nbh(i)}}$ **then**
      $g_{nbh(i)} = i$
    end if
  end for
  {Update velocities and positions}
  for $i = 1$ to λ **do**
    $\vec{v}_i = \chi \left(\vec{v}_i + \vec{U}(0, \varphi_1) \otimes (\vec{p}_i - \vec{x}_i) + \vec{U}(0, \varphi_2) \otimes (\vec{p}_{g_{nbh(i)}} - \vec{x}_i) \right)$
    $\vec{x}_i = \vec{x}_i + \vec{v}_i$
  end for
end while

3.2 Fully Informed Particle Swarm

A popular variant of the canonical PSO is the Fully Informed Particle Swarm (FIPS) as introduced by Kennedy and Mendes [Kennedy and Mendes (2002)] (algorithm 2). They noted that there was no reason not to subdivide the acceleration coefficients into two components (personal best and neighborhood best). Instead, they use a single component for all particles in the neighborhood. In this manner, information of the total neighborhood is used as opposed to information from the best particle alone. The remaining part of the algorithm is the same as the algorithm for the canonical PSO.
The FIPS algorithm does not perform very well while using the global best topology, or, in general, with any neighborhood topology with a high degree of interconnections (a particle is interconnected to many other particles in the swarm). FIPS performs better at topologies with a lower degree such as the local best (ring lattice) topology or topologies where particles have very few (approximately three) neighbors. Intuitively, it seems obvious that information from many neighbors can result in conflicting situations, since these neighbors may have found their successes in different regions of the search space. Therefore, this averaged information is less likely to be helpful as opposed to the canonical PSO where more neighbors will tend to yield better information, as it is more likely to have a particle with a high solution quality in the neighborhood.

3.2.1 Parameters

We eliminated the need for two different acceleration coefficients and replaced them with a single coefficient φ. As we still use Clercs constriction method (explained above), φ is set to 4.1.

3.2.2 Initialization

Initialization procedure for the Fully Informed Particle Swarm is exactly the same as for the Canonical Particle Swarm Optimizer 3.1.

3.2.3 Loop

In the first phase only the personal best positions are updated. There is no need to determine a global best as all particles in a neighborhood influence all

other particles in that neighborhood. The change rule for the Fully Informed Particle Swarm:

$$\vec{v}_i = \chi \left(\vec{v}_i + \sum_{n=1}^{N_i} \frac{\vec{U}(0,\varphi) \otimes (\vec{p}_{nbh(n)} - \vec{x}_i)}{N_i} \right) \tag{3.4}$$

Where:

- $\vec{U}(0,\varphi)$ represents a vector (of size N) of uniformly distributed random values between 0 and φ_i.
- $\otimes$ denotes component-wise multiplication.
- $nbh(n)$ is the particle's n-the neighbor.
- N_i represents the number of neighbors of the particle.

The positions are updated according to 3.3.

Algorithm 2 Fully Informed Particle Swarm

{Initialization}
for $i = 1$ to λ **do**
  $\vec{x}_i \sim \vec{U}(\vec{x}_{lb}, \vec{x}_{ub})$
  $\vec{v}_i \sim \vec{U}(\vec{x}_{lb}, \vec{x}_{ub})$
  $\vec{p}_i = \vec{x}_i$
  $pbest_i = \infty$
end for

{Loop}
while stop criterion not met **do**
  {Update personal best positions}
  for $i = 1$ to λ **do**
    if $f(\vec{x}_i) < pbest_i$ **then**
      $\vec{p}_i = \vec{x}_i$
      $pbest_i = f(\vec{x}_i)$
    end if
  end for
  {Update velocities and positions}
  for $i = 1$ to λ **do**
    $\vec{v}_i = \chi \left(\vec{v}_i + \sum_{n=1}^{N_i} \frac{\vec{U}(0,\varphi) \otimes (\vec{p}_{nbh(n)} - \vec{x}_i)}{N_i} \right)$
    $\vec{x}_i = \vec{x}_i + \vec{v}_i$
  end for
end while

4. BENCHMARK PROBLEMS

In order to test our algorithms we used two sets of (theoretical) test problems. These test problems are widely used in other publications and especially designed to test different properties of optimization algorithms. The first set consists of general benchmark functions which are described in section 4.1. These functions are predominantly used for testing the actual optimization behavior of the algorithms. The Second set of test problems is especially designed for robust optimization. These functions are described in section 4.2.
All benchmark functions are scalable in dimensionality unless otherwise stated. For each function we give a short description, its mathematical representation and the dimensionality and domain (generally) used in this thesis as well as a visual representation in two dimensions. If the benchmark problem is a maximization problem we use the conversion factor: $\alpha = -1$.

4.1 General Benchmark Functions

The following functions are used to test the general performance of our PSO algorithms. In figures 4.1 and 4.2 a two dimensional representation is given for both the entire domain of the search space as well as a close-up of the region around the global optimum.

Sphere: A very simple, unimodal function with its global minimum located at $\vec{x}_{min} = \vec{0}$, with $f(\vec{x}_{min}) = 0$. This function has no interaction between its variables. Defined on: $N = 30, -100 \leq x_i \leq 100, i \in N$.

$$f(\vec{x}) = \sum_{i=1}^{N} x_i^2 \tag{4.1}$$

Rozenbrock: A unimodal function, with significant interaction between some of the variables. Its global minimum of $f(\vec{x}_{min}) = 0$ is located at $\vec{x}_{min} = \vec{1}$. Defined on: $N = 30, -100 \leq x_i \leq 100, i \in N$.

$$f(\vec{x}) = \sum_{i=1}^{N} 100 \left(x_{i+1} - x_i^2\right)^2 + (x_i - 1)^2 \tag{4.2}$$

Griewank: A multi-modal function with significant interaction between its variables, caused by the product term. The global minimum, $\vec{x}_{min} = \vec{0}$, yields a function value of $f(\vec{x}_{min}) = 0$. Defined on: $N = 30, -600 \leq x_i \leq 600, i \in N$.

$$f(\vec{x}) = \frac{1}{4000} \sum_{i=1}^{N} x_i^2 - \prod_{i=1}^{N} \cos \frac{x_i}{\sqrt{i}} + 1 \tag{4.3}$$

Rastrigin: A multi-modal version of the Spherical function, characterised by the deep local minima arranged as sinusoidal bumps. The global minimum is $f(\vec{x}_{min}) = 0$, where $\vec{x}_{min} = \vec{0}$. The variables of this function are independent. Defined on: $N = 30, -10 \leq x_i \leq 10, i \in N$.

$$f(\vec{x}) = \sum_{i=1}^{N} x_i^2 - 10 \cos 2\pi x_i + 10 \tag{4.4}$$

Schaffer's f6: A multimodal function with a global minimum: $\vec{x_{min}} = \vec{0}$ and $f(\vec{x}_{min}) = 0$. Defined on: $N = 2$ (not scalable), $-100 \leq x_i \leq 100, i \in N$.

$$f(\vec{x}) = 0.5 - \frac{\left(\sin \sqrt{x_1^2 + x_2^2}\right)^2 - 0.5}{\left(1.0 + 0.001 \left(x_1^2 + x_2^2\right)\right)^2} \tag{4.5}$$

Ackley: A multi-modal function with deep local minima. The global minimum is $f(\vec{x}_{min}) = 0$, with $\vec{x_{min}} = \vec{0}$. Note that the variables are independent. Defined on: $N = 30, -32 \leq x_i \leq 32, i \in N$.

$$f(\vec{x}) = -20 * e^{-0.2\sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}} - e^{\frac{1}{N}\sum_{i=1}^{N}\cos(2\pi x_i)} + 20 + e \tag{4.6}$$

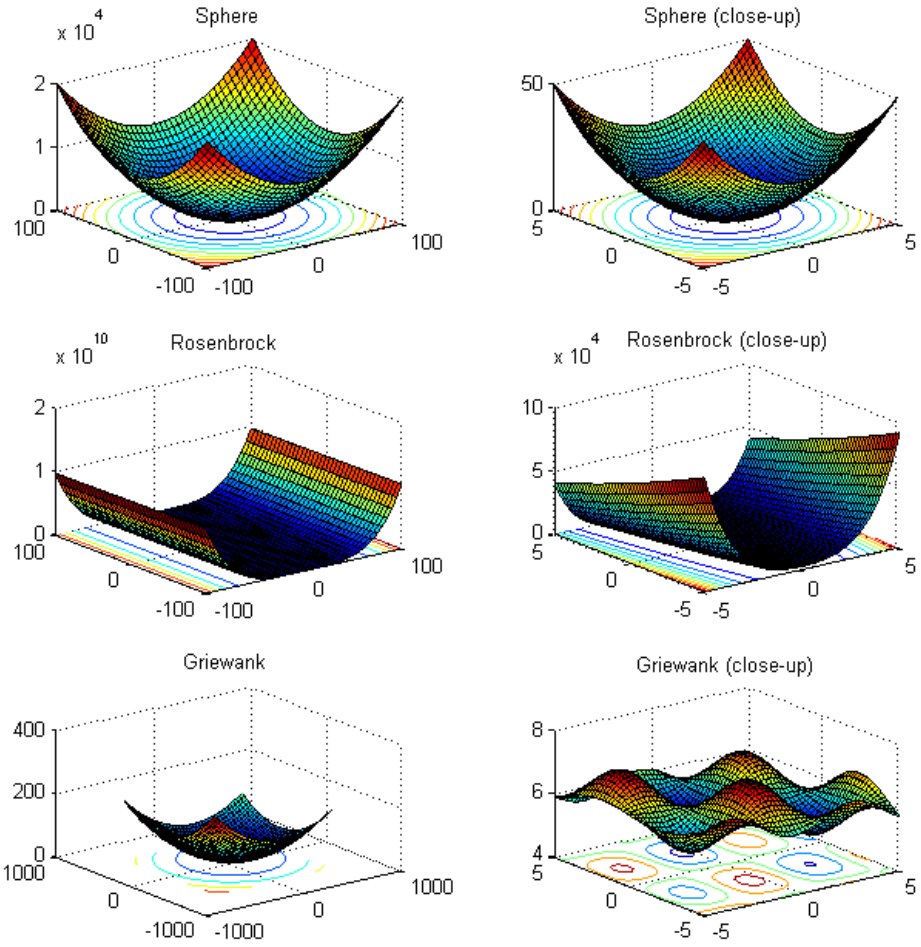

Fig. 4.1: General test functions in two dimensions (1)

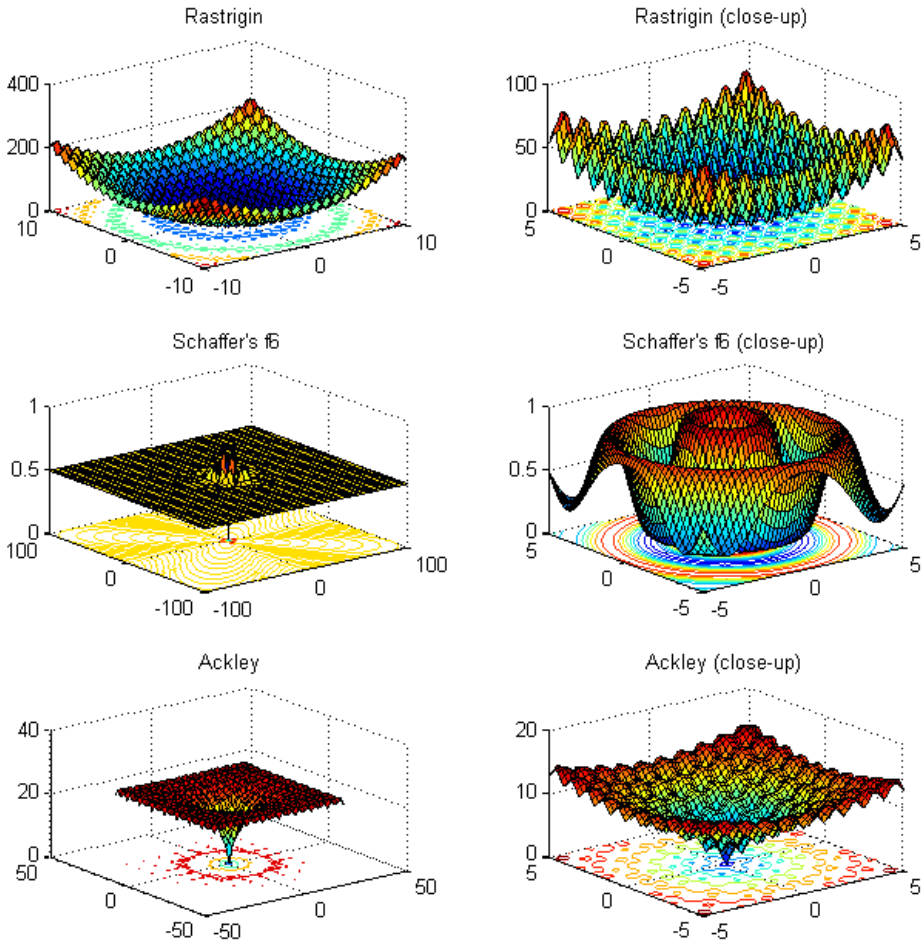

Fig. 4.2: General test functions in two dimensions (2)

4.2 Robust Design Benchmark Functions

In this section we use mostly the functions presented by Branke [Branke (2001)]. For every test function f_i, we give the exact definition for a single dimension, denoted by $\hat{f}_i$. These are scaled to N dimensions by summation

over all dimensions:

$$f_i(\vec{x}) = \sum_{j=1}^{N} \hat{f}_i(x_j) \tag{4.7}$$

As the noise model we use the pdf $\left(\vec{\delta}\right)$: $\vec{\delta} \sim \vec{U}(-0.2, 0.2)$ as is used by Branke [Branke (2001)].

Test Function f_1: In each dimension there are two alternatives: a high sharp peak and a smaller less sharp peak. In a deterministic setting the sharp peak would be preferable, but the smaller peak has a higher effective fitness. Defined on: $N = 20, -2 \leq x_j \leq 2, j \in N$.

$$\hat{f}_1(x_j) = \begin{cases} -(x_j + 1)^2 + 1.4 - 0.8|\sin(6.283 \cdot x_j)| & \text{if } -2 \leq x_j < 0 \\ 0.6 \cdot 2^{-8|x_j - 1|} + 0.958887 - 0.8|\sin(6.283 \cdot x_j)| & \text{if } 0 \leq x_j < 2 \\ 0 & \text{otherwise} \end{cases} \tag{4.8}$$

Test Function f_2: This is a very simple function as it consists of a single peak. Defined on: $N = 20, -0.5 \leq x_j \leq 0.5, j \in N$.

$$\hat{f}_2(x_j) = \begin{cases} 1 & \text{if } -0.2 \leq x_j < 0.2 \\ 0 & \text{otherwise} \end{cases} \tag{4.9}$$

Test Function f_3: Designed to show the influence of asymmetry on the robust optimization behaviour. It consists of a single peak with a sharp drop on one side. Defined on: $N = 20, -1 \leq x_j \leq 1, j \in N$.

$$\hat{f}_3(x_j) = \begin{cases} x_j + 0.8 & \text{if } -0.8 \leq x_j < 0.2 \\ 0 & \text{otherwise} \end{cases} \tag{4.10}$$

Test Function f_4: This function has two peaks in very dimension. The height of each peak can be set by the parameters h_l and h_r as well as the ruggedness of the peaks by the parameters η_l and η_r. Defined on: $N = 20, -2 \leq x_j \leq 2, j \in N$.

$$\hat{f}_4(x_j) = \begin{cases} g(h_r)(x_j-1)^2 + h_r + 10\eta_r\left(x_j - 0.2\left\lfloor\frac{10x_j}{2}\right\rfloor\right) - \frac{3}{2}\eta_r & \text{if } 0.1 < x_j \leq 1.9 \quad \wedge \\ & \lfloor 10x_j \rfloor \bmod 2 = 1 \\ g(h_r)(x_j-1)^2 + h_r - 10\eta_r\left(x_j - 0.1 - 0.2\left\lfloor\frac{10x_j}{2}\right\rfloor\right) - \frac{1}{2}\eta_r & \text{if } 0.1 < x_j \leq 1.9 \quad \wedge \\ & \lfloor 10x_j \rfloor \bmod 2 = 0 \\ g(h_l)(x_j+1)^2 + h_l + 10\eta_l\left(x_j + 2 - 0.2\left\lfloor\frac{10x_j}{2}\right\rfloor\right) - \frac{3}{2}\eta_l & \text{if } -1.9 < x_j \leq -0.1 \quad \wedge \\ & \lfloor -10x_j \rfloor \bmod 2 = 1 \\ g(h_l)(x_j+1)^2 + h_l - 10\eta_l\left(x_j + 1.9 - 0.2\left\lfloor\frac{10x_j}{2}\right\rfloor\right) - \frac{1}{2}\eta_l & \text{if } -1.9 < x_j \leq -0.1 \quad \wedge \\ & \lfloor -10x_j \rfloor \bmod 2 = 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

with $h_l = 1.0, h_r = 1.0, \eta_l = 0.5, \eta_r = 0.0$ and $g(h) = \frac{657}{243} - h\frac{900}{243}$.

Test Function f_5: Again two peaks in every dimension with a very similar effective fitness value. Defined on: $N = 20, -1.5 \leq x_j \leq 1.5, j \in N$.

$$\hat{f}_5(x_j) = \begin{cases} 1 & \text{if } -0.6 \leq x_j < -0.2 \\ 1.25 & \text{if } (0.2 \leq x_j < 0.36) \vee (0.44 \leq x_j < 0.6) \\ 0 & \text{otherwise} \end{cases} \tag{4.12}$$

4.2.1 A Simple Test Function

In contrast to the functions described in section 4.2 we needed a simpler function to modify our PSO algorithms for robustness. We designed our own test function which was used for that purpose (for use with 4.7):

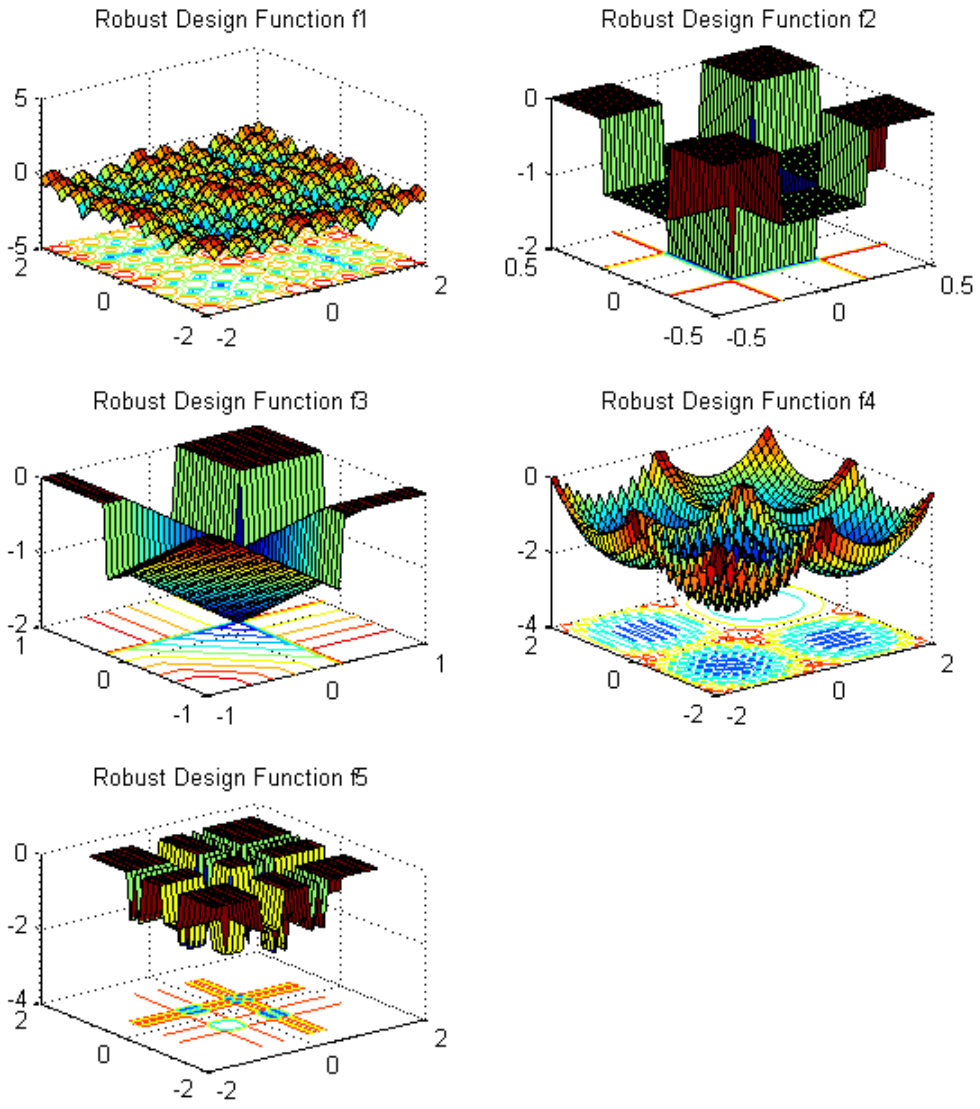

Fig. 4.3: Robust test functions in two dimensions

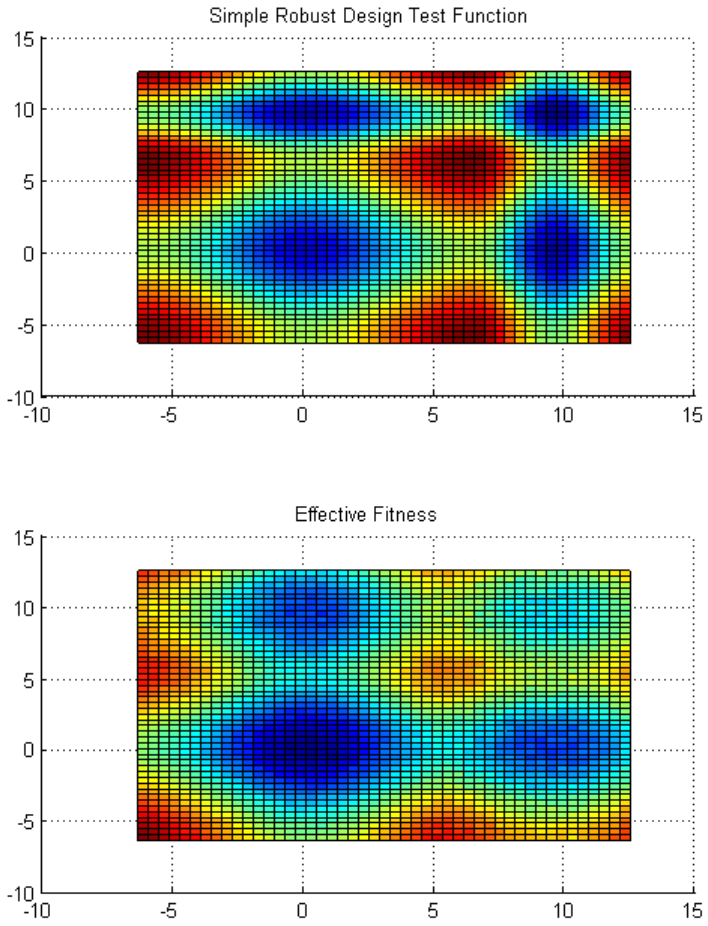

Fig. 4.4: Simple robust design test function with its effective fitness

$$\hat{f}_{\text{simple}}(x_j) = \begin{cases} \cos\left(\frac{1}{2}x_j\right) + 1 & \text{if } -2\pi \leq x_j < 2\pi \\ 1.1 \cdot \cos(x_j + \pi) + 1.1 & \text{if } 2\pi \leq x_j < 4\pi \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$

We use this function only in two dimensions. In figure 4.4 we can clearly

observe that the global optimum becomes a sub-global optimum when we regard the effective fitness.
As the noise model we use: pdf $\left(\vec{\delta}\right)$: $\vec{\delta} \sim \vec{U}(-3, 3)$.

5. EMPIRICAL PERFORMANCE

This chapter investigates the performance of the original PSO algorithms as well as their robust counterparts. We use a neutral parameter setting (as is most commonly found in literature), but we focus on the influences of the two different neighborhood topologies as explained in chapter 3.
Next we make a comparison between our different variants of the PSO algorithms against Evolutionary Algorithms which are commonly used in the field of optimization. These comparisons will focus on solution quality and robustness, but convergence velocity, not considered as a key objective, is briefly discussed.
All investigations and comparisons are made using the benchmark functions found in chapter 4. Unless otherwise stated we use the there mentioned dimensionality and domain.
In chapter 6 we will discuss the results presented here and draw some conclusions.

5.1 PSO Performance for general optimization

A comparison overview between our algorithms and the PSO algorithms used in other literature is found in appendix F. This comparison is primarily used to determine if our algorithms are functioning correctly.
In table 5.1 we present the results of our original PSO algorithms. All results are averaged over 30 runs of $300,000$ objective function evaluations. Values $< 10^{-10}$ are rounded down to 0.0. Comparison against the results found in appendix F yields that our algorithms function correctly.

Function	PSO Variant	Average Solution Quality	Standard Deviation
Sphere (4.1)	CPSO (Global)	0.0	0.0
	CPSO (Local)	0.0	0.0
	FIPS (Global)	0.0	0.0
	FIPS (Local)	0.0	0.0
Rosenbrock (4.2)	CPSO (Global)	25.2	45.2
	CPSO (Local)	28.3	57.2
	FIPS (Global)	7.17	6.84
	FIPS (Local)	31.3	25.7
Griewank (4.3)	CPSO (Global)	0.0168	0.0203
	CPSO (Local)	0.000247	0.00140
	FIPS (Global)	0.00470	0.00540
	FIPS (Local)	0.0	0.0
Rastrigin (4.4)	CPSO (Global)	74.8	19.9
	CPSO (Local)	72.9	19.5
	FIPS (Global)	0.0	0.0
	FIPS (Local)	34.8	8.94
Schaffer's f6 (4.5)	CPSO (Global)	0.0	0.0
	CPSO (Local)	0.0	0.0
	FIPS (Global)	0.00140	0.00470
	FIPS (Local)	0.0	0.0
Ackley (4.6)	CPSO (Global)	1.07	0.913
	CPSO (Local)	0.0	0.0
	FIPS (Global)	2.68	0.681
	FIPS (Local)	0.0	0.0

Tab. 5.1: Solution quality of various PSOs on the general test functions

5.2 PSO Performance for robust optimization

In this section we will make some comparisons between our PSOs and some standard Evolutionary Strategies (ES). In order to have a more compact representation of figures and tables we use a shorthand for the different varieties used:

ES1: A $(1, \lambda)$ ES [Rechenberg (1994)] with $\lambda = 10$.

ES2: A (μ, λ) ES [Rechenberg (1994)] with $\mu = 15$ and $\lambda = 100$.

ES3: The CMA ES [Hansen and Ostermeier (2001)].

PSO1: The Canonical PSO (algorithm 1) with $\lambda = 50$, $\phi_1 = 2.05$, $\phi_2 = 2.05$ and a global best neighborhood topology.

PSO2: The Canonical PSO (algorithm 1) with $\lambda = 50$, $\phi_1 = 2.05$, $\phi_2 = 2.05$ and a local best neighborhood topology.

PSO3: The Fully Informed Particle Swarm (algorithm 2) with $\lambda = 50$, $\phi = 4.1$ and a global best neighborhood topology.

PSO4: The Fully Informed Particle Swarm (algorithm 2) with $\lambda = 50$, $\phi = 4.1$ and a local best neighborhood topology.

Although convergence velocity is not a key objective in this thesis, we first present a comparison between PSOs and ES's. The results in figure 5.1 are based on a single run of $8,000$ objective function evaluations of the benchmark function 4.11.

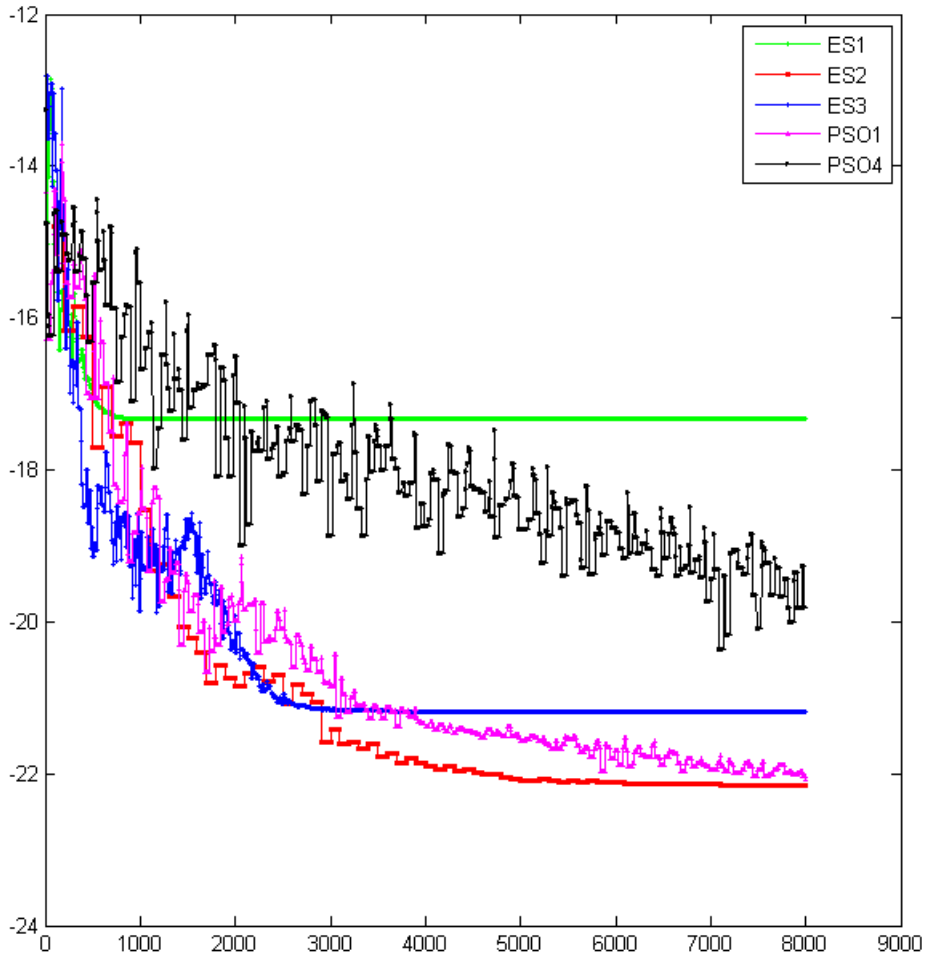

Fig. 5.1: Convergence velocity comparison between various ES's and PSOs

Without modifying our original algorithms we investigated the capabilities of finding robust optima of the PSO's and ES's. The results in table 5.2 and figure 5.2 are based on 30 run of 10,000 objective function evaluations.

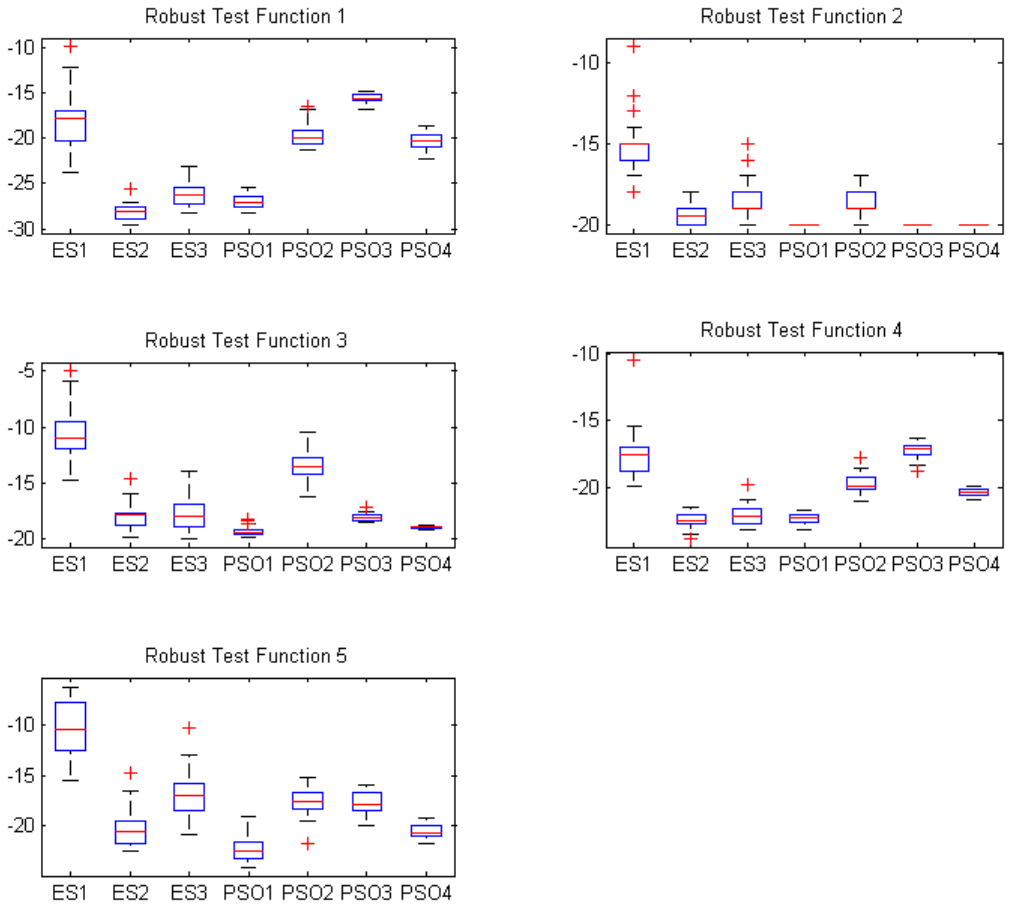

Fig. 5.2: Solution quality comparison between various ES's and PSOs

Function	Optimizer	Average	Standard Deviation	Robustness
f1 (4.8)	ES1	-18.1396	2.9188	6.938
	ES2	-28.0954	0.8066	6.9415
	ES3	-26.2012	1.3015	6.9408
	PSO1	-27.0534	0.8202	6.9421
	PSO2	-19.7769	1.2557	6.9365
	PSO3	-15.6044	0.4709	6.9374
	PSO4	-20.217	0.842	6.9376
f2 (4.9)	ES1	-15.0667	1.6802	6.9389
	ES2	-19.0	0.6747	6.9399
	ES3	-18.4333	1.3047	6.9401
	PSO1	-20.0	0.0	6.9377
	PSO2	-18.8333	0.8743	6.9386
	PSO3	-20.0	0.0	6.9427
	PSO4	-20.0	0.0	6.9409
f3 (4.10)	ES1	-10.6764	2.1648	6.9384
	ES2	-17.9871	1.1103	6.9427
	ES3	-17.7299	1.4413	6.9409
	PSO1	-19.3174	0.4261	6.9387
	PSO2	-13.5944	1.2649	6.9394
	PSO3	-18.0849	0.3125	6.9407
	PSO4	-18.9434	0.123	6.94
f4 (4.11)	ES1	-17.6173	1.781	6.9407
	ES2	-22.4118	0.5724	6.9436
	ES3	-22.0516	0.7679	6.9376
	PSO1	-22.2777	0.406	6.94
	PSO2	-19.6779	0.6753	6.9403
	PSO3	-17.2189	0.5111	6.9376
	PSO4	-20.3218	0.2363	6.9365
f5 (4.12)	ES1	-10.5667	2.9405	6.9402
	ES2	-20.18546	1.8546	6.9393
	ES3	-16.9086	2.3086	6.9376
	PSO1	-22.325	1.1562	6.9385
	PSO2	-17.5833	1.3714	6.9406
	PSO3	-17.7583	1.1073	6.9404
	PSO4	-20.5583	0.6749	6.9379

Tab. 5.2: Robustness comparison between various ES's and PSOs

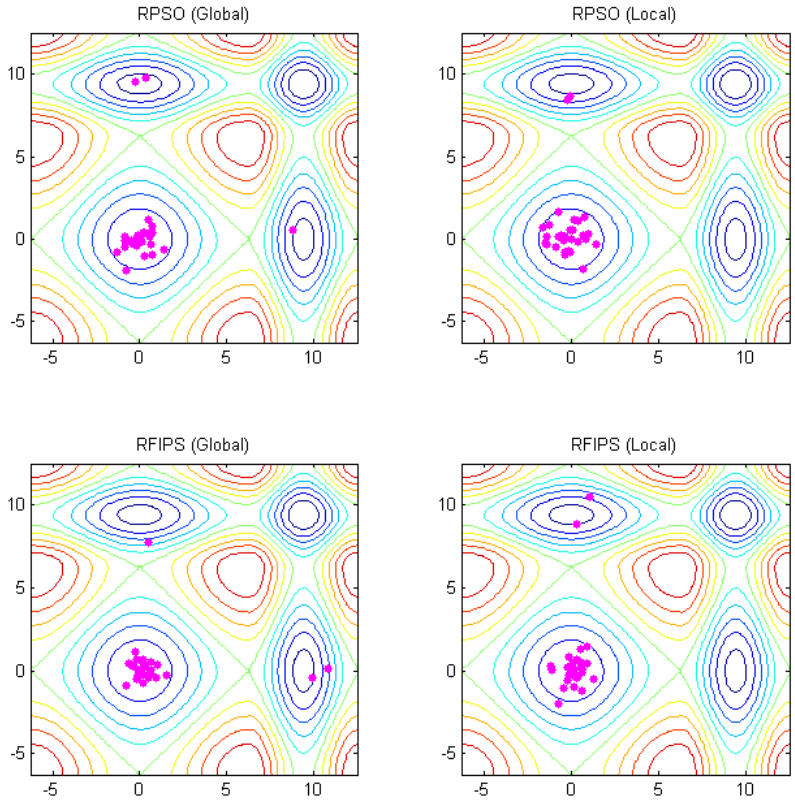

Fig. 5.3: Optima locations with $\mu = 10$

5.2.1 Proof of Concept

In this section we will adjust our original PSO algorithms for robust optimization. We use a Monte Carlo sampling method to adjust the solution quality. For each individual in the population we sample a certain number of points (μ) in the search space according to pdf $\left(\vec{\delta}\right)$. The solution quality of the individual becomes the average solution quality of these points.
We use two settings for the sample size: $\mu = 10$ (figure 5.3) and $\mu = 1$ (figure 5.4). These results are based on 30 runs of $5,000$ objective function

evaluations.
In figures 5.5 and 5.6, we present the performance of the three ES variants (which are modified for robust optimization using the same technique as the PSOs).

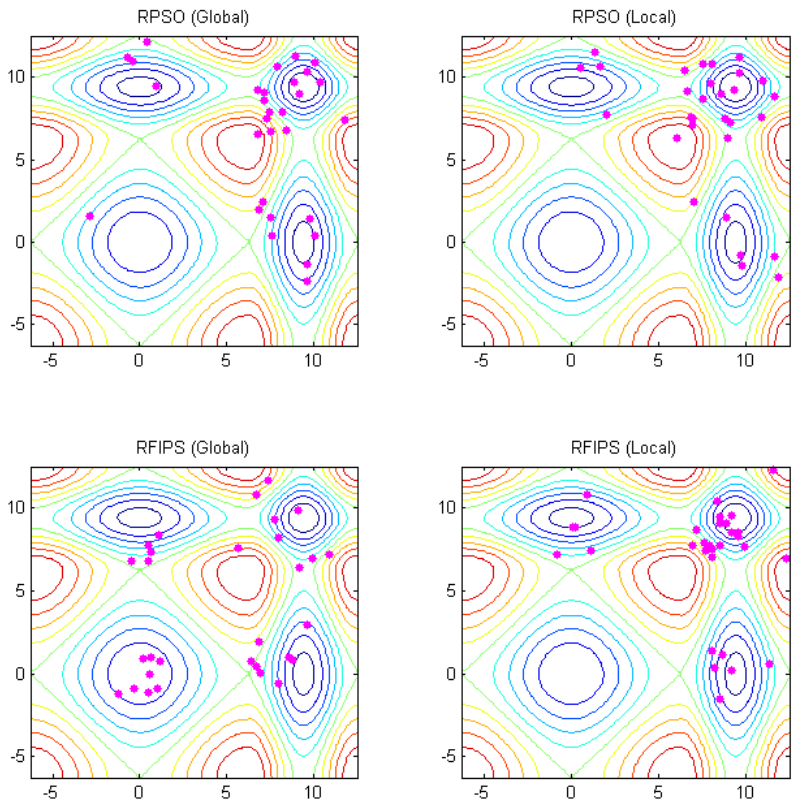

Fig. 5.4: Optima locations with $\mu = 1$

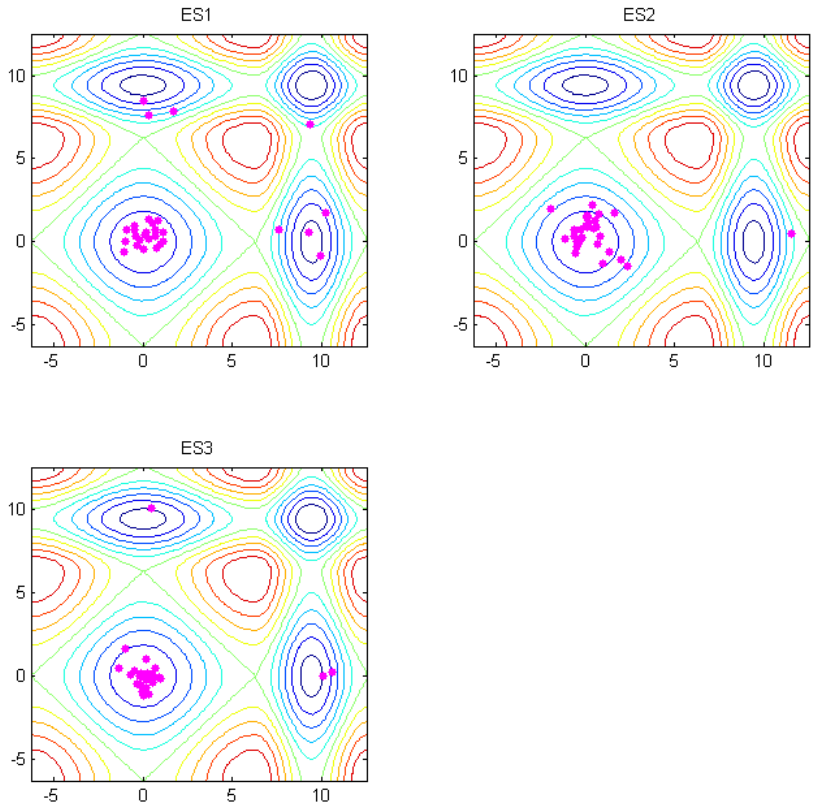

Fig. 5.5: Optima locations with $\mu = 10$ (ES's modified for robustness)

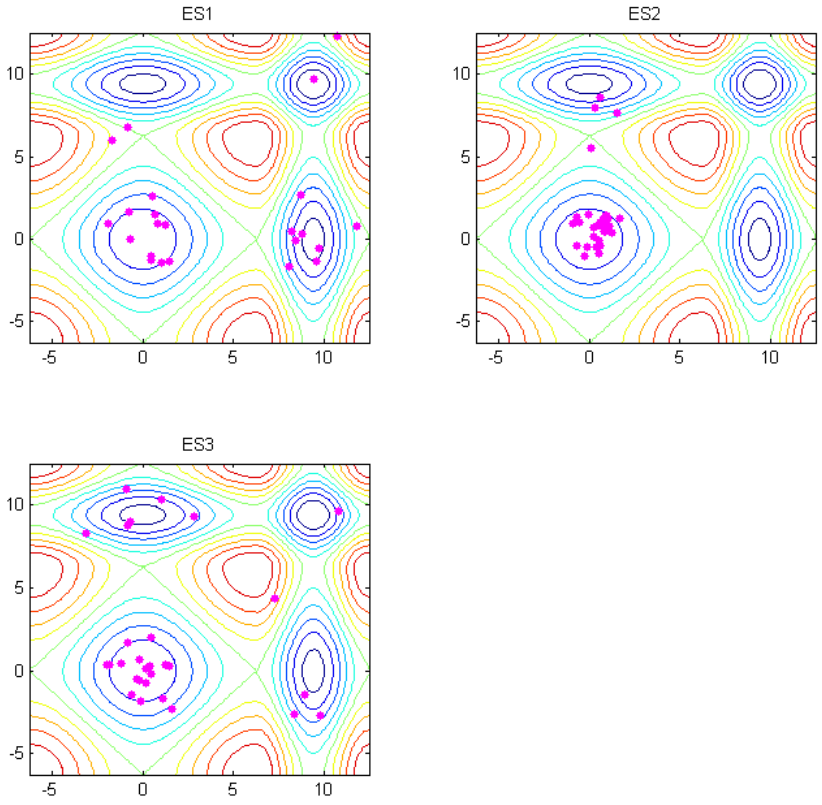

Fig. 5.6: Optima locations with $\mu = 1$ (ES's modified for robustness)

6. CONCLUSIONS AND DISCUSSION

This chapter discusses the resut presented in chapter 5 and summarises the findings and contributions of this thesis, followed by a discussion of directions for future research.

This thesis investigated the behavior of various PSO variants in finding robust optima. As well as a comparison against the behavior of Evolutionary Strategies. We rely only on empirical research as theoretical analysis of PSO algorithms is very hard and generally only possible for simplified versions. However, from the result in chapter 5 we can draw some conclusions.

Comparisons between table 5.1 and appendix F imply that our implementation of the PSO algorithms complies with other implementations found in literature. From here, we used our implementations for comparison against ES's applied to robust optimization. These results are presented in section 5.2. First we studied the convergence velocity. We note that in table 5.2 the solution quality of the PSOs does not significantly differ from the solution quality of the ES's. However, in figure 5.1, we can clearly see that ES's are generally faster to converge in the early stages of optimization. This is especially the case for the FIPS algorithm.

As is clearly shown in table 5.2 the optima found by all algorithms (PSOs and ES's) are very similar in both solution quality and robustness. Maybe with an exception for **ES1**, which is a very simple ES algorithm and is therefore likely to yield less qualitative solutions. So our first conclusion is that PSOs and ES's do not significantly differ and, as a second conclusion, both kind of algorithms have the same preference for the same kind of optima as is indicated by the Robustness figure in table 5.2. We measured robustness

by averaging over 30 runs and using the effective fitness function (2.1).

From figure 5.2, we conclude that, generally, **PSO1** and **PSO4** are out performing the other PSO variants. However, some PSO favour particular benchmark problems. Further, we see that the PSO algorithm, generally, find solutions with less variance in their quality, which can be preferable.

In section 5.2.1, we have modified our original algorithms to optimize for robustness and we use our simple benchmark function introduced in 4.2.1. In figures 5.3 and 5.4, we plotted for 30 runs the found optima after $5,000$ objective function evaluations. In figure 5.3, we used a sample size of $\mu = 10$ so the solution quality of each individual in the population is influenced by 10 samples (Monte Carlo sampling). As is clearly shown most runs ended near the robust optimum at $(0, 0)$. A small percentage ($\leq 10\%$) converged to one of the sub-optimal robust optima. None converged to the global (non-robust) optimum.
In figure 5.4, each particle was only influenced by one random sample. As this method introduces no additional objective function evaluations as opposed to the original algorithm, this would be the preferred sample size. However, as is clearly shown, most PSO algorithms are predominately attracted to the global (non-robust) optimum. In the best case (Robust FIPS with global best topology), the particles converged equally to each of the optima. This is not sufficient for a reliable robust optimizer.

If we compare the performance of robust optimization between the ES's and PSOs, we notice immediately that they behave very simmilary with a sample size $\mu = 10$. For $\mu = 1$ they display a radically different behavior. Where the PSO's fail to converge to the robust optimum, the ES's are much more succesful at finding this optimum. We have to conclude that ES's are naturally better suited to find robust optima with a small sampling sizes.

6.1 Future Research

We have shown that our robust PSO algorithms are able to find robust optima. However, this is at the expense of additional objective function evaluations. When enforcing a fixed number of evaluations the number of loop iterations of the algorithm diminish. This would be an undesired effect especially for high dimensional problems. Therefore, we suggest research towards other methods of robust optimization. For example it would be ideal to design a neighborhood topology (or modified change rules) which intrinsically yield robust solutions.

BIBLIOGRAPHY

Branke, J. (2001). *Evolutionary Optimization in Dynamic Environments*. Kluwer Academic Publishers, Norwell, MA, USA.

Carlise, A. and Dozier, G. (2001). An off-the-self pso. In *Proceedings of the Workshop on Particle Swarm Optimization*, pages 1–6.

Clerc, M. and Kennedy, J. (2002). The particle swarm - explosion, stability, and convergence in a multidimensional complex space. *Evolutionary Computation, IEEE Transactions on*, 6(1):58–73.

Eberhart, R. and Kennedy, J. (1995). A new optimizer using particle swarm theory. In *Micro Machine and Human Science, 1995. MHS '95., Proceedings of the Sixth International Symposium on*, pages 39–43.

Hansen, N. and Ostermeier, A. (2001). Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195.

Heppner, F. and Grenander, U. (1990). A stochastic nonlinear model for coordinated bird flocks. In Krasner, E., editor, *The ubiquity of chaos*, pages 233–238. AAAS Publications.

Kennedy, M. and Mendes, R. (2002). Population structure and particle swarm performance. In *In: Proceedings of the Congress on Evolutionary Computation (CEC 2002)*, pages 1671–1676. IEEE Press.

Kruisselbrink, J. (2009). Robust optimization optimization under uncertainty and noise. In *LIACS, Technical Report*.

Paenke, I., Branke, J., and Jin, Y. (2006). Efficient search for robust solutions by means of evolutionary algorithms and fitness approximation. *Evolutionary Computation, IEEE Transactions on*, 10(4):405–420.

Poli, R., Kennedy, J., and Blackwell, T. (2007). Particle swarm optimization. *Swarm Intelligence*, 1(1):33–57.

Rechenberg, I. (1994). *Evolutionsstrategie'94*, volume 1 of *Werkstatt Bionik und Evolutionstechnik*. Friedrich Frommann Verlag (Günther Holzboog KG), Stuttgart.

APPENDIX

A. MATLAB CODE FOR CANONICAL PSO

```
function [xopt, fopt, stat] = cpso(fitnessfct, N, lb, ub, stopeval, epsilon, param)
%       [xopt, fopt, stat] = cpso(fitnessfct, N, lb, ub, stopeval, epsilon, param)
%       Canonical Particle Swarm Optimizer
%   the standard layout of an optimization
%   function for N-dimensional continuous minimalization problems
%   with the search space bounded by a box.
%
% Input:
%   fitnessfct      - a handle to the fitness function
%   N               - the number of dimensions
%   lb              - the lower bounds of the box constraints
%                     this is a vector of length N
%   ub              - the upper bounds of the box constraints
%                     this is a vector of length N
%   stopeval        - the number of function evaluations used
%                     by the optimization algorithm
%   epsilon         - not applicable: set it to -1
%   param           - a MATLAB struct containing algorithm
%                     specific input parameters
%     .lambda       - number of particles in the swarm
%     .phi1         - cognitive ratio
%     .phi2         - social ratio
%     .nbh          - neighborhood incidence matrix
%
% Output:
%   xopt            - a vector containing the location of the optimum
%   fopt            - the objective function value of the optimum
%   stat            - a MATLAB struct containing algorithm
%                     specific output statistics
%     .histf        - the best objective function value history
%

        % Canonical parameter settings
        if epsilon ~= -1
                warning('cpso:epsilon_ignored', 'epsilon is ignored')
        end
        % c (particle count) = 30
        if isfield(param, 'lambda')
                lambda = param.lambda;
        else
                lambda = 30;
        end
        % phi1 (cognitive ratio) = 2.05
        if isfield(param, 'phi1')
                phi1 = param.phi1;
        else
                phi1 = 2.05;
        end
        % phi2 (social ratio) = 2.05
        if isfield(param, 'phi2')
                phi2 = param.phi2;
        else
                phi2 = 2.05;
        end
        % nbh (neighbourhood matrix) initialization at global best topology
        if isfield(param, 'nbh')
                nbh = param.nbh;
        else
                nbh = nbh_global_best(lambda);
        end
        % For this algorithm we set non-neighboring particles to distance infinity
        nbh(nbh == 0) = Inf;
```

```
        % phi (for use in Clerc's constriction method) = phi1 + phi2 > 4
        phi = phi1 + phi2;
        % chi (Clerc's constriction constant) = 2 / (phi - 2 + sqrt(phi .^ 2 - 4 * phi))
        chi = 2 / (phi - 2 + sqrt(phi .^ 2 - 4 * phi));

        % Initialization
        % Initialize v (velocity) uniform randomly between lb and ub
        v = repmat(lb', lambda, 1) + repmat(ub' - lb', lambda, 1) .* rand(lambda, N);
        % Initialize x (position) uniform randomly between lb and ub
        x = repmat(lb', lambda, 1) + repmat(ub' - lb', lambda, 1) .* rand(lambda, N);
        % Initially the personal best position is the starting position
        p = x;
        % Initialize the personal best objective function evaluation to infinity to always allow improvement
        p_best = ones(lambda, 1) * Inf;
        % Initialize the number of objective function evaluations to zero
        evalcount = 0;
        % Preallocate an array that will hold the objective function evaluations
        f = zeros(lambda, 1);
        % Preallocate the stat.histf array
        stat = struct();
        stat.histf = zeros(stopeval, 1);

        % Loop while number of objective function evaluations does not exceeds the stop criterion
        while evalcount < stopeval

                % Evaluate for all particles the objective function
                for i = 1 : lambda
                        f(i) = feval(fitnessfct, x(i, :));
                        % Update stat.histf array
                        stat.histf(evalcount + i) = min(f);
                end

                % Update the personal best positions if the current position is better than the current personal best position
                p = repmat(f < p_best, 1, N) .* x + repmat(~(f < p_best), 1, N) .* p;
                % Update the personal best objective function evaluation
                p_best = min(f, p_best);

                % Calculate the best particle in each neighborhood
                [g_best, g] = min(repmat(p_best, 1, lambda) .* nbh);

                % Update the velocities using Clerc's constriction algorithm
                v = chi * (v + (p - x) .* rand(lambda, N) * phi1 + (p(g, :) - x) .* rand(lambda, N) * phi2);
                % Update the positions
                x = x + v;

                % Update the number of objective function evaluations used
                evalcount = evalcount + lambda;

        end

        % Select the global minimum from the personal best objective function evaluations
        [fopt, g] = min(p_best);
        % Select the global mimimal solution
        xopt = p(g, :);

end
```

B. MATLAB CODE FOR FIPS

```
function [xopt, fopt, stat] = fips(fitnessfct, N, lb, ub, stopeval, epsilon, param)
%       [xopt, fopt, stat] = fips(fitnessfct, N, lb, ub, stopeval, epsilon, param)
%       Fully Informed Particle Swarm
%   the standard layout of an optimization
%   function for N-dimensional continuous minimalization problems
%   with the search space bounded by a box.
%
% Input:
%   fitnessfct      - a handle to the fitness function
%   N               - the number of dimensions
%   lb              - the lower bounds of the box constraints
%                     this is a vector of length N
%   ub              - the upper bounds of the box constraints
%                     this is a vector of length N
%   stopeval        - the number of function evaluations used
%                     by the optimization algorithm
%   epsilon         - not applicable: set it to -1
%   param           - a MATLAB struct containing algorithm
%                     specific input parameters
%     .lambda       - number of particles in the swarm
%     .phi          - acceleration coefficient
%     .nbh          - neighborhood incidence matrix
%
% Output:
%   xopt            - a vector containing the location of the optimum
%   fopt            - the objective function value of the optimum
%   stat            - a MATLAB struct containing algorithm
%                     specific output statistics
%     .histf        - the best objective function value history
%

        % Parameter settings
        if epsilon ~= -1
                warning('fips:epsilon_ignored', 'epsilon is ignored')
        end
        % c (particle count) = 30
        if isfield(param, 'lambda')
                lambda = param.lambda;
        else
                lambda = 30;
        end
        % phi (for use in Clerc's constriction method) = 4.1 > 4
        if isfield(param, 'phi')
                phi = param.phi;
        else
                phi = 4.1;
        end
        % nbh (neighbourhood matrix) initialization at local best topology
        if isfield(param, 'nbh')
                nbh = param.nbh;
        else
                nbh = nbh_local_best(lambda);
        end
        % chi (Clerc's constriction constant) = 2 / (phi - 2 + sqrt(phi .^ 2 - 4 * phi))
        chi = 2 / (phi - 2 + sqrt(phi .^ 2 - 4 * phi));

        % Initialization
        % Initialize v (velocity) uniform randomly between lb and ub
        v = repmat(lb', lambda, 1) + repmat(ub' - lb', lambda, 1) .* rand(lambda, N);
        % Initialize x (position) uniform randomly between lb and ub
        x = repmat(lb', lambda, 1) + repmat(ub' - lb', lambda, 1) .* rand(lambda, N);
        % Initially the personal best position is the starting position
```

```
        p = x;
        % Initialize the personal best objective function evaluation to infinity to always allow improvement
        p_best = ones(lambda, 1) * Inf;
        % Initialize the number of objective function evaluations to zero
        evalcount = 0;
        % Preallocate an array that will hold the objective function evaluations
        f = zeros(lambda, 1);
        % Preallocate the stat.fhist array
        stat = struct();
        stat.histf = zeros(stopeval, 1);

        % Loop while number of objective function evaluations does not exceeds the stop criterion
        while evalcount < stopeval

                % Evaluate for all particles the objective function
                for i = 1 : lambda
                        f(i) = feval(fitnessfct, x(i, :));
                        % Update stat.fhist array
                        stat.histf(evalcount + i) = min(f);
                end

                % Update the personal best positions if the current position is better than the current personal best position
                p = repmat(f < p_best, 1, N) .* x + repmat(~(f < p_best), 1, N) .* p;
                % Update the personal best objective function evaluation
                p_best = min(f, p_best);

                % Update the velocities using Clerc's constriction algorithm
                for i = 1 : lambda
                        v(i, :) = chi * (v(i, :) + (nbh(i, :) * ((rand(lambda, N) * phi) .* ...
                                (p - repmat(x(i, :), lambda, 1)))) / sum(nbh(i, :)));
                end
                % Update the positions
                x = x + v;

                % Update the number of objective function evaluations used
                evalcount = evalcount + lambda;

        end

        % Select the global minimum from the personal best objective function evaluations
        [fopt, g] = min(p_best);
        % Select the global mimimal solution
        xopt = p(g, :);

end
```

C. MATLAB CODE FOR ROBUST PSO

```
function [xopt, fopt, stat] = rpso(fitnessfct, N, lb, ub, stopeval, epsilon, param)
%       [xopt, fopt, stat] = rpso(fitnessfct, N, lb, ub, stopeval, epsilon, param)
%       Robust Particle Swarm Optimizer
%   the standard layout of an optimization
%   function for N-dimensional continuous minimalization problems
%   with the search space bounded by a box.
%
% Input:
%   fitnessfct      - a handle to the fitness function
%   N               - the number of dimensions
%   lb              - the lower bounds of the box constraints
%                     this is a vector of length N
%   ub              - the upper bounds of the box constraints
%                     this is a vector of length N
%   stopeval        - the number of function evaluations used
%                     by the optimization algorithm
%   epsilon         - not applicable: set it to -1
%   param           - a MATLAB struct containing algorithm
%                     specific input parameters
%     .lambda       - number of particles in the swarm
%     .phi1         - cognitive ratio
%     .phi2         - social ratio
%     .nbh          - neighborhood incidence matrix
%     .mu           - sample size
%     .delta        - sample variance
%                     this is a vector of length N
%
% Output:
%   xopt            - a vector containing the location of the optimum
%   fopt            - the objective function value of the optimum
%   stat            - a MATLAB struct containing algorithm
%                     specific output statistics
%     .histf        - the best objective function value history
%

        % Canonical parameter settings
        if epsilon ~= -1
                warning('cpso:epsilon_ignored', 'epsilon␣is␣ignored')
        end
        % c (particle count) = 30
        if isfield(param, 'lambda')
                lambda = param.lambda;
        else
                lambda = 30;
        end
        % phi1 (cognitive ratio) = 2.05
        if isfield(param, 'phi1')
                phi1 = param.phi1;
        else
                phi1 = 2.05;
        end
        % phi2 (social ratio) = 2.05
        if isfield(param, 'phi2')
                phi2 = param.phi2;
        else
                phi2 = 2.05;
        end
        % nbh (neighbourhood matrix) initialization at global best topology
        if isfield(param, 'nbh')
                nbh = param.nbh;
        else
                nbh = nbh_global_best(lambda);
```

```
        end
        % mu (sample size) = 1
        if isfield(param, 'mu')
                mu = param.sampleN;
        else
                mu = 1;
        end
        % delta (sample variance) = 0-vector
        if isfield(param, 'delta')
                delta = param.delta;
        else
                delta = zeros(N, 1);
        end
        % For this algorithm we set non-neighboring particles to distance infinity
        nbh(nbh == 0) = Inf;
        % phi (for use in Clerc's constriction method) = phi1 + phi2 > 4
        phi = phi1 + phi2;
        % chi (Clerc's constriction constant) = 2 / (phi - 2 + sqrt(phi .^ 2 - 4 * phi))
        chi = 2 / (phi - 2 + sqrt(phi .^ 2 - 4 * phi));

        % Initialization
        % Initialize v (velocity) uniform randomly between lb and ub
        v = repmat(lb', lambda, 1) + repmat(ub' - lb', lambda, 1) .* rand(lambda, N);
        % Initialize x (position) uniform randomly between lb and ub
        x = repmat(lb', lambda, 1) + repmat(ub' - lb', lambda, 1) .* rand(lambda, N);
        % Initially the personal best position is the starting position
        p = x;
        % Initialize the personal best objective function evaluation to infinity to always allow improvement
        p_best = ones(lambda, 1) * Inf;
        % Initialize the number of objective function evaluations to zero
        evalcount = 0;
        % Preallocate an array that will hold the objective function evaluations
        f = zeros(lambda, 1);
        % Preallocate the stat.histf array
        stat = struct();
        stat.histf = zeros(stopeval, 1);

        % Loop while number of objective function evaluations does not exceeds the stop criterion
        while evalcount < stopeval

                % Evaluate for all particles the objective function
                for i = 1 : lambda
                        sample = repmat(x(i, :), mu, 1) + repmat(-delta', mu, 1) + ...
                                repmat(2 * delta', mu, 1) .* rand(mu, N);
                        fsample = zeros(mu, 1);
                        for j = 1 : mu
                                fsample(j) = feval(fitnessfct, sample(j, :));
                        end
                        f(i) = mean(fsample);
                        % Update stat.fhist array
                        stat.histf(evalcount + i) = min(f);
                end

                % Update the personal best positions if the current position is better than the current personal best position
                p = repmat(f < p_best, 1, N) .* x + repmat(~(f < p_best), 1, N) .* p;
                % Update the personal best objective function evaluation
                p_best = min(f, p_best);

                % Calculate the best particle in each neighborhood
                [g_best, g] = min(repmat(p_best, 1, lambda) .* nbh);

                % Update the velocities using Clerc's constriction algorithm
                v = chi * (v + (p - x) .* rand(lambda, N) * phi1 + (p(g, :) - x) .* rand(lambda, N) * phi2);
                % Update the positions
                x = x + v;

                % Update the number of objective function evaluations used
                evalcount = evalcount + lambda * mu;

        end

        % Select the global minimum from the personal best objective function evaluations
        [fopt, g] = min(p_best);
        % Select the global mimimal solution
        xopt = p(g, :);

end
```

D. MATLAB CODE FOR ROBUST FIPS

```
function [xopt, fopt, stat] = rfips(fitnessfct, N, lb, ub, stopeval, epsilon, param)
%       [xopt, fopt, stat] = rfips(fitnessfct, N, lb, ub, stopeval, epsilon, param)
%       Robust Fully Informed Particle Swarm
%   the standard layout of an optimization
%   function for N-dimensional continuous minimalization problems
%   with the search space bounded by a box.
%
% Input:
%   fitnessfct      - a handle to the fitness function
%   N               - the number of dimensions
%   lb              - the lower bounds of the box constraints
%                     this is a vector of length N
%   ub              - the upper bounds of the box constraints
%                     this is a vector of length N
%   stopeval        - the number of function evaluations used
%                     by the optimization algorithm
%   epsilon         - not applicable: set it to -1
%   param           - a MATLAB struct containing algorithm
%                     specific input parameters
%     .lambda       - number of particles in the swarm
%     .phi          - acceleration coefficient
%     .nbh          - neighborhood incidence matrix
%     .mu           - sample size
%     .delta        - sample variance
%                     this is a vector of length N
%
% Output:
%   xopt            - a vector containing the location of the optimum
%   fopt            - the objective function value of the optimum
%   stat            - a MATLAB struct containing algorithm
%                     specific output statistics
%     .histf        - the best objective function value history
%

        % Parameter settings
        if epsilon ~= -1
                warning('fips:epsilon_ignored', 'epsilon is ignored')
        end
        % c (particle count) = 30
        if isfield(param, 'lambda')
                lambda = param.lambda;
        else
                lambda = 30;
        end
        % phi (for use in Clerc's constriction method) = 4.1 > 4
        if isfield(param, 'phi')
                phi = param.phi;
        else
                phi = 4.1;
        end
        % nbh (neighbourhood matrix) initialization at local best topology
        if isfield(param, 'nbh')
                nbh = param.nbh;
        else
                nbh = nbh_local_best(lambda);
        end
        % mu (sample size) = 1
        if isfield(param, 'mu')
                mu = param.mu;
        else
                mu = 1;
        end
```

```
        % delta (sample variance) = 0-vector
        if isfield(param, 'delta')
                delta = param.delta;
        else
                delta = zeros(N, 1);
        end
        % chi (Clerc's constriction constant) = 2 / (phi - 2 + sqrt(phi .^ 2 - 4 * phi))
        chi = 2 / (phi - 2 + sqrt(phi .^ 2 - 4 * phi));

        % Initialization
        % Initialize v (velocity) uniform randomly between lb and ub
        v = repmat(lb', lambda, 1) + repmat(ub' - lb', lambda, 1) .* rand(lambda, N);
        % Initialize x (position) uniform randomly between lb and ub
        x = repmat(lb', lambda, 1) + repmat(ub' - lb', lambda, 1) .* rand(lambda, N);
        % Initially the personal best position is the starting position
        p = x;
        % Initialize the personal best objective function evaluation to infinity to always allow improvement
        p_best = ones(lambda, 1) * Inf;
        % Initialize the number of objective function evaluations to zero
        evalcount = 0;
        % Preallocate an array that will hold the objective function evaluations
        f = zeros(lambda, 1);
        % Preallocate the stat.fhist array
        stat = struct();
        stat.histf = zeros(stopeval, 1);

        % Loop while number of objective function evaluations does not exceeds the stop criterion
        while evalcount < stopeval

                % Evaluate for all particles the objective function
                for i = 1 : lambda
                        sample = repmat(x(i, :), mu, 1) + repmat(-delta', mu, 1) + ...
                                repmat(2 * delta', mu, 1) .* rand(mu, N);
                        fsample = zeros(mu, 1);
                        for j = 1 : mu
                                fsample(j) = feval(fitnessfct, sample(j, :));
                        end
                        f(i) = mean(fsample);
                        % Update stat.fhist array
                        stat.histf(evalcount + i) = min(f);
                end

                % Update the personal best positions if the current position is better than the current personal best position
                p = repmat(f < p_best, 1, N) .* x + repmat(~(f < p_best), 1, N) .* p;
                % Update the personal best objective function evaluation
                p_best = min(f, p_best);

                % Update the velocities using Clerc's constriction algorithm
                for i = 1 : lambda
                        v(i, :) = chi * (v(i, :) + (nbh(i, :) * ((rand(lambda, N) * phi) .* ...
                                (p - repmat(x(i, :), lambda, 1)))) / sum(nbh(i, :)));
                end
                % Update the positions
                x = x + v;

                % Update the number of objective function evaluations used
                evalcount = evalcount + lambda * mu;

        end

        % Select the global minimum from the personal best objective function evaluations
        [fopt, g] = min(p_best);
        % Select the global mimimal solution
        xopt = p(g, :);

end
```

E. MATLAB CODE FOR NEIGHBORHOOD GENERATION

E.1 MATLAB Code for Ring Topology

```
function [nbh] = nbh_local_best(lambda)
% [nbh] = nbh_local_best(lambda)
%   Local Neighborhood Incidence Matrix Generator
%   creates an incidence matrix of a ring topology of size lambda x lambda
%   note: incidence is reflexive
%
% Input:
%   lambda          - the number of particles
%
% Output:
%   nbh             - the incidence matrix: 0 represents no incidence,
%                     1 represents an incidence

        nbh = diag(ones(lambda, 1)) + diag(ones(lambda - 1, 1), 1) + diag(ones(lambda - 1, 1), -1) + ...
                diag(ones(1, 1), lambda - 1) + diag(ones(1, 1), -(lambda - 1));

end
```

E.2 MATLAB Code for Fully Connected Topology

```
function [nbh] = nbh_global_best(lambda)
% [nbh] = nbh_global_best(lambda)
%   Global Neighborhood Incidence Matrix Generator
%   creates a fully connected incidence matrix of size lambda x lambda
%   note: incidence is reflexive
%
% Input:
%   lambda          - the number of particles
%
% Output:
%   nbh             - the incidence matrix: 0 represents no incidence,
%                     1 represents an incidence

        nbh = ones(lambda, lambda);

end
```

F. PSO SOLUTION QUALITY COMPARISON

Benchmark function					PSO				Solution			
Function	#Dimensions	Search range	#Runs	#Evaluations	#Particles	Topology	Initialization	Variant	Parameters	Average	Standard Deviation	#Evaluations	Standard Deviation	Source
Sphere	30	(-100, 100)	50	60000	20	Square	Asymmetrical	CPSO		6.88E-24	2.97E-24			[kent04]
								FIPS		3.47E-09	6.45E-09			[mont06]
										5.29E-09	5.29E-09			[mont06]
			30	300000	50	Ring			pH1 = 2.05, pH2 = 2.05	0.00E+00	0.00E+00	97063	398	[brat08]
			100	200000	40	Global	(50, 100)	CPSO		0.00E+00	0.00E+00	46997	421	[pasu06]
								CPSO-async	c1 = 2.0, c2 = 2.0, w1 = 0.9, w2 = 0.4	0.00E+00	0.00E+00	139515		[pasu06]
Rosenbrock	30	(-30, 30)	50	60000	20	Square	Asymmetrical	CPSO		3.66E-01	1.01E+02			[kent04]
								FIPS		6.70E-01	5.75E+01			[kent04]
										3.89E+02	9.09E+02			[mont06]
			30	300000	50	Ring				6.18E+00	1.07E+00			[brat08]
			100	200000	40	Global	(15, 30)	CPSO-async	c1 = 2.0, c2 = 2.0, w1 = 0.9, w2 = 0.4	9.97E+00	2.29E+00			[pasu06]
		(-100, 100)	100	200000	40	Global	(15, 30)	CPSO-async	c1 = 2.0, c2 = 2.0, w1 = 0.9, w2 = 0.4	1.06E+02	1.31E+02			[pasu06]
Griewank	30	(-600, 600)	50	60000	20	Square	Asymmetrical	CPSO		1.17E-02	1.71E-02			[kent04]
								FIPS		9.50E-03	9.50E-03			[mont06]
										8.27E+00	7.53E+00			[kent03]
			30	300000	50	Ring				0.00E+00	0.00E+00	110816		[mont06]
					50	Global		CPSO	pH1 = 2.05, pH2 = 2.05	1.08E-02	4.00E-03		3310	[brat08]
			100	200000	40	Global	(300, 600)	CPSO-async	c1 = 2.0, c2 = 2.0, w1 = 0.9, w2 = 0.4	1.55E-02	1.60E-02			[pasu06]
Rastrigin	30	(-5.12, 5.12)	50	60000	20	Square	Asymmetrical	CPSO		9.51E+01				[kent04]
								FIPS		6.88E+01	2.19E+01			[mont06]
										5.77E+01	1.97E+01			[kent03]
			30	300000	50	Ring			pH1 = 2.05, pH2 = 2.05	1.64E+02	5.84E+00			[mont06]
					50	Global		CPSO		1.40E+02	5.97E+00			[brat08]
			100	200000	40	Global	(2.56, 5.12)	CPSO-async	c1 = 2.0, c2 = 2.0, w1 = 0.9, w2 = 0.4	2.83E+01	7.73E+00			[pasu06]
Schaffer's F6	2	(-100, 100)	50	60000	20	Square	Asymmetrical	CPSO		6.00E-04				[pasu06]
								FIPS		1.68E-03	1.68E-03			[kent04]
										1.68E-03	3.36E-03			[kent04]
			100	200000	40	Global	(15, 30)	CPSO-async	c1 = 2.0, c2 = 2.0, w1 = 0.9, w2 = 0.4	2.97E-03	4.27E-03			[mont06]
			30	300000	50	Ring				0.00E+00	0.00E+00	48981		[pasu06]
					50	Global		CPSO	pH1 = 2.05, pH2 = 2.05	1.83E-01	8.53E-01			[brat08]
Ackley	30	(-32, 32)	100	200000	40	Global	(15, 30)	CPSO-async	c1 = 2.0, c2 = 2.0, w1 = 0.9, w2 = 0.4	1.29E+00	1.59E+00			[pasu06]
										3.42E+00	7.29E+00			[pasu06]

Fig. F.1: Solution Quality Comparison between different PSOs found in literature