Chapter 1

INTRODUCTION

Evolutionary algorithms (EAs) are optimisation algorithms inspired by facets of biological evolution. Populations of potential solutions (individuals) are iteratively evolved by encouraging the reproduction of more fit individuals. Mutations and crossovers of parent individuals are used to increase the genetic diversity of subsequent generations. EAs have been shown to be effective on optimisation problems that pose a challenge to more traditional optimisation methods [Bäck *et al.*, 1997].

Differential evolution (DE) is an EA variant which uses the positional difference between randomly selected individuals to create a mutant vector. A trial vector is created through a crossover between a mutant and a parent individual. Selection is implemented as competition between a parent individual and a trial vector. DE has been shown to be an effective optimisation algorithm in static environments.

A dynamic optimisation problem is a problem in which the fitness landscape varies over time. Dynamic optimisation problems are found in many real world domains, for example in air-traffic control, polarisation mode dispersion compensation in optical fibre, and target tracking in military applications [Gibbon *et al.*, 2008] [Li *et al.*, 2006]. A considerable amount of research has been conducted on applying EA to dynamic optimisation problems, but very few of these algorithms are based on DE. The focus of this thesis is on the application of DE variants to dynamic optimisation problems.

This chapter gives a motivation for this research in Section 1.1. The research objectives are listed in Section 1.2. Section 1.3 gives the methodology followed in this thesis. The main contributions of this study are highlighted in Section 1.4. The scope of this research study is delineated in Section 1.5. A break-down of the thesis structure is presented in Section 1.6.

1.1 Motivation

Despite the fact that evolutionary algorithms often solve static problems successfully, dynamic optimisation problems tend to pose a challenge to evolutionary algorithms [Morrison, 2004]. Lack of diversity is the main disadvantage of most standard evolutionary algorithms when applied to dynamic problems, since the algorithms tend to converge to a single optimum in the fitness landscape and then lack the diversity to locate new global optima when they appear. Differential evolution is one of the evolutionary algorithms that does not scale well to dynamic environments due to lack of diversity [Zaharie and Zamfirache, 2006].

A significant body of work exists on algorithms for optimising dynamic problems (refer to Section 3.3). However, very few of these algorithms are based on DE. DE has been successfully applied to a large number of static optimisation problems (refer to Section 2.4.5), and is generally considered to be a reliable, accurate, robust, and fast optimisation technique [Salman *et al.*, 2007]. DE provides several advantages as an optimisation algorithm:

- The function to be optimised by DE does not have to be differentiable [Ilonen *et al.*, 2003].
- The DE selection and mutation operators are simpler than those of other EA algorithms [Zaharie, 2002b].
- DE has relatively few control parameters [Storn and Price, 1997].
- The DE algorithm exhibits fast convergence [Karaboga and Okdem, 2004].
- The mutation step size of DE is automatically controlled so that the exploration is favoured initially, and exploitation is favoured later in the optimisation process [Xue *et al.*, 2003].

- The space complexity of DE is comparatively low [Das and Suganthan, 2011].

The application of DE to dynamic optimisation problems has only recently begun to receive attention (refer to Section 3.3.3.3), despite the previously mentioned advantages of DE. The relatively small amount of research into DE-based algorithms for dynamic environments means that the performance and behaviour of DE on dynamic optimisation problems have not been thoroughly investigated, and that improvements to current DE-based algorithms can potentially be made. The purpose of this thesis is to investigate, create, and refine DE-based algorithms aimed at dynamic optimisation problems.

1.2 Objectives

The primary objective of this thesis is to create improved DE-based algorithms for dynamic environments. The sub-objectives are as follows:

- Investigate existing algorithms aimed at solving dynamic optimisation problems, specifically focusing on algorithms based on differential evolution.
- Identify specific approaches, tailored to dynamic optimisation problems, that are commonly used in the above algorithms.
- Extend and hybridise existing DE algorithms to create more effective DE-based optimisation algorithms for dynamic environments.
- Identify and investigate dynamic optimisation problem types that have not been thoroughly investigated by other researchers.
- Investigate the use of adaptive control parameters to reduce the number of parameters that must be manually tuned.
- Perform scalability studies on existing and newly created algorithms.
- Compare the performance of the newly created algorithms to the reported results of other algorithms.

1.3 Methodology

The primary objective of this thesis is to create improved DE-based algorithms for dynamic environments. Eight algorithms are created sequentially in this study. Figure 1.1 illustrates how each algorithm relies on one or more base algorithms. Names of new algorithms, developed in this thesis, are printed in bold italics. DynDE [Mendes and Mohais, 2005], a DE-based multi-population algorithm for dynamic environments, is used as the root algorithm.

Two novel algorithms are directly based on DynDE. The first is competitive population evaluation (CPE), which improves the performance of DE by evolving only one sub-population at a time. The selection of sub-populations for evolution is based on performance. Reinitialisation midpoint check (RMC) is a technique aimed at improving the detection process that DynDE uses to determine if more than one sub-population converges to the same optimum. CPE and RMC are combined to form competing differential evolution (CDE). The performance and scalability of DynDE, CPE, RMC and CDE are evaluated on a standard benchmark for dynamic environments. The results of these experiments are used to draw conclusions regarding the effectiveness of the new algorithms. The general applicability of the CPE and RMC approaches is demonstrated by incorporating their respective algorithmic components into *jDE*, a DE-based optimisation algorithm that was developed by Brest *et al.* [2009]. Comparisons are drawn with the reported results of other algorithms in the literature which focus on dynamic optimisation problems.

Dynamic population differential evolution (DynPopDE) is an extension of CDE which dynamically adjusts the number of sub-populations. DynPopDE is aimed at dynamic optimisation problems in which the number of optima is unknown or fluctuates over time. Fluctuating numbers of optima have not been investigated by previous researchers (although an environment of this type has been included in the benchmark set of the IEEE WCCI-2012 competition on evolutionary computation for dynamic optimisation problems [Li *et al.*, 2011]). An adaptation is made to a standard benchmark to investigate the behaviour of DynPopDE on environments in which the number of optima fluctuates over time. A scalability study identifies problem instances where DynPopDE performs better than CDE and DynDE.

CDE is used as base algorithm to investigate the incorporation of self-adaptive control parameters. Three algorithms that self-adapt the DE scale and crossover factors, which were developed for static environments, are identified as potential approaches to work effectively in conjunction with CDE. Adaptations, aimed at improving their effectiveness in dynamic environments, are made to the three approaches. A total of 13 algorithms are consequently evaluated and compared to find the most effective, i.e. jSA2Ran.

Four strategies for self-adapting the Brownian radius (refer to Section 3.4.1.3) are investigated. SABrNorRes is shown to be the most effective of the four strategies. The final algorithms developed are SACDE and SADynPopDE, which are formed by incorporating the self-adaptive components from jSA2Ran and SABrNorRes into CDE and DynPopDE, respectively. The performances of SACDE and SADynPopDE are evaluated on a benchmark function and are compared to the performances of their base algorithms.

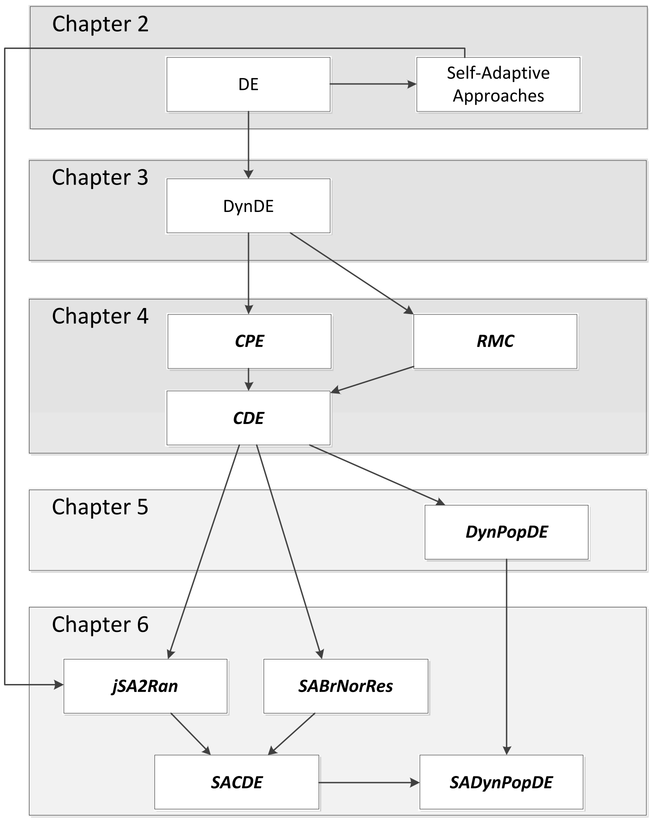

Figure 1.1: Thesis outline

1.4 Contributions

The main contributions of this thesis can be classified under the headings practical and theoretical. These contributions are briefly highlighted below.

Practical Contributions

- A novel approach to improving optimisation in dynamic environments based on sub-populations competing for function evaluations.
- An improvement of the approach used to prevent sub-populations from converging to the same optima by checking for valleys in the fitness landscape.
- A novel algorithm that dynamically spawns and removes sub-populations. This algorithm is aimed at dynamic environments where the number of optima is unknown or fluctuating.
- A novel algorithm to self-adapt the Brownian radius.
- An extension of a standard benchmark that allows the study of optimisation algorithms on environments in which the number of optima fluctuates over time.

Theoretical Contributions

- A unified taxonomy for classifying dynamic environments.
- Algorithmic approaches which are commonly used in dynamic optimisation problems are identified.
- Appropriate values for the sub-population size and number of Brownian individuals are determined.
- The general applicability of the competitive population evaluation and reinitialisation midpoint check approaches is illustrated.
- Evidence is presented to prove that fewer sub-populations should be used than the number of optima that are present in the environment, especially when the number of optima is large.

- An investigation into the most effective self-adaptive scale and crossover approach to use in conjunction with algorithms for dynamic environments is provided.
- An investigation into the most effective approach to self-adapting the Brownian radius is presented.
- Extensive scalability studies of existing and novel algorithms.
- The convergence profiles of the algorithms are investigated.

1.5 Scope

The scope of this research is delineated as follows:

- Dynamic environments in which the number of dimensions vary over time are excluded.
- The effect of different change detecting strategies is not investigated. The algorithms considered are independent of the change detection mechanism. The default detection strategy used is a change detection "oracle" which uses no function evaluations and detects changes perfectly. However, the optimisation algorithms employ actual change detection strategies when comparing these results with the published results of other researchers that used detection strategies.
- Only dynamic environments in which changes are periodic are considered. This constraint applies to the environments that are investigated, but is not exploited by the algorithms presented in this thesis. Specifically, at no point is it known to the algorithms how many function evaluations remain before the next change in the environment.

1.6 Thesis structure

This thesis is structured as follows:

Chapter 2 contains a brief overview of optimisation and evolutionary algorithms. Differential evolution is discussed in detail with reference to DE schemes and control parameters. A literature review of approaches to self-adapt or eliminate control parameters is included. Dynamic environments are described and the main factors which influence optimisation algorithm effectiveness are identified. The benchmarks and performance measures that are used in this study are discussed.

Chapter 3 introduces related work solving dynamic optimisation problems. The chapter commences with a discussion on how normal DE performs on dynamic optimisation problems. Algorithms presented by other researchers in the literature are reviewed. Three broad categories of strategies, which are followed to adapt optimisation algorithms to dynamic environments, are given. Seven specific strategies that fall within these categories are identified. It is argued that most current algorithms aimed at dynamic environments utilise at least one of the seven strategies. Two DE-based algorithms, DynDE and *jDE*, are reviewed in detail. The chapter concludes with a brief description of techniques used to detect changes in the environment.

Chapter 4 presents novel extensions to DynDE. An alternative method of calculating the DynDE exclusion threshold in the absence of knowledge regarding the number of optima present in the environment is given. Competitive population evaluation (CPE) is described and motivated. Reinitialisation midpoint check (RMC) is presented and potential pitfalls with the use of this approach are described. CPE and RMC are combined to form competing differential evolution (CDE). The scalability and performances of DynDE, CPE, RMC and CDE are compared. The general applicability of the CPE and RMC approaches is illustrated by their incorporation into another DE-based algorithm aimed at dynamic optimisation problems. CDE is compared to other algorithms in the literature.

Chapter 5 presents DynPopDE, an extension of CDE aimed at dynamic optimisation problems where the number of optima is unknown, or unknown and fluctuating over time. The technique used by DynPopDE to spawn or remove sub-populations is discussed and motivated. The performance of DynPopDE on problems where the number of optima is unknown is compared to that of DynDE and CDE. In addition, DynPopDE results are compared to results of other algorithms reported in the literature. DynPopDE is tested on an extension of one of the benchmarks that allows for fluctuating numbers of optima. Comparative results with DynDE and CDE are presented.

Chapter 6 incorporates self-adaptive control parameter approaches into CDE. Several approaches and variations of the approaches are investigated to self-adapt the DE scale and crossover factors. A new approach to self-adapting the Brownian radius is presented. The self-adaptive algorithms are combined to form SACDE. The performance and scalability of SACDE are compared to that of its predecessor algorithms and results of other algorithms reported in the literature. The self-adaptive approach is incorporated into DynPopDE to form SADynPopDE. The performance of SADynPopDE on problems where the number of optima is unknown or fluctuates is compared to the performance of DynPopDE and SACDE.

Chapter 7 summarises the conclusions found in this study and describes avenues of future research.

Several appendices are included in this thesis. Appendix A empirically shows the parameter dependence of *jDE*, a DE-based algorithm aimed at dynamic optimization problems (refer to Section 3.4.2).

Appendices B, C and D, provide additional results relevant to Chapters 4, 5, and 6, respectively. Appendix E lists the symbols used in this thesis, while Appendix F gives a list of acronyms that are used. Appendix G lists the publications that were derived from this research study.

Chapter 2

BACKGROUND

This chapter provides a contextual background for the rest of the thesis. A brief overview of optimisation techniques and general evolutionary algorithms is followed by a discussion of the differential evolution algorithm, common variations and control parameters. Related research on eliminating and adapting differential evolution control parameters is briefly outlined. Dynamic environments with specific focus on the benchmarks used in this research are described. A review of performance measures available in the literature is presented.

2.1 Introduction

Optimisation is an important branch of mathematics, engineering, and computer science. Many real-world problems can be reduced to optimisation problems, for which solutions can be found iteratively using optimisation algorithms (refer to Section 2.2 where optimisation algorithms are described). Differential evolution (DE) is an optimisation algorithm which is based on ideas from biological evolution. Section 2.4 describes the basic DE algorithm, its common variations, and DE control parameters. Best results are found if DE control parameters are fine-tuned for each specific problem. This process can be time consuming and should ideally be avoided. A literature survey on adapting and eliminating DE control parameters is included in Section 2.4.

The focus of this thesis is on the application of DE to dynamic optimisation problems (dynamic environments). A dynamic environment is a problem space that does not remain static over time, but which changes periodically. Section 2.5 formally defines the concept of dynamic environments and describes the different types of dynamic environments.

The development of a new algorithm requires a test-bed of problems on which to evaluate the effectiveness of the new algorithm, explore the scalability of the algorithm, and compare the performance of the algorithm to existing algorithms. Researchers have developed benchmark problems to provide a common platform for the evaluation and comparison of algorithms aimed at optimisation in dynamic environments. Two benchmarks are used in this study, namely the moving peaks benchmark (MPB) and the generalised dynamic benchmark generator (GDBG). These benchmarks are discussed in Section 2.5, which also contains a survey of approaches to measure the performance of optimisation algorithms in dynamic environments. Conclusions are drawn in Section 2.6.

2.2 Optimisation

Optimisation refers to the process of iteratively refining the proposed solution to a problem until a satisfactory result is reached. In the n_d dimensional real-valued number space, a minimisation optimisation problem is defined as the process of finding $\vec{x}^*$ with $x_j^* \in \mathbb{R}$ for $j = 1, \ldots, n_d$ such that

$$F(\vec{x}^*) \leq F(\vec{x}) \; \forall \; \vec{x} \in \mathcal{S}^{n_d} \tag{2.1}$$

where F is a function that gives the value of the potential solution vector $\vec{x}$ and $\mathcal{S}^{n_d} \subseteq \mathbb{R}^{n_d}$ denotes the search space. The vector $\vec{x}^*$ is referred to as a global optimum of F. Note that multiple global optima may exist. A local optimum is a point in the fitness landscape which gives the lowest value for F within a subregion of $\mathcal{S}^{n_d}$, but does not represent the lowest value of F in the entire fitness landscape. Equation (2.1) is an example of a function minimisation problem (i.e. the goal is to find the lowest value of function F). Problems where it is necessary to find the maximum value of a function is referred to as function maximisation problems. A global optimum position, $\vec{x}^*$, must thus be found such that

$$F(\vec{x}^*) \geq F(\vec{x}) \; \forall \; \vec{x} \in \mathcal{S}^{n_d} \tag{2.2}$$

The value of the solution vector, $\vec{x}^*$, can typically not be found analytically and large branches of applied mathematics and computer science have been dedicated to optimisation. Historically, many optimisation techniques are derivative-based [Burden and Faires, 2001]. Essentially, derivative-based approaches make use of the gradient of the function to perform stepwise updates to the solution vector. A basic derivative-based function minimisation optimisation approach is given in Algorithm 1.

Algorithm 1: Simple Derivative-Based Optimisation

```
Randomly select vector $\vec{x}$ in the $n_d$ dimensional search space;
$t = 0$;
while $\vec{x}(t)$ is not a satisfactory solution do
    foreach $j \in \{1, \ldots, n_d\}$ do
        $x_j(t+1) = x_j(t) - \eta \cdot \frac{\partial F(\vec{x}(t))}{\partial x_j(t)}$    (2.3)
        $t = t + 1$;
    end
end
```

The learning rate, η, is a constant that controls the size of the steps taken up (for maximisation) or down (for minimisation) the gradient of the fitness landscape. A large step size increases the rate of convergence of Algorithm 1, but increases the risk of stepping over a good optimum. A small step size typically increases the convergence time and the risk of becoming stuck in a local optimum, but ensures a finer grained search. An adaptive function, $\eta(t)$, is often used to provide a large step size initially and an increasingly smaller step size to refine the solution during the later stages of optimisation. It is important to note that Algorithm 1 is not guaranteed to converge to a global optimum.

A disadvantage of derivative-based optimisation is that two requirements must be met. Firstly, a continuous function must exist that fully describes the problem space. Secondly, the function must be differentiable. These requirements are not met for all optimisation problems, which means that derivative-based approaches cannot be applied to all optimisation problems. A further disadvantage of derivative-based approaches is that, in the process of descending down the gradient, the algorithm could become trapped in a local optimum. Since the algorithm cannot recover from such situations, suboptimal results may be found.

Research conducted into the simulation of biological evolution by scientists like Barricelli [1957] and Fraser [1957] on early electronic computers marked the advent of the field of evolutionary computation. Evolutionary algorithms (EAs) draw on ideas from genetics and Darwinian evolution [Darwin, 1859] to stochastically evolve solutions to optimisation problems. An elementary EA to minimise a function F is given in Algorithm 2.

Algorithm 2: Elementary Evolutionary Algorithm

```
Generate a population, $P_x$, of $I$ individuals by creating vectors of random
candidate solutions, $\vec{x}_i$, $i = 1, \cdots, n_I$;
Evaluate the fitness, $F(\vec{x}_i)$, of all individuals;
while termination criteria are not met do
    Select parent individuals from $P_x$ to reproduce;
    Create offspring from parent individuals through reproduction;
    Produce the next population from current parents and offspring;
    Evaluate the fitness, $F(\vec{x}_i)$, of all individuals;
end
```

Candidate solutions (individuals) are referred to as chromosomes, in an analogy to biological evolution. Individuals are given a representation based on the problem and the specific evolutionary algorithm used. Binary vectors were initially used in genetic algorithms (GAs), popularised by Holland [1975], while more recently real-valued numbers are commonly used as the representation scheme. In genetic programming, individuals are normally represented as trees [Koza, 1992]. Evolutionary programming, which is an example of phenotypic evolution rather than genotypic, use individuals to represent behavioural traits.

The initial population of individuals is randomly generated based on the representation used. Ideally, the initial population will uniformly represent the entire search space.

A fitness function is created to quantify the quality of each potential solution. This can, for example, take the form of a function that gives the error of any individual within the search space, or merely the function value at the point represented by an individual.

Termination criteria depend on the problem, but commonly, a set number of generations or a threshold for fitness is used.

The selection process in Algorithm 2 is generally done based on fitness. More fit individuals are assigned a higher likelihood of reproducing than less fit individuals. Components from good solutions are consequently spread through the population. A strong focus on the most fit individual increases the selection pressure, which may lead to the population being dominated by a single solution [Engelbrecht, 2007].

In GAs, reproduction usually involves crossover and mutation. Crossover is the process of combining genetic building blocks from two or more parent vectors to form one or more new offspring individuals. Mutation is the process of injecting random noise into offspring vectors to form a slightly different offspring individual, thereby increasing the genetic diversity of the population.

Broad and often overlapping subfields of evolutionary computation include genetic algorithms, genetic programming [Koza, 1992], evolutionary programming [Fogel, 1962], evolution strategies [Rechenberg, 1973], cultural evolution [Reynolds and Sverdlik, 1994] and differential evolution [Storn, 1996].

Particle swarm optimisation (PSO) [Kennedy and Eberhart, 1995] is often included in the above list, although it is strictly not an evolutionary algorithm [Engelbrecht, 2006]. PSO mimics the swarming behaviour of bird flocks to explore optimisation problem spaces.

Evolutionary algorithms and swarm-based algorithms solve the challenges faced by derivative-based approaches which were mentioned earlier. EAs are global optimisation techniques and thus more, global information about the fitness landscape to determine new candidate solutions [Engelbrecht, 2007], making EAs less likely to converge to a local optimum. Furthermore, the function to be optimised need not be differentiable or continuous. Technically, all that is required is a function that, given two potential solutions to the problem, can indicate which one is the better solution.

This thesis focuses on differential evolution, which is discussed in Section 2.4.

2.3 Genotypic Diversity

Central to the behaviour of an EA in a search space is the concept of genotypic diversity. Genotypic diversity (subsequently referred to simply as diversity) refers to the dispersion of individuals in the search space.

A large portion of the fitness landscape is searched when individuals are spread over a large area. This corresponds to a highly diverse population and such a wide search of the fitness landscape is referred to as exploration. A fine-grained search of the fitness landscape results from low diversity (i.e. individuals are clustered closely together). This is referred to as exploitation.

Typically, EAs commence with a high diversity population which favours exploration. As time passes the individuals converge which lowers the diversity and leads to exploitation. Premature convergence, which refers to the population's converging to a local optimum, can be the result of transitioning from exploration to exploitation too early [Riget and Vesterstrøm, 2002]. A balance must thus be reached between maintaining diversity and rapid convergence [Ursem, 2002].

Several diversity measures are available in the literature [Olorunda and Engelbrecht, 2008]. Diversity measures may be algorithm specific, for example tree similarity measures used for genetic programming [Burke *et al.*, 2004]. However, a basic diversity measure, D_{basic}, for an EA with a population of size n_I in n_d dimensions can be formulated as:

$$D_{basic} = \sum_{i=1}^{n_I} \|\vec{d} - \vec{x}_i\|_2 \tag{2.4}$$

where $\vec{d}$ is the average location of all individuals, calculated as:

$$d_j = \frac{\sum_{i=1}^{n_I} x_{i,j}}{n_I}, \ \forall \ j \in n_d \tag{2.5}$$

The diversity is thus calculated as the sum of the Euclidian distances of all the individuals from their average location. A low value of D_{basic} implies that the individuals are grouped spatially close together, while a large value implies that individuals are dispersed throughout the fitness landscape.

The diversity measure as calculated in equation (2.4) does not take into account that a large population size may artificially increase the diversity value (since the sum over the Euclidian distances would be taken over a large number of individuals). This problem can be remedied by normalising with respect to the population size, by changing the diversity calculation to give the normalised diversity, D_{norm} [Krink *et al.*, 2002]:

$$D_{norm} = \frac{\sum_{i=1}^{n_I} \|\vec{d} - \vec{x}_i\|_2}{n_I} \tag{2.6}$$

The magnitude of the range of the search space has an influence on how the calculated diversity value is interpreted. For example, $D = 0.8$ would be considered a high diversity in a two dimensional search space with range $[0, 1]$ for each dimension, but low in a 100 dimensional search with range $[0, 100000]$ for each dimension. Therefore, to simplify the interpretation of the diversity value, it is common to normalise with respect to the length of the longest diagonal in the search space, L [Ursem, 2002][Riget and Vesterstrøm, 2002]. Equation (2.6) is thus further adapted to calculate diversity, D, as

$$D = \frac{\sum_{i=1}^{n_I} \|\vec{d} - \vec{x}_i\|_2}{n_I L} \tag{2.7}$$

Equation (2.7) will be used as diversity measure throughout this thesis.

2.4 Differential Evolution

The purpose of this section is to describe differential evolution (DE) and to discuss variations of the basic DE algorithm. The original DE algorithm is discussed in Section 2.4.1, followed by a discussion of DE schemes in Section 2.4.2 and DE control parameters in Section 2.4.3. A review of approaches to eliminate or adapt DE control parameters is given in Section 2.4.4. DE applications are listed in Section 2.4.5.

2.4.1 Basic Differential Evolution

Differential evolution (DE) is a population-based, stochastic optimisation algorithm, created by Storn and Price [Price *et al.*, 2005] [Storn and Price, 1997]. DE differs from other EAs in the following aspects:

- Mutational step-sizes are not sampled from a prior specified distribution. Rather, mutations are made based on the spatial difference between two or more individuals added to a base vector.
- Mutation is applied first and thereafter the parent is combined with the mutated individual.

- Selection is applied as competition between the parent and the offspring.

Several variants of the DE algorithm have been suggested, but the original algorithm is given in Algorithm 3.

The following definitions are used for symbols in Algorithm 3: Each $\vec{v}_i$ is known as a mutant vector, $\mathcal{F}$ is known as the scale factor, $\vec{x}_{i_1}$ is referred to as the base vector, Cr is referred to as the crossover probability, $\vec{u}_i$ is referred to as a trial or offspring vector, and each $\vec{x}_i$ that is tested for replacement by $\vec{u}_i$ is known as a target or parent vector. The subscripts of $\vec{x}_{i_1}$, $\vec{x}_{i_2}$ and $\vec{x}_{i_3}$ are simplified to $\vec{x}_1$, $\vec{x}_2$ and $\vec{x}_3$, when no confusion is possible.

A thorough comparison of DE with 16 other optimisation algorithms showed that, while DE is not always the fastest method, it frequently produced the best results on a large number of benchmark problems [Price *et al.*, 2005]. In the first international contest on evolutionary optimisation, held in 1996, the performance of DE on benchmark functions was ranked third amongst several competing algorithms [Bersini *et al.*, 1996]. Vesterstrøm and Thomsen [2004] showed that DE performed better than PSO variants and a simple evolutionary algorithm on several benchmark functions. DE is now generally considered to be a reliable, accurate, robust and fast optimisation technique [Salman *et al.*, 2007].

2.4.2 Differential Evolution Schemes

Most variations of DE (called schemes) are based on different approaches to create the mutant vectors, $\vec{v}_i$ [Storn, 1996] (see equation (2.8)), and different approaches to create trial vectors. One of two crossover schemes are typically used to create trial vectors. The first, binomial crossover, is used in equation (2.9). The second approach is called exponential crossover, given in Algorithm 4.

By convention, schemes are labelled in the form DE/α/β/γ, where α is the method used to select the base vector, β is the number of difference vectors, and γ is the method used to create offspring. The scheme used in Algorithm 3 is referred to as DE/rand/1/bin.

Several methods of selecting the base vector have been developed and can be used with either of the crossover methods. Popular base vector selection methods include [Storn, 1996][Storn and Price, 1996] [Mezura-Montes *et al.*, 2006] (in each case the selected vectors' indexes are assumed to be unique):

Algorithm 3: Basic Differential Evolution

```
Generate a population, $P_x$, of $n_I$ individuals by creating vectors of random
candidate solutions, $\vec{x}_i$, $i = 1, \cdots, n_I$ and a dimensionality of $n_d$;
Evaluate the fitness, $F(\vec{x}_i)$, of all individuals.;
while termination criterion are not met do
    foreach $i = 1, \ldots, n_I$ do
        Select three individuals, $\vec{x}_{i_1}$, $\vec{x}_{i_2}$ and $\vec{x}_{i_3}$, at random from the current
        population such that $i_1 \neq i_2 \neq i_3 \neq i$;
        Create a new mutant vector $\vec{v}_i$ using:
            $$\vec{v}_i = \vec{x}_{i_1} + \mathcal{F} \cdot (\vec{x}_{i_2} - \vec{x}_{i_3}) \quad (2.8)$$
        where $\mathcal{F} \in (0, \infty)$;
        Create trial vector $\vec{u}_i$ as follows:
            $$u_{i,j} = \begin{cases} v_{i,j} \text{ if } (U(0,1) \leq Cr \text{ or } j == r) \\ x_{i,j} \text{ otherwise} \end{cases} \quad (2.9)$$
        where $Cr \in (0, 1)$ is the crossover probability and $r$ is a randomly selected
        index, i.e. $r \sim U(1, n_d)$;
        Evaluate the fitness of $\vec{u}_i$;
    end
    foreach target vector, $\vec{x}_i$, in the current population do
        Select corresponding $\vec{u}_i$ from trial population;
        If $\vec{u}_i$ has a better fitness value than $\vec{x}_i$ then replace $\vec{x}_i$ with $\vec{u}_i$;
    end
end
```

Algorithm 4: Exponential Crossover

```
$r \sim U(1, n_d)$;
$j = r$;
**repeat**
    $u_{i,j} = v_{i,j}$;
    $j = ((j-1) \bmod (n_d - 1)) + 1$;
**until** $U(0,1) \geq Cr$ or $j == r$;
**while** $j \neq r$ **do**
    $u_{i,j} = x_{i,j}$;
    $j = ((j-1) \bmod (n_d - 1)) + 1$;
**end**
```

DE/rand/2: Two pairs of difference vectors are used:

$$\vec{v}_i = \vec{x}_1 + \mathcal{F} \cdot (\vec{x}_2 + \vec{x}_3 - \vec{x}_4 - \vec{x}_5) \tag{2.10}$$

DE/best/1: The best individual in the population is selected as the base vector:

$$\vec{v}_i = \vec{x}_{best} + \mathcal{F} \cdot (\vec{x}_1 - \vec{x}_2) \tag{2.11}$$

Using the best individual as the base vector encourages fast convergence (a consequence of which is exploitation) since offspring individuals will always be a mutation of the best individual.

DE/best/2: The best individual in the population is used as base vector in conjunction with two pairs of difference vectors:

$$\vec{v}_i = \vec{x}_{best} + \mathcal{F} \cdot (\vec{x}_1 + \vec{x}_2 - \vec{x}_3 - \vec{x}_4) \tag{2.12}$$

DE/rand-to-best/1: A point between a randomly selected individual and the current best individual is used as the base vector:

$$\vec{v}_i = \vec{x}_1 + \mathcal{G} \cdot (\vec{x}_{best} - \vec{x}_1) + \mathcal{F} \cdot (\vec{x}_2 - \vec{x}_3) \tag{2.13}$$

where $\mathcal{G}$ is a parameter that scales the contribution of the position of the current best individual to the resultant base vector. Should $\mathcal{G}$ be assigned a value of one, DE/rand-to-best/1 is equivalent to DE/best/1 (which encourages exploitation). A value of zero for $\mathcal{G}$ would result in the position of the best individual being ignored, thus favouring exploration, as the mutant vector would not be biased towards the best individual. $\mathcal{G}$ thus controls the balance between exploitation and exploration of the fitness landscape, and is referred to as the greediness of the scheme.

DE/current-to-best/1: A point between the target vector and the current best individual is used as the base vector:

$$\vec{v}_i = \vec{x}_i + \mathcal{G} \cdot (\vec{x}_{best} - \vec{x}_i) + \mathcal{F} \cdot (\vec{x}_1 - \vec{x}_2) \tag{2.14}$$

where $\mathcal{G}$ is a parameter that scales the contribution made by the target and the best individual to the resultant base vector. A $\mathcal{G}$ value of zero results in only the target vector contributing, while a value of one results in only the best individual contributing. Exploration versus exploitation around the target vector and best individual is thus controlled by $\mathcal{G}$.

DE/current-to-rand/1: A point between a randomly selected individual and the target vector is used as the base vector:

$$\vec{v}_i = \vec{x}_i + \mathcal{G} \cdot (\vec{x}_1 - \vec{x}_i) + \mathcal{F} \cdot (\vec{x}_2 - \vec{x}_3) \tag{2.15}$$

where $\mathcal{G}$ is a parameter that scales the contribution made by the target and the randomly selected individual to the resultant base vector. A $\mathcal{G}$ value of zero results in only the target vector contributing, while a value of one results in only the randomly selected individual contributing. Exploration versus exploitation around the target vector is thus controlled by $\mathcal{G}$.

2.4.3 Differential Evolution Control Parameters

Differential evolution algorithms have several control parameters that have to be set. Ignoring extra parameters introduced by some DE schemes, the main DE control parameters are the population size (n_I), scale factor ($\mathcal{F}$) and crossover factor (Cr).

The scale factor controls the magnitude of the difference vector and consequently the amount by which the base vector is perturbed. Large values of $\mathcal{F}$ encourage large scale exploration of the fitness landscape but could lead to premature convergence, while small values result in a more detailed exploration of the local fitness landscape (exploitation) while increasing convergence time.

The crossover factor controls the diversity of the population, since a large value of Cr will result in a higher probability that new genetic material will be incorporated into the population. Large values of Cr result in broad exploration of the fitness landscape, which in turn may result in slow convergence. Conversely, very small values result in very little genetic material being introduced into the population which may lead to premature convergence. A value for Cr must thus be used that is large enough to ensure sufficient exploration, but small enough to allow exploitation and consequently acceptably fast convergence times.

General guidelines for the values of parameters that work reasonably well on a wide range of problems are known. For example, Zaharie [2002a] showed on three test functions that the smallest reliable value for $\mathcal{F}$ is 0.3 when $Cr = 0.2$ and $n_I = 50$. Storn and Price [1997] showed that $0 \leq Cr \leq 0.2$ worked well on decomposable functions, while $0.9 \leq Cr \leq 1$ worked well on functions that are not decomposable. Other recommendations are available in the literature [Storn, 1996][Rönkkönen *et al.*, 2005][Pedersen, 2010], however, best results in terms of accuracy and convergence time are found if the parameters are tuned for each problem individually [Engelbrecht, 2007].

The population size influences the diversity of the population. A large population size, rather than a small population size, makes a more diverse sampling of the search space likely. The population size also controls the number of function evaluations that are performed per iteration. A small population size would result in a larger number of iteration than would result from a large population size, when a constant number of function evaluations is available. A value for n_I must thus be selected to ensure that an adequate number of iterations can be performed, but also that the fitness landscape is sufficiently explored.

2.4.4 Adapting and Eliminating Control Parameters

A disadvantage of control parameters is that their fine-tuning is a time consuming manual task. Furthermore, ideal values for the control parameters may vary during the evolution process. For example, values that encourage exploration may be desirable initially, while values that encourage exploitation may be effective at a later time during the algorithm's execution.

Three broad strategies have emerged to address the disadvantages associated with DE control parameters. The first strategy uses adaptive control parameters and is discussed in Section 2.4.4.1. The values of the control parameters are adapted using information gathered during the optimisation process or to predetermined values. The second strategy is to use self-adapting control parameters. This is discussed in Section 2.4.4.2. Parameters are incorporated into the evolution process, which results in the optimisation of the control parameters in parallel with optimising the fitness landscape [Eiben *et al.*, 2000]. The third strategy is to explicitly eliminate the need to tune control parameters from the algorithm and is discussed in Section 2.4.4.3.

The computational intelligence community have not reached consensus on when an algorithm should be classified as "adaptive" and when it should be classified as "self-adaptive". For example, Brest *et al.* [2006] and Qin and Suganthan [2005] describe their algorithms as "self-adaptive", while Zhang and Sanderson [2009] argue that these algorithms should be described as "adaptive". The following convention is used in this thesis: Algorithms that explicitly control the values of control parameters during the optimisation process are classified as "adaptive". For example, linearly decreasing the scale factor as a function of time (initially to encourage exploration and later to encourage exploitation) are classified as "adaptive". Algorithms that select control parameters based on the success rate of previous values during the optimisation process are classified as "self-adaptive".

Ideally, algorithms that adapt control parameters should reduce the number of parameters, or preferably eliminate all parameters. However, the literature review presented in this section illustrates that, in practice, adaptive and self-adaptive control parameter algorithms do not in all cases reduce the number of parameters. Several approaches to adapting or eliminating DE parameters are available in the literature and are discussed in this section. Algorithms that are relevant to Chapter 6, which presents the incorporation of self-adaptive control parameters into the algorithms that are developed in this thesis, are described in more detail.

2.4.4.1 Adaptive Control Parameters

A simple method of adapting a control parameter is to linearly vary it between two values as a function of the number of iterations. Das *et al.* [2005] linearly decreased the scale factor from 1.2 to 0.4 during the course of the optimisation process. A disadvantage of this approach is that the number of iterations that will be performed must be known in advance.

Ali and Törn [2004] proposed an algorithm that adapts the scale factor at the end of each generation using:

$$\mathcal{F}_{new} = \begin{cases} \max\left\{0.4, 1 - \left|\dfrac{\max\limits_{\vec{x}_a \in P_x}\{F(\vec{x}_a)\}}{\min\limits_{\vec{x}_b \in P_x}\{F(\vec{x}_b)\}}\right|\right\} & \text{if } \left|\dfrac{\max\limits_{\vec{x}_a \in P_x}\{F(\vec{x}_a)\}}{\min\limits_{\vec{x}_b \in P_x}\{F(\vec{x}_b)\}}\right| < 1 \\ \max\left\{0.4, 1 - \left|\dfrac{\min\limits_{\vec{x}_a \in P_x}\{F(\vec{x}_a)\}}{\max\limits_{\vec{x}_b \in P_x}\{F(\vec{x}_b)\}}\right|\right\} & \text{otherwise} \end{cases} \tag{2.16}$$

The effect of this approach is that small scale factors are used when the difference in function value between the best and the worst individuals in the population is large (this should occur during the initial stages of the evolution). The approach of Ali and Törn [2004] thus initially encourages exploitation. Larger scale factors are produced when the difference in function value between the best and the worst individuals in the population is small (which occurs once the population has converged). Exploration is thus favoured during the later stages of the optimisation process. A disadvantage of this algorithm is that adding a constant to F would result in different scale factors being produced.

Liu and Lampinen [2005] proposed an approach that uses fuzzy logic controllers to adapt the scale factor, $\mathcal{F}$, and crossover factor, Cr. This algorithm is called fuzzy adaptive differential evolution (FADE). FADE employs knowledge of a human expert (in the form a fuzzy rules) in order to adjust values of $\mathcal{F}$ and Cr based on differences in position and fitness of individuals in successive generations. It was shown that this approach outperformed normal DE on high dimensional problems. A disadvantage of FADE is that the fuzzy rules have to be manually created for a specific type of problem and consequently depends on the subjectivity of the human expert [Liu and Lampinen, 2005]. Rules can only be as effective as the specialised knowledge of the expert.

Zaharie [2002a] concluded that premature convergence of the DE algorithm can be avoided by selecting appropriate values for the scale factor, $\mathcal{F}$, and the crossover factor, Cr, during the optimisation process. These parameters should be selected such that higher levels of divesity are maintained for longer to facilitate better exploration. Consequently, an algorithm that adapts $\mathcal{F}$ and Cr to avoid loss of diversity was developed [Zaharie, 2002b]. This approach introduces a new parameter, ϖ, into the algorithm which determines the convergence rate of the DE algorithm. Values of $\varpi > 1$ increases the diversity of the population, thus reducing the probability of premature convergence. Values of $\varpi < 1$ increases the convergence rate. The problem of tuning the two interrelated parameters $\mathcal{F}$ and Cr is thus reduced to tuning only one parameter, ϖ.

2.4.4.2 Self-Adaptive Control Parameters

Abbass [2002] introduced a self-adaptive crossover operator into the Pareto differential evolution algorithm [Abbass and Sarker, 2002], which uses the DE/rand/1/bin scheme. Each individual, $\vec{x}_i$, in the population stores its own crossover value Cr_i. When individuals, $\vec{x}_1$, $\vec{x}_2$ and $\vec{x}_3$, are selected to create a mutant vector, the crossover value, Cr_{new}, to be used in equation (2.9) is calculated as follows:

$$Cr_{new_i} = Cr_1 + N(0,1) \cdot (Cr_2 - Cr_3) \tag{2.17}$$

where Cr_1, Cr_2 and Cr_3 are the crossover values associated with individuals $\vec{x}_1$, $\vec{x}_2$ and $\vec{x}_3$, respectively. The value of Cr_{new_i} is forced to be within the bounds $[0,1]$ by making use of a repair rule. The repair rule essentially keeps the decimal part of Cr_{new_i} (i.e. $Cr_{new_i} - \lfloor Cr_{new_i} \rfloor$). A value of 3.2 would thus be changed to 0.2 by the repair rule. Cr_{new_i} is then associated with the newly created offspring individual. The scale factor $\mathcal{F}_{new_i}$, used in equation (2.8), is selected randomly from a Gaussian distribution, $N(0,1)$, clipped to the range $[0,1]$.

Self-adaptive differential evolution (SaDE), an approach that adapts the values of $\mathcal{F}$, Cr and the DE scheme, was proposed by Qin and Suganthan [2005]. SaDE adapts the DE scheme by utilising DE/rand/1/bin and DE/best/2/bin with probabilities ρ_1 and ρ_2, respectively. The probabilities, ρ_1 and ρ_2, are assigned initial probabilities of 0.5 each, and subsequently receive values based on the rate at which each has been used to generate trial vectors that successfully replaced the target vector. DE was found to be less sensitive to the value of the scale factor, and $\mathcal{F}$ is consequently allowed to assume random numbers sampled from $N(0.5, 0.3)$, clipped to the range $(0, 2]$. Crossover probabilities, Cr_i, are associated with each individual, and is originally created from a normal distribution, $N(0.5, 0.1)$. Each Cr_i is randomly perturbed at regular time intervals ($\rho_3 = 5$ generations) from the distribution $N(\rho_4, 0.1)$, where ρ_4 is originally set to 0.5. The algorithm stores all Cr_i values which result in offspring replacing parent individuals. These successful values are averaged and assigned to ρ_4 every $\rho_5 = 25$ generations, whereafter the list of successful values is cleared. Qin and Suganthan [2005] do not explicitly explain how the algorithm treats randomly generated values of Cr_i that are larger than 1.0 or smaller than zero. However, since the background section of [Qin and Suganthan, 2005] states that Cr is a number in the range $[0, 1)$, it is presumed that Cr_i is clipped to the range $[0, 1)$. This algorithm is classified as self-adaptive since only successful values of Cr_i are averaged to replace ρ_4.

A weakness of SaDE is that five potentially problem dependent parameters are introduced: ρ_3, ρ_5, and the original values of ρ_1, ρ_2 and ρ_4. Furthermore, ρ_3 and ρ_5 clearly depend on the population size (since the population size affects the number of function evaluations before Cr is adapted).

The self-adaptive differential evolution (SDE) algorithm [Omran *et al.*, 2005a] uses a technique similar to that of Abbass and Sarker [2002], but in this case the scale factor is adapted rather than the crossover factor. Each individual, $\vec{x}_i$, stores its own scale factor $\mathcal{F}_i$. When individuals, $\vec{x}_1$, $\vec{x}_2$ and $\vec{x}_3$, are selected to create mutant vectors using the DE/rand/1/bin scheme, three extra vectors $\vec{x}_4$, $\vec{x}_5$ and $\vec{x}_6$ are selected from the population in order to create a new scale factor. The scale factor to be used in equation (2.8) is then calculated as follows:

$$\mathcal{F}_{new_i} = \mathcal{F}_4 + N(0, 0.5) \cdot (\mathcal{F}_5 - \mathcal{F}_6) \tag{2.18}$$

where $\mathcal{F}_4$, $\mathcal{F}_5$ and $\mathcal{F}_6$ are the scale factors associated with $\vec{x}_4$, $\vec{x}_5$ and $\vec{x}_6$, respectively. The crossover factor, Cr_{new}, is sampled from a Gaussian distribution $N(0.5, 0.15)$ to create each new offspring individual. It is not clear from [Omran *et al.*, 2005a] how random values of Cr_{new} outside the range $(0, 1)$ are treated. However, since 99.7% of the numbers sampled from the Gaussian distribution $N(0.5, 0.15)$ falls within the range $(0.05, 0.95)$, it can be assumed that the values are clipped to the range $(0, 1)$. Values outside the range $(0, 1)$ occur extremely infrequently and the assumption should not have a significant impact on the performance of the algorithm. It was shown that SDE produced favourable results when compared to other versions of DE [Salman *et al.*, 2007].

Brest *et al.* [2006] presented an algorithm that self-adapts the values of both the crossover factor and the scale factor. This algorithm is referred to here as jSADE. Each individual, $\vec{x}_i$, stores its own value for the crossover factor, Cr_i, and scale factor, $\mathcal{F}_i$. Before equation (2.8) is used to create a new mutated individual, the scale factor and crossover factor of the target individual ($\vec{x}_i$ in equation (2.9)) are used to create new values for the scale factor and crossover factor to be used in equations (2.8) and (2.9). The new scale and crossover factors are calculated as follows:

$$\mathcal{F}_{new_i} = \begin{cases} \mathcal{F}_l + U(0,1) \cdot \mathcal{F}_u \text{ if } (U(0,1) < \tau_1) \\ \mathcal{F}_i \text{ otherwise} \end{cases} \tag{2.19}$$

$$Cr_{new_i} = \begin{cases} U(0,1) \text{ if } (U(0,1) < \tau_2) \\ Cr_i \text{ otherwise} \end{cases} \tag{2.20}$$

where τ_1 and τ_2 are the probabilities that the factors will be adjusted. Brest *et al.* [2006] used 0.1 for both τ_1 and τ_2. Other values used were $\mathcal{F}_l = 0.1$ and $\mathcal{F}_u = 0.9$. $\mathcal{F}_l$ is a constant introduced to avoid premature convergence by ensuring that the scale factor is never too small (see Section 2.4.3), while $\mathcal{F}_u$ determines the range of scale factors that can be explored by the algorithm.

Brest *et al.* [2006] showed that this algorithm was better or at least comparable to DE and other evolutionary algorithms, including FADE [Liu and Lampinen, 2005], using various benchmark functions. However, a disadvantage of the algorithm is that, while $\mathcal{F}$ and Cr need not be tuned, four new parameters are introduced (τ_1, τ_2, $\mathcal{F}_l$ and $\mathcal{F}_u$). Fine tuning of these parameters may be required for best results on specific problems.

jSADE was extended by Brest *et al.* [2007] to form jSADE2. This algorithm utilises the approach from jSADE to self-adapt the scale and crossover factors for two DE schemes, DE/rand/1/bin and DE/best/1/bin. A pair of scale and crossover values are stored by each individual for each of the schemes. Schemes are selected using the same approach as

SaDE. jSADE2 further differs from jSADE in that the τ_3 worst performing individuals are randomly reinitialised every τ_4 generations. A drawback of jSADE2 is that it introduces two additional parameters (τ_3 and τ_4) to be tuned.

Zhang and Sanderson [2009] proposed JADE, a DE algorithm that self-adapts the scale and crossover factors. JADE employs a new scheme, which is similar to DE/current-to-best/1/bin. The new scheme is called DE/current-to-κbest/1/bin by Zhang and Sanderson [2009] and is given by

$$\vec{v}_i = \vec{x}_i + \mathcal{F}_i \cdot (\vec{x}^{\kappa_1}_{best} - \vec{x}_i) + \mathcal{F}_i \cdot (\vec{x}_1 - \vec{x}_2) \tag{2.21}$$

where $\kappa_1 \in (0, 100]$ and $\vec{x}^{\kappa_1}_{best}$ is randomly selected from the best κ_1% individuals in the the population. The crossover factor for each individual at each generation is calculated as

$$Cr_{new_i} = N(\mu_{Cr}, 0.1) \tag{2.22}$$

where μ_{Cr} (which is originally set to $\kappa_2 = 0.5$) is updated at the end of each generation using

$$\mu_{Cr,new} = (1-\kappa_3)\mu_{Cr} + \kappa_3 \frac{\sum\limits_{Cr_O \in \mathcal{O}_{Cr}} Cr_O}{|\mathcal{O}_{Cr}|} \tag{2.23}$$

where $\mathcal{O}_{Cr}$ is the set of all successful crossover factors (i.e. values of Cr that resulted in trial vectors that replaced parent vectors), and $\kappa_3 \in [0, 1]$ is a parameter to the algorithm. The value of $\mu_{Cr,new}$ is thus calculated as the weighted average of the current value and the mean of all previous successful crossover factors. Cr_{new_i} is truncated to the range $[0, 1]$.

JADE calculates the scale factor for each individual at each generation as

$$\mathcal{F}_{new_i} = C(\mu_{\mathcal{F}}, 0.1) \tag{2.24}$$

where $\mu_{\mathcal{F}}$ (which is originally set to $\kappa_4 = 0.5$) is updated at the end of each generation using

$$\mu_{\mathcal{F},new} = (1-\kappa_3)\mu_{\mathcal{F}} + \kappa_3 \frac{\sum\limits_{\mathcal{F}_O \in \mathcal{O}_{\mathcal{F}}} \mathcal{F}_O^2}{\sum\limits_{\mathcal{F}_O \in \mathcal{O}_{\mathcal{F}}} \mathcal{F}_O} \tag{2.25}$$

where $\mathcal{O}_{\mathcal{F}}$ is the set of all successful scale factors. Equation (2.25) uses the Lehmer mean to place more emphasis on larger scale factors, while equation (2.24) uses a Cauchy distribution to encourage more diverse values of $\mathcal{F}_{new_i}$. $\mathcal{F}_{new_i}$ is truncated to the range $[0, 1]$.

JADE also contains an optional archive component which was found to be beneficial for high dimensional problems. The archive population, which is initially empty, is grown by adding trial vectors that do not replace parent vectors in the selection step of the DE algorithm. Individuals are randomly removed from the archive when the size of the archive exceeds the size of the normal DE population. The archive is utilised by the DE algorithm by randomly selecting individuals for mutation from the union of the archive and normal DE populations.

Zhang and Sanderson [2009] report better results for JADE in comparison with SaDE and jSADE on several benchmark problems. A disadvantage of the JADE algorithm is that four new parameters are introduced (κ_1, κ_2, κ_3 and κ_4). The authors do, however, argue that JADE is relatively insensitive to these parameters, and that JADE works effectively on a large range of values of the parameters.

The preceding approaches that were discussed all focused on adapting the scale and crossover factors. An algorithm that self-adapts the population size was presented by Teo [2006]. In this algorithm, each individual encodes a value for population size which is evolved along with the individual's position. The average of all the individual population size values is then used to calculate the actual population size for the next generation. When the population size is increased, copies of the best individual in the current population are inserted (in addition to all the individuals in the current population) into the new population until the new population has grown to the appropriate size. In cases where the population size is reduced, the weakest individuals in the current populations are not copied to the population of the next generation.

Brest *et al.* [2008] also adapted the population size with jSADE3, an extension to jSADE. jSADE3 halves the population size every τ_5 function evaluations. The fitness of each individual in the first half of the population is compared to the fitness of the corresponding individual in the second half of the population (i.e. the first individual in the first half is compared to the first individual in the second half, etc.). After each comparison, the fittest individual is placed in the new population. The old population is discarded after the comparison process and evolution continues using the new, smaller, population. jSADE3 does not provide a mechanism to increase the population size.

jSADE3 further differs from jSADE in that the sign of the scale factor is changed, with a probability of $\tau_6 = 0.75$, if $F(\vec{x}_2) > F(\vec{x}_3)$ (assuming a function minimisation problem), where $\vec{x}_2$ and $\vec{x}_3$ are the randomly selected difference vectors used in the DE/rand/1/scheme. The difference vector thus points towards the individual with the lowest error value after changing the sign. Note that neither of the changes made to jSADE to form jSADE3 can be classified as "self-adaptive". jSADE3 was, however, included in this section because of the self-adaptive components inherited from jSADE.

A significant disadvantage of the jSADE3 algorithm is that the number of function evaluations that are used during the optimisation process must be known in advance in order to set parameter τ_5. jSADE3 could potentially reduce the population size to the point where it is impossible to execute the DE algorithm (i.e. the population could become too small to select unique individuals for the trial-vector creation step) if τ_5 is set too low.

2.4.4.3 Eliminating Control Parameters

Three of the the self-adaptive approaches described in the previous section eliminated the need to fine-tune control parameters by sampling their values from a distribution. The approaches of Abbass and Sarker [2002] and Qin and Suganthan [2005] used random values for the scale factor, while Omran *et al.* [2005a] used random values for the crossover factor. These parameters were thus not self-adapted. The randomisation of control parameters to eliminate the need for fine-tuning is a common approach, and has been used by several researchers [Price *et al.*, 2005] [Das *et al.*, 2005].

Several researchers have investigated DE and PSO hybrids [Hendtlass, 2001], [Zhang and Xie, 2003]. Omran *et al.* [2009] presented a self-adaptive hybrid PSO and DE algorithm, called the Barebones DE, as a means of eliminating DE control parameters. Three significant changes were made to the standard DE algorithm. Firstly, each individual is given a memory that stores the best location that the individual has occupied during the execution of the algorithm. This is called the *personal best* value and is labelled $\vec{y}_i$ for individual $\vec{x}_i$. The fittest *personal best* value within the entire population is called the *global best* and is labelled $\vec{\hat{y}}$. In the second place, the competition between parents and offspring of DE is eliminated and offspring are placed directly into the main population, even if they are less fit than the original individual. Good solutions are not lost, however, since these solutions will be stored in the *personal best* value. The third change is in how offspring are created. The trial vector is calculated as:

$$v_{i,j} = r_{j_1} \cdot y_{i,j} + (1 - r_{j_1}) \cdot \hat{y}_j + r_{j_2} \cdot (x_{1,j} - x_{2,j}) \tag{2.26}$$

where r_{j_1},$r_{j_2} \sim U(0,1)$. The trial vector is thus computed as the weighted average of the current individual's personal best and the global best. The choice for trial vector is motivated by studies that showed that PSO particles converge to the weighted average of their personal and neighbourhood best positions [van den Bergh and Engelbrecht, 2006][Trelea, 2003][Clerc and Kennedy, 2002]. Note that the scale factor has been eliminated.

The crossover step (see equation (2.9)) is replaced by:

$$x_{i,j} = \begin{cases} v_{i,j} \text{ if } (U(0,1) \leq Cr) \\ y_{3,j} \text{ otherwise} \end{cases} \tag{2.27}$$

Crossover is thus performed with the *personal best* of one of the randomly selected parent individuals. Note that the crossover factor has not been eliminated.

Omran *et al.* [2009] compared Barebones DE to DE, PSO and Barebones PSO. It was concluded that the Barebones DE is more noise tolerant than the other approaches and that it gave superior performance on high-dimensional problems.

2.4.4.4 Summary

The majority of approaches discussed in this section are aimed at adapting, or eliminating the need to tune the scale and crossover factors. The purely adaptive approaches removed the need to tune both the scale and crossover factors, but in the case of the approach of Zaharie [2002b], a new parameter was introduced which must be fine-tuned. In the case of FADE, the approach of Liu and Lampinen [2005], the fuzzy rules must be manually created, which is arguably as onerous as fine-tuning control parameters and subjective.

An advantage of the self-adaptive approaches described in this section is that they are significantly simpler than the adaptive approaches that were discussed. However, only two of the self-adaptive approaches, [Zhang and Sanderson, 2009][Brest *et al.*, 2006], applies self-adaption to both the scale and crossover factors, but at the expense of introducing four new parameters into the algorithm. The other self-adaptive approaches only self-adapt one of the two factors, while the need to tune the other factor is eliminated by repeatedly selecting it from a normal distribution.

The hybrid DE and PSO approach of Omran *et al.* [2009] completely removes the scale factor from the algorithm, but retains the crossover factor.

2.4.5 Differential Evolution Applications

Differential evolution has been applied to a wide range of problems, including image analysis [Li *et al.*, 2003][Xu and Dony, 2004][Omran *et al.*, 2005b], neural network training [Chen *et al.*, 2002][Magoulas *et al.*, 2004][Moalla *et al.*, 2002], scheduling [Lin *et al.*, 2000][Rae and Parameswaran, 1998][Rzadca and Seredynski, 2005], and controllers [Chang and Du, 2000][Chiou and Wang, 1998][Cruz *et al.*, 2003]. This thesis investigates the use of DE in dynamic environments.

2.5 Dynamic Environments

This section commences with a formal definition of dynamic environments in Section 2.5.1. Section 2.5.2 provides a discussion of the different types of dynamic environments and the factors that influence an algorithm's ability to effectively locate optima in a dynamic environment. The moving peaks benchmark and the generalised dynamic benchmark generator, two benchmarks used to simulate dynamic environments, are discussed in Sections 2.5.3 and 2.5.4 respectively. Various performance measures have been suggested for use in dynamic optimisation problems, which are described in Section 2.5.5.

2.5.1 Formal Definition

The solution to real-world optimisation problems often vary over time. Consider, for example, the optimal air-fuel mixture of a aircraft during flight. The optimal mixture at any point in time depends on the prevailing winds, the altitude and the speed of the aircraft, and also varies due to the change in mass caused by burning fuel.

A dynamic environment, in the context of optimisation problems, is a fitness landscape that varies over time. Formally, $\exists\ t, t' \in \mathcal{T}$, where $\mathcal{T}$ is the set of all time steps, and $\exists\ \vec{x} \in \mathcal{S}^{n_d} \subseteq \mathbb{R}^{n_d}$ such that

$$F(\vec{x}, t) \neq F(\vec{x}, t') \tag{2.28}$$

where F is a dynamic function. Consequently, the optima in the fitness landscape may vary in number, location, and function value over time. The objective of an algorithm applied to a dynamic optimisation problem (DOP) is to find the best solutions at all time steps during the optimisation process [Weicker, 2002], i.e. for all $t \in \mathcal{T}$, assuming a minimisation problem, find

$$\vec{x}^*(t) : F(\vec{x}^*(t), t) \leq F(\vec{x}, t)\ \forall\ \vec{x} \in \mathcal{S}^{n_d} \tag{2.29}$$

where $\vec{x}^*(t) \in \mathcal{S}^{n_d}$ is the location of a global optimum at time step t.

The above discussion gives a broad definition of dynamic environments. The following section explores various types of dynamic environments and the characteristics of each.

2.5.2 Types of Dynamic Environments

Several classifications of dynamic environments based on different characteristics are available in the literature [Angeline, 1997][Trojanowski and Michalewicz, 1999a][Branke, 2002][Li *et al.*, 2008][Hu and Eberhart, 2001][De Jong, 1999]. This section endeavors to synthesise disparate classification approaches into a single unified classification scheme by which different types of dynamic environments can be identified.

The three mayor considerations to be taken into account when classifying dynamic environments are the fitness landscape composition, the types of changes and the pervasiveness of changes. An individual discussion of these considerations follows:

1. **Fitness landscape composition**: Changes in a dynamic environment depend in a large extend on the underlying composition of the fitness landscape. The composition of a search space can be classified as either homogeneous or heterogeneous:

   (a) A homogeneous fitness landscape is made up of a single underlying function. Such a fitness landscape was suggested by Angeline [1997]. The underlying function is moved with respect to the coordinate system of the search space to create a change in the environment. The number of optima and the function value of each optimum thus depends purely on the underlying function that is used and does not change over time.

(b) A heterogeneous fitness landscape contain multiple underlying functions. Two ways of how underlying functions are combined have been suggested:

i. At each point in the fitness landscape the sum of the function values of all underlying functions is taken [Li *et al.*, 2008], i.e. $F(\vec{x}, t) = \sum_{p=1,\ldots,n_p} f_p(\vec{x})$, where f_p is one of the n_p underlying functions. As the underlying functions are moved with respect to the coordinate system of the search space, optima in the fitness landscape are formed at locations where the optima of the underlying functions correspond, analogous to constructive interference in wave theory [Feynman *et al.*, 1963]. The number of optima in the fitness landscape and their respective function values thus not only depend on the number of optima and function values of the underlying functions, but also on the relative position and orientation of the underlying functions with respect to the coordinate system of the search space.

ii. At each point in the fitness landscape the maximum function value of all underlying functions is taken for function maximisation problems (i.e. $F(\vec{x}, t) = \max_{p=1,\ldots,n_p}\{f_p(\vec{x})\}$, where f_p is one of the n_p underlying functions) and the minimum is taken for function minimisation problems (i.e. $F(\vec{x}, t) = \min_{p=1,\ldots,n_p}\{f_p(\vec{x})\}$) [Branke, 2002]. The effect of this approach is that some optima may be obscured by other optima, thus varying the number of optima in the fitness landscape over time. The function value of the global optimum, however, depends purely on the function value of the most optimal underlying function.

2. **Change types**: The nature of changes in a dynamic environment influence approaches that can be followed by algorithms to recover from changes. The three broad change types that have been identified:

(a) Random changes imply that a particular change in the environment does not depend on a previous change [Trojanowski and Michalewicz, 1999a], i.e. no pattern governing changes exist.

(b) Chaotic changes refer to changes where a dependency, or pattern, between successive changes exists. This is in contrast to random changes where each change is completely independent of the previous. According to Trojanowski and Michalewicz [1999a], the dependency between changes may be so complex that changes appear random and may thus not be predictable. On the other hand, if the relationship between successive changes is simple enough, changes may be considered predictable, and could be exploited by an optimisation algorithm. Angeline [1997] identified two types of predictable changes:

   i. Linear, where successive changes are linearly correlated.

   ii. Cyclic (or recurrent), where the dynamic environment periodically returns to the same state.

(c) Combinations of chaotic and random changes are changes where some relationship between successive changes exist, but which also contain a random component [Li *et al.*, 2008][Branke, 2002].

3. **Change Pervasiveness**: This metric classifies dynamic environments on the attributes of the environment that are modified. Hu and Eberhart [2001] identified three classifications based on properties of a dynamic environment that can change, to which a fourth category can be added:

   (a) The locations of optima in the environment change (referred to by Hu and Eberhart [2001] as Type I environments).

   (b) The locations of the optima remain the same, but the values of the optima change (referred to by Hu and Eberhart [2001] as Type II environments). A consequence of this is that different optima can become the global optimum over time.

   (c) The locations and the values of the optima change (referred to by Hu and Eberhart [2001] as Type III environments).

   (d) A classification not included in the categorisation of Hu and Eberhart [2001] is represented by environments in which new optima are explicitly introduced or old optima are removed, i.e. the number of optima does not remain constant.

The above taxonomy (i.e. fitness landscape composition, change type and change pervasiveness) can be used to classify a dynamic environment, but does not encompass all the physical characteristics that describes a dynamic environment. The type of dynamic environment also depends on factors that influence an optimisation algorithm's ability to successfully locate optima.

It is argued here that the three main factors that combine to influence an optimisation algorithm's ability to locate optima successfully in a dynamic environment are the hardness of the fitness landscape, the frequency at which changes occur, and the severity of changes to the environment. The the three factors are respectively described below:

Hardness of the fitness landscape: Hardness is a term that is used in this thesis to describe the intuitive concept of the difficulty involved in optimising an objective function by an optimisation algorithm. Intuitively, an easy optimisation problem implies that a specific optimisation algorithm is likely to successfully locate a global optimum within relatively few iterations. Conversely, a hard optimisation problem implies that the algorithm typically requires a relatively large number of iterations to locate the global optimum, or that the algorithm lacks the ability to locate the global optimum. This implies that the algorithm is likely to either converge to an abitrary point or to a local optimum in the fitness landscape of a hard problem. Note that the No-Free-Lunch theorems [Wolpert and Macready, 1997] imply that not all optimisation problems are equally hard for all optimisation algorithms.

Several factors can influence the hardness of a fitness landscape, for example, the modality of the fitness landscape (i.e. the number of optima), the gradient at various points in the fitness landscape and the presence of ridges, valleys and plateaus. In addition, the number of dimensions of a search space usually influences the hardness, since increasing the number of dimensions increases the size of the search space. Note that there are exceptions to this, such as the Griewank function, which has been shown to become easier as the dimension increases [Locatelli, 2003].

Scientists have endeavored to quantify the hardness of fitness landscapes by means of metrics like ruggedness [Malan and Engelbrecht, 2010], fitness distance correlation

[Jones and Forrest, 1995], and the dispersion metric [Lunacek and Whitley, 2006]. To date, no single satisfactory problem hardness measure has been found [He *et al.*, 2007][Malan and Engelbrecht, 2009].

An optimisation algorithm has a limited number of iterations within which to locate a global optimum before a change in a dynamic environment occurs. The quality of the solution found by the optimisation algorithm is thus directly influenced by the hardness of the search landscape.

Frequency of changes: Changes in a dynamic environment can either be at discrete time intervals (the environment is static for periods between changes) or continuous (the environment changes every time it is sampled) [Trojanowski and Michalewicz, 1999a]. As the frequency of changes tends to infinity, changes intervals tend from discrete to continuous. As the frequency of changes tend towards zero, the dynamic environment tends towards a static environment.

Assume that a particular optimisation algorithm has the ability to locate a global optimum in specific static fitness landscape. This assumption does not imply that the algorithm will locate a global optimum in a dynamic version of the same fitness landscape, as the frequency of changes to the fitness landscape will influence the likelihood of locating a global optimum. For example, in situations where changes are highly infrequent (i.e. many iterations can be performed before a change occurs), the optimisation algorithm has more time to locate optima and the likelihood of locating a global optimum is thus increased. However, if changes are frequent, the optimisation algorithm will be able to perform fewer iterations between changes, and the chances of locating optima are reduced. Performance is consequently adversely affected by frequent changes as algorithms must locate optima in fewer iterations.

Severity of changes: Changes in a dynamic environment change the fitness landscape surface. The degree of correlation between the fitness landscape before a change and after a change depends on the severity of the change.

Factors that influence the severity of changes in a dynamic environment include: the amount by which the locations of optima are changed, how much the values of the optima change, and whether new optima are introduced or old optima are removed [De Jong, 1999]. If the changes in the environment are so severe that the fitness landscape after the change bears no relation to the search space before the change, an optimisation algorithm cannot exploit any information collected regarding the fitness landscape before the change, and optimisation might as well resume from scratch [Branke *et al.*, 2000].

Conversely, very small changes in the environment, which result in the location of an optimum remaining close to its old position, makes it possible for an optimisation algorithm to exploit the information regarding the old position of the optimum in order to find the new position. The severity of changes thus determines how much useful information regarding the shape of the search space can be transferred from before a change in the environment to after the change. A large amount of useful information can be exploited by an optimisation algorithm to recover quickly from a change in the environment, while if only a small amount of useful information can be exploited, the recovery rate of the algorithm is likely to be reduced.

The interaction between change frequency and change severity has a considerable impact on whether it is feasible for an optimisation algorithm to optimise a dynamic environment. Figure 2.1 illustrates how the change frequency and change severity impact on a feasible region (where it is possible to solve the problem) which exists for any algorithm on any dynamic optimisation problem. The feasible region corresponds to either changes that are small enough or changes that are infrequent enough (or both), so that an optimisation algorithm is able to function adequately.

A high change severity, even in the presence of a low change frequency, results in successive environments that are too uncorrelated for any useful information from before the change to be transferred to after the change. Similarly, if the change frequency is too high in the presence of even small changes, the optimisation algorithm has too little time to recover from changes. There thus exist upper bounds on the frequency of changes and the severity of changes beyond which it is no longer viable to perform optimisation successfully. Figure 2.1 is shown to illustrate to the reader the general case, but similar curves can be drawn for all optimisation algorithms for dynamic environments, although the gradient of the curve may differ from algorithm to algorithm.

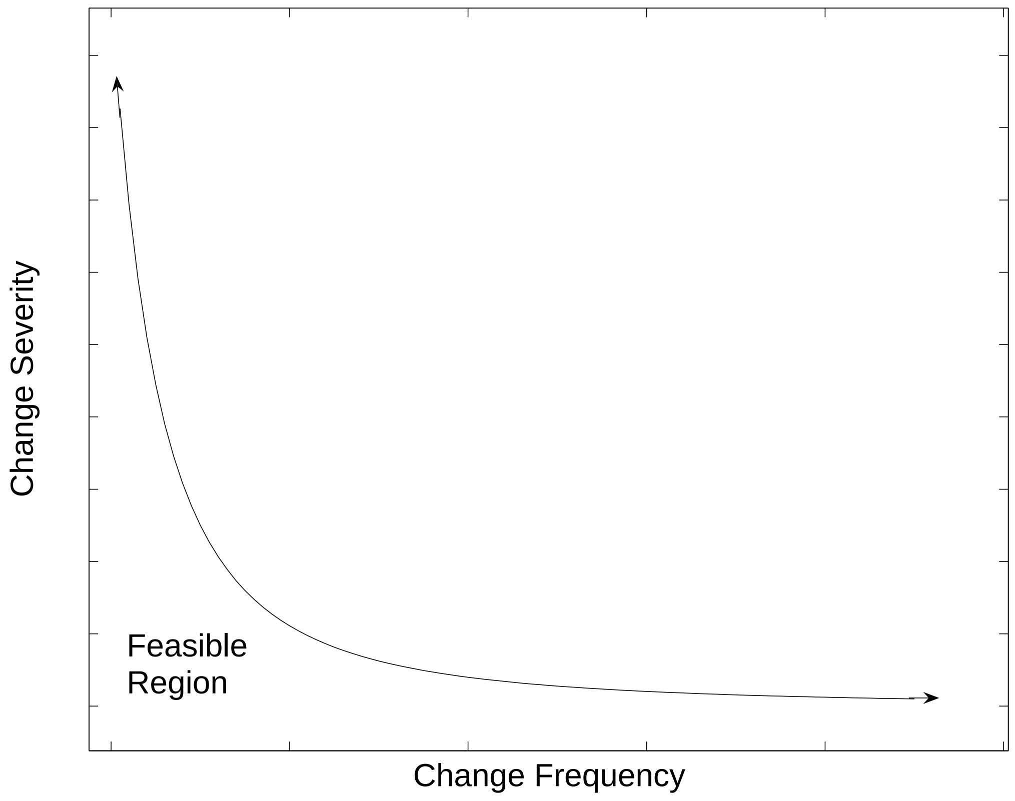

Figure 2.1: Feasible optimisation region in terms of change frequency and severity

This section described various types of dynamic environments. An investigation into the effectiveness of optimisation algorithms aimed at dynamic environments necessitates the evaluation of the optimisation algorithms on a wide range of environment types. The following two sections describe two benchmarks that are used in this study to simulate several types of dynamic environments discussed in this section.

2.5.3 Moving Peaks Benchmark

Branke [2007] created the moving peaks benchmark (MPB) to address the need for a single, adaptable benchmark that can be used to compare the performance of algorithms aimed at dynamic optimisation problems. The benchmark consists of a moving peaks maximisation function to test the effectiveness of an algorithm. The multi-dimensional problem space of the moving peaks function contains several peaks, or optima, of variable height, width, and shape. These peaks move around with height and width changing periodically.

For n_p peaks in n_d dimensions, the moving peaks function is given by [Branke, 2002]:

$$F(\vec{x}, t) = \max\left\{B(\vec{x}), \max_{p=1,\ldots,n_p} \{f_p(\vec{x}, h_p(t), w_p(t), \vec{l}_p(t))\}\right\} \tag{2.30}$$

where $B(\vec{x})$ is a constant basis landscape and f is a peak shape function. $h_p(t)$, $w_p(t)$ and $\vec{l}_p(t)$ are the height, width and position of the respective peaks, given by

$$h_p(t) = h_p(t-1) + \mathit{height\_severity} \cdot N(0,1) \tag{2.31}$$

$$w_p(t) = w_p(t-1) + \mathit{width\_severity} \cdot N(0,1) \tag{2.32}$$

$$\vec{l}_p(t) = \vec{l}_p(t-1) + \vec{c}_p(t) \tag{2.33}$$

where *height_severity* and *width_severity* are the deviation of changes to the width and height, respectively, and $\vec{c}_p(t)$ is given by

$$\vec{c}_p(t) = \frac{C_s((1-\lambda)\vec{r} + \lambda\vec{c}_p(t-1))}{|\vec{r} + \vec{c}_p(t-1)|} \tag{2.34}$$

where C_s is the change severity, $\vec{r}$ is a random vector created by selecting uniform random numbers for each dimension, $r_j \sim U(-0.5, 0.5)$, and normalising the vector length to C_s, and $\lambda \in (0,1)$ is the degree of correlation to previous shifts. For $\lambda = 1$ the direction of the positional shift is equivalent to the direction of the previous shift, while for $\lambda = 0$ the positional shift is not correlated to the previous shift.

The MPB allows the following parameters to be set:

- Number of peaks
- Number of dimensions
- Maximum and minimum peak width
- Maximum and minimum peak height
- Change period (the number of function evaluations between successive changes in the environment)

- Change severity (how much the locations of peaks are moved within the fitness landscape)
- Height severity (standard deviation of changes made to the height of each peak)
- Width severity (standard deviation of changes made to the width of each peak and consequently the gradient of each peak)
- Peak function
- Correlation (between successive movements of a peak)

A static basis function, $B(\vec{x})$, can optionally be added to the problem space if a more complex fitness landscape is required. The fitness landscape of the MPB is fundamentally a heterogeneous fitness landscape (refer to point 1 in Section 2.5.2) although a homogeneous fitness landscape can be simulated by setting the number of peaks to one.

Type I, II and III environments (see point 3 in Section 2.5.2) can be simulated by changing the parameters for change and height severity. Three scenarios of settings of MPB parameters were suggested by Branke [2007]. However, the majority of researchers using the MPB employ only variations of Scenario 2 settings, as these settings were used in an early paper by [Branke and Schmeck, 2003] and were subsequently used by other researchers to facilitate comparisons between algorithms. The Scenario 2 settings are listed in Table 2.1.

A graphical depiction of the moving peaks function in two dimensions with a conical peak function is given in Figure 2.2, and with a spherical peak function in Figure 2.3.

The value of the conical peak function with location $\vec{l}$ and height h is calculated in n_d dimensions as

$$f_{conical}(\vec{x}) = h - w \cdot \sqrt{\sum_{j=1}^{n_d}(x_j - l_j)^2} \tag{2.35}$$

while the value of the spherical peak function is calculated as

$$f_{spherical}(\vec{x}) = h - \sum_{j=1}^{n_d}(x_j - l_j)^2 \tag{2.36}$$

The most apparent difference between the conical and spherical functions is that, because spherical peaks do not have a constant gradient like the conical peaks, the absolute

Table 2.1: MPB Scenario 2 settings

Setting	**Value**
Number of Dimensions	5
Number of Peaks	10
Max and Min Peak height	[30,70]
Max and Min Peak width	[1.0,12.0]
Change period	5000
Change severity	1.0
Height severity	7.0
Width severity	1.0
Peak function	Cone
Correlation	[0.0,1.0]

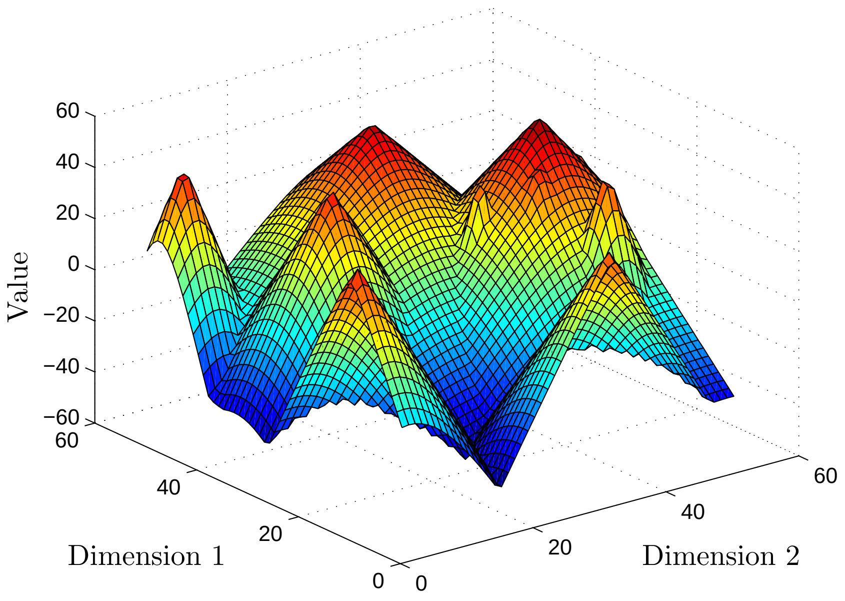

Figure 2.2: The moving peaks function using a conical peak function in two dimensions

minimum of the moving peaks function with spherical peaks is much lower than that of the conical peaks. It is thus possible to find much greater errors when using spherical peaks compared to using conical peaks. On the other hand, the steeper slopes on the spherical

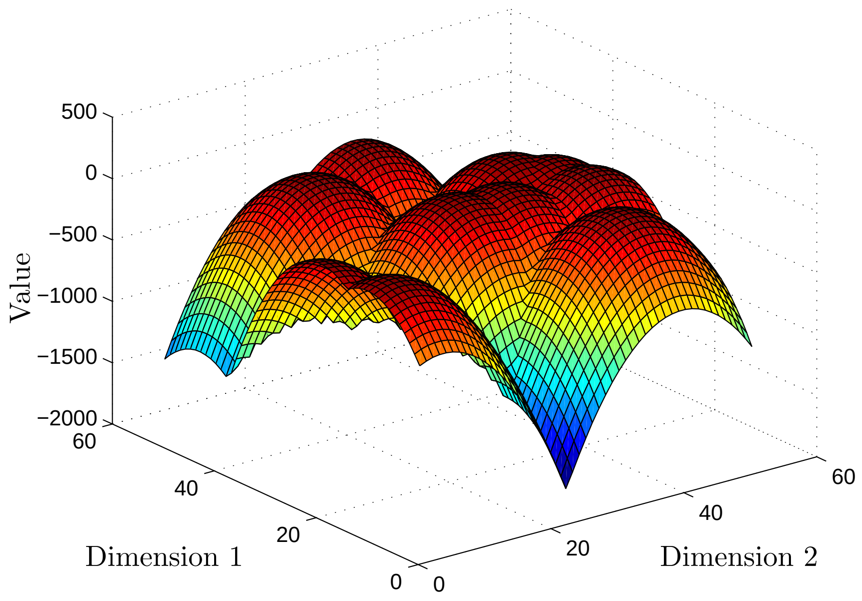

Figure 2.3: The moving peaks function using a spherical peak function in two dimensions

peaks should make initial optimisation easier, but only until a point close to the absolute minimum (where the gradient is small), where optimisation should become harder.

2.5.3.1 Extensions to the Moving Peaks Benchmark

Part of this study involves investigating dynamic environments in which the number of peaks fluctuates over time (see point 3 in Section 2.5.2). The MPB was therefore adapted in this thesis to allow the number of peaks to change when a change in the environment occurs. For the adapted MPB, the number of peaks, $n_p(t)$, is calculated as:

$$n_p(t) = \begin{cases} \max\{1,\ n_p(t-1) - M_{n_p} \cdot U(0,1) \cdot M_c\} \\ \qquad \text{if } U(0,1) < 0.5 \\ \min\{M_{n_p},\ n_p(t-1) + M_{n_p} \cdot U(0,1) \cdot M_c\} \\ \qquad \text{otherwise} \end{cases} \tag{2.37}$$

where M_{n_p} is the maximum number of peaks and M_c is the maximum fraction of M_{n_p} that can be added or removed from the population after a change in the environment. M_c thus controls the severity of the change in the number of peaks. For example, $M_c = 1$ will result in up to M_{n_p} peaks being added or removed, while $M_c = 0.1$ will result in a change of up to 10% of M_{n_p}. M_{n_p} and M_c are included as parameters to the benchmark function.

2.5.3.2 Moving Peaks Benchmark Critical Discussion

The MPB has been used by several researchers [Trojanowski, 2007] [Janson and Middendorf, 2003][Moser and Hendtlass, 2007][Ayvaz *et al.*, 2011][Kiraz *et al.*, 2011] and has served as a valuable method of comparison between different algorithms. The simplicity of the method used to produce changes in the dynamic environment makes the MPB ideal for studying the scalability of an optimisation algorithm in terms of change severity. Unfortunately, apart from three broad scenarios, the benchmark does not contain general guidelines specifying a set of parameters to investigate. As a result, researchers often report on different sets of parameters which complicates comparisons among algorithms.

Another disadvantage of the MPB is that it only provides a limited number of peak functions. Benchmarks for static environments typically contain large sets of complex functions, for example, the Griewank, Ackley, Rastrigin and Weierstrass functions [Qu and Suganthan, 2010]. The MPB does not provide a platform on which to identify optimisation algorithms suited to specific underlying functions that are typically used as benchmarks in static environments. Furthermore, the MPB does not provide functionality to investigate the effect of a broad range of change types (refer to point 2 discussed in Section 2.5.2), as only random and linearly correlated change types are supported.

The MPB does not provide functionality to investigate dynamic environments where optima are explicitly introduced or removed from the dynamic environment. This limitation has been remedied by the extension made to the MPB which is described in Section 2.5.3.1.

2.5.4 Generalised Dynamic Benchmark Generator

The generalised dynamic benchmark generator (GDBG) of Li *et al.* [2008] [Li and Yang, 2008] is a recent benchmark created to investigate optimisation in dynamic environments. This benchmark was used during the special session on evolutionary computation in dynamic and uncertain environments at the 2009 IEEE Congress on Evolutionary Computation to compare the performance of algorithms submitted by several researchers [Korošec and Šilc, 2009, Brest *et al.*, 2009, De França and Von Zuben, 2009, Li and Yang, 2009].

The GDBG dynamic function is defined as

$$F = f(\vec{x}, \vec{\phi}, t) \tag{2.38}$$

where $\vec{\phi}$ represents the system control parameters for height ($\vec{h}$), width ($\vec{w}$), location of the global optimum ($\vec{l}$, assuming a single global optimum), and underlying function rotation ($\vec{\theta}$). The notation ϕ_Ω with $\Omega \in \{\vec{h}, \vec{w}, \vec{l}, \vec{\theta}\}$ is used to refer to the control parameters in general. Minimum ($\phi_{\Omega,min}$), maximum ($\phi_{\Omega,max}$), and severity ($\phi_{\Omega,sev}$) values are defined by the creators of the benchmark for each control parameter.

Changes in the environment are enacted by periodically changing the control parameters. A function, *DynamicChanges*, provides the next values of each control parameter vector:

$$\vec{\phi}_\Omega(t+1) = DynamicChanges(\vec{\phi}_\Omega(t)) \;\; \text{with } \Omega \in \{\vec{h}, \vec{w}, \vec{l}, \vec{\theta}\} \tag{2.39}$$

DynamicChanges takes the form of various change types, presented in Section 2.5.4.2.

2.5.4.1 Generalised Benchmark Generator Functions

The benchmark consists of six main functions. The first, F_1, is a peak function with similarities to the MPB of Branke. The main differences between these two benchmarks are the peak shapes and the way that peaks are moved. The peak function, F_1, at time t is given by:

$$F_1(\vec{x}, t) = \max_{p=1,\ldots,n_p} \left\{ h_p \Big/ \left(1 + w_p \cdot \sqrt{\sum_{j=1}^{n_d} \frac{(x_j - l_{p,j})^2}{n_d}} \right) \right\} \tag{2.40}$$

where h_p, w_p and $\vec{l}_p$ contain the values for height, width and position of the n_p peaks in n_d dimensions. Equation (2.40) results in a peak which height decreases with the square of the distance to the optimum of the peak.

The method used to randomly move peaks around in the MPB results in peaks being reflected when the boundary of the fitness landscape is reached, analogous to a ball bouncing off a wall. This is seen as a disadvantage, since it implies an unequal challenge per change for an optimisation algorithm [Li *et al.*, 2008]. The disadvantage is remedied in the GDBG by using a rotation matrix to affect positional changes in the environment by independently rotating the dimensional components of the positions of the peaks as described in Algorithm 5 [Li and Yang, 2008].

Algorithm 5: Position rotation algorithm for the p-th underlying function

```
Randomly select $s = 2 \cdot \lfloor n_d/2 \rfloor$ dimensions ($s$ is the largest even integer smaller or equal to $n_d$) from the $n_d$ dimensions to compose a vector $\vec{o} = [o_1, o_2, ..., o_s]$;
Let $\vec{\theta}_p(t) = DynamicChanges(\vec{\theta}_p(t-1))$;
foreach pair of dimensions, $(o_j, o_{j+1})$ do
    construct a rotation matrix using $\vec{\theta}_p(t)$ as follows: $\mathbf{R}_{o_j,o_{j+1}}(\theta_{p,(j+1)/2})(t))$,
end
A transformation matrix $\mathbf{T}(t)$ is obtained by:
$\mathbf{T}_p(t) = \mathbf{R}_{o_1,o_2}(\theta_{p,1}(t)) \cdot \mathbf{R}_{o_3,o_4}(\theta_{p,2}(t)) \cdots \mathbf{R}_{o_{s-1},o_s}(\theta_{p,s/2}(t))$;
$\vec{l}_p(t+1) = \vec{l}_p(t) \cdot \mathbf{T}_p(t)$;
```

For positional changes, the values $\phi_{\theta,sev} = 1$, $\phi_{\theta,max} = \pi$, and $\phi_{\theta,min} = -\pi$ are used in *DynamicChanges*.

The vectors containing height and width information of the n_p peaks ($\vec{h} = [h_1, \dots, h_{n_p}]$ and $\vec{w} = [w_1, \dots, w_{n_p}]$) are changed using the *DynamicChanges* function:

$$\vec{h}(t+1) = DynamicChanges(\vec{h}(t)) \tag{2.41}$$

$$\vec{w}(t+1) = DynamicChanges(\vec{w}(t)) \tag{2.42}$$

Settings associated with F_1 are:

- The number of peaks: $n_p = 10, 50$
- Width range: $\phi_{w,max} = 10$ and $\phi_{w,min} = 1$
- Width severity: $\phi_{w,sev} = 0.5$
- Initial width: $initial\ width = 5$

A graphical depiction of a two-dimensional instance of F_1 is given in Figure 2.4. F_1 is a multi-modal maximisation problem with n_p optima (although some optima may be obscured by others).

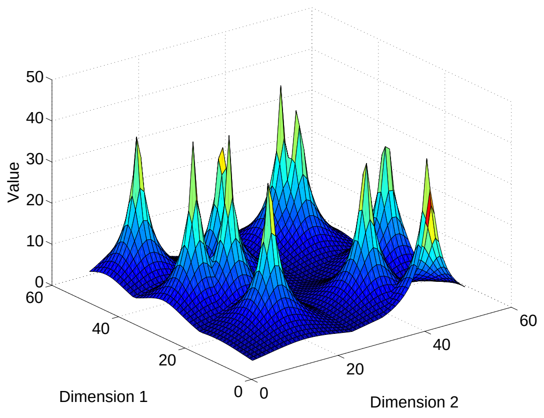

Figure 2.4: F_1 from the GDBG

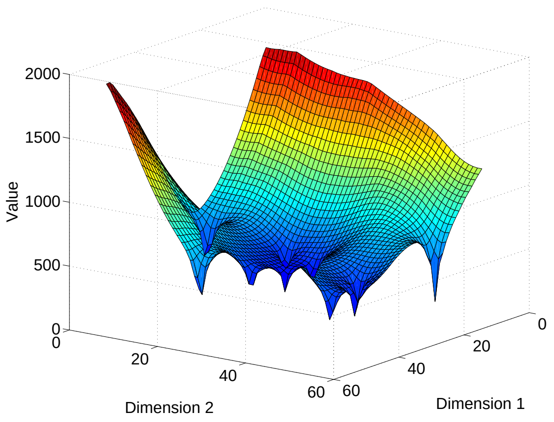

Figure 2.5: F_2 from the GDBG

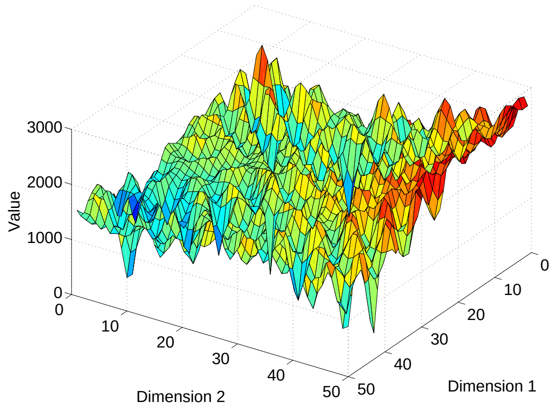

Figure 2.6: F_3 from the GDBG

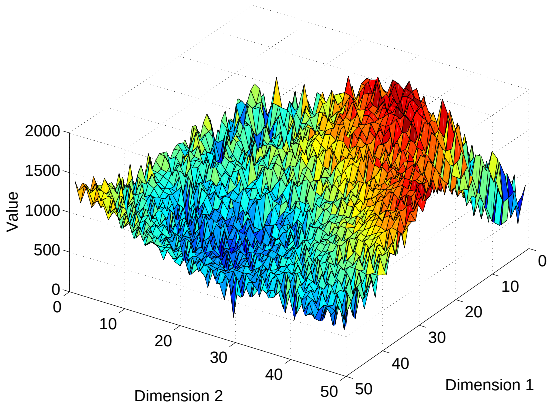

Figure 2.7: F_4 from the GDBG

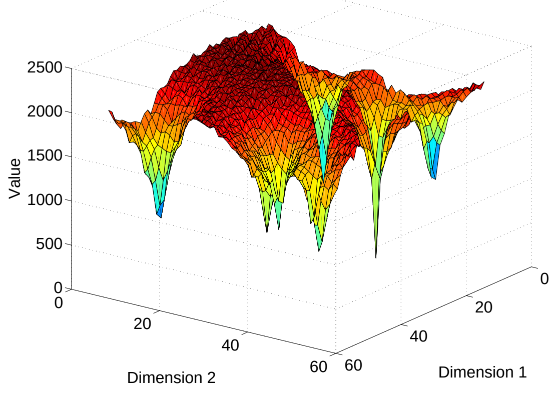

Figure 2.8: F_5 from the GDBG

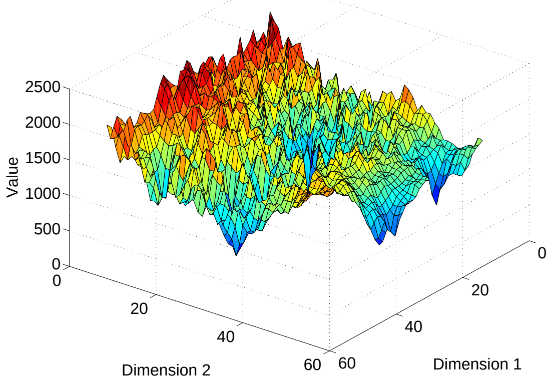

Figure 2.9: F_6 from the GDBG

Functions F_2 through F_6 are composition functions made up from basic functions standardly used as static benchmarks in the field of evolutionary computing. Each function is constructed from a composition of 10 underlying functions with different orientations, whose height and position are changed individually, resulting in a complex dynamic environment. The composite function values are calculated as:

$$F(\vec{x},t) = \sum_{p=1}^{10}\left(\psi_p \cdot \left(2000 \cdot \frac{f_p\Big(\big(\vec{x} - \vec{l}_p(t)\big)/\sigma_{f_p} \cdot \mathbf{T}_p\Big)}{f_{p,opti}} + h_p(t)\right)\right) \tag{2.43}$$

where $\vec{l}_p(t)$ is the location of the global optimum (with value $f_{p,opti}$) of underlying function f_p, and $\mathbf{T}_p$ is the rotation matrix for each underlying function. $\mathbf{T}_p$ is generated once for each underlying function and is not changed thereafter. The value of ψ_p is calculated for each f_p in three steps:

1.
$$\psi_p = \exp\left(-\sqrt{\frac{\sum_{j=1}^{n_d}(x_j - l_{p,j})^2}{2n_d}}\right) \tag{2.44}$$

2.
$$\psi_p = \begin{cases} \psi_p & \text{if } \psi_p = \max_{a=1}^{10}\{\psi_a\} \\ \psi_p \cdot \left(1 - (\max_{a=1}^{10}\{\psi_a\})^{10}\right) & \text{if } \psi_p \neq \max_{a=1}^{10}\{\psi_a\} \end{cases} \tag{2.45}$$

3.
$$\psi_p = \psi_p / \sum_{a=1}^{10}\psi_a \tag{2.46}$$

Note that equation (2.43) allows a single function to be composed of versions of itself with different orientations. The individual underlying functions that are used are listed in Table 2.2. Each function has a different range, so a stretch factor is employed to ensure a uniform range of the composition function $F(\vec{x})$. The stretch factor σ_{f_p} is calculated for each underlying function as

$$\sigma_{f_p} = \frac{V_{max,F} - V_{min,F}}{V_{max_{f_p}} - V_{min_{f_p}}} \tag{2.47}$$

where $[V_{max,F}, V_{min,F}]^{n_d}$ is the search range of $F(\vec{x})$ which is set to $[5,5]^{n_d}$ and the search range $[V_{max_{f_p}}, V_{min_{f_p}}]^{n_d}$ is that of the underlying function f_p.

Table 2.2: Details of the underlying benchmark functions of the GDBG

Name	**Function**	**Range**
Sphere	$f(\vec{x}) = \sum_{j=1}^{n_d} x_j^2$	[-100,100]
Rastrigin	$f(\vec{x}) = \sum_{j=1}^{n_d} (x_j^2 - 10\cos(2\pi x_j) + 10)$	[-5,5]
Weierstrass	$f(\vec{x}) = \sum_{j=1}^{n_d} \left(\sum_{a=0}^{20} \left(0.5^a \cos\left(2\pi \cdot 3^a (x_j + 0.5)\right)\right)\right) - n_d \sum_{a=0}^{20} \left(0.5^a \cos(\pi \cdot 3^a)\right)$	[-0.5,0.5]
Griewank	$f(\vec{x}) = \frac{1}{4000} \sum_{j=1}^{n_d} x_j^2 - \prod_{j=1}^{n_d} \cos(\frac{x_j}{\sqrt{j}}) + 1$	[-100,100]
Ackley	$f(\vec{x}) = -20 \exp\left(-0.2 \sqrt{\frac{1}{n_d} \sum_{j=1}^{n_d} x_j^2}\right) - \exp\left(\frac{1}{n_d} \sum_{j=1}^{n_d} \cos(2\pi x_j)\right) + 20 + e$	[-32,32]

Changes in the environment occur when $\vec{l}_p$ and $\vec{h}$ are changed using the $DynamicChanges$ function:

$$\vec{h}(t+1) = DynamicChanges(\vec{h}(t)) \tag{2.48}$$

$$\vec{l}_p(t+1) = DynamicChanges(\vec{l}_p(t)) \ \forall \ p \in \{1, 2, \ldots, n_p\} \tag{2.49}$$

F_2 is made up of a composition of the Sphere function, illustrated in two dimensions in Figure 2.5. F_2 has 10 optima. The third function, F_3, contains a composition of Rastrigin's function, illustrated in two dimensions in Figure 2.6. F_4 is a composition of the Griewank function, illustrated in two dimensions in Figure 2.7. The fifth function, F_5, is conctructed from a composition of Ackley's function, illustrated in two dimensions in Figure 2.8. The last function is a composition of the Sphere, Ackley, Griewank, Rastrigin and Weierstrass functions. Figure 2.9 depicts F_6 in two dimensions. Functions F_3 to F_4 all have a large number of optima.

F_1 is maximised, while F_2 to F_6 are minimised. For all functions (F_1 to F_6) the following settings are used:

- Height range: $\phi_{h,max} = 10$ and $\phi_{h,min} = 1$
- Height severity: $\phi_{h,sev} = 5.0$
- Initial height: $initial\ width = 50$

2.5.4.2 Generalised Benchmark Generator Change Types

Six different change types are used in the benchmark to investigate algorithm performance under different types of changes. The experimenter selects one of the following change types which governs the functioning of the function *DynamicChanges*:

1. Small step changes, T_1:

$$\phi_{\Omega,q}(t+1) = \phi_{\Omega,q}(t) + 0.02 \cdot (\phi_{\Omega,max} - \phi_{\Omega,min}) \cdot U(-1,1) \cdot \phi_{\Omega,sev} \tag{2.50}$$

2. Large step changes, T_2:

$$\phi_{\Omega,q}(t+1) = \phi_{\Omega,q}(t) + (\phi_{\Omega,max} - \phi_{\Omega,min}) \cdot (0.02 \cdot sign(U(-1,1)) + 0.08 \cdot U(-1,1)) \cdot \phi_{\Omega,sev} \tag{2.51}$$

3. Random changes, T_3:

$$\phi_{\Omega,q}(t+1) = \phi_{\Omega,q}(t) + N(0,1) \cdot \phi_{\Omega,sev} \tag{2.52}$$

4. Chaotic changes, T_4:

$$\phi_{\Omega,q}(t+1) = 3.67 \cdot (\phi_{\Omega,q}(t) - \phi_{\Omega,min}) \cdot \left(1 - \frac{(\phi_{\Omega,q}(t) - \phi_{\Omega,min})}{(\phi_{\Omega,max} - \phi_{\Omega,min})}\right) \tag{2.53}$$

5. Recurrent changes, T_5:

$$\phi_{\Omega,q}(t+1) = \phi_{\Omega,min} + (\phi_{\Omega,max} - \phi_{\Omega,min}) \cdot \left(\sin\left(2\pi(\frac{t}{12} + \frac{q}{10})\right) + 1\right)/2 \tag{2.54}$$

6. Recurrent changes with noise, T_6:

$$\phi_{\Omega,q}(t+1) = \phi_{\Omega,min} + (\phi_{\Omega,max} - \phi_{\Omega,min}) \cdot \left(\sin\left(2\pi(\frac{t}{12} + \frac{q}{10})\right) + 1\right)/2 + N(0,0.8) \tag{2.55}$$

where $\phi_{\Omega,sev}$, $\phi_{\Omega,max}$ and $\phi_{\Omega,min}$ correspond to the upper bound, lower bound and severity values of the specific control parameter to be changed. The creators of the GDBG suggested that changes occur once every 100 000 function evaluations.

The GDBG also provides a change type that allows the number of dimensions to vary over time. This type of dynamic environment is outside the scope of this thesis.

2.5.4.3 Generalised Benchmark Generator Critical Discussion

Each of the six functions of the GDBG provide a heterogeneous fitness landscape (refer to point 1 discussed in Section 2.5.2) which consists of underlying functions typically found in benchmarks for static environments. The simulated dynamic environments can all be classified as type III in terms of change pervasiveness (refer to point 3 discussed in Section 2.5.2).

The GDBG provides six change types which allows researchers to investigate the performance of the optimisation algorithms under different types of changes. All the change types identified under point 2 in Section 2.5.2 are represented in the GDBG. A disadvantage of the rigidly defined and complex change types of the GDBG is that investigations into the scalability of algorithms in terms of change severity are not explicitly supported. The settings of the GDBG can be altered in a limited extent to vary the change severity, by using different change types, but much of what is intuitively understood as change severity is lost. The GDBG does not have a parameter, like it has with the change period, that can be varied over a range of values to observe the scalability of an algorithm with respect to change severity.

Recent years have seen the development of a considerable number of algorithms to solve dynamic optimisation problems (refer to Section 3.3). It is important to be able to compare these algorithms in order to determine which algorithms perform best on various problems. The next section reviews metrics that have been suggested to measure performance.

2.5.5 Performance Measures for Dynamic Environments

The purpose of performance measures is firstly to measure how effective an algorithms is at solving an optimisation problem, and secondly to provide a baseline for comparisons among optimisation algorithms. The goal of this section is to discuss proposed performance measures for dynamic optimisation problems. The most appropriate performance measure for this study can be identified from this discussion. According to Morrison [2003], a good performance measure should:

1. have intuitive meaning,

2. allow plain statistical significance testing, and

3. have a sufficiently large exposure to landscape dynamics.

The third requirement implies that performance measures which are normally used in static environments are not appropriate for dynamic environments, since these are, generally, only focused on the performance of an algorithm at the conclusion of the optimisation process. In dynamic environments, the effectiveness of an algorithm has to be considered over the entire span of the optimisation process and over all environment changes.

An important consideration when choosing a performance measure is the information that is available to the observer. Benchmark problems typically provide information regarding the function value and location of the global optimum, and when changes in the environment occur. In the case of the moving peaks benchmark (refer to Section 2.5.3) the function values and locations of local optima are also available. This information can be used to measure the performance of an algorithm more accurately. However, the above mentioned information may not be available when solving real world dynamic optimisation problems [Nguyen *et al.*, 2012].

A survey of the literature identified 13 performance measures:

1. Graphs of the average best-of-generation at each generation [Grefenstette, 1999] [Bäck, 1998], calculated at the g-th generation as:

$$H_{AG,g} = F(\vec{x}_{best}(g), t_{\vec{x}_{best}(g)}) \tag{2.56}$$

   where $\vec{x}_{best}(g)$ is the best performing individual in generation g and $F(\vec{x}, t)$ is a dynamic optimisation function. The time step in which $\vec{x}_{best}(g)$ is evaluated is $t_{\vec{x}_{best}(g)}$. The value of each $H_{AG,g}$ is averaged over many repeats of the same experiment, and are subsequently plotted on a graph of $H_{AG,g}$ versus generations.

   The graphs provide a visual representation of how fitness improves after each change in the environment. However, a visual comparison of the graphs of more than one algorithm does not provide clear information as to which algorithm performed the best [Morrison, 2003]. It is also impossible to do statistical significance testing between the results of multiple algorithms when using $H_{AG,g}$ graphs, as they support visual comparisons rather than numerical comparisons.

A further disadvantage of the $H_{AG,g}$ measure is that it is sampled once every generation, hence making the time between samples dependent on the population size. An algorithm with a large population size has a natural advantage over an algorithm with a small population size, since the larger population size ensures a broader exposure to the fitness landscape. A fair comparison of two algorithms using $H_{AG,g}$ thus requires equal population sizes, as $H_{AG,g}$ does not consider the computational cost per generation.

2. The average error of the best individuals just before a change in the environment [Trojanowski and Michalewicz, 1999b]:

$$H_{AEBI} = \frac{\sum_{c=1}^{n_c} E(\vec{x}_{best}(t_c), c)}{n_c} \tag{2.57}$$

where n_c is the total number of changes in the environment, t_c is the time step before the c-th change in the environent, $\vec{x}_{best}(t)$ the best individual found since the last change in the environment and $E(\vec{x}, c)$ is the error of individual $\vec{x}$ immediately before the c-th change in the environment.

This performance measure provides an indication of how well the fitness landscape is optimised before changes in the environment and allows for statistical significance testing. A further advantage of this performance measure is that it allows for uniform treatment of function maximisation and minimisation problems, since the error and not the function value is used. It is thus always desirable to achieve a low value for H_{AEBI}.

A disadvantage of this performance measure is that it only considers the performance immediately before changes in the environment. The performance during periods between changes are ignored. An algorithm that recovers quickly after changes in the environment will not be rated better than an algorithm with a slow recovery time.

3. The lowest error value found of the best individual just before a change, over all changes [Li *et al.*, 2008]:

$$H_B = \min_{c=1,\ldots,n_c} \{E(\vec{x}_{best}(t_c), c)\} \tag{2.58}$$

This measure is meant to be used in conjunction with performance measure number 2, and gives the minimum error value found. Very little can be learnt by using this measure, since an algorithm that achieves a single very low error value at some point in the optimisation process, but high error values during the rest of the optimisation process, would be judged to be more effective than an algorithm that consistently produces moderately low error values.

4. The worst error value found of the best individual just before a change, over all changes [Li *et al.*, 2008]:

$$H_W = \max_{c=1,\ldots,n_c} \{E(\vec{x}_{best}(t_c), c)\} \tag{2.59}$$

This measure is meant to be used in conjunction with performance measure numbers 2 and 3, and gives the worst just-before-change error value. H_W does not represent an accurate reflection of an algorithm's performance, since the error value at just a single point in the optimisation process is reported.

5. The online performance is the running average of all fitness evaluations [Branke, 2002]:

$$H_{ON} = \frac{\sum_{t=1}^{n_t} F(\vec{x}(t), t)}{n_t} \tag{2.60}$$

where $\vec{x}(t)$ is the individual evaluated at function evaluation t, $F(\vec{x}, t)$ is the dynamic function and n_t is the total number of function evaluations.

A disadvantage of this measure is that fitness values of all individuals are taken into account (not only the best performing individuals). An algorithm is penalised for exploring sub-optimal regions of the fitness landscape even if it locates the global optimum. This performance measure is thus biased towards algorithms with low population diversity and fast convergence.

On the other hand, an observer using this performance measure does not require information regarding when changes in the environment occur and the location or function value of the global optimum.

6. The offline performance is the running average of the best-so-far fitness found since the last change in the environment [Branke, 2002]:

$$H_{OP} = \frac{\sum_{t=1}^{n_t} F(\vec{x}_{best}(t), t)}{n_t} \tag{2.61}$$

   where n_t is the total number of function evaluations and $F(\vec{x}, t)$ is a dynamic function.

   This measure addresses the disadvantages of measure number 5 by averaging over the function value of the best individual found since the last change in the environment at each time step. A disadvantage of using H_{OP} is that function maximisation and minimisation problems cannot be treated uniformly, since a large value for H_{OP} is desired for maximisation problems and a low value is desired for minimisation problems. Furthermore, averaging over the fitness makes intuitive interpretation of the results difficult, since the optimum value of the dynamic fitness function may vary over time, leaving the observer without any knowledge of how good a particular fitness value is.

7. The offline error is the running average of the lowest-so-far error found since the last change in the environment [Branke, 2002]:

$$H_{OE} = \frac{\sum_{t=1}^{n_t} E(\vec{x}_{best}(t), t)}{n_t} \tag{2.62}$$

   where n_t is the total number of function evaluations and $E(\vec{x}_{best}(t), t)$ is the error of the best individual found since the last change in the environment.

   This performance measure is similar to measure number 6, but averages over the error of the best found individual since the last change in the environment. It addresses a disadvantage of performance measure number 2 in that the error during the entire optimisation process is taken into account (not only the error just before changes). The measure thus also rewards algorithms for recovering quickly after changes in the environment, not only for achieving a low error value.

   An advantage of averaging over the error and not the fitness provides a more intuitive interpretation of the results, since the observer knows that the ideal error value is zero. In addition, by making use of the error rather than the function value allows for uniform treatment of maximisation and minimisation problems. H_{OE} can only be used when information regarding the value of the global optimum and when changes in the environment occurs, is available.

The offline error performance measure was suggested as part of the moving peaks benchmark (refer to Section 2.5.3), and has become one of the most commonly used performance measures for dynamic optimisation problems.

8. The collective mean fitness is the average of best-of-generation over all generations [Morrison, 2003]:

$$H_{CM} = \frac{\sum_{g=1}^{n_g} F(\vec{x}_{best}(g), t_{\vec{x}_{best}(g)})}{n_g} \tag{2.63}$$

where $\vec{x}_{best}(g)$ is the best performing individual in generation g, n_g is the total number of generations and $F(\vec{x}, t)$ is a dynamic optimisation function. The time-step in which $\vec{x}_{best}(g)$ is evaluated is $t_{\vec{x}_{best}(g)}$.

This performance measure requires no information regarding when changes in the environment occur, or the value of the global optimum. However, as was the case with performance measure number 6, averaging over the function value complicates interpretation of the results.

Another disadvantage is that the population size may influence the value of H_{CM} because the average is taken over the number of generations. For example, an algorithm with a small population size may have a larger value for n_g over a constant number of fitness evaluations than an algorithm with a large population size. The average is taken over the function value of the best individual in each generation, so the algorithm with the smaller population size is more likely to have generations included in the average where the best individual performed relatively poorly.

9. The average minimum Euclidean distance to optimum at each generation [Weicker

and Weicker, 1999]:

$$H_{AD} = \frac{\sum_{g=1}^{n_g} \left(\min_{i=1,\ldots,n_I} \{ \|\vec{l}_F(t) - \vec{x}_i(t)\|_2 \} \right)}{n_g} \tag{2.64}$$

where n_g is the total number of generations and $\vec{l}_F(t)$ is the location of the global optimum of the dynamic function (assuming a single global optimum). This measure requires information regarding the location of the global optimum.

The major disadvantage of this performance measure is that only the global optimum is taken into account. Algorithms are not rewarded of locating local optima. Ideally, the global optimum should be located, but local optima may have function values very close to that of the global optimum, and could represent an adequate solution to the optimisation problem. A further disadvantage of using H_{AD} is that it ignores both the function value and the error of individuals. The gradient around the global optimum can be steep for a particular optimisation problem which would result in a high error value for an individual that is relatively close, but this high error value would not be reflected in H_{AD}.

10. The ratio of the difference between the best-of-generation and the worst fitness found within a window of recent generations, over the difference of the best fitness found within the window and the worst fitness found within the window [Weicker, 2002]:

$$H_{RW} = \frac{\sum_{g=1}^{n_g} \left(\frac{F(\vec{x}_{best}(g), t_{\vec{x}_{best}(g)}) - W_{worst}(g)}{W_{best}(g) - W_{worst}(g)} \right)}{n_g} \tag{2.65}$$

$$W_{worst}(g) = \min_{g'=g-\omega,\ldots,g} \left\{ \min_{i=1,\ldots,n_I} \{F(\vec{x}_i(g'), t_{\vec{x}_i(g')})\} \right\} \tag{2.66}$$

$$W_{best}(g) = \max_{g'=g-\omega,\ldots,g} \left\{ \max_{i=1,\ldots,n_I} \{F(\vec{x}_i(g'), t_{\vec{x}_i(g')})\} \right\} \tag{2.67}$$

where $W_{best}(g)$ is the function value of the best performing individual within a window of ω generations and $W_{worst}(g)$ is the function value of the worst performing individual within a window of ω generations. The i-th individual in generation g is represented by $\vec{x}_i(g)$, and the individual is evaluated at time step $t_{\vec{x}_i(g)}$. The best individual within generation g is denoted by $\vec{x}_{best}(g)$. The above equations assume a function maximisation problem. This performance measure does not require information regarding the value of the global optimum or when changes occur. A value close to 1.0 represents good performance while a value close to zero represents poor performance.

The mayor disadvantage of using this performance measure is that performance is measured relative to the difference between the best and worst performing individuals within the window. Consequently, a poor performing population that converged to the extend where all individuals have a very similar fitness value, will still receive a relatively high value for H_{RW}. Misleading results can thus ensue from using this performance measure.

A further disadvantage of using H_{RW} is that results will be affected by the window size ω and the population size, since both influence the number of function evaluations that are included within the window.

11. The sampled relative error, H_{RE} [Li *et al.*, 2008]. Let the relative error $RE(t)$ be defined for maximisation problems as:

$$RE(t) = \frac{F(\vec{x}_{best}(t), t)}{F(\vec{l}_F(t), t)} \tag{2.68}$$

and for minimisation problems as:

$$RE(t) = \frac{F(\vec{l}_F(t), t)}{F(\vec{x}_{best}(t), t)} \tag{2.69}$$

A relative error close to 1.0 indicates good performance while values smaller than 1.0 indicate poorer performance. The relative error is sampled n_s times between successive changes in the environment, i.e. $RE(t_{1,c}), RE(t_{2,c}), \ldots, RE(t_{n_s,c})$ before the c-th change in the environment. The sampled relative error performance measurement, H_{RE}, is calculated as:

$$H_{RE} = \frac{\sum_{c=1}^{n_c} \left(\frac{RE(t_{n_s,c})}{1 + \sum_{a=1}^{n_s} (1 - RE(t_{a,c}))/n_s} \right)}{n_c} \tag{2.70}$$

where n_c is the total number of changes in the environment. This performance measure requires information regarding when changes in the environment occur and the value of the global optimum.

There are several disadvantages to H_{RE} as a performance measure. Firstly, H_{RE} is inappropriate for problems where the function value of the global optimum is equal to zero since this will always result in a RE value of zero for a minimisation problem and a divide by zero for a maximisation problem. Secondly, the value of the global optimum must be positive for function minimisation problems, otherwise RE values of greater than 1.0 is found. Thirdly, errors are not treated consistently for minimisation and maximisation problems, because RE increases linearly for maximisation problems as $F(\vec{x}_{best}(t), t)$ tends to $F(\vec{l}_F(t), t)$, while RE increases hyperbolically for minimisation problems as $F(\vec{x}_{best}(t), t)$ tends to $F(\vec{l}_F(t), t)$.

In addition to the above mentioned disadvantages, the fact that errors are only sampled periodically means that the performance of the algorithm is not taken into account during the intermediate periods of the optimisation process.

12. Peak cover, a performance measure specifically created for the moving peaks benchmark (refer to Section 2.5.3), measures the average ratio of the number of peaks on which there are individuals over the number of peaks whose optima are not obscured by other peaks [Branke, 2002]:

$$covered\_peaks = \sum_{p=1}^{n_p} \begin{cases} 1 \text{ if } \left(\exists i \in \{1, \ldots, n_I\} \text{ such that } f_p(\vec{x}_i, t) = F(\vec{x}_i, t)\right) \\ \qquad \text{and } \left(f_p(\vec{l}_p, t) = F(\vec{l}_p, t)\right) \\ 0 \text{ otherwise} \end{cases} \tag{2.71}$$

$$not\_hidden\_peaks = \sum_{p=1}^{n_p} \begin{cases} 1 \text{ if } f_p(\vec{l}_p, t) = F(\vec{l}_p, t) \\ 0 \text{ otherwise} \end{cases} \tag{2.72}$$

$$H_{PC} = \frac{\sum_{g=1}^{n_g} \frac{covered\_peaks}{not\_hidden\_peaks}}{n_g} \tag{2.73}$$

$$\tag{2.74}$$

where $f_p(\vec{x}, t)$ is the function value of the p-th peak, $\vec{l}_p$ is the location of the optimum of the p-th peak, $F(\vec{x}, t)$ is the dynamic function that contains the p peaks and n_p is the number of peaks. A common strategy for optimising dynamic problems is to track all optima in the environment in order to recover quickly from changes in the environment (refer to Section 3.3). H_{PC} measures how well all optima are tracked.

Although several algorithms attempt to track all optima, it can be argued that the possibility should not be excluded that an effective algorithm could be developed that does not make use of this strategy. Accordingly, H_{PC} is, in general, not an appropriate performance measure.

A further disadvantage of the peak cover measure is that it can only be used when information regarding the locations of all peaks are available, and if $F(\vec{x}, t)$ is calculated using the maximum value of all peaks, as described in point 1(b)ii on page 33. This is not the case with all benchmark functions and very unlikely in real world problems.

13. Best known peak error, a performance measure specifically created for the moving peaks benchmark, was designed by Bird and Li [2007] to be used in conjunction with performance measure number 12. The goal of this measure is to measure the convergence speed of an algorithm once it has been found to cover a peak. This is achieved by calculating the minimum error found for each peak at the end of each generation:

$$\xi_{p,g} = \min_{i=1,\ldots,n_I} \{E(\vec{x}_i(g), t_{\vec{x}_i(g)}) \mid f_p(\vec{x}_i, t) = F(\vec{x}_i, t)\} \tag{2.75}$$

where $\xi_{p,g}$ is the minimum error found on the p-th peak during generation g, $\vec{x}_i(g)$ is the location of the i-th individual during the g-th generation and $t_{\vec{x}_i(g)}$ is the time step in which $\vec{x}_i(g)$ is evaluated.

When a change in the environment occurs, the index, a_c , of the peak with the lowest error during the last generation before the change is found:

$$a_c = p \text{ where } \xi_{p,c\times n_{g,c}} = \min_{p'=1,\ldots,n_p} \{\xi_{p',c\times n_{g,c}}\} \tag{2.76}$$

where $n_{g,c}$ is the number of generations that takes place between two changes in the environment. The best known peak error is then calculated as:

$$H_{BKPE} = \frac{\sum_{c=1}^{n_c} PE_{a_c,c}}{n_g} \tag{2.77}$$

where n_g is the total number of generations, n_c is the total number of changes and the function $PE_{p,c}$ is defined as:

$$PE_{p,c} = \sum_{g=(c-1)\times n_{g,c}+1}^{c\times n_{g,c}} \xi_{p,g} \tag{2.78}$$

This performance measure requires information regarding when changes in the environment occur and the function values of all the local optima.

The major disadvantage of using H_{BKPE} is that only the error of the best known peak is taken into consideration. Since the global optimum is not taken into account, an algorithm with fast convergence to inferior optima can receive a high ranking from H_{BKPE}.

A further disadvantage is that the measure is sampled every generation. The assumption is that the number of function evaluations required to evaluate a generation is much smaller than the number of function evaluations between changes in the environment. This is not the case when a large population size is used, or when changes in the environment occur frequently.

The above discussion of the 13 performance measures pointed out several disadvantages of sampling the error after each generation. A further disadvantage is that the number of fitness evaluations per generation is not constant in several modern algorithms aimed at dynamic optimisation problems (refer to Section 3.3). The number of generations that an algorithm performs will consequently not always be the same when experiments are repeated (when allowing a constant number of fitness evaluations to ensure equal exposure to the fitness landscape).

Considering the positive and negative aspects of the performance measures discussed in this section it was concluded that the most appropriate performance measure to use in this thesis is number 7, the offline error. The offline error measure has several benefits: It conforms to all three requirements of Morrison [2003]: it has intuitive meaning, allows statistical significance testing, and has sufficient exposure to landscape dynamics since the error at all function evaluations contribute to the final offline error. A further advantage of the offline error is that it does not suffer from any of the disadvantages associated with generation based performance measures. In addition, error values are taken into account as opposed to function values, which means that maximisation and minimisation problems can be treated uniformly. A potential disadvantage of the offline error is that it requires information regarding when changes in the environment occur and the function value of the global optimum. However, this is not seen as a problem, as the required information is available from the benchmark functions used in this thesis.

The offline error averages over the lowest errors found since the last change in the environment (referred to by Branke [2002] as the *current error*). The average current error over several repeats of the optimisation process can be graphed analogously to performance measure number 1 (average best-of-generation graphs) to provide a visual representation of the optimisation algorithm's performance during the optimisation process. Figure 2.10 illustrates the functioning of the offline error measurement and the current error obtained by an elementary random guessing algorithm (given in Algorithm 6) on the Scenario 2 settings of the MPB. Ten changes in the environment are depicted. The offline error is the average of all previous current errors and thus produces a smoother curve than the current error. The current error spikes after each change in the environment and drops as better solutions are found.

Algorithm 6: Elementary random guessing algorithm

```
$t = 0$;
while maximum number of function evaluations is not exceeded do
    Uniformly select random vector $\vec{x}(t)$ from the $n_d$ dimensional search space;
    Evaluate $F(\vec{x}(t), t)$;
    $t = t + 1$;
end
```

Random guessing is not an effective optimisation strategy, and it will be shown in the next chapter that much better results are found even with algorithms not tailored to dynamic environments.

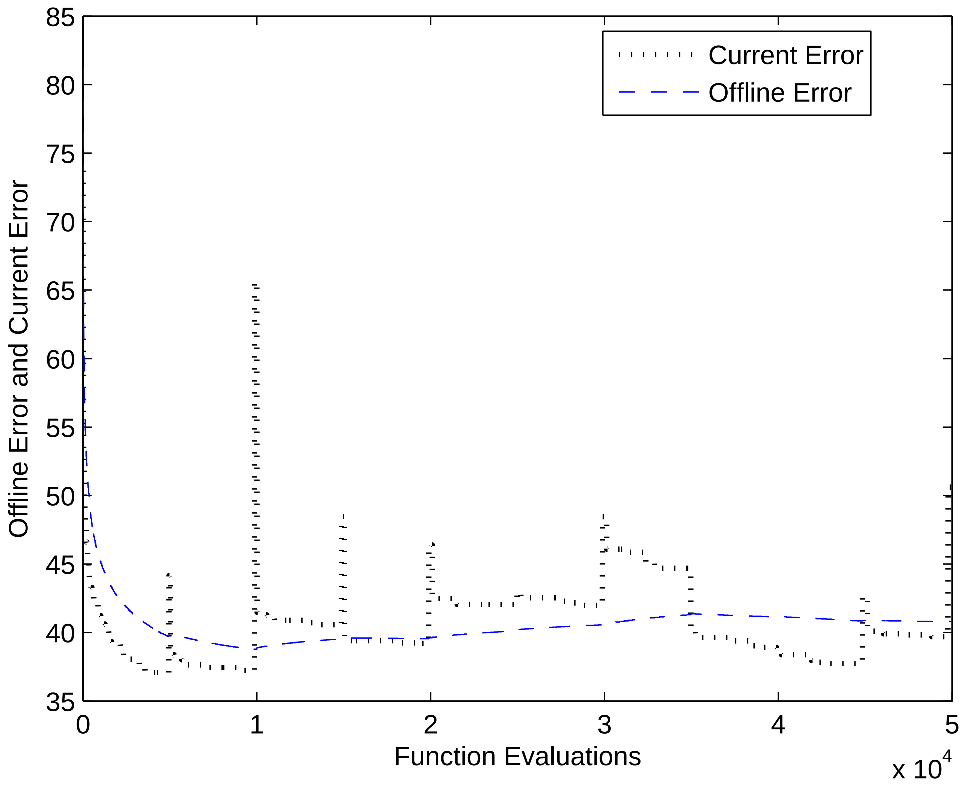

Figure 2.10: Current and offline errors found on the MPB through evaluating random individuals

2.6 Conclusions

This chapter provided background information regarding optimisation with emphasis on DE. DE is a evolutionary optimisation algorithm that employs the spatial difference between individuals to perform mutations. Common DE schemes were reviewed along with the standard DE control parameters. The control parameters can be altered to change the exploration and exploitation characteristics of the DE algorithm. However, fine-tuning the control parameters for a specific problem is a time consuming manual task. Work aimed at reducing control parameters was reviewed with special emphasis on self-adapting parameters.

Dynamic optimisation environments were formally defined. It was found that dynamic environments are typically classified in the literature based on the types of changes that occur, the fitness landscape composition and the pervasiveness of changes. It was argued that the three most important factors influencing the intricacy of an dynamic optimisation problem is hardness of the fitness landscape, the frequency at which changes occur, and the severity of changes to the environment. An thorough investigation into the effectiveness of an optimisation algorithm should include a scalability study on how the algorithm scales over various values of the three factors that influence the intricacy of the dynamic environment.

Two benchmarks were discussed that can be used to simulate dynamic environments. Both are used in this thesis to investigate the performance of an algorithm with respect to various settings of the number of dimensions and the change period. The first benchmark that was discussed is the moving peaks benchmark. This benchmark has been used by several other researchers, and is simple enough to provide an intuitive understanding of the fitness landscape. The MPB is ideally suited to investigate the scalability of algorithms in terms change severity. The MPB allows researchers to set the number of peaks in the environment, and was extended by the author of this thesis to support the simulation of dynamic environments in which the number of peaks fluctuates over time. The extended MPB is thus ideal for studying how the number of optima in the environment influences the performance of an optimisation algorithm.

The second benchmark is the generalised dynamic benchmark generator. This benchmark simulates dynamic environments composed of benchmark functions typically used in static optimisation. The GDBG provides six different change types. The GDBG can be used to evaluate the performance of an optimisation algorithm on six different functions and change types.

The chapter concluded with description of common performance measures for dynamic environments which were identified in the literature. The advantages and disadvantages of 13 performance measures were discussed. The offline error performance measure, which averages the lowest error value found since the last change in the environment over all function evaluations, was concluded to be the most appropriate performance measure for the current study.

The next chapter introduces related research on optimising dynamic environments with specific focus on DE-based algorithms.