Models for Evolutionary Algorithms and Their Applications in System Identification and Control Optimization

Rasmus K. Ursem

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

EVALife

Complex systems
and the mutual influence between
Information Technology and Life Sciences

Models for Evolutionary Algorithms and Their Applications in System Identification and Control Optimization

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Rasmus K. Ursem

1st April 2003

Supervisors: Thiemo Krink and Brian H. Mayoh

Colophon
First print: 1st April 2003
Typesetting: LaTeX
Font: Times
References: Bibtex
Figures: Xfig and Gnuplot
Graphs: Gnuplot

Abstract

In recent years, optimization algorithms have received increasing attention by the research community as well as the industry. In the area of evolutionary computation (EC), inspiration for optimization algorithms originates in Darwin's ideas of evolution and survival of the fittest. Such algorithms simulate an evolutionary process where the goal is to evolve solutions by means of crossover, mutation, and selection based on their quality (fitness) with respect to the optimization problem at hand. Evolutionary algorithms (EAs) are highly relevant for industrial applications, because they are capable of handling problems with non-linear constraints, multiple objectives, and dynamic components – properties that frequently appear in real-world problems.

This thesis presents research in three fundamental areas of EC; fitness function design, methods for parameter control, and techniques for multimodal optimization. In addition to general investigations in these areas, I introduce a number of algorithms and demonstrate their potential on real-world problems in system identification and control. Furthermore, I investigate dynamic optimization problems in the context of the three fundamental areas as well as control, which is a field where real-world dynamic problems appear.

Regarding fitness function design, smoothness of the fitness landscape is of primary concern, because a too rugged landscape may disrupt the search and lead to premature convergence at local optima. Rugged fitness landscapes typically arise from imprecisions in the fitness calculation or low relatedness between neighboring solutions in the search space. The imprecision problem was investigated on the Runge-Kutta-Fehlberg numerical integrator in the context of non-linear differential equations. Regarding the relatedness problem for the search space of arithmetic functions, Thiemo Krink and I suggested the smooth operator genetic programming algorithm. This approach improves the smoothness of fitness function by allowing a gradual change between traditional operators such as multiplication and division.

In the area of parameter control, I investigated the so-called self-adaptation technique on dynamic problems. In self-adaptation, the genome of the individual contains the parameters that are used to modify the individual. Self-adaptation was developed for static problems; however, the parameter control approach requires a significant number of generations before superior parameters are evolved. In my study, I experimented with two artificial dynamic problems and showed that the technique fails on even rather simple time-varying problems. In a different study on static problems, Thiemo Krink and I suggested the terrain-based patchwork model, which is a fundamentally new approach to parameter control based on agents moving in a spatial grid world.

For multimodal optimization problems, algorithms are typically designed with two objectives in mind. First, the algorithm shall find the global optimum and avoid stagnation at local optima. Additionally, the algorithm shall preferably find several candidate solutions, and thereby allow a final human decision among the found solutions. For this objective, I created the multinational EA that employs

a self-organizing population structure grouping the individuals into a number of subpopulations located in different parts of the search space. In a related study, I investigated the robustness of the widely used sharing technique. Surprisingly, I found that this algorithm is extremely sensitive to the range of fitness values. In a third investigation, I introduced the diversity-guided EA, which uses a population diversity measure to guide the search. The potential of this algorithm was demonstrated on parameter identification of two induction motor models, which are used in the pumps produced by the Danish pump manufacturer Grundfos.

The field of dynamic optimization has received significant attention since 1990. However, most research performed in an EC-context has focused on artificial dynamic problems. In a fundamental study, Thiemo Krink, Mikkel T. Jensen, Zbigniew Michalewicz, and I investigated artificial dynamic problems and found no clear connection (if any) to real-world dynamic problems. In conclusion, a large part of this research field's foundation, i.e., the test problems, is highly questionable. In continuation of this, Thiemo Krink, Bogdan Filipič, and I investigated online control problems, which have the special property that the search changes the problem. In this context, we examined how to utilize the available computation time in the best way between updates of the control signals. For an EA, the available time can be a trade-off between population size and number of generations. From our experiments, we concluded that the best approach was to have a small population and many generations, which essentially turns the problem into a series of related static problems. To our surprise, the control problem could easily be solved when optimized like this. To further examine this, we compared the EA with a particle swarm and a local search approach, which we developed for dynamic optimization in general. The three algorithms had matching performance when properly tuned. An interesting result from this investigation was that different step-sizes in the local search algorithm induced different control strategies, i.e., the search strategy lead to the emergence of alternative paths of the moving optima in the dynamic landscape. This observation is captured in the novel concept of *optima in time*, which we introduced as a temporal version of the usual optima the in search space.

Acknowledgements

I owe a dept of gratitude to many people who have been important for my studies and the completion of this thesis.

In particular, I am deeply grateful to my two supervisors Thiemo Krink and Brian Mayoh. I would like to express my sincere thanks to Thiemo Krink for devoting a vast amount of hours to review, guidance, discussion, and moral support. To Brian Mayoh I owe a lot of gratitude for guidance and always being very open-minded and supportive regarding my projects.

I am very thankful to Bogdan Filipič for inspiring collaboration, interesting discussions, and for always taking good care of me during my stays in Slovenia. Special gratitude is also extended to Pierré Vadstrup for valuable insight into the field of system identification and for inspiring collaboration regarding the induction motors.

Many thanks are also due to Zbigniew Michalewicz who woke my interest in evolutionary computation and helped me with my first publications. Finally, I am also in dept to René Thomsen, Lars Bach, Mikkel T. Jensen, and Peter Rickers for valuable discussions and their time spent on commenting my papers.

Rasmus K. Ursem
Aarhus, 1st April 2003

Table of contents

Chapter 1

Introduction

In recent years, optimization algorithms have received increasing attention by the research community as well as the industry. Scientifically, the field of optimization algorithms is a highly relevant research area, because these algorithms can find approximate solutions to NP-hard problems and solutions to problems where no analytic method exists, e.g. for solving non-linear differential equations. Optimization algorithms have a very broad range of application, since many problems in science and industry can be formulated as an optimization task where the objective is to minimize or maximize a given objective function f. In other words, to find a solution s in the search space S of possible solutions such that $f(s)$ is minimized (or maximized). A simple example, though easily solved analytically, is the following.

Example 1.1:
Minimize

$$f(x_1, x_2) = x_1^2 + x_2^2 \qquad \text{where } x_1, x_2 \in [-2, 2]$$

Here is S the subset of $\mathbb{R}^2$ determined by the bounds $-2 \leq x_i \leq 2$, $i = 1, 2$.

$\diamondsuit$

From an industrial point of view, optimization algorithms are of major economic importance, because they can be used to improve or automate several processes in the company. For example, such algorithms can often be used to improve the quality of products, to lower the production cost, or increase efficiency in logistics and scheduling-related problems. In this context, even a few percent improvement of existing solutions may give the company a significant competitive advantage. Hence, optimization techniques can be an important key to success when considering the slim margin between expenses and revenues of today's companies. In contrast to the algorithmic approach, the manual search of a solution with a slight improvement is often tedious, if not impossible, because manual optimization requires a great deal of insight and patience. Furthermore, manual optimization often limits the scope of the search process to what the human expert is trained to consider as a good solution. Conversely, optimization algorithms automate the search and are not biased in scope regarding the solutions.

The wide range of real-world optimization problems and the importance of finding good approximative solutions have lead to a great variety of optimization techniques (for a comprehensive survey, see [100]). In this context, the so-called evolutionary algorithms (EAs) are a particularly promising approach, because this technique has shown good and robust performance on a broad range of real-world problems, e.g., [74; 104; 133; 52; 29; 113; 105; 78].

1.1 Evolutionary Algorithms

In short, evolutionary algorithms are iterative and stochastic optimization techniques inspired by concepts from Darwinian evolution theory. An EA simulates an evolutionary process on a *population* of *individuals* with the purpose of evolving the best possible approximate solution to the optimization problem at hand. In the simulation cycle, three operations are typically in play; recombination, mutation, and selection. Recombination and mutation create new candidate solutions, whereas selection weeds out the candidates with low fitness, which is evaluated by the objective function[1]. Figure 1.1 illustrates the initialization and the iterative cycle in EAs. Chapter 2 gives an elaborate introduction to EAs.

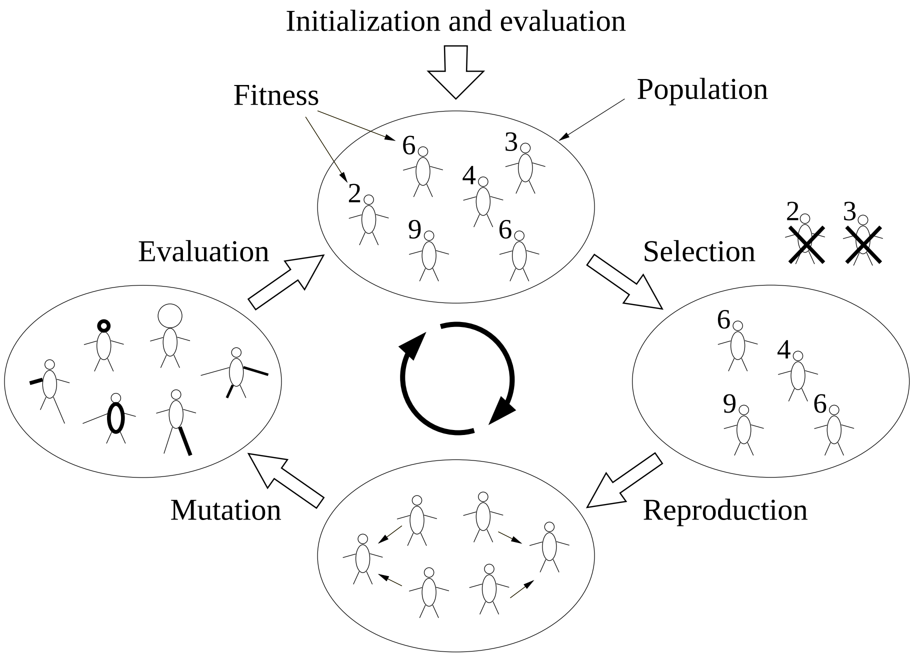


Figure 1.1: Initialization and the iterative cycle in evolutionary algorithms.

Historically, EAs were first suggested in the 1940'ties [51]. However, the founding fathers of modern EAs are considered to be Lawrence Fogel (Evolutionary Programming [53]), Ingo Rechenberg and Hans-Paul Schwefel (Evolution Strategies [113]), and by John Holland (Genetic Algorithms [68]). Several years later, *Evolutionary Algorithms* (EAs) and *Evolutionary Computation* (EC) were introduced as unifying terms for the forest of optimization techniques inspired by biological evolution. For a comprehensive overview, see [10].

[1] In EAs, the objective function is often referred to as the fitness function.

1.2 Why evolutionary algorithms?

In general, most real-world optimization problems have several challenging properties. Nearly all problems have a significant number of local optima, and the search space can be so huge that the exact global optimum cannot be found in reasonable time. Additionally, the problems may have multiple conflicting objectives that should be considered simultaneously (e.g., cost versus quality). Moreover, there may be a number of non-linear constraints to be fulfilled by the final solution. Furthermore, the problem may have dynamic components altering the location of the optimum during the optimization process. For some problems, variants of the local search approach have proven to be very efficient, e.g., Lin-Kernighan's algorithm for the Traveling Salesman Problem[2]. However, deterministic local search algorithms, such as steepest decent, do not allow a decrease in the solution's quality during the search. For this reason, these algorithms often stagnate at a local optimum, which makes local search less desireable for many real-world problems. Valuable alternatives are stochastic search methods such as simulated annealing, tabu search, and evolutionary algorithms. Among these techniques, EAs seem to be a particularly promising approach for several reasons. EAs are very general regarding the problem types they can be applied to (continuous, mixed-integer, combinatoric, etc.). Furthermore, these algorithms can easily be combined with existing techniques such as local search and other exact methods. In addition, it is often straightforward to incorporate domain knowledge in the evolutionary operators and in the seeding of the population. Moreover, EAs can handle problems with any combination of the above mentioned challenges in real-world problems (local optima, multiple objectives, constraints, and dynamic components). In this connection, the main advantage lies in the EA's population-based approach. For local optima, the genetic diversity of the population allows the algorithm to explore several areas of the search space simultaneously. This is of course no guarantee against premature convergence to a local optimum, but the population improves the EAs robustness on such problems. In multiobjective problems, EAs provide *a set of trade-off solutions* to the problem's conflicting objectives in a single run, whereas traditional approaches typically only produce one solution per run. Regarding constraint problems, EAs typically allow a mix of feasible and infeasible solutions in the population. This improves the algorithms capabilities of exploring the boundary between feasible and infeasible search space, and the capabilities for "crossing" infeasible regions. Finally, the population gives EAs an advantage on dynamic problems, because the population is likely to contain a good solution after the problem changes.

Naturally, EAs do also have some disadvantages. Unfortunately, they are rather computationally demanding, since many candidate solutions have to be evaluated in the optimization process. However, there has been a recent increase in interest in dealing with this problem and some techniques have been suggested (see section 3.2.3). Furthermore, EAs should not be applied blindfoldedly to any problem. As mentioned, many simpler and faster techniques exist and they should typically be

[2] Although one could argue that this is a problem is perhaps a bit "academic", it still represents many real-world problems, in particular scheduling and other permutation problems.

tried first. In this context, EAs offer the possibility to further improve solutions found by simpler techniques, which can be done by incorporating them in the start population (see section 2.3). In addition, EAs typically have a few more algorithmic parameters to tune compared with simpler techniques. These parameters are unfortunately problem dependent, but this is also the case for simpler techniques though fewer parameters need to be tuned. Consequently, a substantial amount of research has been carried out on methods for dealing with this issue (see chapter 4).

1.3 Objectives, contributions, and limitations

The primary objectives in this thesis are to investigate three fundamental challenges in evolutionary computation, to suggest algorithms dealing with these challenges, and to demonstrate the potential of the proposed algorithms on real-world problems in system identification and control. The three fundamental challenges are: Fitness function design, methods for parameter control, and techniques for multimodal optimization problems[3]. Furthermore, dynamic optimization problems were studied in the context of the three fundamental challenges as well as control.

Fitness function design is of major importance for EAs and other optimization algorithms, because the fitness function essentially determines how hard the problem is to optimize. There are several sub-aspects of fitness function design. The smoothness of the function is one of primary concern, because a too rugged fitness landscape may distrupt the search and trap the algorithm in a local optimum. Rugged fitness landscapes typically arise from imprecision in the fitness calculation or from the structure of the search space, i.e., that neighboring solutions are too unrelated. Regarding imprecise fitness calculations, I experienced this problem in connection with Runge-Kutta-Fehlberg approximation of non-linear differential equations for the greenhouse problem, which I investigated in collaboration with Thiemo Krink and Bogdan Filipič [88; 141; 143]. The imprecision problem with the Runge-Kutta-Fehlberg approach is discussed in section 3.2.2. For the search space structure, Thiemo Krink and I studied the neighborhood problem on arithemetic expression trees[4] [144]. To reduce the ruggedness of the landscape, we suggested the smooth operator genetic programming, which was tested on a simple black box structural identification problem. The proposed technique not limited to black box identification, but can be applied in other areas where the objective is to discover arithmetic expressions, e.g., fitness function approximation of computationally demanding problems (section 3.2.3).

As mentioned in section 1.2, parameter control in EAs has been studied extensively and several methods have been suggested to deal with this issue. One approach is to evolve the parameters along with the solution to the problem, and thereby let the algorithm self-adapt to the problem. Self-adaptation has been studied on static problems; however, this self-adaptive control approach requires a significant number of generations before superior parameters are evolved. Hence,

[3]Problems with many local optima.

[4]EAs with expression tree encodings are generally referred to as genetic programming.

it may not be adequate for dynamic problems, because they may change too fast. To study this further, I experimented with two artificial benchmark problems and showed that the self-adaptive approach fails on even rather simple time-varying problems [138]. This study is described in section 4.4.2. Regarding real-world application of self-adaptation, I included two self-adaptive evolution strategies in a comparison of techniques a parameter identification study on two induction motors (see chapter 7). This study was performed in collaboration with chief engineer Prof. Pierré Vastrup, Grundfos A/S. A journal paper is currently in submission [147]. In a different study, Thiemo Krink and I investigated a spatial agent-based approach to parameter control. Spatial grid models have been suggested as an approach to improve the performance of EAs on multimodal problems (section 5.2). For parameter control, we examined a variant of the patchwork model where the individual's position corresponds a parameter setting for that particular individual [87]. This terrain-based patchwork model (TBPM) allowed the algorithm to self-organize the population around the best parameters and thereby achieve better performance, because more individuals exploited good parameters and no evaluations were wasted on bad parameters. Spatial parameter interpretation was suggested by Gordon et al. [59]. Our contribution was to use spatially self-organizing agents, which is fundamentally different from the study by Gordon et al. and other control techniques. The novel TBPM is described in section 4.5.2.

Multimodal optimization is a third fundamental branch of research in EC. The field is related to the area of fitness function design, because the modality of the landscape is defined by the fitness function. For real-world problems, it is usually impossible to design the fitness function without local optima. Instead, multimodal optimization techniques are developed to handle such problems. In an EC-context, multimodal optimization is performed with two goals in mind. The algorithm shall find the global optimum while avoiding stagnation at local optima. Additionally, the algorithm shall preferably find several optima. The purpose of the latter goal is to allow a final decision by a human expert among the found solutions. For this goal, I suggested the multinational EA [137]. In this algorithm, the basic idea is to determine the location of valleys by calculating the fitness in sample points of the fitness landscape. The information about valleys is then used to divide the population into a number of self-organized subpopulations and thereby focus the search on several promising local optima. The multinational EA is described in section 5.4.3. In the initial study, the multinational EA was compared with the well-known sharing technique, which gave some rather strange results. In a follow up study, I investigated the behavior of sharing on simple variants of the two problems used in Goldberg and Richardson's original paper [58]. To my surprise, I found that the sharing algorithm was extremely sensitive to the range of fitness values [139]. This discovery was particularly surprising, because more than 100 papers and theses utilizing sharing have been published. The sharing approach is described in section 5.3.1, which also contains my criticism of the technique. As a continuation of these two studies, I investigated the role of genetic diversity in multimodal optimization. As mentioned in section 1.2, maintaining genetic diversity gives the algorithm an advantage in escaping local optima. In most studies, genetic diversity is achieved indirectly by various techniques, e.g., having slower genetic diffusion in the population. In contrast, I suggested the diversity-guided EA (DGEA), which use a diversity measure to directly control the population diversity [140]. In a more general context, my experiments indicated that both high and low diversity is important in multimodal optimization, which is somewhat controversy to the general belief that the maintaining of high diversity is the key to success in multimodal optimization. The algorithm is described in section 5.6.2. In collaboration with Pierré Vadstrup, I applied the DGEA to parameter identification of two induction motors [147]. For more information on this study, see chapter 7.

In my research, investigations on dynamic optimization problems have been a red line through much of the work. As mentioned earlier, some real-world problems have dynamic components and thus a time-varying fitness landscape. For this reason, EAs for dynamic problems have been investigated since the beginning of the nineties. Regarding my work on this topic, I carried out an initial investigation showing the potential of multimodal optimization algorithms, in this case the multinational EA, on artificial dynamic problems [138]. Theoretically speaking, the fitness of the currently located optimum may decline and thereby no longer be the global optimum. Hence a multimodal approach can continuously track local optima and thereby have a potential optimum located before it becomes the global optimum. This study was based on artificial dynamic problems, which I realized were too simplistic to allow conclusions about real-world problems. For this reason, Thiemo Krink, Mikkel T. Jensen, Zbigniew Michalewicz, and I conducted an investigation on the relationship between the widely used artificial dynamic problems and real-world dynamic problems [146][5]. As suspected, we found that the used artificial benchmark problems had no clear connection (if any) to real-world dynamic problems. Conclusively, a considerable part of this research area's foundation, i.e., the benchmark problems, is highly questionable (see section 3.3.3). As a continuation of this study, Thiemo Krink, Bogdan Filipič, and I started to investigate online control problems, which is a subgroup of dynamic problems that capture many interesting issues. Online control problems have the special property that the *search changes the problem*, since there is an interaction between the controller (the EA) and the controlled system. The primary limitation in online control is the available computation time between updates of the control signals. In an EA-context, the available time can be a trade-off between population size and number of generations, which we investigated on a simple greenhouse simulator [88]. However, the results were rather inconclusive and a more realistic and complex simulator was developed [145] (appendix C). The initial study was repeated and extended with an investigation of look-ahead times [141] (first published as a conference paper [142]). From the experiments in this study, we concluded that available time was best invested by having a small population size and many generations between problem updates. Hence, the problem was essentially turned into a series of related static problems. To our surprise, the problem was extremely easy to solve once it was treated as a series of static problems. This was further examined in a follow up investigation that compared the EA with a particle swarm optimization and a local search approach, which we specifically

[5]Published rather late because of a long review process.

developed for dynamic problems [143]. The three algorithms had nearly matching performance when properly tuned. An interesting result from this investigation was that different step-sizes in the local search algorithm induced different control strategies, i.e., the search strategy lead to the emergence of alternative optima in the dynamic fitness landscape. This observation is captured in the novel concept of *optima in time*, which we introduced as a temporal version of the usual optima in the search space [143]. In addition to the published investigations, I examined four multi-valued control approaches, which encode a control value per time-step instead of one value for the entire prediction horizon. Surprisingly, the simple approach with constant control signals turned out to be the best. The greenhouse study is described in chapter 8, which also contains the description of the novel particle swarm and local search algorithms.

Regarding the limitations of my Ph.D. project, there are a number of additional issues I would have liked to investigate. In this thesis, I studied three fundamental issues in EC: fitness function design, methods for parameter control, and techniques for multimodal optimization problems. I would also have liked to study techniques for handling problems with correlated parameters, i.e., problems where several search parameters co-vary and have to be changed simultaneously to improve the fitness (see section 3.1.3 for further information). Evolution strategies and particle swarm optimization algorithms are two techniques for this purpose. Although I experimented with these techniques in connection with the induction motor study, I would have liked to perform a broad investigation in this area. In system identification and control, a wide selection of approaches exist. Naturally, I would have liked to investigate more of these techniques; however, this was not possible in the given time frame.

1.4 Thesis outline

The thesis is written in book-style as a combination of survey chapters and case studies. The survey chapters describe the investigated fundamental research areas in EC, and put my research in perspective to other approaches and investigations. The case study chapters are included to demonstrate the potential of the algorithms on realistic problems.

In short, the thesis is structured as follows: Chapter 2 introduces the basics of evolutionary algorithms. This includes an overview of the encodings and their associated operators, the aspects of initialization, and the most commonly used selection operators. In chapter 3, the aspects of fitness function design is extensively discussed. The chapter contains a survey of theoretical aspects, practical aspects, and an introduction to the three special properties of real world problems (constraints, multiple objectives and dynamic components). Chapter 4 focuses on the issues related to parameter control in EAs. In this chapter, I describe a new taxonomy for parameter control approaches and give a survey of the techniques in this area of EC. The use of EAs in multimodal optimization is described in chapter 5. Here, I present an overview of the main algorithmic ideas and my own work on this subject. Chapter 6 gives an introduction to system identification and control, and a short survey of various techniques in these fields. This is followed by the case studies on parameter identification of induction motors and on direct control of a crop-producing greenhouse, which is the investigation on a realistic dynamic optimization problem. Finally, chapter 9 concludes the thesis and chapter 10 contains my plans for future research.

Chapter 2

Basics of evolutionary algorithms

Evolutionary algorithms (EAs) are iterative optimization techniques inspired by concepts from Darwinian evolution theory. However, the evolutionary process in EAs is extremely simplified compared with the process in nature. Although many terms used in connection with EAs have been adopted from biology, only a few modern approaches have implemented biological concepts in a realistic manner. Conceptually, an EA maintains a *population* of *individuals* that are *selected* and *created* in an iterative process. An individual consist of a *genome*, a *fitness*, and possibly a number of auxiliary variables such as age and sex. The genome consist of a number of *genes* that altogether *encode* a solution to the optimization problem. The *encoding* is the internal representation of the problem, i.e., the datastructure holding the genes. The fitness represents the quality of the solution encoded in the individual's genome, and it is usually calculated by a so-called *fitness function*. The surface obtained by the *fitness landscape* is the search space in relation to the fitness function.

Regarding the implementation of EAs, there is a great variety in population structures and evolutionary operators. However, all EAs have an initialization phase followed by an iteration phase that evolves the initial population to a (hopefully) better set of solutions to the problem. Figure 2.1 illustrates the pseudocode of a simple EA.

EA Main
  $\texttt{t} = 0$
  initialize population $P(0)$
  evaluate population $P(0)$
  while(!(termination condition)) {
    $\texttt{t} = \texttt{t} + 1$
    select population $P'(\texttt{t})$ from $P(\texttt{t}-1)$
    create population $P(\texttt{t})$ from $P'(\texttt{t})$
    evaluate population $P(\texttt{t})$
  }

Figure 2.1: A simple evolutionary algorithm.

In EAs, the population is usually initialized with randomly created individuals that are evaluated with respect to the fitness function. After initialization, the iteration phase loops until some termination criterion is met. This may be a maximal number of generations, a maximal number of fitness evaluations, or that a desired fitness is reached. The loop consist of four parts. First, the generation counter `t` is increased. Next, *selection* is applied to form the population at generation `t` from the population at generation `t-1`. Naturally, individuals with better fitness are more likely to be represented in the new population. After selection, a new population is typically created by *recombination*[1] and *mutation* of the solutions in the selected population $P'(\mathtt{t})$. The recombination operator creates one or two new solutions by mixing (crossing over) the genomes of two or more parents. The mutation operator alters the genome of one individual to create a new individual. A typical approach is to add a bit of stochastic noise to the existing solution. Finally, the new population is evaluated and the process is repeated.

During the run, the fitness of the best individual (hopefully) improves over time and typically tends to stagnate towards the end of the run (see figure 2.2).

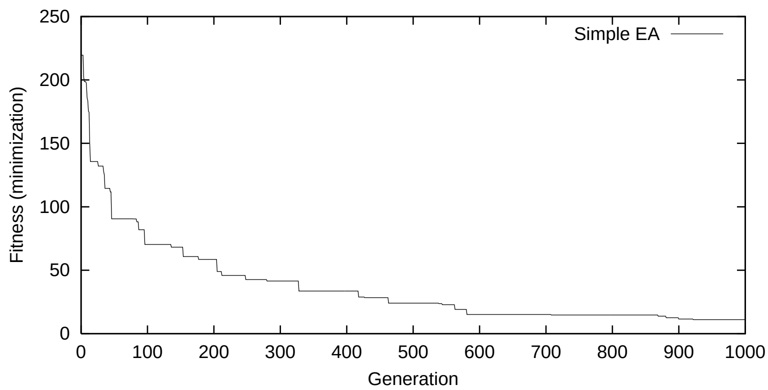


Figure 2.2: Gradual fitness improvement during the run.

Ideally, the stagnation of the process coincides with the successful discovery of the global optimum. However, stagnation also occurs on local optima, which is usually an unwanted result and one of the key problems in EAs and other iterative search algorithms. Typically, the performance stagnation is caused by genetic convergence of the individuals in one part of the search space, i.e., the genes of all individuals have become very similar. At this point, mutation is the only way to explore other areas of the search space, which corresponds to random steps away from the current location in the search space.

2.1 Basic terminology

The terminology of evolutionary computation (EC) is, to a large extent, borrowed from biology, but many terms have a different meaning in an EC-context. Unfortunately, there is no agreement on a large part of the basic terminology used in connection with EC. In general, researchers agree on the meaning of selection, mutation, and recombination, which is as described above. However, the terms

[1]Recombination is frequently called crossover.

related to the *problem*, the *objective*, and the *representation* are very vaguely defined and call for more concise and unifying descriptions. A system identification problem is used for illustrative purposes. The introduced terms are displayed in figure 2.3 for the example.

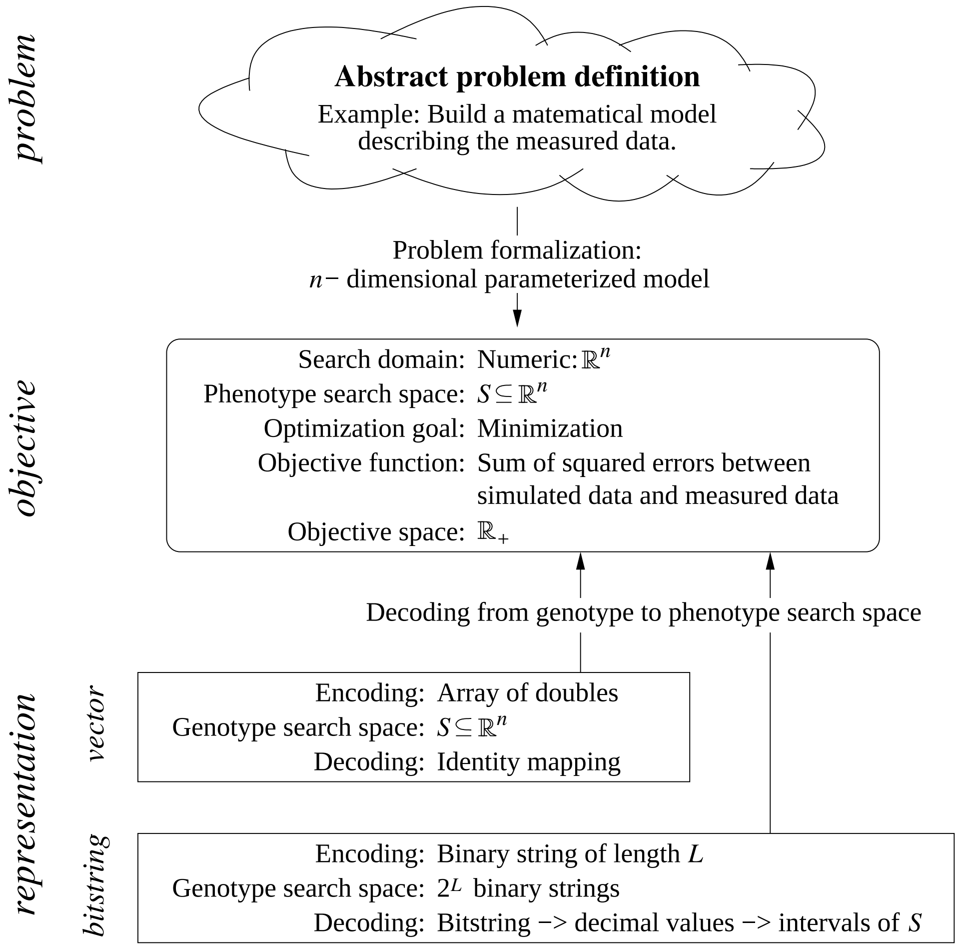


Figure 2.3: Terms related to the problem, the objective, and the representation.

The given problem is often described in an abstract way (top of figure 2.3). A system identification problem may be described as "find a mathematical model describing the measured data". First, the abstract problem description needs to be formalized. This can be done in a number of ways. In the system identification example, an engineer (a domain expert) may derive an n-dimensional parameterized model of the process that generated the data (the formalization used in figure 2.3). A completely different approach may be to use an artificial neural network to approximate the true system.

Assuming that the problem should be solved using a parameterized model, the *objective* is to find the values of the n model parameters that generate a behavior matching the measured data in the best possible way. Hence, the *search domain* is numeric and in this case $\mathbb{R}^n$. The actual *search space* S is usually defined by an interval for each of the n variables, i.e., $S \subseteq \mathbb{R}^n$. The search space is called the *phenotype search space*[2] when there is a difference between the encoding's domain and the search domain (see later).

The next step is then to define the *objective function*, or *fitness function*. In the system identification example, the objective function could be the sum of squared error between the simulated and the measured data. Note that there may be several meaningful functions for a given problem. To this end, a number of important issues arise when designing fitness functions. They are discussed in chapter 3. The objective function defines the *objective space*, which is the set of possible fitness values. In the case of a single objective, the objective space is usually a subset of $\mathbb{R}$. For multiobjective problems, the objective space is a subset of $\mathbb{R}^m$ where m is the number of objectives. See section 3.3.2 for further information on multiobjective optimization.

Settling on a problem formalization and a phenotype search space narrows the number of meaningful representations. A representation consist of an *encoding* datastructure and a *decoding* function. The encoding is used to store the actual solution in. The encoding defines the *genotype search space*[3] and also the size of this search space, i.e., the number of possible solutions. The decoding scheme is a mapping from the genotype search space to the phenotype search space. It may be the simple identity mapping[4] if the search space is a natural subset of the search domain (e.g., an interval in $\mathbb{R}$). In all other cases, a decoding scheme must be implemented. In the system identification example, the most straightforward approach is to use vectors, which may be represented as arrays of doubles. Another possible approach is to use binary strings of length L. Here, the decoding scheme must map solutions from the search space of L-bit binary strings to $\mathbb{R}^n$. Finally, the choice of encoding determines the set of possible evolutionary operators. Encodings and evolutionary operators are closely connected because the operators access the datastructure of the encoding directly. However, it should be mentioned that a great variety exist for each encoding, and that several new operators are introduced every year. For a comprehensive survey of the most commonly used operators, see [10]. The next sections describe the encodings and operators relevant for system identification and control problems. Furthermore, the remaining components of evolutionary algorithms are introduced.

2.2 Encoding, mutation, and crossover

The optimal type of parameter encoding in the genome of the individual depends on the definition of the problem. In principle, any problem parameters can be encoded by a binary representation. However, it is often convenient to use a high-level problem representation and implement specialized mutation and recombina-

[2] The term "phenotype" (alone) denotes the individual's solution in the search space *and* its corresponding fitness as well as other traits such as age and gender.

[3] The term "genotype" is often used in connection with the representation. However, there is no consensus regarding what genotype exactly denotes. Some researchers use genotype for the encoding; other researchers use it for both the encoding and the decoding scheme.

[4] The phenotype and the genotype search spaces are usually just called "search space" when the identity mapping is used.

tion operators for the particular encoding. The wide variety of EA-applications have created a great variety of encodings and operators. The most frequently used are encodings for numeric domains, permutation domains, matrix domains, and function domains. It is beyond the scope of this thesis to describe all of them in detail. Here, I will focus on the numeric and function domains, which are the two primary domains relevant for system identification and control.

2.2.1 Numeric search domains

Numeric domains cover problems where the objective is to find a numerical vector. The majority of EA-applications originate in this domain and therefore a significant amount of work has been devoted to investigate and develop encodings and operators for this domain. The two main encodings are the binary string encoding and the real-valued vector encoding.

An important issue in the representation of numerical problems is the precision of the encoding. A discrete encoding of a continuous interval can never be accurate, since any finite set of numbers leaves gaps in a continuous interval. The precision of the representation can be improved by increasing the number of bits in the binary representation. However, this improvement in precision also increases the size of the search space, which grows exponentially with the number of bits. For instance, the size of a search space in a 16-bit problem representation is $2^{16} = 65,536$. To double the precision, the genomes have to consist of 17 bits, which doubles the size of the search space. The same consideration applies to real value encoded problems. In high-level programming languages, the binary encoding is hidden from the programmer and the precision, and thus the size of the search space, is given by the internal representation of the used floating point data type.

Binary strings

Binary encoding is the traditional way to represent parameters in EAs. The data structure used for binary encoding is a bit-vector with fixed length L, which corresponds to 2^L different solutions in the search space. Apart from numerical problems, binary encoding is often used in permutation and combinatorial problems, such as the 0-1 knapsack problem. To use binary encoding with numeric domains, one has to specify a decoding function that maps the binary representation of a gene to a floating-point number. The decoding function converts the binary number to a decimal number, and then it is mapped to the real variable's search interval. Suppose a gene x is encoded by L bits, then the corresponding floating-point value x_{value} is calculated according to equation 2.1.

$$x_{value} = x_{min} + \frac{x_{max} - x_{min}}{2^L - 1} \left(\sum_{i=0}^{L-1} x[i] \cdot 2^{L-1-i} \right) \tag{2.1}$$

where x_{value} is the floating-point value, x_{min} and x_{max} are the minimal and maximal values of x, and $x[i]$ is the i'th bit in the binary encoding. If x is encoded by 8 bits, $x_{min} = -2$, and $x_{max} = 2$, then the binary number $\boxed{\texttt{01100111}} = 103$ is translated as follows:

$$x_{value} = -2 + \frac{2 - -2}{255} 103 \approx -0.3843 \qquad (2.2)$$

The simple two-dimensional test problem introduced in example 1.1 can, for instance, be encoded using 16 bits with 8 bits for x_1 and x_2 (see figure 2.4).

$$\underbrace{\boxed{11010010}}_{x_1}\underbrace{\boxed{00111001}}_{x_2}$$

Figure 2.4: The genome of a two parameter function with 8-bit encoding per parameter.

Another way to map a binary encoding to a numeric domain is called *Gray decoding*. The advantage of Gray decoding is that similar parameter values in the floating point representation correspond to adjacent numbers in the binary representation. For instance, the binary number $\boxed{\texttt{00011111}}$=31 is not adjacent to $\boxed{\texttt{00100000}}$=32 in the traditional binary encoding, although 31 and 32 are adjacent integers. If 32 is a better solution than 31 then the EA has to change six bits in the representation to change the value from 31 to 32. The Gray decoding function solves this problem such that neighboring integers are represented by binary numbers that differ in only one bit. Figure 2.5 shows a Gray decoding algorithm with $O(L)$ time complexity. However, in both binary decoding techniques there is the problem that a small change of the binary genome can lead to very large jumps in the floating point search space, such as in $\boxed{\texttt{00000001}} = 1$ and $\boxed{\texttt{10000001}} = 129$.

```
function Gray Decode(bit-string x) : integer
  ones = 0
  intvalue = 0
  for (i = 0; i<|x|; i++) {
    if (x[i]==1)
      ones++
    intvalue =intvalue + (ones mod 2)*2^(|x|-1-i)
  }
  return intvalue
```

Figure 2.5: Pseudocode for Gray decoding in linear time.

Bit-flip mutation

Bit-flip mutation is the most widely used mutation operator for binary encoded problems. The operator procedure consists of an iteration over all genes, where the bit in a gene g[i] is flipped if a uniform random number u of $U(0,1)$ is smaller than a certain probability threshold p_m. The main drawback of this operator is the time complexity, which is $O(L)$ for bit-strings of length L. However, the distance between two changed bits follows the geometric distribution, i.e., if p_m is the probability of changing a bit then $T \sim ge(p_m)$ is a stochastic variable describing the distance between changed bits. The number of bits t to skip can be calculated from the following function.

$$t = 1 + \left\lfloor \frac{\ln(u)}{\ln(1 - p_m)} \right\rfloor, \tag{2.3}$$

where u is uniformly distributed according to $U(0, 1)$. If the position t' of the next bit-flip is not in the current genome then the first bit flipped in the next mutated genome should be the $(t' - L)$'th bit. Empirical studies have suggested values for $p_m \in [0.001, 0.01]$ (e.g. [35] and [62]). Bremermann [23], and later Bäck [8], showed that the value $p_m = 1/L$ is optimal for simple sphere problems. For this reason, $1/L$ is usually used as a lower bound on p_m.

N-point and uniform crossover

A widely used crossover operator for binary (and also real encoding) is the n-point crossover operator, which recombines the genes of two or more parents in order to create two offspring genomes.

In one-point crossover, the parent genomes of size n are cut and reassembled at a random position p of the genome. The first offspring genome receives its genes between gene[1] and gene[p-1] from parent 1 and its remaining genes gene[p] to gene[n] from parent 2. The second offspring genome is assembled with the mirror-image of the first offspring genome, i.e., gene[1] to gene[p-1] are from parent 2 and gene[p] to gene[n] are from parent 1 (see figure 2.6).

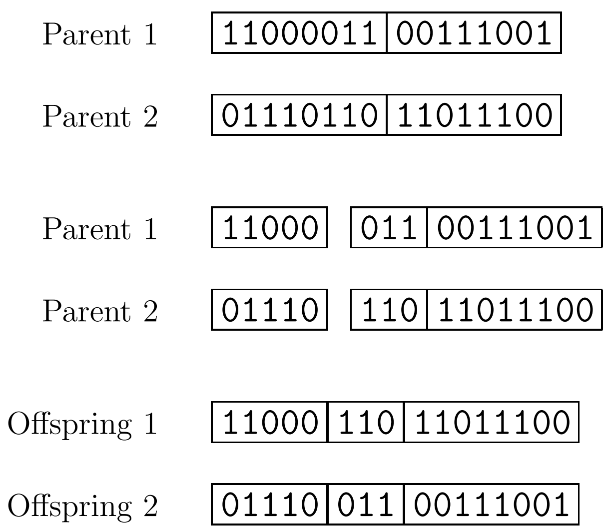


Figure 2.6: One-point crossover operator.

The difference between n-point and one-point crossover is the use of n crossover points instead of one. At each crossover point, the source of gene[i] alternates between the two parents. Usually n is a value between 1 and 4. Another frequently used crossover operator is the *uniform crossover*. In uniform crossover the offspring is generated by picking each gene[i] randomly from one of the parent's gene[i]'s.

Real-valued vectors

Another popular way to encode numerical domains is to represent the genes directly by (pseudo-)real numbers. Here, the search space is a subset of the objective domain. Thus, no decoding is necessary. The direct representation of the real values allows the design of mutation and crossover operators that are based on arithmetic operations and stochastic distributions.

Gaussian and uniform mutation

Most mutation operators for real valued vectors alter the solutions by adding a randomly generated vector $M = (m_1, m_2, \ldots, m_n)$ to the solution vector $\mathbf{x}$, i.e.,

$$\mathbf{x'} = \mathbf{x} + M \tag{2.4}$$

It is important that the m_i in M are generated from a distribution with zero as mean value, otherwise the solutions will drift due to the mutation. The common choice for the generation of M is the Gaussian distribution $N(0, \alpha)$ (figure 2.7).

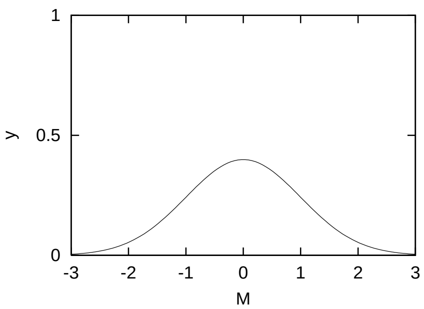
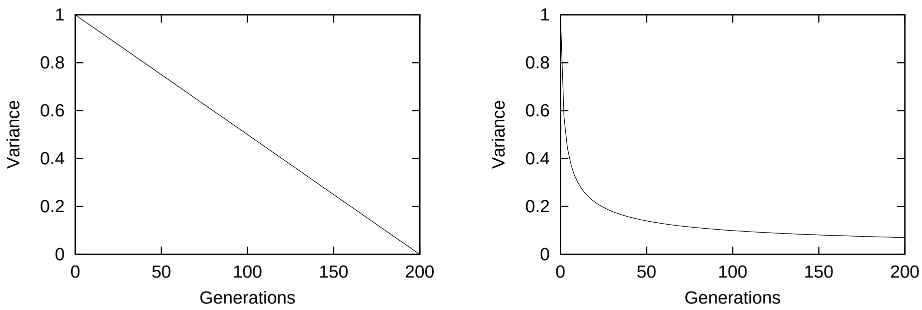


Figure 2.7: The Gaussian distribution for $N(0, 1)$.

A rather uncommon mutation is based on the uniform distribution $U(-\alpha, \alpha)$, where M is a value between $-\alpha$ and α with equal probability. A special case of the uniform mutation is $\mathbf{x'} = M$ with $M \in U(geneRange_{min}, geneRange_{max})$, which can be useful for the encoding of an enumerable parameter other than binary.

The performance of the mutation operator strongly depends on the parameter α. If α is set too high the algorithm has difficulties in fine-tuning the solutions, while if set too low the population might end up in a local optimum. Several techniques have been suggested to control α, such as self-adaptation in Evolutionary Strategies [113].

A very simple but effective solution is to define α as a function of the generation number. A well-supported hypothesis is that, in general, the population will converge towards a local or global optimum. To improve the chances of locating the global optimum the algorithm should start with a broad search strategy that gradually narrows as the population converges, i.e., α should be calculated from a decreasing function[5]. Two decreasing functions are displayed in figure 2.8.



(a) $\alpha(t) = 1 - 0.005t$ (b) $\alpha(t) = 1/(1+\sqrt{t})$

Figure 2.8: Decreasing functions for calculation of α.

Arithmetic crossover

Arithmetic crossover is an operator for real encoded genomes in which an offspring genome is generated by the weighted mean of each gene in the two parent genomes.

$$\mathbf{x'} = w\mathbf{x}_1 + (1-w)\mathbf{x}_2, \tag{2.5}$$

where w is the weight and $\mathbf{x}_1$ and $\mathbf{x}_2$ are the genomes of the parents. If $w = 0.5$ then arithmetic crossover calculates the offspring genome as the arithmetic mean of the two parents. The weight w is often generated according to the uniform distribution $U(0,1)$, which will place the offspring genome numerically between the parent genomes (figure 2.9(b)). A variant of arithmetic crossover generates a specific weight w_i for each gene x_i in the genome vector $\mathbf{x'} = (x'_1, x'_2, \ldots, x'_n)$.

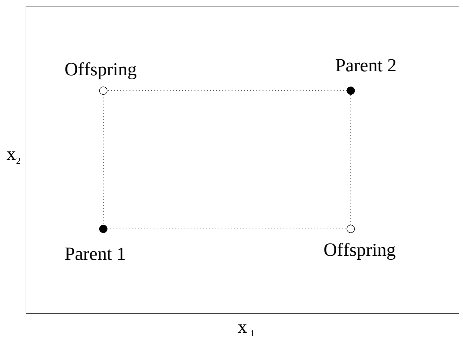
$$x'_i = w_i x_{1i} + (1-w_i)x_{2i} \tag{2.6}$$

In this variant, the offspring is placed at a random location inside the hypercube spanned by the two parents, see figure 2.9(c). A third variant of the arithmetic crossover generates the offspring of $k > 2$ parents. The offspring is created by combining the parents according to a number of weights, which define the amount of contribution from each of the parents. The offspring is created according to equation 2.7.

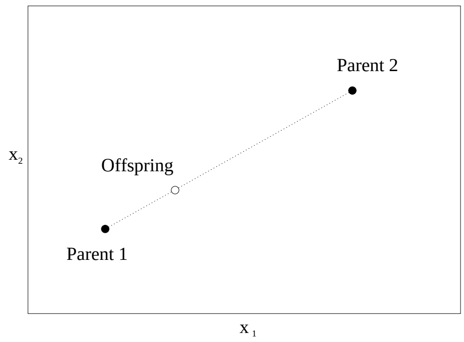
$$\mathbf{x'} = \sum_{j=1}^{k} w_j \mathbf{x}_j, \quad \text{where } w_j \in [0,1], \quad \sum_{j=1}^{k} w_j = 1 \tag{2.7}$$

In this setup the offspring is created in the convex hull defined by the k parents (figure 2.9(d)).

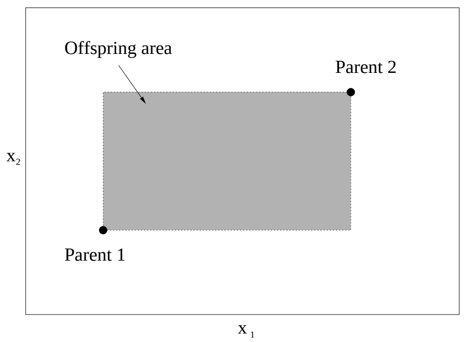
[5]This idea is often referred to as annealing.



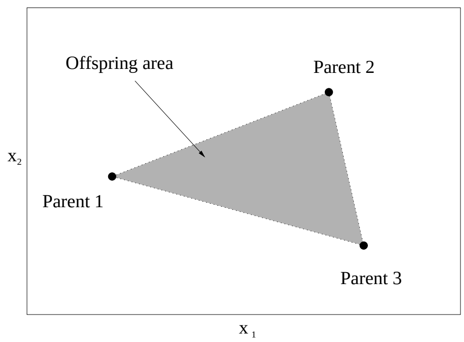
(a) Uniform and n-point crossover.



(b) Arithmetic crossover with one weight.



(c) Arithmetic crossover with n weights.



(d) Arithmetic crossover with 3 parents and one weight per parent.

Figure 2.9: Crossover for real valued vectors on a two-dimensional problem. The outer box illustrates the two dimensional search space.

2.2.2 Function search domains

In problems with function domains, the objective is to evolve a mathematical expression. EAs evolving expressions are usually called Genetic Programming (GP) in the literature. In GP, the evolved expressions act as problem solvers rather than particular problem solutions. This idea is closely related to the much older idea of Evolutionary Programming [53], which is an approach for evolving automata that can learn symbolic patterns.

The key data structure in GP is the parse tree representation. A parse tree consists of terminals and non-terminals. The terminals are the leaves of the tree, while the non-terminals are the nodes. The terminals may be constants and variables related to the problem. The non-terminals are operators such as +, /, and `if-then-else` constructions. The difference between terminals and non-terminals is that the non-terminals have subtrees under them. For instance, the + operator has a left and a right subtree. Non-terminals can have different numbers of subtrees. For instance, the unary minus has one subtree, plus has two, while the `if-then-else` construction has three subtrees (condition, then part, and else part). A tree is evaluated by recursively traversing the tree. Naturally, a non-terminal cannot be evaluated unless its subtrees have been evaluated.

The choice of terminals and non-terminals is of course dependent on the kind of parse trees that shall be evolved. They must be carefully selected to allow just the kind of expressions that are needed to represent the problem solution. A too limited set may lead to functions with rather poor performance. On the contrary, a large set of operators could make the search difficult because the search space grows with the available operators. Another approach to limit the search space is to introduce a maximal depth of the evolved trees. Additionally, the choice of operators and terminals might introduce some technical problems. For instance, the return type of the subtrees of the `if-then-else` operator are both boolean and the type of the expression in the `then` and `else` branch. The evolutionary operators have to ensure that the tree only contains syntactically legal expressions. Another kind of problem are illegal arithmetic expressions such as division by zero and square-root of negative numbers. This problem is usually handled by letting the operator return a fixed value when it would otherwise have rendered an illegal value. For instance, the division operator may simply return 1 when a division by zero occurs.

Grow, shrink, switch, and cycle mutation

Mutation operators for function encodings either alter the structure of the parse tree or the internal value of nodes and leaves. Angeline defines four forms of mutation called grow, shrink, switch, and cycle [4] The grow operator replaces a random leaf with a new randomly generated subtree. The shrink operator selects a random node and replaces it with a random leaf. The switch operator exchanges two subtrees of a random node (provided that the selected node has two subtrees). Finally, the cycle operator replaces a node's operator by another operator with the same number of subtrees. Figure 2.10 illustrates an example of the four operators.

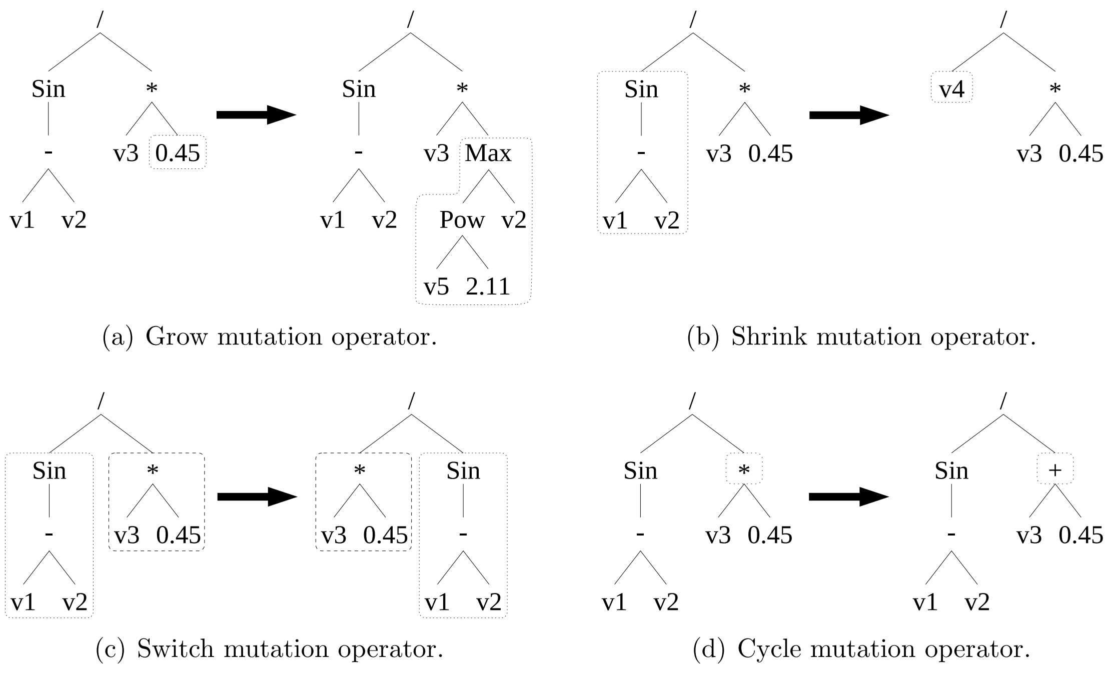


Figure 2.10: Grow, shrink, switch, and cycle mutation for function encodings.

Subtree crossover

Crossover in parse trees is surprisingly simple. The most widely used crossover operator is the subtree crossover. The operator selects two nodes with the same return type in the parents. Two children are created by swapping the subtrees starting at the selected nodes (figure 2.11).

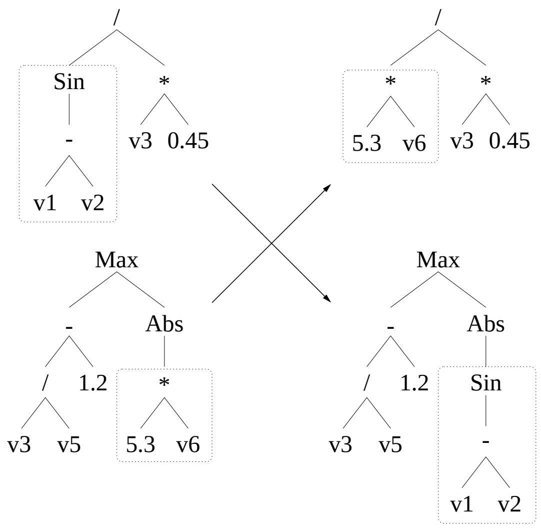
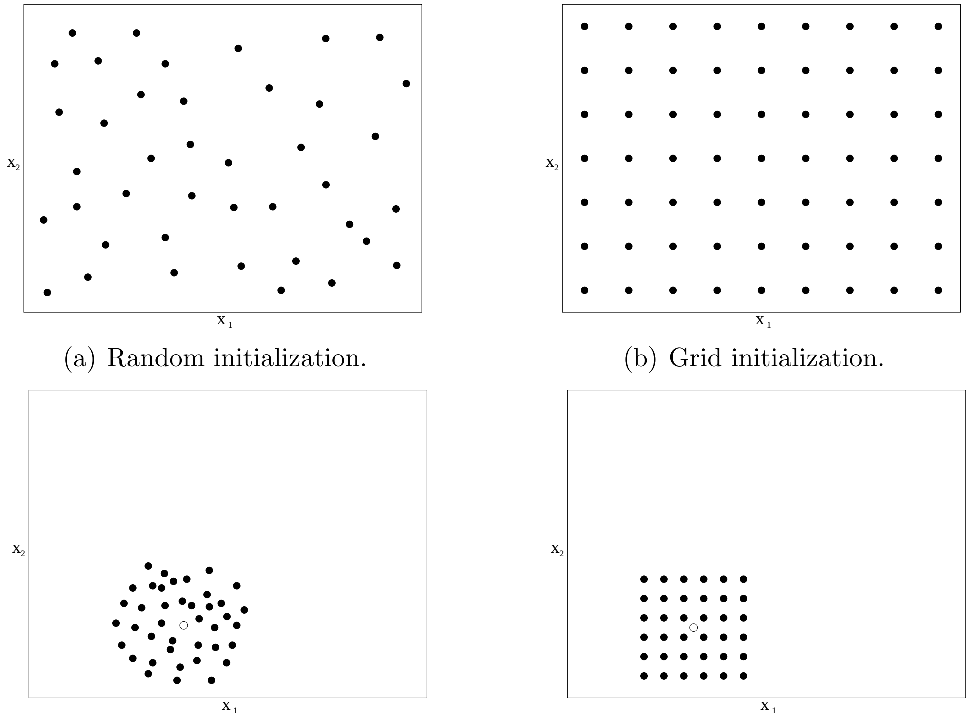


Figure 2.11: Subtree crossover.

2.3 Population initialization

The initialization of the population specifies the starting points of the search. The initial population can be created in a number of ways (see figure 2.12). The most common setup is the random initialization where the chromosomes are randomly assigned, preferably using a uniform distribution. The goal is to create a population with a good coverage of the search space, and thereby have a gene pool with good potential for breeding better solutions. Alternatively, genomes can be evenly scattered over the whole search space according to a regular grid-layout. However, deterministically determined search space positions can be suboptimal starting points. In particular, a random setup can take advantage of a completely new selection of starting points when runs are repeated. A third approach is to incorporate expert knowledge into the initialization. In some cases, it is possible to assign the initial search space positions based on specific knowledge about the objective function. Domain experts usually have an idea of what a reasonably good solution is. Furthermore, the current best known solution may easily be incorporated in the search by just inserting the solution as one of the starting individuals. The remaining individuals could then be randomly scattered or arranged in a grid near the best known solution. A problem with such an initialization is that the search may be too focused on the area around the special solution. A randomly initialized population may allow the EA to discover fundamentally different solutions in comparison with what a human would have proposed. Several examples can be found in the literature, e.g., Rechenberg's early and famous tube-bending study [113]. Finally, including solutions created by other search techniques seem to be an extremely promising approach, although rarely used. To this end, Thomsen et al. recently investigated the potential of this approach for multiple sequence alignment in bioinformatics [129]. Their approach improved the initially generated solution by 10% and reduced the running time of the EA from several hours to a few minutes.



(a) Random initialization.

(b) Grid initialization.

(c) Knowledge-based random initialization.

(d) Knowledge-based grid initialization.

Figure 2.12: Examples of initialization methods. The boxes denote two-dimensional search spaces. The circle in (c) and (d) represents the current best known solution.

In summary, the choice of initialization methods depends on the study one is performing. Random initialization is used in most general investigations on EAs, because the global optimum is usually known for test functions used in this context. For real-world applications, a rule of thumb is to incorporate as much expert knowledge as possible in initialization as well as operator design.

2.4 Selection operators

Selection is an essential process in EAs that removes individuals with a low fitness and drives the population towards better solutions. In this section, I will describe the four most common selection operators and manual selection, which is used when a formal description of the fitness is impossible. The selection operator essentially defines how the algorithm updates the population from one iteration to the next. In general, selection either replaces the entire population or only a fraction of it. The former approach is used in *generational EAs* whereas the latter is employed in *steady-state EAs*. There are a few major differences between the two approaches. First, the selection procedure is stochastic in generational EAs, but deterministic in steady-state EAs. Hence, generational EAs may accidently not select the currently best solution. However, it is generally considered a good idea to ensure the survival of the best individual. This scheme is referred to as *elitism* or k*-elitism* if k individuals are saved as the elite. Second, individuals are cloned in generational EAs whereas steady-state EAs select a deterministic subset of the candidate solutions. Steady-state selection is mainly used by the Evolution Strategies [113].

An important aspect of selection is the selection pressure, which governs the individual's survival rate. It is important to balance the selection pressure. A too high pressure usually leads to convergence to a small area of the search space and thus possibly premature stagnation on a suboptimal solution. A too low pressure will result in a very slow convergence.

2.4.1 Tournament selection

Tournament selection creates the next generation by holding a tournament for each slot in the population of the next generation. In each tournament, the process picks k random individuals, compares their fitnesses, and copies the individual with the best fitness to the slot. The tournament size k is usually set to two individuals and rarely above five, since this would impose a too strong selection pressure and lead to premature convergence. Figure 2.13 shows the pseudocode for tournament selection with a tournament size of two. The "source" population is usually fixed during the selection of the next generation, which allows good individuals to be copied multiple times.

```
tournament selection(P(t))
  for (j = 0; j<|P(t)|; j++) {
    Pick two random individuals I1 and I2 in P(t).
    Compare the fitness of I1 and I2.
    Insert a copy of the fitter individual in P(t + 1) at position j.
  }
```

Figure 2.13: Tournament selection with a tournament size of two.

Tournament selection is easy to implement, produces good results within short time, requires very little computing time, and is controlled by only a few parameters. For these reasons, tournament selection is probably the most commonly used selection operator nowadays.

The selection pressure in tournament selection can be increased by letting more individuals compete. A tournament size of two will, on average, ensure that the best individual is copied twice to the next generation. Increasing the tournament size to three will also increase the better individuals' winning chances, because all individuals on average take part in three tournaments instead of two. On the other hand, the selection pressure can be lowered by introducing stochastic winners in tournaments with two individuals. Hence, the fittest individual wins with probability $p > 0.5$. Typical values are $p = 0.75$ or $p = 0.8$. Setting $p = 0.5$ is equivalent to random selection.

2.4.2 Proportional selection

Proportional selection assigns the probability of an individual's survival according to the fitness of the individual. The probability is calculated by dividing the fitness of the individual by the fitness sum of the whole population, i.e., an individual's chance of survival depends on its relative fitness to the other individuals.

$$p_{survival}(I) = \frac{fitness(I)}{\sum_{j=1}^{popsize} fitness(I_j)}, \qquad \text{i.e.,} \sum_{j=1}^{popsize} p_{survival}(I_j) = 1 \tag{2.8}$$

Each individual is assigned to a "slot" of the interval $[0, 1]$ according to the individual's $p_{survival}$. An individual is selected if a random number of the interval $[0, 1]$ is within its slot. This selection method is often illustrated as a biased roulette wheel[6], where the interval slots correspond to the slots of a roulette wheel and the "winners" are copied to the next generation.

The drawback of proportional selection is that the selection pressure depends on the relative fitness of the individuals instead of a parameter such as tournament size. In proportional selection, a few very good individuals can quickly take over the entire population, because they dominate a large part of the roulette wheel and is therefore frequently copied when the next generation is formed. For this reason, proportional selection is not as popular as it used to be.

2.4.3 Ranking selection

Ranking selection is a variant of proportional selection that deals with the uncontrolled selection pressure. In ranking selection, the selective superiority of an individual is determined by a fixed probability $p_{survival}$ according to its fitness rank. The ranking is obtained by sorting the individuals according to their fitness. Each individual is then assigned a probability $p_{survival}$, which is determined by the used ranking scheme (see table 2.1 for an example). The selection is performed using the roulette wheel approach.

Rank	$p_{survival}$	Rank	$p_{survival}$	Rank	$p_{survival}$	Rank	$p_{survival}$
1	0.100	6	0.075	11	0.045	16	0.020
2	0.095	7	0.070	12	0.040	17	0.015
3	0.090	8	0.065	13	0.035	18	0.010
4	0.085	9	0.060	14	0.030	19	0.005
5	0.080	10	0.055	15	0.025	20	0.000

Table 2.1: Example of ranking scheme for population size of 20 individuals.

[6] Proportional selection is sometimes called roulette wheel selection.

The difficult part of applying ranking selection is to determine a good probability $p_{survival}$ for each rank. A scheme that is too generous towards low-fit solutions might slow down the convergence, while a scheme favoring the best individuals might lead to a premature loss of genetic diversity.

2.4.4 Steady-state selection

Evolutionary algorithms that are based on steady-state selection, also known as steady-state EAs, update only a small fraction of the population in every iteration. The evolutionary operators create λ potential solutions from the parent population with size μ. Afterwards, the $(\mu + \lambda)$ individuals are sorted and λ individuals with the lowest fitness are discarded[7]. Common values are $\mu = 100$ and $\lambda = 15$. This approach is fundamentally different from tournament, proportional, and ranking selection. In steady-state selection the populations are overlapping and *all* the surviving individuals are deterministically selected, which is only the case for the elite individuals in the other three selection techniques.

2.4.5 Manual selection

In some applications, the quality of a solution is based on a subjective evaluation of issues that are hard or impossible to capture mathematically; for instance, the beauty of a design. Instead, the selection process can be handled by a human operator. The algorithm displays the current solutions and asks the operator to select a subset of the presented solution. The selected solutions are then used to create a new population and the process is repeated. Examples of manual selection include evolution of robot controllers [96], mixing of food-colors [66], and more experimental applications in evolutionary art [42].

2.5 Summary

In this chapter, I have outlined the components of evolutionary algorithms and given a few guidelines of how to combine them. It is rather difficult to give general advises on what to choose. This is partly because of the so-called “no free lunch” theorem stating that no search algorithm is better on all problems [153]. However, the theorem is mainly of theoretical value because “all problems” literally means *all problems* including those that map a search point to an arbitrary value. Obviously, random search is as good as any advanced EA on this type of problem. In practice, problems are not that noisy and the no free lunch theorem is therefore primarily of theoretical interest. Unfortunately, practical experimentation has shown that the performance of encoding and operators is problem dependent. Hence, some experience with the techniques is an advantage when handling real problems. In this context, a catalog of guidelines would be incredible valuable. The first step in designing such a catalog would be to define a system for categorizing problems based on the problem definition and the local topology of the fitness landscape. It is clearly not sufficient to base such a guideline on general properties such as

[7] μ and λ are commonly used in the literature on steady-state EAs.

number of peaks. The performance ranking of encodings and operators is not consistent for even very similar problems. Fortunately, most combinations give a reasonably performance and there are a few heuristics regarding implementation and values of parameters. Consequently, researchers tend to stick to their favorite encoding and operators.

2.6 Future work

Regarding new research on encodings, the main objective domains are well investigated. For these domains, the focus is primarily on developing novel alteration operators. In less explored domains, some work is being performed on developing novel encodings and operators.

Somewhat surprising, the population initialization has not received much attention so far. One explanation may be that most EA investigations are performed on artificial benchmark problems where the global optimum is known in advance and any application of "domain knowledge" would be considered cheating. However, even in this scenario some important issues need to be investigated. For instance, the initial population may be generated randomly, but currently the initial sample of individuals is usually equal to the population size. To my knowledge, no one has attempted to generate, e.g., ten times the population size and then choose the most promising candidates for the initial population. Fortunately, the entire field seems to focus more and more on real-world applications. Therefore, the initialization phase may receive increasing attention in the future.

In a wider context, a significant amount of work should be done on comparing various implementations of EAs. For instance, tournament selection can be implemented in at least two ways. One approach is to iteratively fill the array of the next generation by filling position j with the best of two randomly chosen individuals from the previous generation. Another approach is to let the individual at position j be one of the contesters.

Chapter 3

Aspects of fitness function design

The fitness function essentially determines how difficult the problem is to search. For real-world problems, the fitness function is often defined in collaboration with an expert from the application domain. From an EC point-of-view, there are a number of important theoretical and practical issues to consider when defining the fitness function. Additionally, constraint, multiobjective, and dynamic problems are introduced as the three main fitness aspects of real-world problems requiring specialized techniques. These issues are covered in this chapter and put in context of system identification and control problems.

3.1 Theoretical aspects of fitness function design

In most numerical problems, the fitness function is explicitly given by a mathematical equation. However, many real world problems are usually not well-defined and their representation is up to the designer of the EA. The primary criterion is that the fitness function properly ranks the individuals so the most desirable solution is assigned to the best fitness (maximization or minimization). Otherwise, selection will choose the wrong individuals when forming the next generation.

In addition to the ranking criterion, a number of important properties of fitness functions exist. These are not strict requirements, but issues that should be considered when designing the fitness function, because the search performance depends on the topology of the fitness landscape. First, the fitness landscape should not contain plateaus. Selection fails on plateaus, because all individuals have the same fitness in such a scenario. Consequently, the EA essentially degrades to random search when the landscape mainly consists of plateaus. Second, the fitness landscape should be somewhat smooth, i.e., adjacent solutions in the search space should have similar fitness values. Searching a very irregular landscape corresponds to finding a needle in a haystack, and no algorithm can do this any better than random search. Third, ridges in the search space may pose an additional challenge to the algorithm, because a ridge in the fitness landscape corresponds to correlation between the parameters in the search space. In the vicinity of a ridge, any movement in the search space that is not in the direction of the ridge orientation will lead to a fitness reduction. Fourth, local optima may attract the population of the algorithm and lead to a premature convergence of the search.

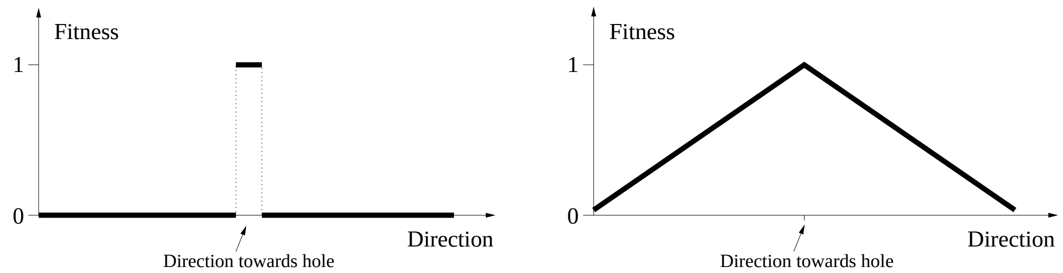
Plateaus can usually be handled by transforming the fitness function to a more gradual design, whereas problems with smoothness may be solved by using a different encoding. The last two issues, ridges and local optima, are difficult to find a fitness design solution to. Instead, they are tackled by various extensions of the basic algorithm.

3.1.1 Plateaus

Plateaus can often be avoided by transforming the original objective function into one with a more gradual fitness landscape. The following example illustrates a transformation of a fitness function. Let us assume that we want to evolve a controller for a golf-putting robot. A simple fitness function could be defined such that the fitness of a genome is 1 if the robot puts the ball into the hole and 0 otherwise. In this setup, a robot controller that slightly misses the hole will receive the same fitness as a robot that shoots the ball in the opposite direction. Hence, the EA is practically performing a random search, if no robot in the initial population can hit the hole. The corresponding landscape would consist largely of plateaus and a tall narrow peak that represents the successful robot behavior. Instead of a simple success-or-failure evaluation, the fitness should incorporate the concept of a "near miss" by a measure of the closest distance between the ball and the hole; for example:

$$f(I) = 1 - min\_dist(\text{ball, hole})$$

This fitness function assigns 1 to the individuals that are able to hit the hole and slightly smaller values to near misses. The two fitness functions are displayed in figure 3.1.



(a) Success-or-failure fitness function. (b) Distance-to-hole fitness function.

Figure 3.1: Two fitness functions for the golf-putting robot.

Plateaus are unfortunately unavoidable for some problems. In function domains, the used non-terminals may introduce occasional plateaus. In particular, the if-construction can cause rather large plateaus when the boolean expression returns a constant value.

if (<B-exp>) **then** return <A-exp1> **else** return <A-exp2> (3.1)

For instance, if <B-exp> is false then the return value of the if-sentence in equation 3.1 is completely independent of the arithmetic expression <A-exp1>. Hence, any change in <A-exp1> will not affect the fitness of the individual.

3.1.2 Smoothness

Achieving landscape smoothness is somewhat the opposite problem of avoiding plateaus. On plateaus, EAs have difficulties because the landscape does not contain any information for directing the search. An extremely rugged landscape has more or less the same problem, because solutions in the vicinity of a solution may have an arbitrary fitness value. Hence, the algorithm is essentially searching for a needle in a haystack. Figure 3.2 illustrates a rugged fitness landscape.

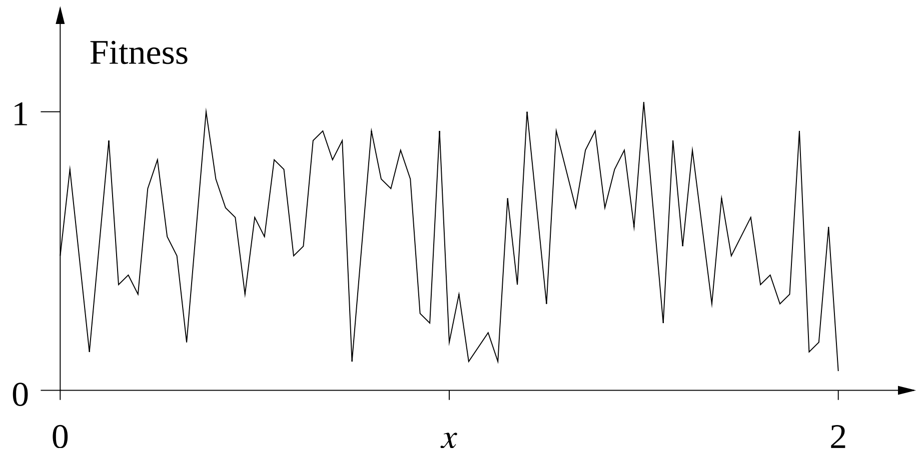


Figure 3.2: A rugged fitness landscape for the search space $x \in [0, 2]$.

Smoothness of the fitness landscape is primarily related to the encoding, although ruggedness may also appear because of imprecision in the fitness calculations (see section 3.2.2). In theory, numerical problems are normally rather smooth because the variables are often continuous and a small change in the value will not result in a large change in the fitness. In contrast, defining a proper neighborhood in the function search domain[1] of genetic programming is far from easy. Until now, most genetic programming studies (if not all) define the neighborhood of an expression as the set of expressions reachable with a "minimal change" of the expression; for instance, exchanging a binary operator (+) with another binary operator ($*$). Although this is the smallest possible change, apart from modifying a constant, it can have quite a drastic effect on the expression. For example, mutating $500 + x$ into $500 * x$ is a minimal change that could vastly alter the genome's fitness. Hence, neighboring solutions in the search space may receive completely different fitness values although their edit-distance in the search space is minimal. To deal with this problem, smoother operators have been suggested for genetic programming. The idea in smooth operator genetic programming is to combine several "ordinary" operators (the non-terminals) into one and allow a gradual change from one operator to another. In boolean function domains, Poli et al. suggested smooth operators for boolean parity problems [107; 108]. Their study is based on the idea of reducing ruggedness by introducing a smoother representation of boolean operators. For this purpose, they replace the standard operators `AND`, `OR`, and `NAND` by a sub-symbolic representation where the output of a node is represented as a

[1]The search space of expression trees, see section 2.2.2.

truth-table. In their study, `1000` represents an `AND` while `1110` corresponds to `OR`. An in-between operator can for example be `1100`. Combining this encoding with traditional bit-flip mutation and a specialized crossover operator allows the GP to interpolate between operators and thus search a smoother fitness landscape.

For arithmetic function domains, Thiemo Krink and I introduced the two smooth operators *diviplication* and *subdition* [144]. Diviplication combines division and multiplication, whereas subdition integrates subtraction and addition. Diviplication (DP) and subdition (SD) are defined as follows:

$$xDP[a,b]\,y \;\equiv\; sgn(x) * sgn(y) * |x|^a * |y|^b \tag{3.2}$$
$$xSD[a,b]\,y \;\equiv\; a * x + b * y \tag{3.3}$$

where $sgn(x)$ is the sign of x and $|x|$ is the absolute value of x. The slightly long definition of diviplication with $sgn(\cdot)$ and $|\cdot|$ is merely to avoid problems with undefined values of the power function such as $(-1)^{0.5} = \sqrt{-1}$. A more intuitive (but troublesome) definition of diviplication is:

$$xDP[a,b]\,y \;\equiv\; x^a * y^b \tag{3.4}$$

Defining diviplication according to equation (3.2) or (3.4) does not eliminate the ordinary operators division and multiplication – they are inherent in diviplication and can be obtained by setting a and b to 1 or -1.

$$\begin{array}{lll} a=1 & b=1 & : \; xDP\,y \equiv x*y \\ a=1 & b=-1 & : \; xDP\,y \equiv x/y \\ a=-1 & b=1 & : \; xDP\,y \equiv y/x \end{array}$$

As mentioned, the goal in introducing the smooth operators is to allow a gradual change from one operator to another. For instance, a diviplication node with $a = 1$ and b changing gradually from 1 to -1 corresponds to a smooth change from $x * y$ to x/y. Some intermediate expressions are:

$$\begin{array}{lll} a=1 & b=1 & : \; xDP\,y \equiv x*y \\ a=1 & b=0.5 & : \; xDP\,y \equiv x*\sqrt{y} \\ a=1 & b=0 & : \; xDP\,y \equiv x \\ a=1 & b=-0.5 & : \; xDP\,y \equiv x/\sqrt{y} \\ a=1 & b=-1 & : \; xDP\,y \equiv x/y \end{array}$$

Figure 3.3 illustrates the gradual transition from multiplication to division. As seen on the center graph for “x”, the output of diviplication is not exactly x because the sign of y affects the output. However, this is only a problem if y is close to zero and both positive and negative. A way solving this is to define an *interpolating diviplication* (IDP) as a linear interpolation between multiplication and division.

$$xIDP[a]\,y \;\equiv\; a(x*y) + (1-a)(x/y) \tag{3.5}$$

The sign problems are not present in the subdition operator. Here, the ordinary addition operator can be changed gradually into subtraction.

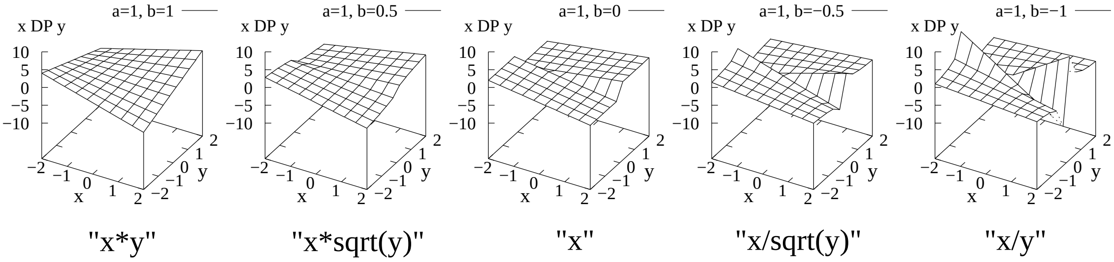


Figure 3.3: Gradual change from multiplication to division in diviplication.

An important aspect of SOGP is setting the range and the step-size for a and b. The *order* of diviplication and subdition can be controlled by the range. For instance, setting it to [-2:2] allows diviplication expressions such as $x^2 * y$. The *discreteness* of the operator can be controlled by the step-size. For example, a step-size of 0.5 and a range of [-1:1] limits the possibilities to expressions having exponent a and b in $\{-1, -0.5, 0, 0.5, 1.0\}$. Hence, only expressions based on $x, \sqrt{x}, 1, 1/\sqrt{x}, 1/x$ and $y, \sqrt{y}, 1, 1/\sqrt{y}, 1/y$ are possible.

SOGP with diviplication and subdition was compared with traditional GP on a simple black box system identification problem [144]. The two approaches had similar performance with a slight advantage to SOGP. Furthermore, SOGP seemed to be more robust when the found models were evaluated on the test data.

3.1.3 Ridges

Ridges in the fitness landscape are generally hard to avoid because it requires a rewrite of the fitness function that removes the correlation among the problem parameters. Instead, problems with correlated parameters are usually handled by various extensions of the basic algorithms. The algorithmic challenge in handling ridges is to change multiple problem parameters simultaneously and thereby avoid reduction in fitness. In other words, to search in the direction of the ridge orientation. A simple ridge is illustrated in figure 3.4.

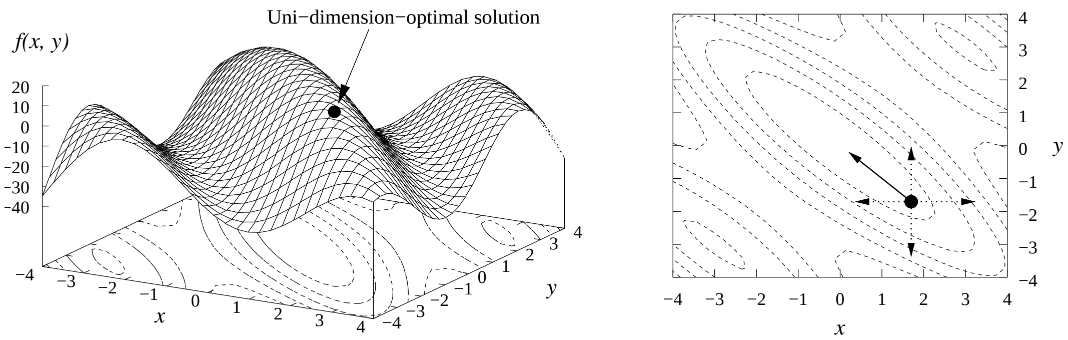


Figure 3.4: A simple problem with a ridge. $f(x, y) = 20\cos(y + x) - y^2 - x^2$.

In this example, a solution located near the axis $x = -y$ going through (0,0) is *uni-dimension-optimal*, i.e., optimal if only one parameter, x or y, is changed.

Hence, the EA has to change both parameters to achieve an increase in fitness. This is illustrated in the right graph of figure 3.4. The dashed lines are contour lines as shown on the left graph. The dotted lines with arrows illustrate changes in only one parameter whereas the solid line with the arrow is the direction towards increasing fitness values.

Handling ridges through specialized alteration operators has been investigated for at least 25 years. In 1977, Schwefel extended his work with Rechenberg on the Evolution Strategies (ESs) and suggested the self-adaptive ES with correlated mutations [122]. The idea in self-adaptation is to encode algorithmic parameters in the genome and use these parameters to modify the individual. The hypothesis is that good solutions carry good parameters; hence, evolution discovers good parameters *while* solving the problem. Particularly for ridges, the self-adaptive ES with correlated mutations encodes a set of rotation angles that allow the algorithm to generate new solutions with correlated parameter values. The main drawback of encoding rotation angles is the rather large number necessary for high-dimensional problems. For instance, a 100-dimensional problem requires 4950 angles to allow rotation between all dimensions [12]. This makes the approach rather unsuitable for these problems, because the adaptation of the angles takes additional time. For problems with rather few parameters, a self-adaptive ES is often a good choice. This is further discussed in chapter 7 where two self-adaptive ESs are compared with respect to system identification of two induction motors. In addition to handling ridges, self-adaptation seems to be a promising approach for controlling the parameters of the algorithm. This topic is covered in section 4.4.

A recent and vastly simpler approach is the so-called Differential Evolution (DE) suggested by Storn and Price [127]. DE algorithms create new individuals by adding the vector difference between two randomly chosen individuals to a third individual. The main difference between DE and ESs is that DE utilizes the information from the population whereas self-adaptive ESs encode the information in each separate individual.

3.1.4 Local optima

Local optima are practically impossible to avoid in real-world applications, and redesigning the fitness function to remove local optima is extremely difficult in most cases. For this reason, a significant amount of work has been dedicated to develop methods for handling problems with many local optima[2]. Figure 3.5 illustrates a simple example of a function with two local optima and one global optimum.

Local optima are particularly problematic for some methods such as deterministic local search techniques, because these algorithms only use one current solution to create new candidate solutions. Furthermore, new solutions are usually generated as immediate neighbors of the current solution. Consequently, these algorithms have a tendency to stagnate on a local optimum because escaping such optima may require a significant amount of backtracking, or “downhill movement”,

[2]Such problems are called multimodal in the literature, whereas problems with only one global optimum are named unimodal or convex problems.

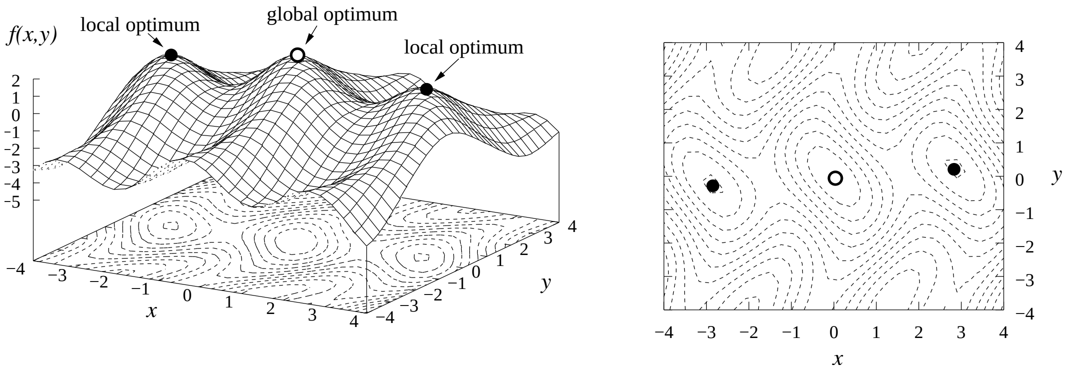


Figure 3.5: A simple multimodal problem. $f(x, y) = \cos(2x + \sin(y)) + \cos(y) - 0.1(x^2 + y^2)$.

before new fitness improvements occur. Naturally, other optimization algorithms incorporate various techniques for handling multimodal problems. A widely used technique is the so-called Simulated Annealing (SA). For an overview of SA and other related techniques, see [100].

Evolutionary algorithms are particularly well-suited for handling multimodal problems. First, EAs maintain several solutions simultaneously, which provide a significantly better foundation for escaping local optima. Second, solutions are not necessarily created as neighbors of existing solutions. Over the years, several hundred algorithms have been suggested. Chapter 5 contains an overview of the most widely used techniques as well as some recent approaches.

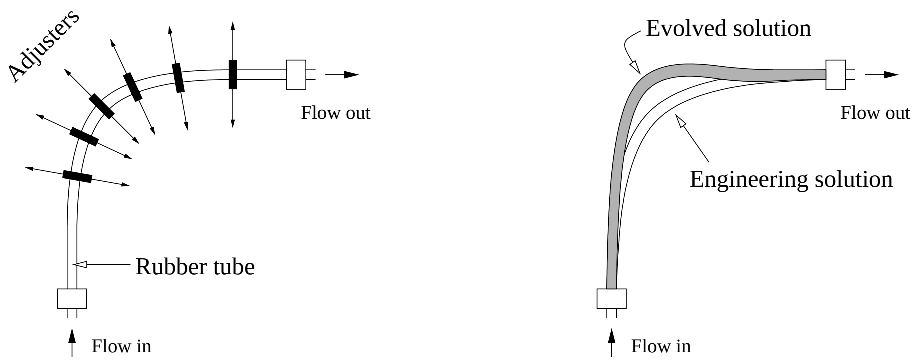
3.2 Practical aspects of fitness function design

In addition to the theoretical aspects, a number of practical aspects should be considered when designing fitness functions for real-world problems. First, it may be possible to use the real system to evaluate candidate solutions. The primary advantage of this approach is that the candidate solutions are evaluated on the physical system instead of a model. Hence, the usual mismatch between the model and the true system is avoided in this setup. However, this approach is often impossible because of physical or safety reasons. For instance, it may be too expensive, too dangerous, or too time-consuming to evaluate a solution using a real system. Instead, the optimization algorithm uses a simulation-based fitness function. A fundamental problem in both cases is that it can be very time-consuming to evaluate a solution. For example, complex simulations of fluid flows may be necessary to obtain a precise fitness value. Fortunately, techniques exist for handling such problems.

3.2.1 System-based fitness functions

In system-based fitness functions the physical system serves as a test bed for evaluating candidate solutions of the problem. The main motivation for using a system-based fitness function is that it eliminates the problem of transferring a simulated solution to the real system. Naturally, system-based fitness evaluation is not always possible. First, it may be too dangerous to use a real system for evaluation. For instance, evolving a controller for a nuclear power plant is not a feasible approach. Second, system-based evaluation may be too expensive. For example, testing a large number of settings for a production plant may be very costly. Third, the evaluation may take several hours to perform, which often makes this approach impractical. Fourth, the evaluation may affect the system state in an irreversible way and thereby unfairly bias the evaluation and thus selection.

One of the first implementations of system-based fitness functions was performed by Rechenberg and Schwefel in the early seventies [113]. Among other things, they worked on design of bending a tube 90°, with the purpose of minimizing the pressure loss. In their experiment, they placed six adjusters along a rubber tube that was fixed at both ends of the 90° turn (figure 3.6(a)). Rechenberg and Schwefel used a simple (1+1) Evolution Strategy and produced a rather surprising results. Figure 3.6(b) shows the symmetric engineering solution and the asymmetric solution found by the algorithm. Interestingly, the evolved solution resembles the one appearing in nature when, e.g., a river makes a 90° unhindered turn.



(a) Experimental setup with tube adjusters.

(b) Engineering solution (symmetric) and evolved solution (asymmetric).

Figure 3.6: Tube design of 90° turn using a system-based fitness function.

Naturally, system-based fitness functions are not always straightforward to implement and use. First, the system usually needs to settle between experiments to ensure a fair evaluation. Furthermore, the system is often influenced by the immediate environment. For example, many systems change behavior depending on the temperature. Hence, the evolved solution may only work for certain temperatures. An example is the recent investigations on evolvable hardware where the evolved physical circuits are often sensitive to even small changes in the room temperature. Finally, the evaluation may be noisy because of small differences in the start conditions of the experiment, imprecise sensors, and other factors. A simple approach to tackle this problem is to resample the fitness. However, that requires additional experimentation, which increases the fitness calculation accordingly. Another technique is to inherit fitness values from the parents. For instance, storing the fitness obtained in the last ten generations and let the offspring inherit the nine most recent fitness values during crossover. The offspring's fitness then consists of its own fitness plus the average of the nine inherited fitness values. Yet another approach is to apply a Kalman filter to lower the noise from sensors [128].

3.2.2 Simulation-based fitness functions

Simulation is necessary when real-system evaluation is impossible. There are several important issues when basing fitness evaluation on a simulation of the real system. A fundamental problem is that a simulator can never be 100% accurate. Hence, a solution found in simulation may not render the same performance when transferred to the real system. This is particularly a problem when using simulators to design robot controllers. One problem in this context is the lack of fuzziness and randomness in the simulation. For instance, real sensors are usually never completely noise-free. One solution to this problem may be to introduce some noise in the simulated sensor input and thereby hope that the evolved controller will be more robust. Another problem in robot simulation is the unmodeled physical components of the robot and the environment. For example, the robot's wheels may have some friction that is hard to capture. In addition to friction, it may be very difficult to model every single component of the physical world such as sand on the floor and changing light conditions.

Another fundamental problem in simulation is that it sometimes relies on inaccessible data; for instance, weather data from the future. In this case, the simulator must be able to predict the missing data. The precision of the prediction may have a substantial impact on the performance of the evolved solution. Hence, both a system simulator and an predictor must be developed to cope with such problems.

Simulating system identification and control problems

System identification and control problems can be modeled by the interactions between the controller, the system, and the surrounding environment (see figure 3.7). Here, the vector $\mathbf{x}(t)$ represents the internal state of the system at time t, $\mathbf{v}(t)$ is the environment state, $\mathbf{u}(t)$ is the control signal from the controller, and $\mathbf{y}(t)$ is the output from the system. The environment state $\mathbf{v}(t)$ and the output $\mathbf{y}(t)$ from the system serve as feedback input to determine the control signal for the next time-step. In system identification, the objective is to find a system model of the controlled system, whereas in controller design problems the focus is on the decision maker (see figure 3.7).

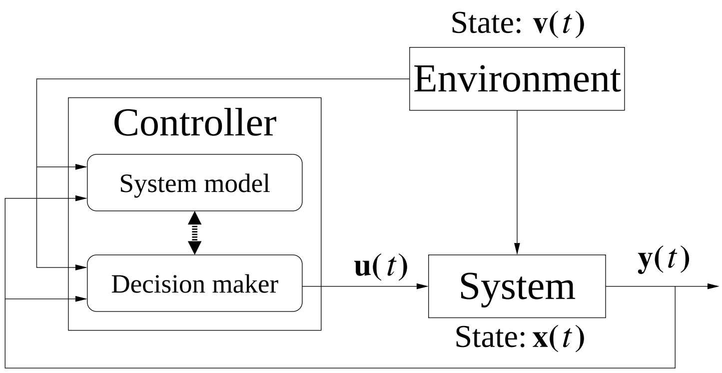


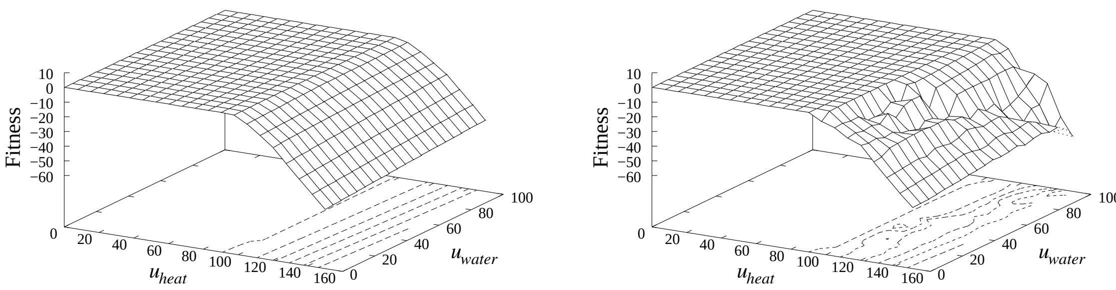
Figure 3.7: Model for controller, system, and environment.

The change in system state is often modeled by a number of difference equations of the form:

$$x_i(t+h) = x_i(t) + \Delta x_i(\mathbf{u}, \mathbf{x}, \mathbf{v}, t, h) \tag{3.6}$$

where x_i is the i-th system variable in $\mathbf{x}$, $\Delta x_i(\cdot)$ is the update function for x_i, t is the time, h is the length of a time-step, and $\mathbf{u}$, $\mathbf{x}$, and $\mathbf{v}$ are the control signal, the system state, and the environment state of the previous time-step (sometimes several steps in the past). Real systems are often described by a system of non-linear differential equations. In these cases, an approximation method, such as Runge-Kutta [1], is used as the update function $\Delta x_i(\cdot)$.

The Runge-Kutta approximation method is probably one of the most frequently used techniques for handling non-linear differential equations. The traditional Runge-Kutta approach has an error bounded by $\mathcal{O}(h^4)$, i.e., dividing the step-size h by 2 gives a factor 16 in precision improvement. In practice, a Runge-Kutta-Fehlberg approach with adaptive step-size is often used [46] (for implementation details, see [109]). The basic idea is to use an estimate of the error to control the step-size. The method adjusts the step-size to ensure that the error is less than a predefined threshold. Hence, a large step-size is used when the function is rather flat and small steps are used when the function is rugged. Consequently, the algorithm saves a significant amount of computation time. An important problem arises when the adaptive step-size technique is used in connection with optimization algorithms. The adaptive approach can accidently introduce "phantom peaks" in the fitness landscape, i.e., peaks that appear because of the adaptive step-size control. As a consequence, the algorithm may stagnate on a local optimum that is an artifact of the approximation method. Figure 3.8 illustrates two fitness landscapes from the greenhouse control problem investigated in chapter 8 (see appendix C for a detailed description of the simulator). The only difference between the two plots is that one uses a constant step-size whereas the other uses an adaptively controlled step-size.



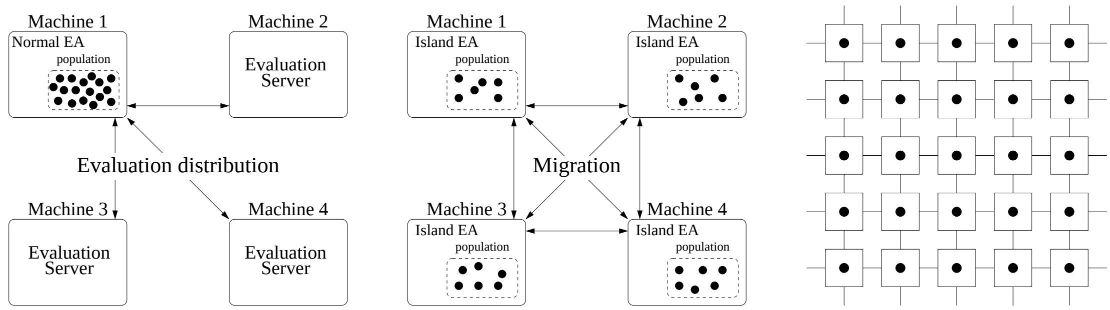
(a) Fixed step-size. (b) Adaptive step-size.

Figure 3.8: Phantom peaks introduced by adaptive step-size control in the Runge-Kutta-Fehlberg approximation of non-linear differential equations.

3.2.3 Computationally demanding fitness functions

Another main challenge when applying EAs to real-world problems is the computation time required to evaluate a candidate solution. EAs can be rather demanding with respect to number of evaluations. It is not unusual that the algorithm needs 50-100,000 evaluations to reach a reasonably good solution. Assuming that one evaluation takes 10 seconds, a run with 100,000 evaluations will then take about 278 hours to perform. Now, 10 seconds is rather fast in relation to some real-world problems. For instance, industrial design problems are often investigated with simulators for fluid flow dynamics (Computational Fluid Dynamics, or CFD). In such problems, one simulation may take several hours to perform. Hence, certain techniques are necessary when applying EAs (and other optimization algorithms) to computationally demanding problems. In general, three approaches exist. First, the EA can be parallelized in a number of ways. Second, the algorithm can store solutions in a relational database for later reuse. Third, expensive fitness functions can be approximated and thereby reduce the number of real evaluations.

Regarding parallel EAs, three basic architectures exist; the master-slave model, the island model, and the diffusion model (see figure 3.9). The master-slave approach is the most widely used in practice. In this approach, one machine runs the main algorithm and the rest acts as evaluation servers. In the island approach, each machine holds a local population. Migrations between the islands then occur at fixed intervals. Island models should be used when an individual is too large to fit an entire population on a single machine. Finally, the diffusion model is used when the parallel computer consists of many small interconnected units (hypercube architecture). Here, each unit holds one individual and the interaction range is then the unit's immediate neighbors. Somewhat surprising, a considerable number of papers have been published on the subject particularly on migration schemes in the island model, e.g., [27; 91]. Nevertheless, the rather pragmatic master-slave model exploits the available parallel computer best in most cases. This is mainly because EAs require a reasonably large population of at least 50-100 individuals.



(a) Master-slave model. (b) Island model. (c) Diffusion model.

Figure 3.9: Parallel architectures for EAs.

Another simple idea is to store all evaluated solutions along with their fitness values in a database for later reuse of the fitness calculation. Naturally, this may cause some problems if the solution requires a significant amount of memory to store. However, this simple approach can save a substantial amount of computation time. Another idea in this connection is to implement a random search algorithm and run this when nodes are idle. This ensures that the computing facility is almost fully exploited. The randomly generated solution may then be incorporated in either the initial population or during the execution of the EA.

In addition to using parallel computers and relational databases, computation time may be saved by simply reducing the number of fitness evaluations. The most straightforward approach is of course to use a smaller population size. However, this usually deteriorates the performance of the algorithm. A better approach may be to approximate the fitness function by a so-called metamodel, which is a model of the fitness landscape. Combining EAs with metamodels have received increasing attention during the last five years. So far, rather few investigations have been reported in the EC-literature although metamodels have been studied for several years in other disciplines. In an EC-context, El-Beltagy et al. investigated a a statistical approximation technique known as Kriging [43]. They showed good approximation on a number of simple benchmark problems and used the technique to optimize the design of a beam structure. Giannakoglou et al. used radial basis function (RBF) neural networks for airfoil optimization [57]. They compared three approaches in their study; a simple EA without approximation, a RBF approach, and a RBF approach with adaptive importance factors for each of the optimization parameters. The last technique was superior on all four airfoil problems. In a similar study, Emmerich et al. tested the Kriging approach on one of the airfoil design problems investigated by Giannakoglou et al. [44]. The two techniques have their advantages and disadvantages. The RBF approach is relatively easy to implement but does not guarantee the true value when a training pattern is evaluated. Naturally, this can be done by simply checking if the solution exist in the database of evaluated points. Kriging guarantees exact values in the training patterns, but the technique requires substantial effort to implement. Furthermore, Kriging requires that a simple minimization problem is solved before each evaluation. Fortunately, this can be done rather quickly with a local search technique. These two techniques are those mainly applied in connection with EAs. Many other model building methods can be used to approximate fitness functions; for instance, feedforward neural networks or regression models.

3.3 Special properties of real-world problems

A large number of real-world problems have additional issues to consider when designing the fitness function. First, there may be constraints limiting the set of feasible solutions. For instance, certain requirements with respect to safety may have to be fulfilled. Second, multiple conflicting objectives must often be optimized simultaneously. The task in these problems is to find a set of reasonable trade-offs between the conflicting objectives. Third, the problem may change over time. Consequently, the algorithm has to be able to adapt to a dynamic fitness landscape. Naturally, real-world problems can have any combination of these special properties.

3.3.1 Constraints

In constraint problems, the found solutions must fulfill a number of requirements. These requirements, or constraints, are usually formulated as a set of inequalities that must hold for the final solutions. Generally speaking, a constraint problem on the search space $\mathcal{S}$ may be formulated as follows:

Definition 3.1: Constraint problem
Optimize

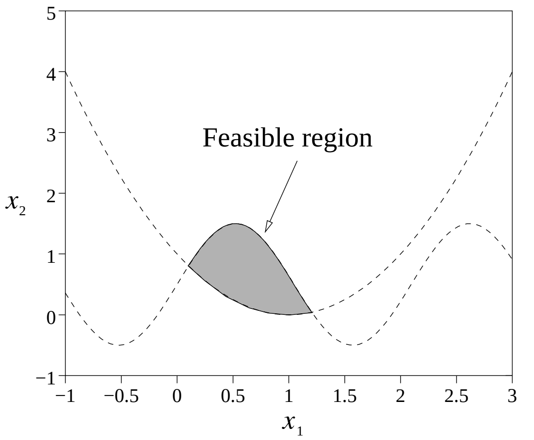
$$f(s), \quad s \in \mathcal{S}$$

subject to $p \geq 0$ inequalities and $q - p \geq 0$ equalities

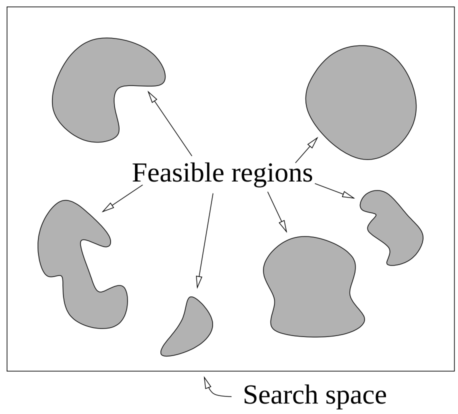
$$\begin{aligned} g_j(s) &\leq 0 \qquad & j &= 1, 2, \ldots, p \\ h_j(s) &= 0 \qquad & j &= p+1, p+2, \ldots, q \end{aligned}$$

△

Solutions fulfilling the constraints are *feasible solutions*, and those violating the constraints are *infeasible solutions*. Hence, the constraints partition the search space $\mathcal{S}$ into two disjoint subsets; the feasible region $\mathcal{F}$ and the infeasible region $\mathcal{U}$ (see figure 3.10(a) for a simple example). In general, the search space may consist of any number of disconnected feasible regions (figure 3.10(b)).



(a) Search space with respect to the two constraints $(x_1 - 1)^2 - x_2 \leq 0$ and $x_2 - \sin(3x_1) - 0.5 \leq 0$.



(b) Abstract search space with a number of feasible regions.

Figure 3.10: Feasible regions in the search space of constraint problems.

Constraints are often considered to be either *hard* or *soft*. Hard constraints must be fulfilled by the solution whereas soft constraints may be slightly violated. For example, assume that we are planning the air traffic over Europe on a given day. The safety margins for air traffic are then hard constraints that must be fulfilled by the solution. A soft constraint in this example could be that planes leave no later than 15 minutes after their designated time of departure, i.e., leaving 20 minutes late is acceptable but should preferably be avoided. A measure of the constraint violation is often useful when handling both hard and soft constraints. A straightforward measure of constraint j's violation is:

$$f_j(s) = \begin{cases} \max\left(0, g_j(s)\right) & 1 \leq j \leq p \\ |h_j(s)| & p+1 \leq j \leq q \end{cases} \tag{3.7}$$

The sum of all constraint violations is zero for feasible solutions and positive when at least one constraint is violated. An obvious application of the constraint violation is to use it to guide the search towards feasible areas of the search space.

Regarding EC-research on constraint problems, a substantial amount of work has been published; in particular on algorithms for numerical constraint optimization. The most widely used technique is probably the penalty approach. In this method, the constraint violation is used to calculate a penalty, which is added to the fitness for minimization problems and subtracted for maximization problems. Hence, infeasible solutions receive a fitness penalty for violating the constraints. Michalewicz and Fogel provide an extensive survey of penalty approaches and other methods for constraint handling [100, Chapter 9]. In system identification and control problems, constraints play an important role. For system identification, the constraints may express some relationship among the variables that must be fulfilled. With respect to control problems, constraints typically express a system behavior that is considered to be feasible, e.g., that certain safety regulations are fulfilled. This is further discussed in chapter 6.

3.3.2 Multiple objectives

In some problems, the task is to optimize with respect to multiple objectives instead of just one. A typical example is car engine design, where the task may be to maximize the performance while minimizing the fuel consumption. Multiobjective optimization problems usually involves a number of conflicting objectives that have to be handled simultaneously. The goal for the algorithm is to find a set of tradeoffs between the objectives and thereby allow a final human decision among the solutions. The objectives does not necessarily have to be conflicting but they are in most problems. In some cases, it may be unclear from the beginning whether or not objectives are in conflict with each other. Multiobjective optimization problems may be defined as follows:

Definition 3.2: Multiobjective optimization problem
Find a set of solutions optimizing

$$F(s) = [f_1(s), f_2(s), \ldots, f_m(s)]$$

where $f_i(s)$ is either a maximization or a minimization problem[3].

$\triangle$

Hence, the objective space is now m-dimensional instead of one-dimensional. For a n-dimensional numerical problem, the fitness mapping is as follows:

$$\begin{array}{lll} \text{Single objective:} & F(s): & \mathbb{R}^n \mapsto \mathbb{R} \\ \text{Multiple objectives:} & F(s): & \mathbb{R}^n \mapsto \mathbb{R}^m \end{array}$$

[3]Multiobjective optimization problems are often defined with an additional set of constraints. However, constraint handling techniques are usually investigated independently of methods for multiobjective optimization.

For single objective optimization problems, the mapping defines a clear ordering of the individuals that is used in selection. This is a bit more complicated for multiobjective problems because the objective space is m-dimensional. As mentioned earlier, the fitness function should be designed, so that nearby solutions in the search space are smoothly mapped to similar values in the one-dimensional objective space. Figure 3.11 illustrates three cases. The first case (white circles) shows the ideal situation where neighboring solutions are mapped to the same vicinity in the objective space. Case two illustrates a scenario where solutions from different areas of the search space are mapped to nearby points in the objective space. Finally, case three is the situation that should preferably be avoided. Here, adjacent solutions are mapped to different areas of the objective space. It should be noted that these three cases do exist for single-objective optimization as well. Case one is like the smooth distance-to-hole fitness (figure 3.1(b)), case two is the multimodal problem (figure 3.5), and case three resembles the rugged fitness function (figure 3.2).

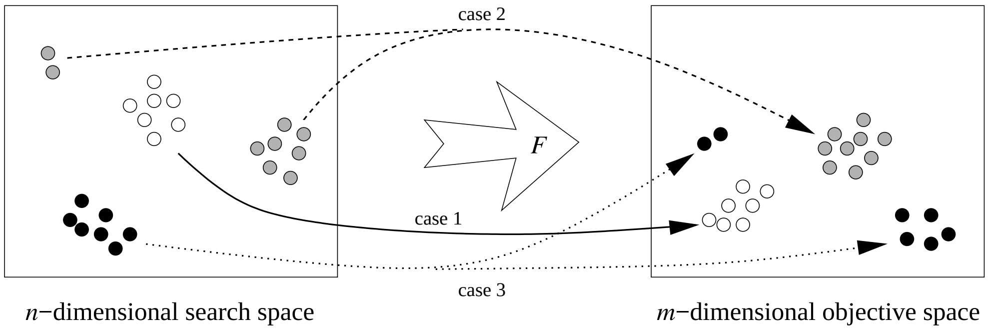


Figure 3.11: Multiobjective mapping from search space to objective space.

As mentioned, selection for multiobjective problems is a bit more complicated than the single-objective case, because the conflicting objectives do not define a clear ordering of all points in the objective space. A partial ordering exists through the concepts of *dominance* and *non-dominance*. Formally, dominance and non-dominance are defined as:

Definition 3.3: Dominance

A solution s is *dominated* by a solution s' if s is "equal or worse" on all objectives and "strictly worse" on at least one objective.

$$s \prec s' \quad \longleftrightarrow \quad s \text{ is dominated by } s'$$

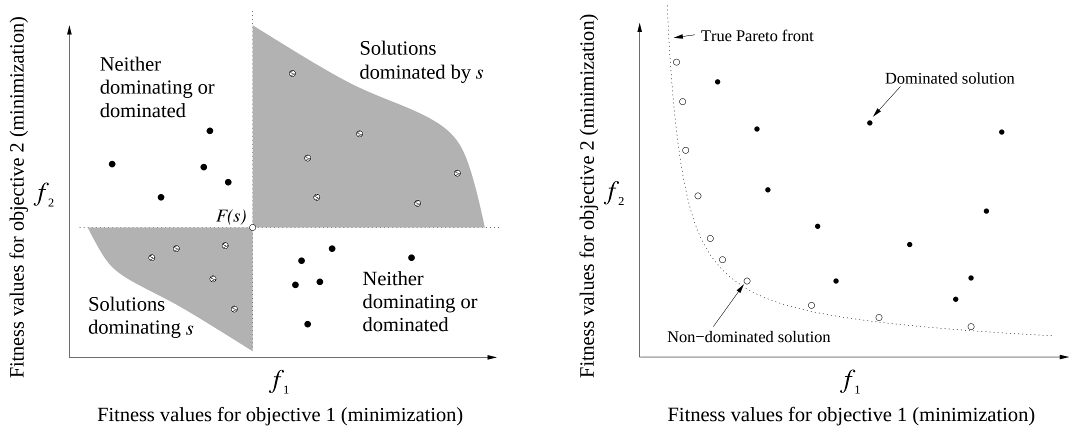
$\triangle$

Definition 3.4: Non-dominance

A solution s is *not dominated* in the set P if $\forall s' \in P : s \neq s' \Rightarrow s \not\prec s'$, i.e., if no other solution in P is dominating it.

$\triangle$

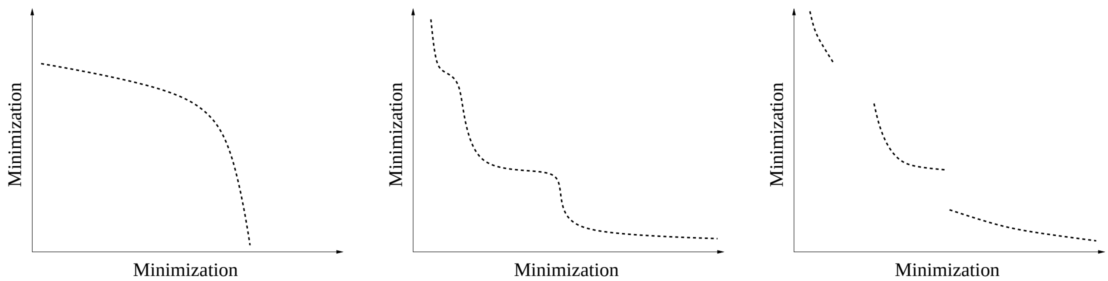
The set of *all* non-dominated solutions in the search space (i.e., $P = \mathcal{S}$) is denoted the *Pareto front* and may in principle be of infinite size. The goal in multiobjective optimization is to find a set of solutions as close to the true Pareto front as possible, and preferably a set with a good coverage of the true front. Figure 3.12(a) illustrates an example of dominance and non-dominance. The figure shows the solutions dominating a solution s, the solutions dominated by s, and the solutions neither dominating s or dominated by s.



(a) Dominance and non-dominance of s. (b) True Pareto front and solutions.

Figure 3.12: Dominance, non-dominance, and Pareto fronts in multiobjective optimization of two minimization objectives.

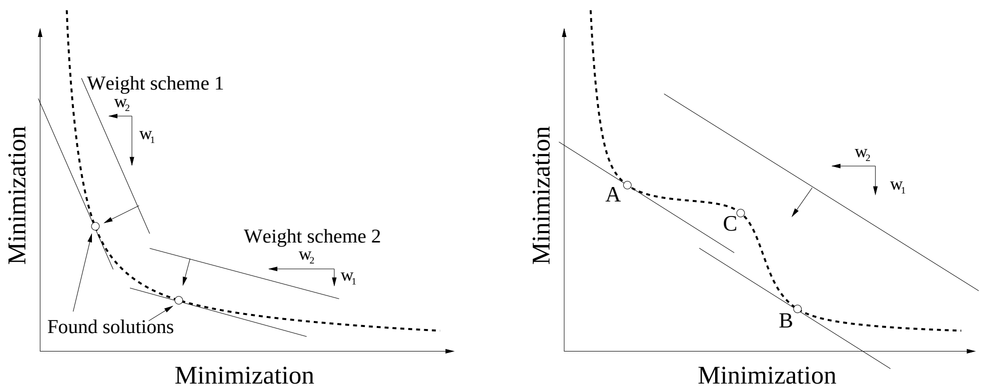
The Pareto front illustrated in figure 3.12(b) is an example of the simple convex type. In practice, Pareto fronts can be somewhat more subtle and create problems for simple multiobjective optimization algorithms. Figure 3.13 displays three different examples of Pareto fronts.



(a) Concave front. (b) Neither convex or concave front. (c) Discontinuous front.

Figure 3.13: Examples of Pareto fronts for minimization.

A simple approach to multiobjective optimization is to turn the multiobjective problem into a single-objective problem by a weighting scheme. The algorithm is then executed several times to obtain a number of points on the Pareto front. Using a weighting scheme corresponds to placing a line in the objective space and search for a solution on the front where the line is a tangent to the front. Figure 3.14(a) illustrates two weighting schemes, the corresponding lines and the points found on the front. The main disadvantage is that the approach fails on non-convex Pareto fronts. For example, the algorithm is unable to locate solution C on the front in figure 3.14(b), because A and B are more attractive in the optimization direction.



(a) Convex front. (b) Neither convex or concave front.

Figure 3.14: Multiobjective optimization with a weighting scheme.

Another simple approach is to turn all but one objective into constraints and then use a constraint technique to solve the problem. The idea is basically to define a feasible fitness range for the $m - 1$ objectives that are handled as constraints. The tradeoffs are then found by running the algorithm with different fitness ranges. This technique works for some of the more challenging Pareto fronts as well. However, the method requires extensive knowledge of the fitness ranges to get an even spread of the solutions on the Pareto front. Furthermore, it uses a significant number of runs to get a reasonable approximation of the front, which may not be a feasible approach if some of the objectives are computationally expensive. The idea of turning multiple objectives into constraints is also possible the other way around. The constraint violation (equation 3.7) may simply be treated as an additional minimization objective. This may indeed be relevant for problems with many soft constraints, because the set of solutions returned by the multiobjective optimization algorithm should give a good coverage of the Pareto front, which corresponds to a lot of tradeoffs between the soft constraints.

Multiobjective optimization in the EC-field have received increasing attention over the past ten years. A substantial number of algorithms and different techniques have been suggested, e.g., the Pareto Envelope-based Selection Algorithm PESA-II [33], the Elitist Non-dominated Sorting Genetic Algorithm NSGA-II [37], and the Strength Pareto Evolutionary Algorithm SPEA2 [156]. The main idea in most evolutionary multiobjective optimization approaches is to diversify the population by incorporating dominance and non-dominance ranking into the selection scheme. For an overview, see [36].

Multiobjective optimization techniques are relevant for system identification and control in various contexts. In system identification, the primary goal of finding a system model can be seen as the task of minimizing the error while maximizing the robustness of the model. Regarding control problems, multiple objectives also exist in a general context. The objective in many control applications is to minimize the response time and the overshoot of the controller. This is further discussed in chapter 6.

3.3.3 Dynamic components

Problems with dynamic components have the special property that the fitness function is not constant during the optimization. Hence, the algorithm must be able to both find *and* track the moving optimum. Dynamic problems are present in very diverse fields of application. For instance, in nurse scheduling the problem might change because employees get sick, medical equipment may break down, or the patient may not show up for an appointment. This requires rescheduling when new information is available. For system identification and control, time-varying problems arise for a number of reasons. On a short time-horizon, a change in temperature or humidity may affect the system, because many materials are sensitive to such changes. On a long time-horizon, effects such as wear out of ball bearings may result in a slow change in behavior. Furthermore, the control problem itself can be seen as a special kind of dynamic problem (see chapter 8). It is generally difficult to give a unifying definition of dynamic problems. The following definition is very general, but it covers all dynamic problems – even those that are only occasionally dynamic.

Definition 3.5: Dynamic problem
Optimize

$$f(s,t), \quad s \in \mathcal{S},\ t \in [t_0, t_{max}]$$

where t is the time when f is calculated, t_0 is the start time of the optimization, and t_{max} is the end time (possibly ∞). The optimization is performed under the assumption that

$$\exists?\ t, t'\ :\ t \neq t' \implies f(s,t) \neq f(s,t')$$

The notation $\exists?$ means "may exist".

$\triangle$

Note that the fitness in definition 3.5 is not a function of the time, which is typically the case for artificial benchmark problems. Furthermore, dynamic problems do not necessarily change over time. For instance, the nurse scheduling problem is static if no unforeseen events occur.

Dynamic problems have been investigated in the EC-community since about 1990. Most of this have been focused on developing various techniques for handling dynamic problems (for a survey see [20] or [21]). In this context, many of the studies have been carried out using rather "academic" time-varying problems. An example is the so-called moving peak problem.

Example 3.1: Moving peak problem
Maximize

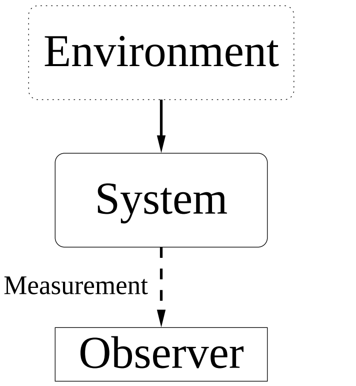
$$f(\mathbf{x}, t) = \sum_{i=1}^{n} (x_i - o_i(t))^2 \quad \mathbf{x} \in \mathcal{S},\ \mathcal{S} \subset \mathbb{R}^n$$

where $\mathbf{x} = [x_1, x_2, \ldots, x_n]$, $o_i(t)$ is the i'th entry in the current location of the moving parabola-shaped optimum $\mathbf{o}(t)$. The peak moves at given intervals according to some function. For instance, linearly along a vector with a new position every 50 generations.

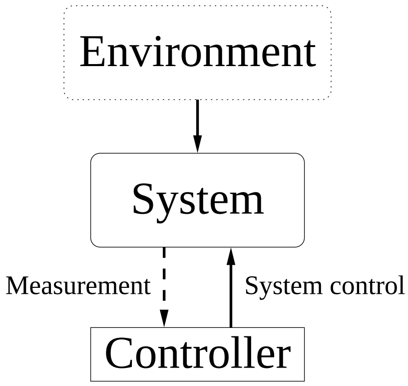
$\diamondsuit$

In 1999, several authors suggested new test-case generators (*TCGs*) for generalizing the moving peak problem ([19], [101], [134], and [63]). These TCGs are based on deterministic or stochastic updating of peak characteristics such as position, height, and width. Although the introduction of these TCGs was important, no research had been conducted to thoroughly evaluate how well they reflect characteristic dynamics of real-world problems. To investigate this, Thiemo Krink, Mikkel T. Jensen, Zbigniew Michalewicz, and I examined the relevance of the TCGs and found that they have limited relation to dynamic real-world problems [146]. The study focused on numerical dynamic problems, because these problems have the same search domain ($\mathbb{R}^n$) as the moving peak problems.

Generally speaking, many dynamic problems can be viewed as either observation or control problems. The main difference between these two types of classes is the feedback from the controller to the system (see figure 3.15).



(a) Observation problem. The environment influences the system; the observer does not affect the system.



(b) Control problem. The environment affects the system. The controller and the system interact; the controller affects the system.

Figure 3.15: Observation and control problems.

The objective in observation problems is either to predict the values of certain variables (dynamic system identification, see section 6.1.3) or to process sampled data (signal processing). The difference between these two subclasses is that dynamic system identification use observations from the past to predict the future, whereas signal processing focuses on continuously extracting information from recorded data (i.e., no prediction). Examples for dynamic system identification are weather forecasting, stock value prediction, and server failure prediction. Signal processing deals with tasks such as speech recognition and adaptive noise filtering.

In control problems, a controller has to operate a system and, in many cases, meet a certain output goal. The input for this process is provided by sensors that measure the state of the system and its environment. In other words, there is a feedback loop in which the controller affects the system variables that it uses as its own input in immediate future. If the EA is running while the system is being controlled it actually has the interesting consequence that *the search itself changes the fitness landscape*. This is further discussed in section 8.1.

A fundamental difference between the artificial moving peak problems and real-world problems is the focus on what generates the dynamics. In moving peak problems, the focus is on the fitness landscape and how this should be changed to make the test-problem challenging. In contrast, dynamics in real-world problems stem from the interactions among the system components. Hence, a shift of focus is necessary to get to the core of what makes dynamic problems challenging. A number of important questions arise when dynamic real-world problems are considered.

1. Do we know when the fitness changes?

2. How fast must we react to a change?

3. Does the search influence the future state of the problem?

Naturally, the answers to these questions depend on the problem at hand, but a few general issues should be considered. Question one circles the problem of resampling and detecting fitness changes. This is particularly important if the fitness calculation is expensive, because a change will require a complete reevaluation of the entire population. It may not be possible to decide exactly at what stage the fitness changes, but in most cases this can be detected somehow. For instance, a significant change in temperature may trigger a reevaluation. In other problems, the need for reevaluation is straightforward. For example, rescheduling of the nurses is obviously necessary if an employee gets sick. Question two deals with the problem of evaluation time. Obviously, an algorithm using five hours to find a solution that is required in only one hour is of little value. The reaction time is particularly important for control problems. For instance, an airplane controller must react in a few hundred milliseconds or less. Question three focus particularly on control problems. Here, the important issue is that control decisions influence the state of the problem and thus future fitness evaluations. In principle, this is relevant for any kind of problem where decisions are based on the found solution. In the nurse scheduling problem, the implemented schedule has some consequences for who is available in the case of a rescheduling event (see [71] for further information on scheduling, rescheduling, and robust scheduling with EAs). Question three has some very interesting consequences in a theoretical EA-context. In short, a search that influences future states of the problem may lead to multiple *optima in time*, because the decisions may drive the system state in different directions [143]. See chapter 8 for further information on the theoretical implications for control problems.

3.4 Summary

Defining the fitness function is by no means an easy task. As described, there is a significant number of issues to consider and pitfalls to avoid. Experience with studies on real-world problems reveal that a surprisingly large part of the time spent on a project is invested in defining the optimization problem. In this context, insight into both theoretical and practical issues play a key role in the success of the approach. Furthermore, special properties such as constraints, multiple objectives, and dynamic components pose additional challenges to the designers of the algorithm. To this end, a wide selection of evolutionary techniques exist. Consequently, some experience with the topic is usually an advantage to fully exploit the potential of evolutionary computation for such problems.

3.5 Future research

There are certainly many open issues regarding fitness function design. For plateaus, a significant amount of work has focused on examining the theoretical aspects of plateaus (neutral networks). However, only a few investigations have been published on techniques for searching problems with neutral networks. Regarding smoothness, the smooth operator genetic programming should be further investigated and tested on more problems. In connection with ridges, only a few algorithms exist although most real-world problems have correlated parameters. A possible explanation might be that a set of mainly uncorrelated benchmark problems are used to test novel algorithms. Regarding local optima, a significant amount of work has been done (see chapter 5). This is indeed a well-investigated area, but radical new ideas are still investigated and they often lead to simpler but more powerful algorithms. An open issue in a more theoretical context is the difficulty of defining what a hard multimodal problem is. Some metrics, perhaps from statistics, may help shed some light on these issues. In general, very few (if any) rules-of-thumb exist regarding how to approach a new multimodal problem.

Moving on to practical design issues, the problem of noise in connection with system-based fitness evaluation calls for further investigation. Again much research is performed using rather artificial noisy problems. In connection with the simulation-based fitness evaluation, further studies of the problems with inaccuracy in the adaptive Runge-Kutta-Fehlberg approach is needed. In this context, a survey in the computing field of numerical analysis may give some insight. Focusing on the problems of time-consuming evaluations, a significant amount of work can be done. This research area is rather new in an EC-context, but is becoming increasingly important as EC is moving into new fields of applications. Of particular interest is the area of industrial design, because such problems often involve computationally expensive fluid simulations. At EVALife, fitness approximation is currently being investigated by MSc student Kim Pedersen, who is also studying the application of smooth operator GP in this context.

Continuing with special properties, constraint problems received much attention in the nineties. Although the interest in the community has dropped recently, many issues are still rather uninvestigated. For instance, the role of subpopulation structures in constraint optimization. It may be an idea to have several subpopulations focusing on different parts of the feasible search space. Multiobjective optimization has more or less taken over the role of being the hot research topic. Nevertheless, many topics have not yet been studied. One direction that, to my knowledge, has received limited attention is to incorporate subpopulation approaches and other multimodal optimization techniques. I would expect this to be an advantage, since each of the fitness functions are typically multimodal. Finally, dynamic optimization has received considerable attention since 1990. Our study on the test-case generators for moving peak problems revealed the absent relationship between the used benchmark problems and real-world dynamic problems. Hence, the fundamental problem in most research on algorithms for dynamic problems until now is the missing experimentation on realistic problems. Undoubtably, many of the algorithms will work significantly better than simple techniques, but this is an open issue at the present stage.

Chapter 4

Methods for parameter control

The first observation when working with EAs is their sensitivity to the algorithmic parameters. For instance, the probability of mutation p_m can have quite an impact on the performance of the algorithm. Unfortunately, every part of an EA has parameters. Hence, there are parameters for the population, the representation, and the evolutionary operators. Parameter setting in EAs is usually a two-stage process; i) choosing the approach and ii) setting the parameters for the chosen approach. For instance, the population structure can be a single population, several subpopulations, or perhaps a two-dimensional grid structure with one individual per cell. The chosen population structure then has a number of parameters to decide upon. For example, the population size has to be set if a simple population structure is used.

Research in parameter control mainly focuses on the parameters for the evolutionary operators (mutation, crossover, and selection), and in this research the primary interest has been in techniques for setting the parameters. A fundamental problem in determining the parameters is that the best parameters depend on the problem. Hence, one set of parameters may yield good performance for some problems but less good performance on other problems. Furthermore, the optimal parameters are not constant during the run of the algorithm. For instance, a large mutation variance in Gaussian mutation may be advantageous in the beginning of the run, but it may hinder finetuning of the solutions in the population at the end of the run. From a theoretical point of view, the problem of determining parameters is implicit in Wolpert and Macready's so-called "No free lunch" theorem for search [153]. The theorem states that no search algorithm is superior on all problems. Consequently, no set of parameters is superior on all problems. Fortunately, the practical implications are rather limited, although it seems to be a very discouraging theorem. The statement *all problems* literally means <u>all</u> problems, including those with completely arbitrary relationship between solution and fitness value (static but random). Naturally, random search can search such problems as well as any other search algorithm.

The sensitivity of EAs to parameter values has lead to a wide variety of parameter control approaches. This makes it difficult to suggest a good taxonomy. Eiben et al. suggest a taxonomy based on *when* (before or during the run) the parameters are determined [41]. However, parameters are seldomly determined without performing a few test runs, which makes the criterion *when* inadequate for distinguishing between techniques. In my view, the key issue is whether or not the control method *adapts* to the search process. In non-adaptive control, the parameters are controlled without considering how the search progresses (e.g., if it stagnates). Conversely, adaptive control utilizes information from the search to tune the parameters while the search is performed. The criterion of adaptation suggests a further grouping of approaches into five sub-categories; two for non-adaptive control and three for adaptive control. Figure 4.1 illustrates the full taxonomy.

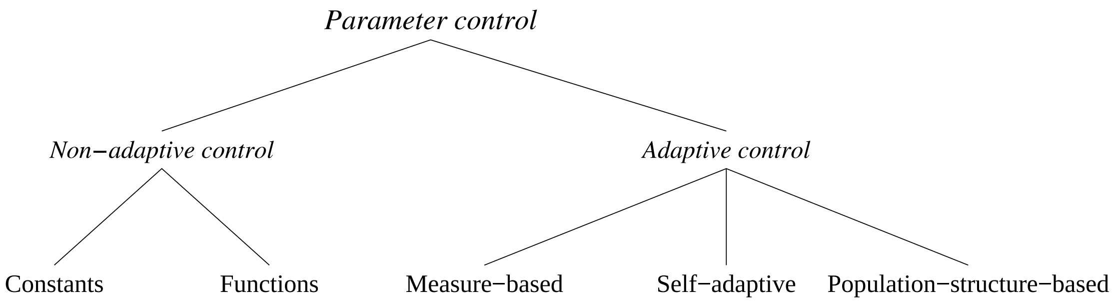


Figure 4.1: Taxonomy for parameter control methods.

In non-adaptive control, the parameter values may either be constant or calculated by a simple parameterized function of the generation counter. A common example is the annealing variance in Gaussian mutation. The main disadvantages of non-adaptive control techniques are that they require a significant amount of manual tuning and that the values are problem dependent. Adaptive control methods use information from the search process to alter the parameter values. A simple approach in this group is to control parameters by functions triggered by measurements on the population; for instance, increase the mutation rate if no improvement has been reported over the last k generations. Another idea used in so-called self-adaptive techniques is to encode the parameters in the individual's chromosome. Hence, this approach uses evolution for both searching the problem and setting the parameters. A third idea is to use the population structure and spatial position of individuals to determine their parameters.

Although adaptive control techniques are better, manual tuning is impossible to avoid completely, because nearly all components of the algorithms have parameters. Hence, all algorithms have at least a few parameters that need to be set to reasonable values. In addition, most advanced parameter control techniques also have parameters. Hence, the focus is shifted one layer up from tuning the actual parameters to tuning the parameter control methods. An obvious question to ask is then why even consider advanced techniques? The superior performance of parameter-varying algorithms is the primary motivation. Furthermore, tuning the control method is often easier than tuning the actual parameters, because the control technique is usually more robust with respect to parameter sensitivity.

4.1 Manual tuning of constants

The simplest possible parameter control approach is to keep values constant. Fortunately, fixing a large part of the EA-parameters will not decrease the performance of the algorithm significantly. The parameters commonly fixed are probability of crossover (p_c), probability of mutation (p_m), population size, and parameters related to selection, e.g., the tournament size.

Finding parameters usually requires a significant amount of experimentation for several reasons. First, it is necessary to repeat runs to get an impression of the performance obtained from a set of parameters, because EAs are stochastic algorithms. Second, parameters may be correlated and can therefore not be tuned independent of each other. This leads to a combinatorial explosion regarding the number of settings to try. For instance, trying five probabilities of mutation with five probabilities of crossover requires $25k$ runs to find the best combination (k is the number of repetitions). From an abstract point of view, manual tuning of parameters corresponds to searching "the universe of evolutionary algorithms".

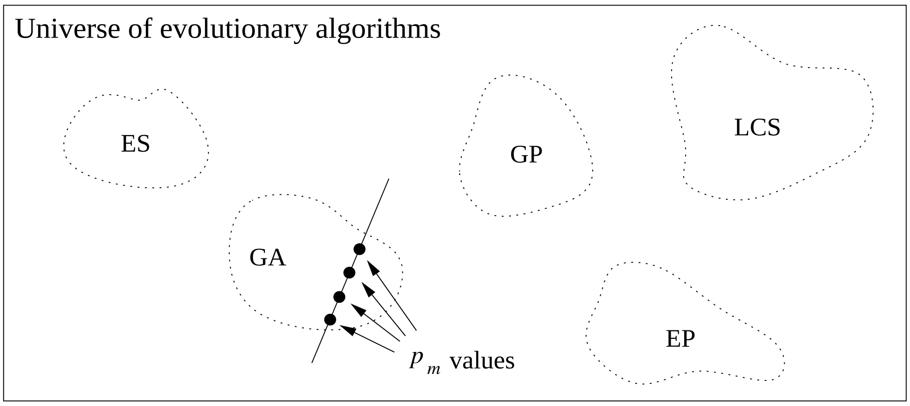


Figure 4.2: Searching the universe of evolutionary algorithms.

On the basis of countless experiments, a number of simple heuristics has been formed. Regarding binary encoding, a good starting point for bit-flip mutation is to use $p_m = 1/L$, where L is the string length. A mutation rate of $p_m = 1/L$ is commonly used as a lower bound, because $1/L$ corresponds to flipping one bit per genome on average. Furthermore, $p_m = 1/L$ has been proven optimal for the sphere problem [23], [8]. Over the years several constants for p_m have been suggested. De Jong found $p_m = 0.001$ to be appropriate on a number of simple benchmark problems [35], Grefenstette suggested $p_m = 0.01$ [62], and Schaffer et al. recommend a range of $p_m \in [0.005, 0.01]$ [120]. For real-value encoding, the probability of mutation is usually $p_m \in [0.6, 0.9]$ and probability of crossover $p_c \in [0.7, 1.0]$. In tournament selection, a tournament size of two often gives the best results. Setting it much higher imposes a too strong selection pressure with premature convergence as the overall result. The population size should typically be set to more than 50 individuals unless the evaluation is very time-consuming.

4.2 Manual tuning of functions

The search performed by an EA can be seen as a process shifting from highly explorative in the beginning of the run to fine-tuning towards the end. With this in mind, it is intuitively obvious that constant values for all parameters may lead to rather poor performance. For instance, a large variance in Gaussian mutation is usually advantageous in the explorative phase while small values are necessary to perform the fine-tuning at the end of the run. The straightforward approach is to replace each constant parameter p_x by a simple function $p_x(t)$ of the generation number t.

4.2.1 Mutation rate in bit-flip mutation

Varying the mutation rate p_m for bit-flip mutation has been investigated on several occasions. In a pioneering study, Fogarty investigated two ideas for controlling the mutation rate in bit-flip mutation [49]. In this study, Fogarty investigated a real-world problem where the objective was to determine seven control settings for an industrial burner. The problem was represented as a bit string of $L = 70$ bits encoding the seven parameters with ten bits each. One idea was to control the mutation rate $p_m(t)$ by an exponential decreasing function of the generation number t. Fogarty experimentally determined the following formula for how to set the mutation rate (see figure 4.3(a)).

$$p_m(t) = \frac{1}{240} + \frac{0.11375}{2^t} \tag{4.1}$$

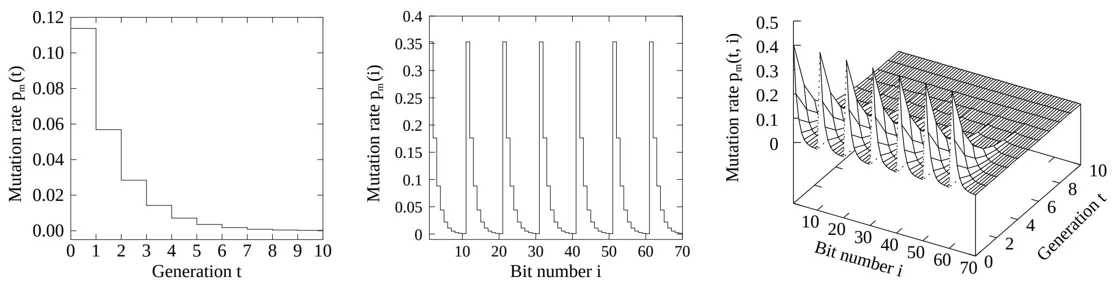
Additionally, Fogarty experimented with varying the mutation rate over bits in the chromosome. The idea is to have high mutation rates on the least significant bits and low mutation rates for the most significant bits. This approach gives better local search while allowing large mutation steps. In this particular setup, the formula for $p_m(i)$ was experimentally found to be:

$$p_m(i) = \frac{0.3528}{2^{i-1}} \tag{4.2}$$

where i is the bit number and $i = 1, 2, \ldots, 10$ with $i = 1$ being the least significant bit. The mutation rates for Fogarty's seven-dimensional real-world problem are illustrated in figure 4.3(b). Fogarty also experimented with a combination of the two ideas. In this case the mutation rates was controlled by:

$$p_m(t, i) = \frac{28}{1905 \cdot 2^{i-1}} + \frac{0.4026}{2^{t+i-1}} \tag{4.3}$$

Unfortunately, these formulas are not present in the original paper, but they are described in the *Handbook of Evolutionary Computation* [10, Part E1.2]. In his study, Fogarty compared these three schemes with a scheme using constant mutation rate of $p_m = 0.01$. Not surprisingly, the varying mutation rate approaches turned out to be the best, and the combination of varying across both generations and bit number was the best of all the four techniques.



(a) Generation $p_m(t)$. (b) Bit number $p_m(i)$. (c) Generation and bit number $p_m(t, i)$.

Figure 4.3: Varying mutation rate for generation $p_m(t)$, bit number $p_m(i)$, and combination of generation and bit number $p_m(t, i)$.

Following Fogarty's idea, Hesser and Männer theoretically derived an exponential function for the counting-ones problem [67]. They suggest a mutation rate of the following form.

$$p_m(t) = \sqrt{\frac{\alpha}{\beta}} \cdot \frac{\exp\left(\frac{-\gamma t}{2}\right)}{\lambda\sqrt{L}} \tag{4.4}$$

where α, β, γ are constants, λ is the population size, and L is the string length. In a comparative study, Bäck and Schütz investigated a linear decreasing function from 0.5 to $1/L$ in generation T [11]. After generation T, the mutation rate remains constant at $p_m(t) = 1/L$. Hence,

$$p_m(t) = \begin{cases} \left(2 + \frac{L-2}{T} \cdot t\right)^{-1} & 0 \le t \le T \\ \frac{1}{L} & T < t \end{cases} \tag{4.5}$$

Janikow and Michalewicz revisited Fogarty's idea of varying mutation rate over both bit number and generation. However, Janikow and Michalewicz perform the mutation slightly different from traditional bit-flip mutation. They suggest the *nonuniform mutation* operator for binary encoding [70]. The operator mutates the k'th binary encoded parameter x_k according to:

$$x'_k = \mathit{flip}(x_k, \nabla(t, L_k)) \tag{4.6}$$

$$\nabla(t, L_k) = \begin{cases} \lfloor \Delta(t, L_k) \rfloor & \text{if a random bit is 0} \\ \lceil \Delta(t, L_k) \rceil & \text{if a random bit is 1} \end{cases} \tag{4.7}$$

$$\Delta(t, L_k) = L_k \cdot \left(1 - r^{(1-\frac{t}{T})^b}\right) \tag{4.8}$$

where $r \sim U(0, 1)$, b is a slope parameter, T is the maximal number of generations, and L_k is the number of bits in x_k. The function $\Delta(t, L_k)$ determines the bit number to flip; however, this is a real value and has to be rounded by $\nabla(t, L_k)$. The function $\mathit{flip}(x_k, i)$ flips bit i of parameter x_k. This binary variant of the nonuniform mutation operator was derived from the version used for real encoding. Janikov and Michalewicz compared the two variants and several other approaches on an artificial control problem. For the nonuniform mutation operator, the real valued version outperformed the binary variant.

4.2.2 Variance in Gaussian mutation

Proper control of the variance in Gaussian mutation is a key issue in successful application of real-encoded EAs. In this context, setting the mutation variance according to a monotonic decreasing function depending on generation number is the standard approach. As outlined in section 2.2.1, Gaussian mutation of a real-encoded variable x_i is usually performed according to:

$$x_i' = x_i + N(0, \sigma_i(t)) \tag{4.9}$$

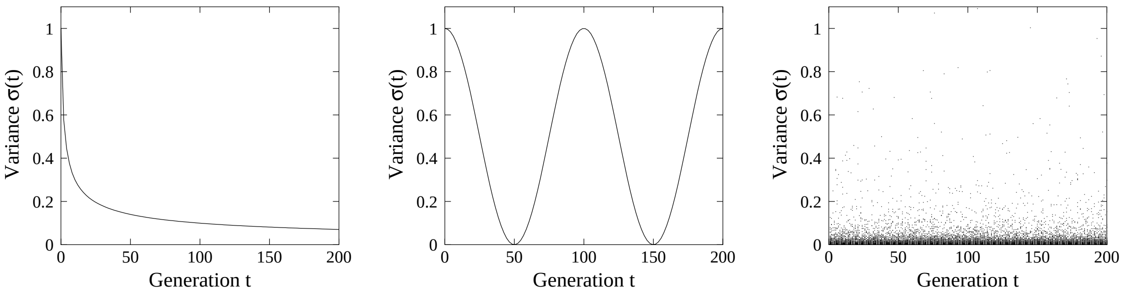
The traditional approach sets the mutation variance using either a linear or an exponentially decreasing function such as $\sigma_i(t) = 1/\sqrt{1+t}$ (see figure 2.8 and figure 4.4(a)). However, recent research revealed that significant improvement can be achieved by controlling the mutation variance by other techniques than a strictly decreasing function. Krink et al. used a so-called sandpile model[1] to generate power-law distributed numbers for controlling the variance in Gaussian mutation [86]. In the power-law distribution, the magnitude of an event has an inverse exponential relationship with its frequency. Hence, many small events occur and large events are rare. A real-world example is the relationship between size and frequency of earthquakes. The SOC mutation operator[2], suggested by Krink et al., scales the power-law distributed numbed according to the interval length of each parameter and use them as variance in Gaussian mutation. A fundamental difference between this approach and the annealing approach is that each individual is mutated using its own variance. Hence, the individuals are mutated using different power-law distributed variances. As a result, the algorithm explores the local neighborhood of a solution intensively *while* placing a few points far from it. For more information on SOC mutation, see [130], [131], and [140]. In a recent study, Brønsted and From investigated ten approaches to variance control [24]. The ten approaches were grouped in three categories with 3-4 functions in each. The categories are: deterministic annealing methods (the classic approach), deterministic non-monotonic methods, and stochastic methods (see figure 4.4 for examples). Brønsted and From concluded that the SOC approach gave the best results, although some tuning of the parameters was required. They also found the traditional deterministic $1/\sqrt{1+t}$ annealing approach to be nearly as good.

4.3 Measure-based control

In measure based control, the algorithm updates the parameter values using measurements of various features of the search process. Hence, the control approach constantly adapts the parameter values to the search and the status of the population. The control functions are usually just simple if-then rules based on measures on the population. A simple approach may be to count the number of generations without fitness improvement and then increase the mutation variance temporarily

[1]Bak suggests the sandpile model as a simple approach to study many complex phenomena found in nature [14]. The sandpile model is an example of how self-organized criticality (SOC) can be generated by very simple means.

[2]Named after Bak's work on self-organized criticality.



(a) Deterministic annealing $\sigma(t)$.

(b) Deterministic non-monotonic $\sigma(t)$.

(c) Stochastic SOC controlled $\sigma(t)$ with 100 individuals per generation.

Figure 4.4: Three approaches to variance control in Gaussian mutation.

if stagnation occurs for more than a given number of generations. Figure 4.5 illustrates a simple example using a stagnation counter to trigger a hypermutation event, in which all individuals are mutated with large variance.

Stagnation triggered hypermutation

if($fit(s_{best}(t-1))$ `==` $fit(s_{best}(t))$) `stagcount++` **else** `stagcount`=0
if(`stagcount`>`stagmax`) {
  Mutate all individuals with large variance.
  `stagcount`=0
}

Figure 4.5: Example of measure based control. $s_{best}(t)$ is the best individual at generation t, `stagmax` is the maximal number of generations before a hypermutation event occurs.

Naturally, measures can be defined in several ways; however, two main types exist. Measures either record features over a number of generations (generational measures) or calculate a value on the basis of the current population (instantaneous measures). Most generational measures either record the fitness improvement or the success rate of operators. Typical examples are:

- Stagnation counting. The number of generations with no improvement in fitness.
- Fitness velocity. The rate of fitness improvement over an number of generations.
- Success rate of an operator. The ratio between number of better individuals and total individuals produced by a given operator.

Instantaneous measures often express the degree of population diversity. Most diversity measures calculate the value using the genes in the population, because it is rather easy to define a measure on a numeric encoding such as real-valued vectors. However, it can be quite difficult to define a proper measure for more complex encodings such as parse trees. Instead, the diversity can be calculated using the fitness values of the individuals. In this, the underlying hypothesis is that similar fitness values correspond to some degree of genetic similarity, which is of course not generally true but acceptable in practice. The main advantage of this approach is the time it takes to calculate the diversity. In summary, the main ideas in instantaneous measures are:

- Genetic diversity. A similarity measure of the individuals' genomes. For instance, the average Hamming distance between the bitstring genome of each individual and the genomes of all other individuals in the population.

- Fitness diversity. A similarity measure of the individuals' fitnesses. For example, the statistical variance of the fitnesses.

- Fitness ratios. A ratio between best fitness and average fitness or worst fitness and average fitness.

Regarding genetic diversity, problems with numeric search domains allow several interesting approaches. The most commonly used are:

- All-to-all diversity. The average distance between each individual's point in $\mathbb{R}^n$ and all other points in the population.

- All-to-midpoint diversity. The average distance between each individual's point and a population midpoint. The midpoint is calculated as the population's average point in $\mathbb{R}^n$.

- All-to-best diversity. The average distance between each individual's point and the point of the best individual.

- Nearest-neighbor diversity. The average distance between each individual's point and the point of the nearest neighbor.

Typically, measure-based techniques employ a combination of both generational and instantaneous measures. The main differences between approaches are in how the rules are obtained and the number of controlled parameters. Rules are either programmed explicitly, evolved with an EA, or found by another learning approach.

4.3.1 Preprogrammed rules

Preprogrammed control rules express a heuristic discovered through extensive experimentation. The heuristic is usually discovered by scrutinizing the experimental data and forming a hypothesis from the relationship between performance of the algorithm and how the parameters were changed. One of the first preprogrammed control rules was Rechenberg's 1/5-rule for Gaussian mutation in (1+1)-Evolution Strategies [113]. The rule states that the ratio of successful mutations to all mutations should be 1/5 measured over a number of generations. The mutation variance should increase if the ratio is above 1/5, decrease if it is below, and remain constant if it is 1/5. The variance is updated every N generations according to:

if (t **mod** $N = 0$)

$$\sigma(t) = \begin{cases} \sigma(t-n)/c & p_s > 1/5 \\ \sigma(t-n)\cdot c & p_s > 1/5 \\ \sigma(t-n) & p_s = 1/5 \end{cases}$$

else

$$\sigma(t) = \sigma(t-1)$$

where $0.817 \leq c \leq 1.0$ [9]. The lower bound $c = 0.817$ was theoretically derived by Schewfel for the Sphere problem [122]. Setting $c > 1.0$ will reverse the effect of the control rule and be in conflict with the underlying hypothesis that a non-optimal solution can always be improved by using a sufficiently small step-size.

4.3.2 Evolved and adaptive rules

The rule discovery process can be automated by various machine learning techniques. In a pioneering study, Lee and Takagi evolved fuzzy control rules [92]. The fuzzy system used the two ratios (average fitness)/(best fitness) and (average fitness)/(worst fitness) in combination with the fitness velocity to control the change in population size, crossover rate, and mutation rate. Lee and Takagi evolved the fuzzy rules using simple artificial benchmark problems. To test the robustness of their approach, the fuzzy rules were then used to set the parameters of an EA evolving a controller for a pole-balancing problem. The fuzzy parameter control approach showed better so-called online performance but similar offline performance in comparison with a simple EA (for details, see [92]). The experiment indicates that the learned fuzzy rules may be generally applicable. In contrast, the approach requires a significant investment in programming, because a fuzzy inference engine needs to be implemented before the technique can be used in practice.

In a recent study, Kee et al. used a combination of one generational measure and two instantaneous measures to set three parameters of the algorithm [79]. In this study, the measures are called the *state vector* and the controlled parameters are the *control vector*. The state vector consist of the fitness velocity (generational), the fitness variance (instantaneous), and the population diversity (instantaneous), which is measured by Hamming distance. The control vector sets the probability of mutation p_m, the probability of crossover p_c, and the so-called power fitness scaling factor α. Their approach is somewhat simpler compared with the fuzzy control approach suggested by Lee and Takagi, because it does not require the implementation of a fuzzy inference engine. Kee et al. use a simple rule system to map the current state vector to a control vector. They partition the state values into three ranges, low, medium, and high, which gives a system with $3^3 = 27$ rules. The control values are set to either a low, a medium, or a high value. Hence, each rule maps a state three-tuple to a corresponding control three-tuple. For instance, (low, high, high) may be mapped to (medium, low, low). The rule learning is carried out while the EA is optimizing the problem.

4.4 Self-adaptive control

The idea in self-adaptive control is to encode algorithmic parameters in the genome. In this setup, the genome consists of a solution s to the problem and an additional set of control parameters p. Most self-adaptive algorithms first apply the algorithmic parameters p on themselves to obtain the new parameters p'. The new parameters p' are then used to create the solution s' from s. The underlying hypothesis in this scheme is that good solutions carry good parameters; hence, evolution discovers good parameters *while* solving the problem.

Some of the earliest studies on self-adaptation were performed by Reed et al. [115] and Bagley [13]. The idea was later refined by Schwefel in the context of numerical optimization with evolution strategies (ES) [122]. Self-adaptation has now become a fixed ingredient in ES and real-valued evolutionary programming.

4.4.1 Gaussian mutation in evolution strategies

Controlling the variance of the Gaussian mutation operator has been the primary objective with self-adaptive techniques. In 1977, Schwefel suggested three approaches to self-adaptation for numerical optimization with ES [122]. Gaussian mutation is the main operator for creating new individuals in ES. In this context, ES use self-adaptation extensively to control the mutation operator's variances. Simple ES only encode one variance σ, which is used to mutate all problem variables. Hence, the chromosome consist of a pair $(x,\ \sigma)$. The mutation of an individual is then performed in two steps:

$$\sigma' = \sigma \exp\left(\tau_0 N(0,1)\right) \tag{4.10}$$

$$x_i' = x_i + \sigma' N_i(0,1) \tag{4.11}$$

where τ_0 controls the adaptation rate. In this formula, the normal distribution $N_i(0,1)$ is sampled for each x_i. Figure 4.6(a) illustrates the level curves with equal probability in mutation with one σ. A slightly more advanced approach encodes one variance σ_i for each variable x_i (figure 4.6(b)). In this case, the mutation formula is:

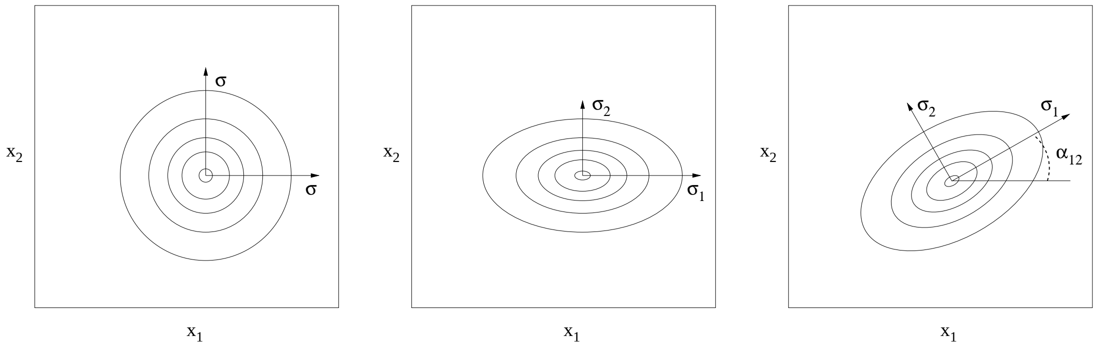
$$\sigma_i' = \sigma_i \exp\left(\tau' N(0,1) + \tau N_i(0,1)\right) \tag{4.12}$$

$$x_i' = x_i + \sigma' N(0,1) \tag{4.13}$$

where τ' determines the overall adaptation rate and τ control the adaptation rate for the variable dependent sampling $N_i(0,1)$. A third variant aims at handling correlation between problem variables. In this approach, the chromosome encodes both variances σ_i and an additional set of rotation angles α_{ij}, which express the correlation between variable x_i and x_j. In this operator, the mutation is performed in three steps according to:

$$\begin{aligned} \sigma'_i &= \sigma_i \exp\left(\tau' N(0,1) + \tau N_i(0,1)\right) && (4.14)\\ \alpha'_{ij} &= \alpha_{ij} + \beta N_{ij}(0,1) && (4.15)\\ x' &= x + \mathbf{N}(\mathbf{0}, \mathbf{C}(\sigma', \alpha')) && (4.16) \end{aligned}$$

where τ' and τ are as before, β controls the adaptation rate of the rotation angles ($\beta = 0.0873 \approx 5°$ is recommended [12, Section 6.4]), and $\mathbf{N}(\mathbf{0}, \mathbf{C}(\sigma', \alpha'))$ denotes the correlation mutation vector. Figure 4.6(c) displays the case with two problem variables, two variances, and one rotation angle for the correlation between the problem variables. For further information on the self-adaptive mutation operator in ES, see [12, Section 6.4].



(a) Self-adaptation of one variance σ.

(b) Self-adaptation of all variances σ_i.

(c) Self-adaptation of variances σ_i and rotation angles α_{ij}.

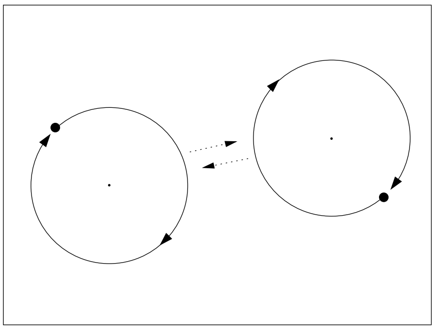
Figure 4.6: Variants of self-adaptive mutation operator for evolution strategies. The boxes denote the search space. Ellipses are level-curves with equal mutation probability density.

4.4.2 Self-adaptation on dynamic problems

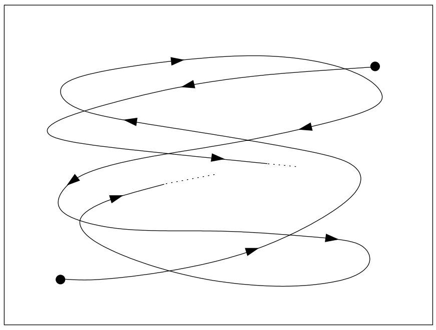
A disadvantage of self-adaptation is the long period it takes for the algorithm to discover the superior parameters. In EA-terms, the selection pressure for good parameters is too low to quickly adapt the parameters to the problem, because suboptimal parameters will yield acceptable solutions. In addition, the optimal parameters may change as the search progresses. Hence, the parameters controlled by a self-adaptive EA will often reflect the state of the search process as it was several generations earlier. To examine this effect, I performed two simple experiments[3] on self-adapting the mutation variance σ [138]. In the study, two simple moving peak problems were implemented using a test-case generator for artificial dynamic problems. The first problem consists of two peaks moving in circles while they slowly exchange position (figure 4.7(a)). The peaks in this problem move with a constant velocity. Hence, the self-adaptive EA may be able to discover the

[3]The experiments were carried out using my Multinational EA [137]. For further information on the algorithm, see section 5.4.3.

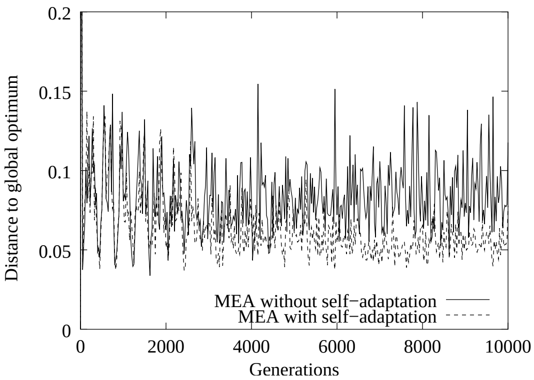
optimal mutation variance. In the second problem, the peaks move with varying velocity (figure 4.7(b)). This makes it impossible for the self-adaptive EA to discover the optimal mutation variance, because the encoded variance will not reflect the current speed of the peaks. Figure 4.7(c) and 4.7(d) illustrate the average distance to the global optimum for the two problems. For the first problem (constant velocity), the self-adaptive EA discovers a good variance, although it takes about 5000 generations before it shows on the graph. On the second problem (variable velocity), the self-adaptive EA fails in obtaining a variance yielding a better performance than the non-adaptive EA. This is most likely because the varying peak velocities make the adaptation impossible.



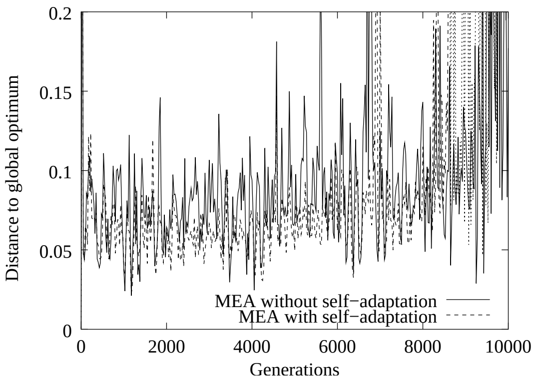
(a) Circular moving peaks problem.



(b) Sinusoidal moving peaks problem.



(c) Distance to global optimum on the circular moving peaks problem.

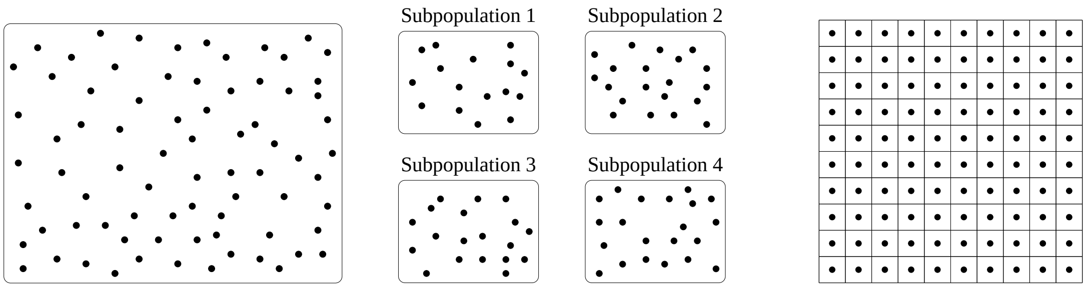


(d) Distance to global optimum on the sinusoidal moving peaks problem.

Figure 4.7: Self-adaptation on dynamic problems.

4.5 Population-structure-based control

The most recent idea in parameter control is to use the population structure to set the parameters of the algorithm. In this, two main approaches exist. One idea is to divide the entire population into a number of disjoint subpopulations and use the success of each subpopulation to control the parameters of other subpopulations. Another idea is to use a spatial population structure and map the position of each individual to a corresponding set of parameter values. For comparison, figure 4.8 illustrates a standard population, a scheme with subpopulations, and a spatial population structure.



(a) Single population. (b) Subpopulations. (c) Spatial grid population.

Figure 4.8: Typical population structures in EAs.

4.5.1 Subpopulation-based control

In a study from 1996, Lis suggested to use a subpopulation-based approach to control the probability of mutation p_m in a binary encoded GA [94]. In this algorithm, each subpopulation has its own level of mutation probability. The subpopulations are assigned to consecutive levels from a predefined scheme of mutation probabilities (see table 4.1).

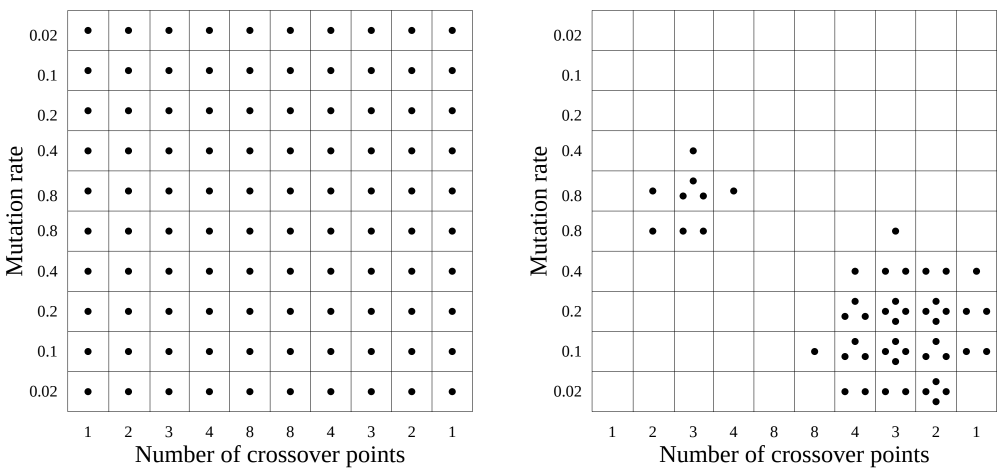
Level	0	1	2	3	4	5
p_m	0.0001	0.0002	0.0005	0.001	0.002	0.005
Level	6	7	8	9	10	11
p_m	0.01	0.02	0.05	0.1	0.2	0.5

Table 4.1: Mutation probability levels.

The algorithm was originally designed for a parallel computer, but can easily be executed on a single-processor architecture. In the parallel version, a main processor is responsible for the coordination and the distribution of subpopulations to the other processors. The main processor first creates a random initial population, which is then copied and distributed to each processor along with the processor's mutation level. The processors then optimize their populations for a given number of generations and send the best individual to the main processor. The best individual in all subpopulations is found and its corresponding mutation level is checked. The mutation level of all processors is increased by one if the best individual's level is the highest among the currently used levels, decreased by one if the best individual's level is the lowest used level, and unchanged if the best individual is found by one of the processors having an inbetween mutation level. Hence, the algorithm constantly adapts the window of mutation probabilities towards the most successful probability. After adjusting the levels, the main processor creates a new population and the process is repeated for a predefined number of times (epochs). Lis experimented with a scheme having 12 levels and four subpopulations. Several starting configurations for the mutation levels were tested to see if there was a trend in how the levels evolved over time. To examine this, the level of the population producing the best individual was recorded and plotted as it developed over the epochs. Interestingly, the development of the mutation level had similar behavior as the traditional annealing of mutation rates.

4.5.2 Spatial control

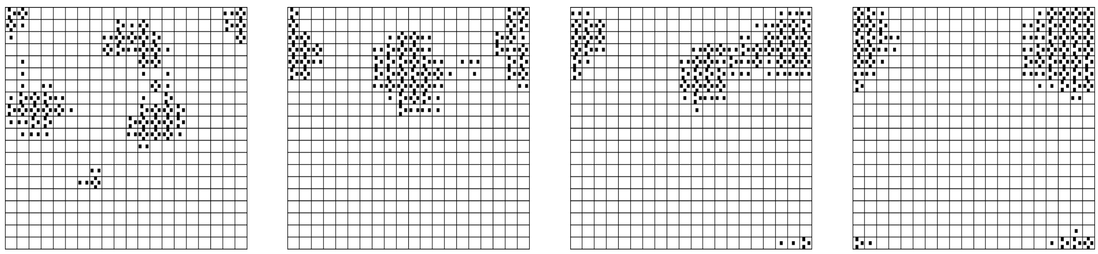
The main idea in spatial control is to represent a large part of the parameter space by a spatial population structure and interpret the location of the individual as a set of parameters. A recent idea in this context is the so-called Terrain-Based Genetic Algorithm (TBGA) introduced by Gordon et al. [59]. The TBGA uses a spatial population structure (figure 4.8(c)) where each cell hold and individual with a binary encoded genome. In TBGAs, the two-dimensional grid position of an individual is interpreted as its offspring's mutation rate and number of crossover points. Thus, the individuals of a TBGA population apply the entire variety of different parameter combinations at each time step. The advantage of this approach is that parameters can be exploited that are optimal for the type of optimization task and the current state of the optimization process. However, since the individuals in the cellular EA are fixed at their grid positions, only a few of them are able to take advantage of this set-up simultaneously. One way to tackle this problem is to use a multi-agent system where agents can move between grid cells and more than one agent can be at the same grid location. In this context, Thiemo Krink and I investigated a Terrain-Based Patchwork Model (TBPM) and compared the performance of this agent based approach with the TBGA [87]. The patchwork model was introduced by Krink et al. as a general agent framework with a spatial grid world [84]. In the Patchwork model, individuals are considered as autonomous mobile agents that choose their actions based on "desires", which are modeled by so-called motivation networks. At each step, the motivation network calculates the desire for performing each action (move, mate, etc.) based on sensors sensing the state of the local environment and the agent's internal state. For more information on the patchwork model, see section 5.2.2, [84], and [87]. Figure 4.9 illustrates an example of how the agent's spatial location is interpretated in the TBGA and the TBPM. For the TBPM, figure 4.9(b) shows an example of flocking near the parameters $(3, 0.2)$ and $(3, 0.8)$.



(a) The TBGA. (b) The TBPM.

Figure 4.9: The population structure of the Terrain-Based Genetic Algorithm (TBGA) and the Terrain-Based Patchwork Model (TBPM).

In the comparison, Krink and Ursem used a 20×20 grid world with connected borders to form a torus-shaped world. For the TBPM, the motivation network was simplified and set up to make an agent move toward the neighboring agent with the highest fitness. Interestingly, the agents quickly flocked around the parameters leading to better performance. Figure 4.10 illustrates a typical run with 200 agents. The algorithm was initialized with a maximum of one agent per patch, i.e., the start position of the agents covered many different parameter settings. After a couple of generations the agents formed small clusters, which then merged to larger clusters and finally into one single cluster. Between generation 10 and 25, the four small clusters merged to two larger clusters (figure 4.10(a) and 4.10(c)). Figure 4.10(c) shows the situation where the center cluster began to merge with the cluster located in the corners[4], which was completed in generation 60. Regarding optimization performance, the TBPM outperformed the TBGA on two of five test problems and had matching performance on the remaining three.



(a) Generation 10. (b) Generation 25. (c) Generation 40. (d) Generation 60.

Figure 4.10: Example of flocking behavior in the Terrain-based Patchwork model. The grid was connected at the borders to form a torus-shaped world.

4.6 Summary

Setting parameters is indeed the Achilles heal of evolutionary computation and a significant amount of work has been carried out to deal with this problem. Fortunately, EAs offer great flexibility and they partially contain the solution to their own problem, because the algorithms can easily be extended to adapt to the problem. In this connection, there are a number of interesting directions. From a scientific point of view, the measure-based approaches are of particular relevance, because they express comprehensible rules for how to control parameters. However, such rules may not be generally applicable since they can be problem dependent. In contrast, approaches encoding parameters in the genome are more general, because the parameters are determined with respect to the problem at hand. Unfortunately, these algorithms have a rather long adaptation horizon, which makes them inadequate for time-varying problems and problems where only few evaluations are possible. One solutions to this problem may be to use a population-structure-based approach, because such methods appear to have rather quick convergence time with respect to the parameters.

[4]The world was defined as a torus so even though it looks like as if the population is scattered over four corner clusters it is in fact a single cluster.

4.7 Future research

Regarding future work on setting algorithmic parameters, the main challenge lies in the analysis of control techniques. So far, a significant number of methods have been suggested. These methods appear to improve performance and are therefore of great value to the field. However, an indepth understanding of the underlying mechanism is, in my view, still missing. In this context, it should be mentioned that some theoretical work has been performed, but this has mainly been focused on rather idealized problems, and the relevance of these results to realistic problems is yet to be determined. One approach to generalizing parameter control rules may be to apply datamining techniques to traces of the optimization process. For instance, logging measures as those described in section 4.3 along with parameters may be used to learn easily comprehensible control rules such as Rechenberg's 1/5 rule for mutation in evolution strategies.

Chapter 5

Techniques for multimodal optimization

The main optimization challenge in many real-world problems is caused by the topology of the fitness landscape, in particular, its ruggedness in terms of local optima. Evolutionary algorithms (and all other iterative optimization algorithms) must be able to avoid and escape local optima to find the optimal or near-optimal solutions for such multimodal problems. EAs for multimodal problems have been investigated since the early days of evolutionary computation, and numerous algorithms and extensions have been suggested to improve their performance on such problems. The main objective in research on multimodal optimization has been to deal with the problem of premature convergence, i.e., stagnation in a local optimum. Moreover, the algorithms have been developed with two alternative objectives in mind. One goal has been to improve the techniques to search for a global optimum; for instance, by keeping track of local optima and use this information to guide the search towards unexplored areas of the search space. Another goal has been to develop algorithms capable of locating multiple good solutions. This is an advantage when the solutions need to be assessed by a human expert before one of them can be used in reality. For example, additional criteria may have to be considered by the human expert before the final solution is put in use. For this purpose, alternative and different solutions are necessary to give the expert the best set of candidate solutions for the final decision. An immediate approach in this context is to re-run a simple EA a number of times and then hope that the population converges to different peaks in every run. However, this is rarely the case in practice, because local optima with a fitness slightly lower than the fitness of the global optimum may be difficult to discover because the global optimum is more attractive. An algorithm that keeps track of multiple optima simultaneously can use this information to spread out the search and thereby discover different optima in one run.

The central concept in most research on EAs for multimodal optimization has been to maintain the genetic diversity of the algorithm's population. Genetic diversity is important because a diverse population allows the algorithm to better exploit the crossover operator when creating new solutions. Crossover on individuals from a fully converged population has no effect, because recombining identical individuals will not generate any new solutions. In such a scenario, the algorithm is fully dependent on the mutation operator to escape the local optimum and further explore the search space. Experience shows that this is extremely difficult in practice, because selection favors the solutions trapped in the local optimum. Hence, full convergence should be avoided by maintaining a sufficient high diversity. A naïve conclusion in this context is that an algorithm would perform well by keeping the highest possible degree of diversity. However, a too high diversity makes the crossover operator less efficient for fine-tuning the solutions, which is important at the end of the run. Recombining very different solutions will often not produce a well-fit solution, because the parents are probably approaching different peaks. Hence, the offspring is likely to be placed somewhere between the two peaks. Conclusively, the optimal level of diversity is somewhere between fully converged and highly diverse. In this context, diversity measures have mainly been used to *analyze* algorithms and reason about their diversity maintaining capabilities. Recently, diversity measures have been used to *control* algorithms, which lead to significant improvements in performance.

Over the years, several hundred algorithms have been suggested. Most of these algorithms combine ideas from a rather small set of approaches. In general, algorithms either avoid convergence or attempt to repair a converged population. The majority of algorithms use the avoid strategy, which can be further refined into strategies attempting to slow down the genetic convergence and strategies trying to prevent overlap of solutions in the search space. This grouping suggests a taxonomy for multimodal optimization techniques (see figure 5.1).

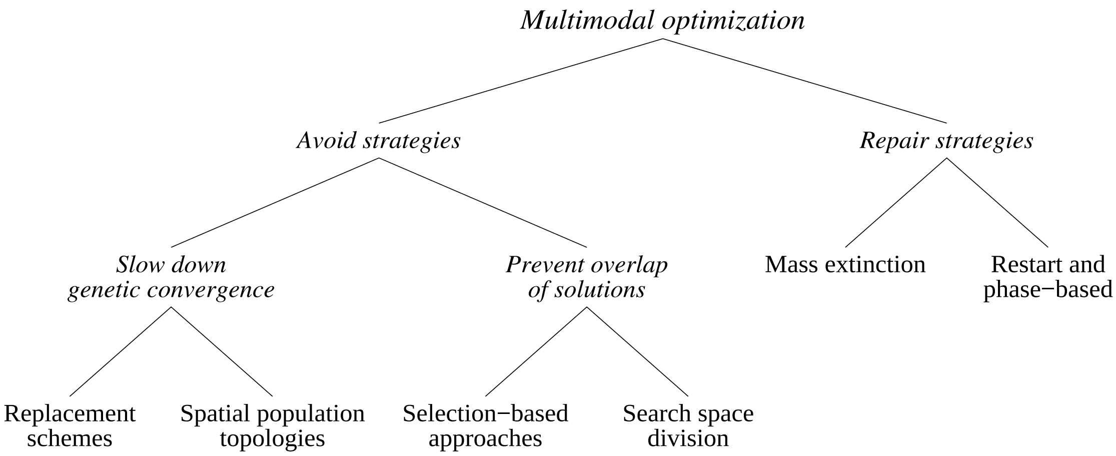


Figure 5.1: Taxonomy for multimodal optimization techniques.

In avoid strategies, the main idea is to prevent premature convergence to a local optimum. The algorithms attempting to slow down genetic convergence aim at maintaining the population's diversity for a longer period and thereby avoid stagnation in a local optimum. Algorithms in this category either use a replacement scheme for updating the population or try to reduce the spread of genes by introducing a spatial population topology. The strategies trying to prevent overlap of solutions either maintain diversity through selection and alterations of the fitness function, or by actively dividing the search space into subspaces. In repair strategies, algorithms either maintain diversity by mass extinction techniques or by introducing new genetic material when population convergence is detected.

The following sections describe these ideas and gives a survey of the more known algorithms in each category.

5.1 Replacement schemes

Evolutionary algorithms using replacement schemes differ from ordinary EAs by not having a classic selection round. Traditional EAs do not make a distinction between new and old individuals. In a replacement scheme, each newly created individual is checked to see if it should replace another individual in the population. Hence, the replacement scheme acts as a kind of selection mechanism that examines each individuals and decides if it should survive. The challenging part in replacement schemes is to choose the best suited individual to replace.

5.1.1 Crowding

Crowding was introduced by De Jong in 1975, and it was one of the first attempts to deal with multimodal optimization problems [35]. De Jong's crowding algorithm creates a certain number G of offspring in every generation. For each offspring o the algorithm selects a small subset C of the population and finds the most similar individual s in C. The offspring o replaces s in the population if o is better than s. De Jong used binary encoding and the Hamming distance to determine the similarity of two genomes. The size of the subset is called the crowding factor CF, while the parameter G is called the generation gap. De Jong reported good results with $CF = 2$ and $CF = 3$.

The drawback of crowding is the risk of replacement errors. The new individual might accidently replace a good solution from another peak, if the small subset of individuals used in the comparison does not contain a solution from a peak in vicinity of the new individual.

5.1.2 Deterministic and probabilistic crowding

Mahfoud examined De Jong's crowding and suggested the *deterministic crowding* algorithm [97]. Deterministic crowding was developed with three objectives in mind. First, to eliminate the need for the crowding factor CF and the generation gap G, which are highly problem dependent parameters. Second, to minimize the number of replacement errors, otherwise, as Mahfoud surmised, the performance of the algorithm would be more sensitive to probabilistic artifacts. Third, to increase selection pressure between individuals that approach the same peak, but favor convergence to multiple peaks.

The idea in deterministic crowding is to calculate the similarity between the genomes of the parents and the offspring. An offspring replaces the most similar parent if it has a better fitness than the parent. This lowers the replacement errors when the population has settled on multiple peaks, because the offspring will be more similar to its parents than to other individuals in the population. The pseudocode for deterministic crowding is listed in figure 5.2.

```
Deterministic Crowding
  Initialize and evaluate population P(0)
  while (not <termination condition>) {
    Permute the population array P(t)
    for (i=0; i<|P|/2; i++) {
      j=i + |P|/2
      Create two offspring o1 and o2 from P(t)[i] and P(t)[j]
      if(d(o1,P(t)[i])+d(o2,P(t)[j])<d(o2,P(t)[i])+d(o1,P(t)[j])) {
        if(Fit(o1) > Fit(P(t)[i])) then P(t)[i] = o1
        if(Fit(o2) > Fit(P(t)[j])) then P(t)[j] = o2
      }
      else {
        if(Fit(o2) > Fit(P(t)[i])) then P(t)[i] = o2
        if(Fit(o1) > Fit(P(t)[j])) then P(t)[j] = o1
      }
    }
    Mutate P
    Evaluate P
  }
```

Figure 5.2: Pseudocode for deterministic crowding. The population size $|P|$ must be an even number, d is a distance measure, and $P(t)$`[i]` *is the* `i`*'th individual of the population P at generation t.*

Mahfoud reported good results on two very simple one-dimensional functions introduced by Goldberg and Richardson [58] (see section 5.3.1). Deterministic crowding was able to maintain individuals near all five peaks on both test problems.

A variant of deterministic crowding was introduced by Mengshoel and Goldberg [98]. The idea in *probabilistic crowding* is to use a stochastic replacement rule instead of the deterministic rule shown in figure 5.2. The probability of replacing the parent is calculated from the fitness of the offspring and the parent (equation 5.1).

$$p(I_{offspring}, I_{parent}) = \frac{Fit(I_{offspring})}{Fit(I_{offspring}) + Fit(I_{parent})} \tag{5.1}$$

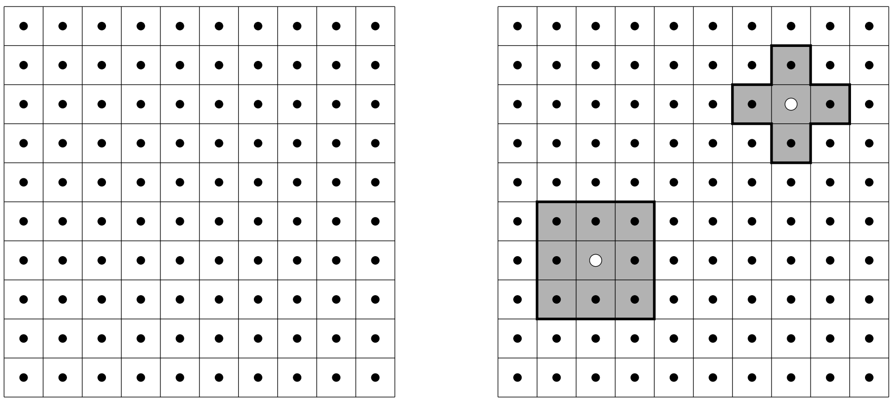
The main disadvantage of probabilistic crowding is its sensitivity to the values of the fitness function. Hence, the selection pressure depends on the problem, which is generally not desirable because it makes the algorithm highly problem dependent.

5.2 Spatial population topologies

Spatial population topologies were originally introduced to exploit the power of parallel computers with the hypercube architecture (see section 3.2.3). The physical design of such computers introduced the idea of local crossover and selection operations. In contrast, crossover and selection in simple evolutionary algorithms induce a global interaction range between individuals. For instance, the participants in tournament selection are picked randomly from the entire population. This population structure allows a high genetic flow throughout the population, because any individual can interact with any other individual. In this scenario, a good, but suboptimal, solution can quickly take over the entire population, since it is competing with all other solutions in the population and not just a small subset of them. A spatial structure with local interactions improves the algorithm's ability to avoid premature convergence, because of the slower diffusion of genetic material. Consequently, spatial models are now in use on single CPU machines.

5.2.1 Cellular EA

The population structure of the cellular EA (also known as the diffusion model) is closely related to the two-dimensional lattice used in cellular automata algorithms [10, C6.4] (see figure 5.3(a)). In the cellular EA, each cell holds an individual that can only interact with its immediate neighbors, which is usually either four or eight individuals (see figure 5.3(b)). The borders of the grid are usually connected to form a torus shaped world. Limiting the interaction range of an individual to its immediate neighbors makes it impossible for a solution to quickly take over the entire population, simply because it takes several generations to diffuse[1] the genes throughout the grid.



(a) Population structure of a 10×10 cellular EA.

(b) Neighborhoods with four or eight individuals.

Figure 5.3: Population structure and interaction range of the Cellular EA. Each circle represents an individual.

The effect of interaction range and neighborhood shape was investigated by Sarma and De Jong in two theoretical studies from 1996 and 1997 [118; 119]. They compared the take-over time[2] of four selection operators in combination with various neighborhood size and shapes in a two-dimensional cellular EA. The selection operators were proportional, linear ranking, tournament, and a so-called

[1] The Cellular EA is sometimes called the diffusion model.

[2] The take-over time is the number of generations it takes for the best individual to dominate the entire population when no mutation or recombination is employed.

random walk approach [119]. The shapes were cross, boxes, and diamonds. As expected, the ratio between the entire grid and the neighborhood size is the main factor influencing the take-over time. The neighborhood shape had no effect.

In an interesting study, Bach and Kjær-Larsen investigated the effect of dimensionality in the cellular EA [7]. They experimented with the individuals' neighborhood by varying the dimensionality of the world between one and six dimensions. The objective was to determine whether it would be beneficial to implement more complex models, or if a simple one-dimensional world would be sufficient. Bach and Kjær-Larsen concluded that the main parameter influencing the performance was the interaction range, i.e., the size of the neighborhood. Somewhat surprising, the one-dimensional model gave similar performance as higher dimensional models with the same interaction range. Hence, the traditional two-dimensional cellular EA may just as well be implemented with a one-dimensional world, which is simpler to implement. In the one-dimensional model, Bach and Kjær-Larsen found an interesting and clear correlation between performance and interaction range. In short, a small interaction range gives a better solution, but at a slower convergence speed. A large interaction range yields fast convergence, but often stagnation on a local optimum. This indicates that the interaction range can be used as a control parameter to balance the quality of the solution versus the convergence speed. Needless to say, such a parameter is important for real world problems where the evaluation is often time consuming.

5.2.2 Patchwork model

The Patchwork model was introduced by Krink et al. and is a hybrid evolutionary algorithm implementing ideas from the cellular EA, multi-agent systems, island models and traditional evolutionary algorithms [84]. The patchwork model was originally introduced as a general approach to modeling biological systems, but it has also been used for optimization [84; 87]. A patchwork model consists of a grid world and a number of interacting agents. In each time step, an agent's behavior is determined by its motivation network, which executes an action, such as moving, mating, or eating. The decision process is based on the input from a number of sensors and their "interpretation" by the motivation network. The sensors detect various properties such as best fitness, population density, and own fitness. Two kinds of sensors exist; i) external sensors returning a value based on the state of the patch it is scanning, and ii) internal sensors reflecting the state of the agent. Each sensor has an output value between 0 and 1. For each possible action the motivation network has a set of functions that map the sensor values to a motivation value for performing the corresponding action. For instance, a setting with two actions and four sensors implies that the motivation network will have eight mapping functions (see figure 5.5 for an example). These functions are encoded in the agent's genome so it can evolve behavior patterns like "follow agents with good fitness, but avoid overcrowded and empty patches". Each sensor is represented by six values, a weight and five points defining the mapping function. See figure 5.4 for an example. The sensor values are combined to a motivation value

M_i as follows:

$$M_i = \sum_{j=1}^{\#sensors} w_{ij} \cdot map_{ij}(s_j) \tag{5.2}$$

where w_{ij} is a weight between 0 and 1 for sensor j in action i, map_{ij} is the mapping function and s_j is the sensor value. Figure 5.4 illustrates an example of an agent's mapping functions for moving when the model supports sensors for best fitness and patch population density. In this case, the agent will be motivated to move to patches with density near 0.5 and high fitness.

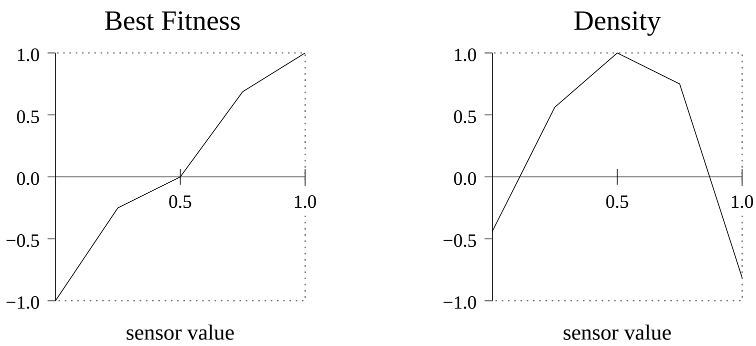


Figure 5.4: Example with sensor mapping functions.

The action of an agent is determined by calculating the motivation for each available action on each of the patches in the agent's neighborhood. If the agent can choose between two actions, e.g., moving and mating, and the neighborhood range is 2 the motivation will be calculated for 50 action-patch pairs. A range of 2 implies 25 neighborhood patches (5×5), whereas a range equal to 3 corresponds to 49 patches (7×7). The motivation network is illustrated in figure 5.5. The action selector is responsible for selecting the agent's next action based on the motivation values. A straightforward idea is to select the action with the highest motivation. In case that multiple action-patch pairs evaluate to the same high motivation, a random action-patch is selected and the action with the corresponding patch is performed.

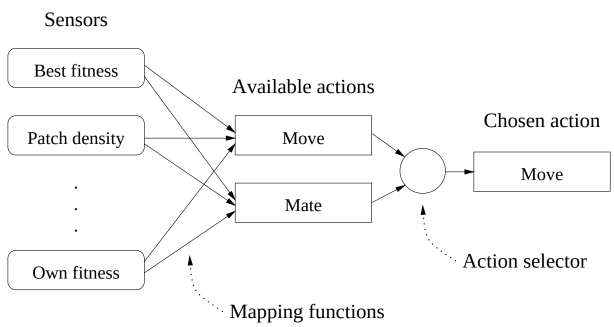


Figure 5.5: Example of a motivation network.

The patchwork model has shown good results on a number of traditional benchmark problems [84; 87].

5.2.3 Religion-based EA

The religion-based EA (RBEA) is inspired by the concepts of religion and how they attract believers. The algorithm was introduced by Thomsen et al. [132]. It extends the ideas of the cellular EA and the patchwork model by adding the concept of religions (subpopulations) and a conversion scheme for attracting new believers to a religion. The RBEA uses a grid world where each cell may be empty or inhabited by a single agent. Like in the patchwork model, the population consists of mobile agents, but RBEA-agents have three fixed actions that are performed in turn. First, each agent tries to do a random walk to an empty cell from its neighborhood. After moving, the agent scans its neighborhood and tries to convert agents belonging to other religions. Conversion is probabilistic and only possible if the current agent has a better fitness than the agent it tries to convert. Finally, the agent attempts to reproduce with a nearby agent from its own religion. Hence, mating is restricted to agents belonging to the same religion, which corresponds to having a number of spatially distributed subpopulations. For details regarding conversion and mating, see [132].

The RBEA was compared with a standard EA and a cellular EA on six numerical benchmark problems. The algorithm outperformed the standard EA on all six problems and the cellular EA on five of six problems.

5.3 Selection-based approaches

High diversity can also be achieved by refining the selection procedure. This can be done by modifying the raw fitness of an individual or by specifically changing the selection operator. In both cases, the objective is to prevent premature convergence by selectively favoring individuals that increase the genetic diversity.

5.3.1 Sharing

Sharing use the idea of modifying the raw fitness to increase the selection pressure on very similar individuals and thereby favor individuals raising the genetic diversity. Goldberg and Richardson introduced the technique with the goal of locating multiple peaks simultaneously [58]. The central concept in sharing EAs is to decrease the fitness value of individuals that are close in the search space, i.e., penalize individuals for clustering. Individuals close in the search space "share" the resources (the fitness), whereas individuals in sparsely populated regions of the search space have fitness equal to the raw fitness. The purpose of this is to prevent individuals from converging to a single peak, which is accomplished by dividing the raw fitness by the so-called sharing factor sh. The "shared" fitness is often calculated according to the following formula:

$$Fit'(I_i) = \frac{Fit(I_i)}{\sum_{j=1}^{\mu} sh(d(I_i, I_j))} \tag{5.3}$$

where $Fit'(I_i)$ is the shared fitness, $Fit(I_i)$ is the raw fitness, μ is the population size, $sh(\cdot)$ is the sharing function, and $d(I_i, I_j)$ is a distance measure between individual I_i and I_j. Goldberg and Richardson suggested the following sharing function, which later became widely used.

$$sh(d) = \begin{cases} 1 - (d/\sigma_{share})^{\alpha} & d < \sigma_{share} \\ 0 & \text{Otherwise} \end{cases} \tag{5.4}$$

The parameters σ_{share} and α specify the shape of the sharing function. A common value for α is 1, whereas the choice of σ_{share} is more problem dependent.

Goldberg and Richardson tested two simple functions in their paper from 1987 [58]. The functions are one-dimensional maximization problems and are defined according to equation 5.5 and 5.6 ($x \in [0, 1]$).

$$F_1(x) = \sin^6(5.1\pi x + 0.5) \tag{5.5}$$

$$F_2(x) = \sin^6(5.1\pi x + 0.5) \cdot e^{\left(-4\ln(2)\frac{(x-0.1)^2}{0.8^2}\right)} \tag{5.6}$$

The algorithm was capable of finding all five peaks on both problems.

A problem with sharing

In a recent study, I investigated the robustness of the sharing technique [139]. For sharing, one property of $F_1(x)$ and also $F_2(x)$ is worth noticing: *The fitness of the minima is zero.* Consequently, no matter how many individuals are near the maxima they will always have a fitness value greater than zero, because the raw fitness is divided by the sharing factor (equation 5.3). Hence, these individuals will always have an advantage in selection when they are compared with individuals at the minima (fitness of zero). However, if a constant is added to the $F_1(x)$ function, i.e.,

$$F_3(x) = F_1(x) + C \tag{5.7}$$

then the properties of selection change substantially. For instance, assume that $C = 10$ in equation (5.7), i.e. $F_3'(x) = F_1(x) + 10$. The topology of the function $F_3'(x)$ is then exactly the same as for $F_1(x)$, except that 10 is added to the function. Now, imagine the situation where two individuals are close to each other and near one of the maxima. In this case the shared fitness of each of these two individuals is

$$Fit'(I) \approx \frac{11}{1+1} = 5.5 \tag{5.8}$$

where the denominator is 2 because the two individuals are close to each other ($sh(\cdot) = 2$). In this scenario, the two individuals near the maximum will actually receive a lower fitness than a single individual near a minimum (see figure 5.6). Hence, the sharing scheme changes the fitness ranking between individuals in such a way that individuals with high fitness with respect to the raw fitness function can easily be deleted during selection. Notice that this problem is independent of the constant added to the function; it only depends on the relative fitness between good and bad solutions. A similar problem arises if good solutions have a fitness of 1.0 while bad solutions have a fitness of, e.g., 0.75.

Figure 5.7 shows two graphs from the experiments I performed [139]. The bars indicate the percentage of the population in the corresponding range of the

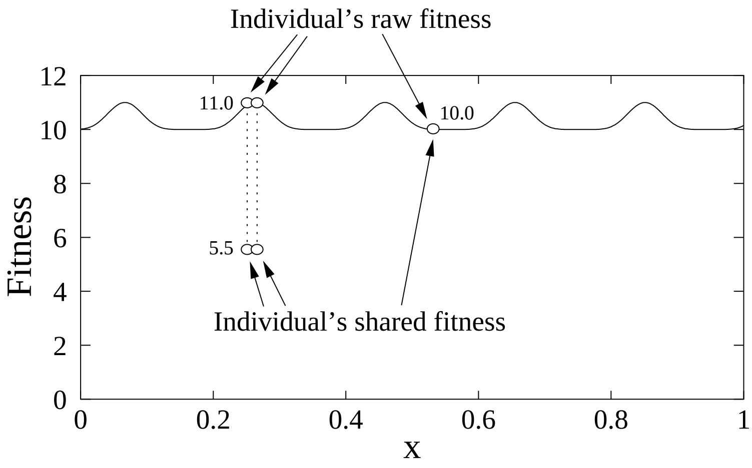
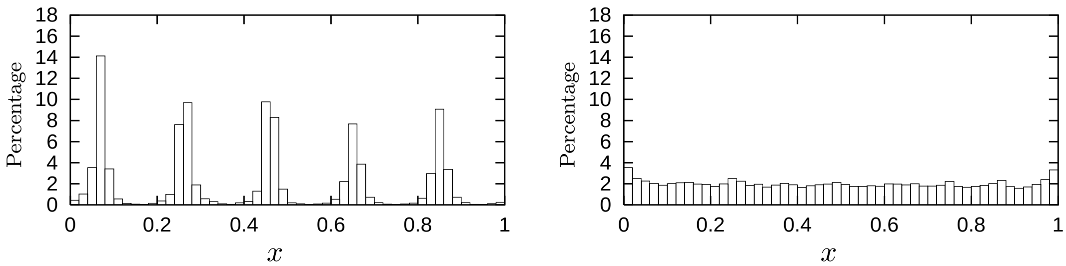


Figure 5.6: $F_3'(x) = F_1(x) + 10$

one-dimensional search space. The graphs clearly show that sharing is extremely sensitive to the fitness range. In fact, the population displayed in figure 5.7(b) roughly corresponds to a population after random initialization.



(a) Sharing on $F_1(x)$ (equation 5.5).

(b) Sharing on $F_3'(x) = F_1(x) + 10$.

Figure 5.7: Some results from the experiments with proportional selection. The x-axis is the search space, and the y-axis is the percentage of individuals in the corresponding part of the search space.

5.3.2 Diversity Control-Oriented EA

Shimodaira recently suggested the diversity control-oriented EA (DCEA), which use the cross-generational probabilistic survival selection (CPSS) [125]. Unlike traditional selection mechanisms, CPSS bases the selection on both fitness and a measure of diversity between each individual and the currently best individual (Shimodaira used binary encoding and the Hamming distance). Hence, the algorithm *directly* incorporate a diversity measure to ensure a sufficient degree of diversity. The main loop of the DCEA works as follows. First, the individuals of the current generation are paired off and each pair creates one offspring by crossover and mutation, i.e., $|P|/2$ individuals are created ($|P|$ is the population size). The $|P|/2$ new individuals are then merged with the parent population and the entire population is sorted on fitness. After sorting, the algorithm removes duplicate individuals and copy the best individual to the next generation. The remaining individuals are then selected with the CPSS operator as follows: The sorted population is traversed from the most fit to the least fit individual. Each candidate individual I_i is stochastically selected using the probability of survival p_s.

$$p_s = \left(\frac{(1-c) \cdot \mathrm{HD}(I_i, I_{best})}{L} + c \right)^{\alpha} \tag{5.9}$$

where $\mathrm{HD}(I_i, I_{best})$ is the Hamming distance between the candidate individual I_i and the population's best individual I_{best}, L is the length of the binary string, and c and α are shape coefficients for controlling the selection pressure. This selection approach does not guarantee that sufficiently many individuals are selected to fill the new population, because each individual is stochastically selected. In case too few individuals are selected, the operator inserts randomly generated individuals in the remaining slots of the new population. Figure 5.8 illustrates the pseudocode of the diversity control-oriented EA.

```
Diversity control-oriented EA
  Initialize and evaluate population P(0)
  while (not <termination condition>) {
    t = t + 1
    Randomly pair all individuals
    Create |P|/2 individuals by mutation and crossover
    Merge new individuals with P(t − 1) and sort on fitness
    Eliminate duplicate individuals
    Select next population with CPSS
    Fill population P(t) with random individuals if too few were selected
  }
```

Figure 5.8: Pseudocode for the diversity control-oriented EA.

The CPSS operator has a number of interesting properties in comparison with traditional selection operators. First, a variable number of individuals are selected in each generation, because the individuals are considered one by one and each individual survives with a certain probability. Second, the operator incorporates both fitness and similarity with the best individual in the selection procedure. The operator favors highly fit individuals that differ significantly from the current best solution.

Shimodaira reported good results on two numerical benchmark problems and three TSP problems [125]. However, from these experiments it seems that the main drawback of the approach is the algorithms sensitivity to the shape parameters controlling the probability of survival in CPSS. Figure 5.9 illustrates a few examples of the probability of survival for some values of c and α.

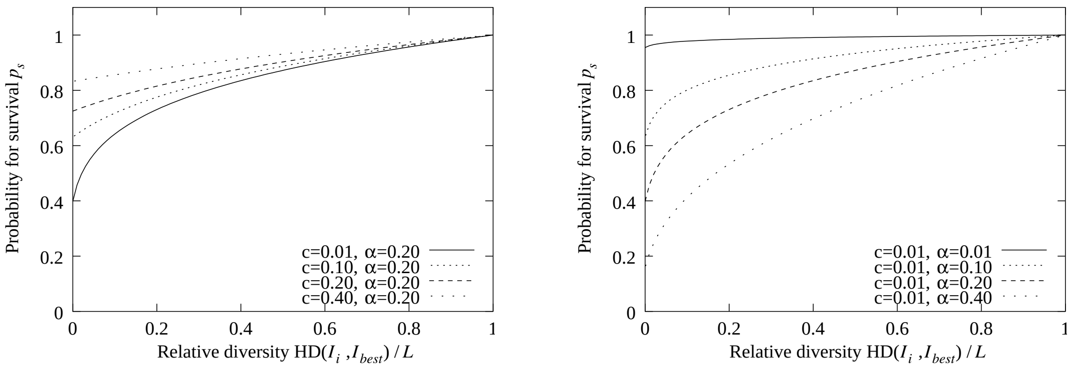


Figure 5.9: Examples of probability of survival p_s for different values of the shape coefficients c and α.

5.4 Search space division

The idea in search space division algorithms is to maintain multiple subpopulations with a minimum overlap in the search space. The goal is to reduce the problem's modality by transforming the problem into a number of easier problems that are searched simultaneously by the subpopulations. In the ideal case, the subpopulations cover different areas of the search space where each subpopulation corresponds to a potential peak in the fitness landscape. A perfect division will allow the algorithm to better exploit the crossover operator, because the parents are approaching the same peak and this will increase the chances of producing a fitter offspring as discussed in the beginning of the chapter.

5.4.1 Forking EA

The forking EA was suggested by Tsutsui and Fujimoto and is one of the first search space division algorithms [135]. The algorithm is developed to search for a single global optimum by keeping track of potential local optima. The population structure consists of a parent population and a variable number of child populations. The algorithm creates a child population when a certain level of similarity is detected in the parent population. The similarity is either calculated from the the binary strings used for the encoding (*genotypic forking*, [135]) or from a phenotypic similarity measure such as Eucledian distance between individuals (*phenotypic forking*, [136]). The parent and the child populations are not allowed to overlap.

The division of the search space in genotypic forking is based on the so-called temporal and salient schemata, which detect the convergence of bit positions in the binary encoding. The schemata are strings consisting of the letters "0", "1", and "*". The temporal schema reflects the state of the population in the current iteration, while the salient schema is calculated from the last K_H temporal schemata. The temporal schema contains a 0 or 1 if more than a predetermined percentage K_{TS} of the individuals have the same value in a gene, otherwise * is inserted. The salient schema is calculated as the temporal schema, but all the K_H temporal schemata have to contain the same value of a gene. Figure 5.10

illustrates a calculation of a temporal schema for a population of 10 individuals. Figure 5.11 illustrates an calculation of a salient schema over 5 generations.

$$
\begin{array}{rcccccccccccccccc}
 & & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
 & & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
 & & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\
 & & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
 & & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
P(t) & = & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
 & & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
 & & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
 & & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
 & & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
\hline
TS^t & = & 1 & * & * & 1 & 0 & * & * & 0 & 1 & * & * & 0 & 1 & 1 & *
\end{array}
$$

Figure 5.10: The temporal schema TS^t for a population of 10 individuals with 15 bit encoding, $K_{TS} = 85\%$.

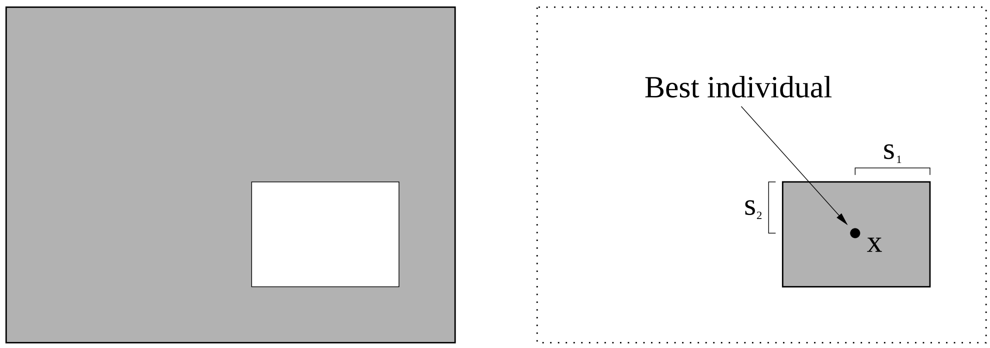
$$
\begin{array}{lcccccccccccccccc}
TS^{t-4} & = & 1 & * & * & 1 & 0 & * & * & 0 & 1 & * & 1 & 0 & 1 & 1 & * \\
TS^{t-3} & = & 1 & * & 1 & 1 & 0 & * & * & 0 & 1 & * & 1 & 0 & 1 & 1 & * \\
TS^{t-2} & = & 1 & 0 & * & 1 & 0 & * & 1 & 0 & 1 & * & * & 0 & 1 & 1 & * \\
TS^{t-1} & = & 1 & * & 0 & 1 & 0 & * & 1 & 0 & 1 & * & * & 0 & 0 & 1 & * \\
TS^{t} & = & 1 & * & * & 0 & 0 & * & 1 & 0 & 1 & * & * & 0 & 1 & 1 & * \\
\hline
SS^{t} & = & 1 & * & * & * & 0 & * & * & 0 & 1 & * & * & 0 & * & 1 & *
\end{array}
$$

Figure 5.11: The salient schema SS^t on 15 bit encoding, $K_H = 5$.

The salient schema is used to extract a child population from the parent population. The subspace searched by the child population is determined by the salient schema leading to the fork. The child population is allowed to alter the bit positions where the salient schema has a '*', i.e., the remaining bits are fixed to the values in the salient schema. In the following generations, individuals in the parent population matching any of the current salient schemata are deleted, which ensures that the parent population is not searching the same subspaces as the child populations. The algorithm limits the number of child populations by either merging salient schemata or overwriting the oldest schema when a new one is needed.

The basic principle of search space division in phenotypic forking is similar to the one used in genotypic forking. The only difference is the procedure for creating the child populations. In phenotypic forking, the child population is a N-dimensional hypercube that is "cut out" of the search space. As in genotypic forking the child population is restricted to this subspace, while the individuals in the parent population search the remaining space. The hypercube determining the boundaries of the child population is defined by a center point $\mathbf{x} = (x_1, x_2, ..., x_N)$

and a range vector $\mathbf{s} = (s_1, s_2, ..., s_N)$, where $s_i > 0$. The hypercube is then defined as $\mathbf{x} \pm \mathbf{s}$. The center point is the genome of the currently best individual in the parent population at the time of the forking, $\mathbf{s}$ is usually calculated from the intervals that define the search space (see figure 5.12).



(a) The parent population searches the gray area.

(b) The search area of the child population defined by the center point and range vector $\mathbf{s}$.

Figure 5.12: Parent and child population in the phenotypic forking EA.

A child population is created if two conditions are fulfilled. i) The best individual in the parent population has not improved for K_H generations and ii) the number of individuals inside the new hypercube is higher than a certain percentage of the size of the parent population. In the following generations, the individuals in the parent population ending up inside the hypercube are replaced with randomly generated individuals such that they are not within any of the hypercubes searched by the child populations.

The forking EA produced good results on four non-trivial test problems including the *frequency modulation sound parameter identification problem* (see [135] for details).

5.4.2 Shifting Balance EA

The shifting balance EA (SBEA) was developed by Oppacher and Wineberg with the purpose of locating the global optimum [103]. The population structure in the SBEA is similar to the one used in the forking EA. The SBEA manages a core population and a number of colony populations. The core population searches for the global optimum, while the colony populations explore areas of the search space that are not covered by the core population.

The best individuals of the colony populations are transferred to the core population at fixed intervals, e.g., every fifth generation. The core population contains the currently best solution while the colonies search for new potential peaks. The role of the core and colony populations is thus the opposite of the setup in the forking EA, where the parent population searches for new peaks and the child populations specialize on already discovered peaks. The core and colony populations are kept apart by enforcing a special kind of selection that drives the colony away from the core if they start to overlap. The idea is to select a percentage of the individuals in the colony based on their distance to the core population and then select the remaining individuals by traditional fitness-based selection. The percentage of individuals that are selected according to their distance is determined by the degree of overlap between the core and the colony. The larger the overlap the higher is the percentage of individuals that are selected based on their distance. The overlap factor is calculated from the Hamming distance between individuals from the two populations. The calculations are based on the all-to-all diversity measure for binary strings.

$$Diversity(P) = \frac{1}{L \cdot |P| \cdot (|P| - 1)} \sum_{i=1}^{|P|} \sum_{j=1}^{|P|} HD(I_i, I_j) \tag{5.10}$$

where P is the population, L is the length of the binary string used in the encoding, $|P|$ is the population size, and I_i is the genome of the i'th individual. A similar function can be used to calculate the distance between an individual and a population.

$$Distance(I, P) = \frac{1}{L \cdot |P|} \sum_{i=1}^{|P|} HD(I, I_i) \tag{5.11}$$

The following function was used to estimate the degree of containment of the colony population A in the core population B.

$$\begin{aligned} Containment(A, B) &= \frac{1}{|P_c|} \sum_{i=1}^{|P_c|} WithinDistance(a_i, B) && (5.12) \\ &= \frac{1}{|P_c| \cdot |P|} \sum_{i=1}^{|P_c|} \sum_{j=1}^{|P|} \delta(a_i, b_j) && (5.13) \\ \text{where} & \\ \delta(a_i, b_j) &= \begin{cases} 1 & \text{if } Distance(a_i, B) < Distance(b_j, B) \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

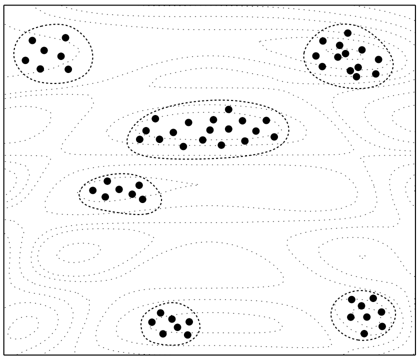
where $|P_c|$ is population size of the colony and $|P|$ is the size of the core population. The function $WithinDistance(a_i, B)$ counts the number of individuals in the B-population that are closer to B than a_i. The $Containment$ function returns a value between zero and one indicating the percentage of individuals from A contained in B. If the colony population A is completely contained in the core population B, then the algorithm selects all individuals in population A according to their distance to population B, which should drive the colony away from the core.

Oppacher and Wineberg tested the SBEA on a static and a simple dynamic problem. The SBEA outcompeted the standard EA on both problems.

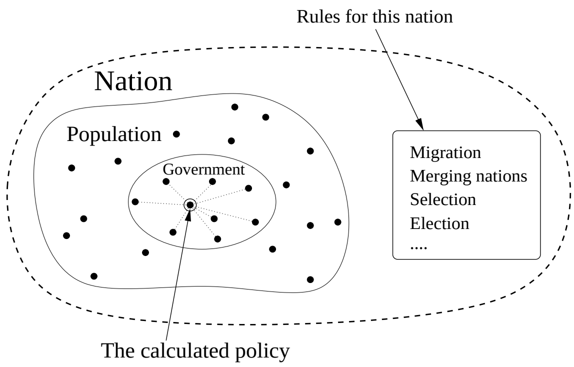
5.4.3 Multinational EA

To locate multiple solutions simultaneously, I suggested the multinational EA (MEA) [137]. The algorithm combines several ideas including self-organization, adaptation to the problem, search space division, and subpopulation mechanisms.

The main idea in the MEA is to automatically divide the population into nations (a kind of subpopulation), each corresponding to a potential peak in the fitness landscape. A situation with six nations is illustrated in figure 5.13(a). A nation consists of a population, a government, and a policy. The government is a subset of the individuals in the population, and it is elected so that its members are the best representatives, e.g., the fittest individuals, of the potential peak the nation is approaching. From these "politicians" the policy is calculated, which is a single point representing the peak the nation is approaching. These concepts are pictured in figure 5.13(b).



(a) Example with six nations.



(b) Concepts of the nation.

Figure 5.13: Population structure in the multinational EA.

The grouping of individuals is done with the *hill-valley detection* procedure that, given two points in the search space, calculates the fitness of a number of random sample points on the line between the points. A valley is detected if the fitness in a sample point is lower than the fitness of both end points. An example of hill-valley detection for a one-dimensional problem is illustrated in figure 5.14.

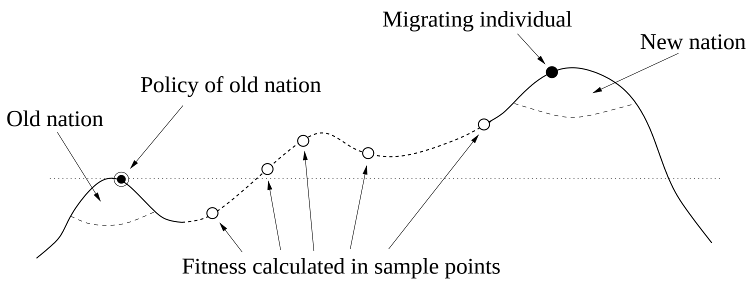


Figure 5.14: Detection of valleys with hill-valley detection in connection with migration.

The hill-valley detection is used in three parts of the MEA; i) migration of individuals between existing nations, ii) creation of new nations in unexplored areas, and iii) merging of nations when the algorithm detects that they approach the same peak. Migration and creation of new nations are performed as follows. In every generation, the algorithm compares each individual with the policy of its nation. The individual migrates if a valley is detected, because it is no longer approaching the same peak as all the other individuals in that nation. The destination is found by comparing the individual to the policy of each of the other nations. This particular individual might have discovered a whole new potential peak if no suitable nation is found. In this case, the individual founds a new nation. As a precaution, the new nation is supported with a number of individuals taken among the lowest fit individuals from the other nations if very few individuals have migrated to this new nation when the migration is over. This ensures that the new nation has sufficiently many individuals to have a fruitful evolutionary process. These individuals are converted to the new nation by overwriting their genome with the position of the policy with some noise added to diversify them a bit. To counterbalance this splitting of nations a merging scheme for nations is enforced. Two nations are merged if a valley is not detected between their policies, because this indicates that the nations are approaching the same peak.

The multinational EA has been applied to a number of static problems [137]. The algorithm was capable of successfully locating a number of peaks in each problem. Furthermore, it has been applied to artificial dynamic problems [138]. In this study, the main idea is to track multiple peaks in the changing landscape and thereby have suboptimal peaks located before they may rise to become the global optimum.

5.5 Mass extinction

In mass extinction models, the diversity is maintained by forcefully replacing a part of the population. Mass extinction models differ in how the individuals are selected for extinction, how many individuals go extinct, and how the new individuals are generated.

5.5.1 Random immigrants EA

The random immigrants EA was suggested by Grefenstette in 1992, and is probably one of the first EAs using mass extinction [61]. The algorithm uses a very simple extinction scheme. In each generation, a proportion of the population is replaced by randomly created individuals. The *replacement rate* is fixed and usually set to a value between 5 and 10 percent of the population size.

The random immigrants EA was suggested as an algorithm for handling time-varying problems. In this context, Cobb and Grefenstette investigated the algorithm on three simple dynamic benchmark problems and a static problem (all two-dimensional problems) [32]. Random immigrants outperformed a standard EA on the dynamic problems, but not on the static problem. Cobb and Grefenstette concluded that this was probably because of the disruptive effect of constantly reintroducing random individuals.

5.5.2 Extinction EP

Greenwood et al. investigated the palaeontologic literature on natural mass extinction and used this as an inspiration for a mass extinction model for evolutionary programming [60]. Their algorithm operates with two types of extinction events; a mass extinction event and a background extinction event. In each generation, a stress factor $\eta(t)$ is generated according to $\eta(t) \sim U(0, 0.96)$. The upper bound of 0.96 may seem to be a rather odd number, but it is motivated by the most devastating event in Earth history where 96% of all marine animals went extinct. To determine the individuals subject to extinction, the algorithm scales the fitness of all individuals to the interval to $[\alpha, 1]$ as follows (assuming a minimization problem):

$$Fit'(I_i) = \alpha + (1 - \alpha) \cdot \frac{Fit(I_i) - Fit(I_{max})}{Fit(I_{min}) - Fit(I_{max})} \tag{5.14}$$

where α is a control parameter, $Fit(I_{max})$ is the fitness of the worst individual, and $Fit(I_{min})$ is the fitness of the best individual. The individuals to kill are those with a scaled fitness below the stress factor, i.e., $Fit'(I_i) < \eta(t)$. The vacant slots in the population are filled with mutated variants of the surviving individuals. The algorithm executes the background extinction if no individual had a scaled fitness below the stress factor. The background extinction mutates the worst individual and five randomly chosen individuals. The pseudocode for the Extinction EP is displayed in figure 5.15.

Extinction EP
Initialize and evaluate population P(0)
t=0
while (not <termination condition>) {
  $t = t + 1$
  Generate stress factor $\eta(t) \sim U(0, 0.96)$
  Find best fitness $Fit(I_{min})$ and worst fitness $Fit(I_{max})$
  `killcount`=0
  for (each individual I_i in $P(t)$) {
    Calculate $Fit'(I_i)$ according to equation 5.14
    if ($Fit'(I_i) < \eta(t)$) {
      Remove I_i from population $P(t)$
      `killcount++`
    }
  }
  if(`killcount`>0)
    Insert `killcount` individuals in $P(t)$
  else
    Perform background extinction event
}

Figure 5.15: Pseudocode for the Extinction EP.

Although this model seems interesting, Greenwood et al. unfortunately only reported results on a rather simple two-dimensional numerical problem [60]. Hence, the performance on a broader selection of problems have not been examined yet.

5.5.3 SOC extinction EA

The idea of self-organized criticality (SOC, [14]) has been used by Krink et al. to control the extinction rate in an EA [86]. The approach is motivated by the fact that mass extinction in nature follows the power law distribution [111; 14]. The SOC extinction EA use a so-called sandpile model [14] to generate power-law distributed numbers that determine the percentage of the population to kill in each generation. The empty slots in the population are then filled with mutated copies of existing individuals. The pseudocode for the extinction procedure is shown in figure 5.16.

```
SOC extinction
  kill_percent = PLDistribution[t]
  if (kill_percent≥1) {
    extinction_rate = kill_percent*|P|
    Remove extinction_rate randomly chosen individuals
    Create extinction_rate from surviving individuals by mutation
    and insert in empty slots
  }
  else {
    Apply tournament selection
  }
```

Figure 5.16: Pseudocode for the extinction procedure in the SOC extinction EA. PLDistribition is an array with power-law distributed numbers, t is the generation number, and $|P|$ is the population size.

Krink et al. compared the SOC extinction EA with a standard EA on six static benchmark problems [86]. The algorithm outperformed the standard EA on all six benchmark problems. In a followup study, Krink and Thomsen combined the idea of SOC and mass extinction with spatially distributed population structures [85]. A related investigation on spatial mass extinction, but without SOC, was performed by Kirley and Green [82].

5.6 Restart and phase-based techniques

The idea in restart and phase-based EAs is to somehow detect convergence and then diversify the population whenever this occurs in the optimization process. In these algorithms, the optimization is performed in a number of consecutive phases where the best individual survives from one phase to the next.

5.6.1 CHC algorithm

The CHC algorithm suggested by Eshelman combines several ideas in one model [45]. The algorithm uses binary encoding and a number of techniques to both slow down convergence and repair a converged population. The genetic convergence is slowed by the way recombination is performed. First, the CHC algorithm uses a special uniform crossover operator that creates two offspring by exchanging exactly half the non-matching bits in the two parents. This produces two individuals with the maximal Hamming distance between parent and offspring. Second, the algorithm employs a so-called *incest prevention* scheme to limit the number of offspring produced by similar individuals. In incest prevention, only parents with a Hamming distance above a certain difference threshold are allowed to reproduce. The difference threshold is initially set to $L/4$, where L is the length of the binary genome. The threshold is decreased when no offspring are created because of too similar parents. In addition to these two techniques for slowing convergence, CHC use a restart strategy when it detects that the population has converged. The restart occurs when the difference threshold has dropped to zero and no new offspring has been accepted for a number of generations. In this case, the population is reinitialized with copies of the best individual where a large part of the genome is mutated. Eshelman suggests to flip 35% of the bits [45]. An interesting aspect of CHC is that it use selection and recombination during the optimization process and only mutation when the population is restarted.

The CHC algorithm was compared with a traditional GA on ten numerical benchmark problems, four so-called deceptive functions, and a 532-city traveling salesman problem (see [45] for details). The CHC algorithm was the best algorithm on nearly all test problems.

5.6.2 Diversity-Guided EA

To improve the control over the population diversity, I introduced the diversity-guided EA (DGEA) [140]. The idea behind the DGEA is simple. Unlike most other EAs the DGEA uses a diversity measure to alternate between exploring and exploiting behavior. To use a measure for this purpose it has to be robust with respect to the population size, the dimensionality of the problem, and the search range of each of the variables. An immediate measure for N-dimensional numerical problems is the “distance-to-midpoint” measure, which is defined as:

$$diversity(P) = \frac{1}{|D| \cdot |P|} \cdot \sum_{i=1}^{|P|} \sqrt{\sum_{j=1}^{N} (s_{ij} - \overline{s}_j)^2} \tag{5.15}$$

where $|D|$ is the length of the diagonal[3] in the search space $S \subseteq \mathbb{R}^N$, P is the population, $|P|$ is the population size, N is the dimensionality of the problem, s_{ij} is the j'th value of the i'th individual, and $\overline{s}_j$ is the j'th value of the midpoint $\overline{s}$. The pseudocode for the DGEA is listed in figure 5.17.

The DGEA applies *diversity-decreasing* operators (selection and recombination) as long as the diversity is above a certain threshold d_{low}. When the diversity drops below d_{low} the DGEA switches to *diversity-increasing* operators (mutation) until a diversity of d_{high} is reached. Hence, phases with exploration and phases with exploitation will occur (see figure 5.18 for an illustration of the process). Theoretically, the DGEA should be able to escape local optima because the operators enforce higher diversity regardless of fitness.

[3]Assuming that each search variable x_k is in a finite range, i.e., $x_{kmin} \leq x_k \leq x_{kmax}$.

DGEA main

$t = 0$
Initialize population $P(0)$
Evaluate population $P(0)$
`mode` = "Exploit"
while(!(termination condition)) {
  $t = t+1$
  if(diversity($P(t)$)$< d_{low}$)
    `mode` = "Explore"
  elseif(diversity($P(t)$)$> d_{high}$)
    `mode` = "Exploit"

  if(`mode ==` "Exploit")
    Select next generation $P(t)$ from $P(t-1)$
    Recombine $P(t)$
  else
    Mutate $P(t)$
  Evaluate population $P(t)$
}

Figure 5.17: Pseudocode for the DGEA.

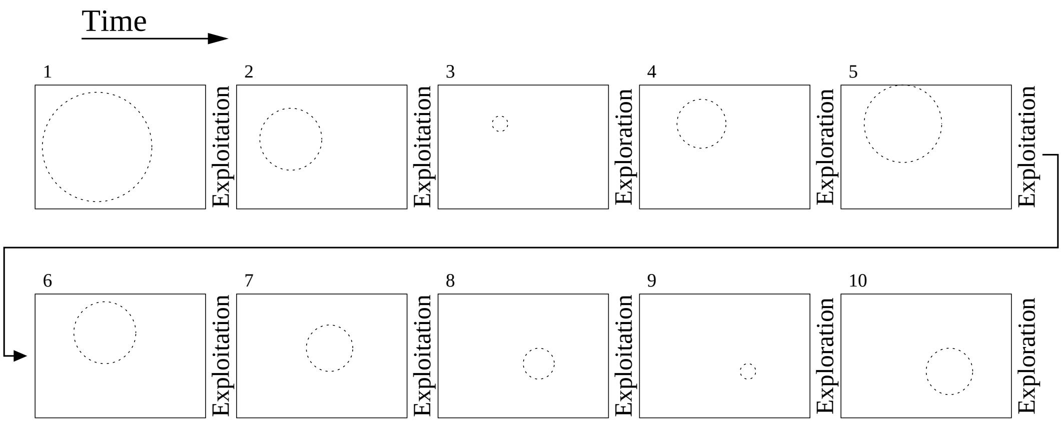


Figure 5.18: Phases in the DGEA. The boxes denote the search space, the dotted circles indicate the diversity and position of the population. The mode is shown as the vertical text between the pictures, i.e. exploitation lowers the diversity in frame 1 and transforms it into frame 2.

An important issue is to apply a mutation operator that rather quickly increases the distance-to-midpoint measure. Otherwise, the algorithm will stay in "explore"-mode for a long time. A straightforward idea for a measure-increasing mutation operator is to use the midpoint of the population to calculate the direction of each individual's mutation. The individual is then mutated with the Gaussian mutation operator, but now with a mean directed away from the midpoint (see figure 5.19). The purpose of this mutation operator is to force the individuals away from the population center.

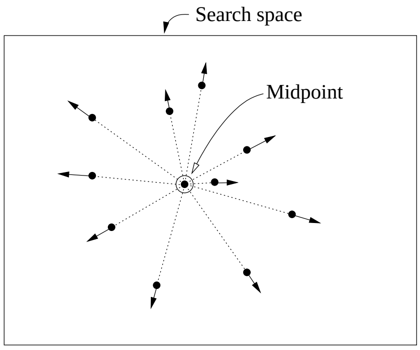


Figure 5.19: Directed mutation in the DGEA.

The DGEA have been tested on 20, 50, and 100 dimensional variants of four standard benchmark problems [140]. In the initial investigation, I logged the diversity when fitness improvements occurred. Interestingly, the majority of improvements occurred at very low diversity. In a later study, I applied the DGEA to system identification of two induction motor models [147]. For more information on this study, see chapter 7.

5.7 Summary

Multimodal optimization is most likely the most extensively investigated issue in evolutionary computation, and hundreds of algorithms have been suggested over the years. In this chapter, I have tried to group the main ideas behind these algorithms and give examples from each category. In my view, the most interesting approaches are the search space division techniques and the methods based on diversity. Search space division techniques are particularly interesting because they adapt to the problem through the use of self-organizing population structures. In this, the algorithms utilize the location of found optima enforce diversity by preventing an overlap of the subpopulations. Diversity-based methods seem promising, because diversity is believed to play a key role in the success on multimodal problems. In these algorithms, diversity measures are directly used to guide the search and prevent premature convergence.

5.8 Future research

The wide selection of algorithms for multimodal optimization should give a good starting point for application to tough real-world problems. Somewhat surprising, few of these algorithms are actually used in practical applications although they show superior performance when compared with simple text book algorithms. One explanation for this mismatch between research on multimodal techniques and practical application may be that many of these algorithms are rather complex to implement and additional parameters need to be tuned. Hence, there is a great need for making these algorithms available and easily approachable for researchers without a background in EC-research. In this context, at least two issues should be taken into account by EA-researchers. First, they should invest more time in demonstrating their techniques on realistic problems. Applying a novel algorithm to artificial benchmark problems as those in appendix B is not very convincing to researchers working on real-world problems. Second, algorithms should be developed with a minimalistic approach in mind. In practical applications, most time is spent on defining the problem and implementing the fitness function. In many cases, only little time is available for the implementation of the optimization algorithm. Unfortunately, many algorithms in multimodal optimization are simply too complex to implement and simpler, but less efficient, algorithms are therefore used instead.

Regarding population diversity, an underlying hypothesis in many studies on multimodal optimization is that maintaining a high level of population diversity is a prerequisite for escaping local optima. This may be true, but my recent work on the Diversity-Guided EA shows the importance of also having an occasional low diversity. Surprisingly, most improvements in fitness occurred at quite low diversity. Hence, both low and high diversity seem to be necessary to escape local optima and exploit crossover. Interestingly, little is actually known about the exact role of diversity in optimization. The common notion is that the population diversity should be kept at "an appropriate level"; however, what this level is is rather unclear, because the algorithm's diversity maintaining capabilities are seldomly thoroughly analyzed. Hence, a great deal of insight could be achieved by thoroughly investigating the algorithm's diversity behavior during the optimization.

Chapter 6

EA approaches to system identification and control

6.1 System identification

System identification is an important part of the scientific fundament of most natural sciences. In short, system identification is about building mathematical models for describing an observed system, which may be anything from planetary movements in Astronomy to algae growth in Biology. In control engineering, system identification[1] is employed to determine a model of the system (plant) subject to control. In this context, system models describe the behavior of the plant over time as it is exposed to control and influence from external factors. System identification consists of two subtasks; i) structural identification of the equations in the model M, and ii) parameter identification of the model's parameters $\hat{\theta}$. A system identification problem can be formulated as an optimization task where the objective is to find a model and a set of parameters that minimize the prediction error between system output $y(t)$, i.e., the measured data, and model output $\hat{y}(t, \hat{\theta})$ at each time-step t (figure 6.1).

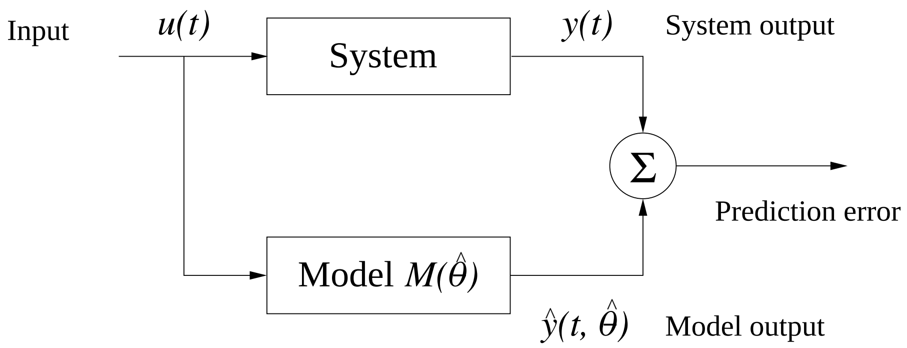


Figure 6.1: Components of system identification.

The data is a vector of system outputs measured at fixed intervals in time,

[1]System identification is the control engineering term for this basic scientific task, but many alternative names exist such as equation discovery, time-series prediction, and inverse problems.

and the data is therefore referred to as time-series data. The sum of squared error (SSE) is a commonly used measure of the prediction error.

$$SSE(\hat{y}, \hat{\theta}) = \sum_{t=1}^{T} (y(t) - \hat{y}(t, \hat{\theta}))^2 \tag{6.1}$$

Naturally, system identification is a huge research area in control engineering. Thus, providing a complete survey is beyond the scope of this thesis (for an overview of linear identification theory and some non-linear techniques, see [95]). In short, the engineering approach is highly analytical and based on mathematical derivation of a system model. At the present stage, the system identification theory is well-developed for linear systems, but many systems are non-linear and limited theory has been developed to identify such systems analytically. For non-linear system, evolutionary computation seems to be a very promising approach, because EAs can easily be combined with a number of other techniques from control engineering, machine learning, artificial intelligence. Non-linear system identification techniques can generally be grouped into white box and black box approaches. Table 6.1 gives an overview of EA-relevant approaches.

	Parameter identification	**Structural identification**
White box	Numeric optimization EA (Parameters in engineering models)	Unit-typed genetic programming (Equations with correct SI units)
Black box	Neural networks (Weights of the network)	Fuzzy logic predictors (Membership functions)
	Non-linear regression models (Weights of the terms)	Untyped genetic programming (Equations regardless of SI units)

Table 6.1: Grouping of techniques for non-linear system identification. Text in parentheses denote the part evolved by the EA.

In white box models, the system is typically described by a set of non-linear differential equations expressing the physics behind the system. In white box parameter identification, the model is derived manually by an engineer. The EA's task is then to determine the model parameters that minimize the prediction error (for examples, see [123; 65]). White box structural identification may also be automated by unit-typed genetic programming, in which differential equations with correct SI units are evolved, e.g., [6]. In contrast, black box models are not concerned with the physical soundness of the model, but just to match the system output in the best possible way. Black box models usually predict the next system output $\hat{y}(t, \hat{\theta})$ as a function of previously recorded system inputs and outputs, i.e., $\hat{y}(t, \hat{\theta}) = f(\varphi(t-1))$ where $\varphi(t-1) = [u(t-1), ..., u(t-n), y(t-1), ..., y(t-n)]$. The vector $\varphi(t-1)$ is called a *regression vector* and the elements of $\varphi(t-1)$ are called *regressors*. Black box approaches include neural networks, regression models, fuzzy logic predictors, and untyped genetic programming. Neural networks and regression models are parameter identification problems, in which the task is to identify the weights in the network or the terms of the regression model, which is typically a polynomial of the regressors[2]. In black box structural identification, fuzzy logic predictors express human readable rules describing the system. The EA's task is to select and design the membership functions for the fuzzy predictor. Another approach is to use untyped genetic programming, which can be seen as a generalization of the regression model technique. For instance, polynomials can be expressed in genetic programming by only using plus and multiplication in the nodes of the expression trees and regressors and constants as the leaves [116].

After deriving the model and its parameters, it should be tested to make sure it properly predicts the system behavior. This is usually done by a manual verification. For instance, by performing a whiteness test to see if the residuals are white noise (see [95, pp. 511]). The residuals are the difference between the measured data and the prediction.

$$\varepsilon(t) = y(t) - \hat{y}(t, \hat{\theta}) \tag{6.2}$$

Obviously, the model cannot be further improved when the residuals are white noise.

6.1.1 Fitness function design

The sum of squared error (equation 6.1) is one of the most widely used prediction error measure. There are a number of related error measures such as the mean squared error (MSE) and the root mean squared error (RMS). In summary, these measures are defined as follows:

$$SSE(\hat{y}, \hat{\theta}) = \sum_{t=1}^{T} (y(t) - \hat{y}(t, \hat{\theta}))^2 \tag{6.3}$$

$$MSE(\hat{y}, \hat{\theta}) = \frac{1}{T} \sum_{t=1}^{T} (y(t) - \hat{y}(t, \hat{\theta}))^2 \tag{6.4}$$

$$RMS(\hat{y}, \hat{\theta}) = \sqrt{\frac{1}{T} \sum_{t=1}^{T} (y(t) - \hat{y}(t, \hat{\theta}))^2} \tag{6.5}$$

These error measures often appear in the prediction literature, because they usually have smooth surfaces with respect to the model parameters $\hat{\theta}$. From an EA perspective, these measures give the same ranking of individuals, since the relative relationship between two solutions are the same regardless of the error measure, i.e., if individual A is better than individual B with respect to SSE, then A will also be better than B if evaluated using the other measures. However, measures based on squared error are not entirely fair regarding the influence of each error term in the sum. In fact, absolute prediction errors below 1 are underestimated whereas absolute prediction errors above 1 are overestimated. For instance:

$$|y(t) - \hat{y}(t, \hat{\theta})| = 0.5 \quad \Longrightarrow \quad ((y(t) - \hat{y}(t, \hat{\theta}))^2 = 0.25$$
$$|y(t) - \hat{y}(t, \hat{\theta})| = 2.0 \quad \Longrightarrow \quad ((y(t) - \hat{y}(t, \hat{\theta}))^2 = 4.0$$

[2] Neural networks and regression models can be considered as structural identification when the network layout or model complexity is evolved.

Hence, choosing a squared error measure implies the preference of many small deviations over few large deviations, which may not be desirable. Squared error measures are probably so popular because they can be used in connection with gradient methods such as back-propagation training of neural networks. However, EAs do not require that the fitness function has a derivative. Consequently, more advanced and perhaps more fair measures can be used in combination with EAs. An obvious choice is to use the sum of absolute errors (SAE).

$$SAE(\hat{y}, \hat{\theta}) = \sum_{t=1}^{T} |y(t) - \hat{y}(t, \hat{\theta})| \tag{6.6}$$

The maximal error (ME) is another measure that is occasionally used.

$$ME(\hat{y}, \hat{\theta}) = \max_{t} |y(t) - \hat{y}(t, \hat{\theta})| \tag{6.7}$$

However, this may result in a fitness landscape with many plateaus, because only one measuring point is used as the fitness. Hence, a few outliers in the measured data may easily disrupt the search, since the EA will first focus on minimizing the prediction error in these few points.

Adaptive fitness functions

The flexibility of EAs regarding the fitness function have been exploited by Eggermont and van Hemert who suggested an adaptive fitness approach [40]. Their method is called "stepwise adaptation of weights" (SAW). Although the approach was tested on non-time-series data, it is straightforward to adapt the method to time-series data. The idea in SAW is to have a weight w on each of the error terms, i.e.

$$SAW(\hat{y}, \hat{\theta}) = \sum_{t=1}^{T} w(t) \cdot |y(t) - \hat{y}(t, \hat{\theta})| \tag{6.8}$$

The weights are then increased at fixed intervals in the optimization, which is to put more focus on the largest deviating terms in the sum of prediction errors. Eggermont and van Hemert suggest two methods; classic SAW and precision SAW. In classic SAW, the weights where $|y(t) - \hat{y}(t, \hat{\theta})| > 0$ are increased by $\Delta w(t) = 1$. In precision SAW, the error at time t is used to increase the corresponding weight, i.e., $\Delta w(t) = |y(t) - \hat{y}(t, \hat{\theta})|$. For time-series data, I will expect precision SAW to be the best, because few terms will have an error of zero. However, this is yet to be investigated for system identification problems. Finally, it should be mentioned that this approach turns the identification into a dynamic optimization problem, because the fitness function is altered at fixed intervals.

6.1.2 Multiobjective and constraint system identification

System identification problems can also be considered as multiobjective optimization problems with constraints. As mentioned, the main objective is to produce a model that matches the measured data in the best possible way. However, the model complexity should also be minimized when black box methods are used. This means minimizing the tree size in genetic programming, reducing the number of terms in regression models, and limiting the number of rules in fuzzy predictors. A third objective is to evolve a valid model. This is particularly important for black box methods, because it is not possible to check that the model is physically sound. In summary, the three main objectives in system identification are:

- **Model performance:** The accuracy of the model with respect to the measured data. This is typically expressed by minimizing one of the error measures used in single-objective system identification (section 6.1.1).
- **Model complexity:** The size and complexity of the model. The model's complexity should be as small as possible. For example, concise GP expressions are preferred over complex expressions, since this makes the model more understandable and give a better generalization ability[3].
- **Model validity:** The validity of the model can be expressed by various measures. A common test is to see if the residuals are white noise, in which case the model is valid (see [116] for techniques).

According to Rodríguez-Vásquez, the model validity measures should preferably be treated as constraints to ensure that valid models are evolved [116]. So far, multiple objectives and constraints in system identification have received very limited attention. The PhD thesis of Rodríguez-Vásquez is probably the most comprehensive work until now [116]. To my knowledge, only a few other papers have been published [126; 54].

6.1.3 Dynamic system identification

A fundamental problem in many system models is that they often cannot incorporate all aspects of the system they describe. One solution to this problem is to adjust the model while it is in use, e.g., whenever the error between the model and the system exceeds a threshold. This is particularly important in control applications where simple, but fast, models are used. Dynamic system identification in combination with control is further discussed in section 6.2.4. For other system identification problems, such as time-series prediction, dynamic optimization is typically performed to update the model when new data is available [75; 154]. This is usually done with a sliding window approach that considers data until a certain point in the past.

6.2 Control

Control problems are present in many industrial areas ranging from electro-mechanical systems to bio-chemical processes. The design process of a controller resembles the process for system identification problems. Controller design consist of two

[3] This objective is sometimes referred as Occam's razor.

subtasks; i) structural design of the controller and the equations, and ii) determination of the controller's parameters. Traditionally, control can be seen as the task of operating the system to match a certain reference signal $w_r(t)$, i.e., minimize the error between the system output $y(t)$ and the desired reference signal $w_r(t)$. An electro-mechanical example is the moving of a harddisk's read-write head to a new position. Figure 6.2 illustrates a simple view of the components in reference signal control problem. In other control problems, the task is operate the system to maximize or minimize some measure of the system. For instance, maximize the production or minimize the cost associated with the system. To some degree, such control problems can be considered as a reference signal problem where $w_r(t)$ is set to $\pm\infty$. However, fitness functions from reference signal control problems are typically not used, because they are based on deviation from the reference signal and that the system settles at some point. Instead, a function of the output is used as a performance criterion. This approach is used in the greenhouse control problem studied in chapter 8.

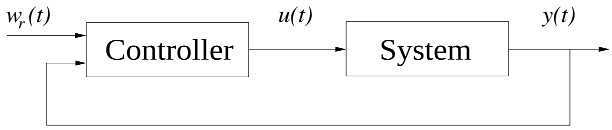


Figure 6.2: Components of reference signal control problems.

For reference signal control, the objective is to design a controller that can quickly and safely operate the system whenever a new reference signal is set. In this connection, the objective is typically to minimize the maximal overshoot and the settling time of the system. The system is settled when the deviation from the reference signal is less than a predetermined error threshold ε. Figure 6.3 shows the maximal overshoot, the settling time, and the error threshold for a typical system behavior in reference signal control.

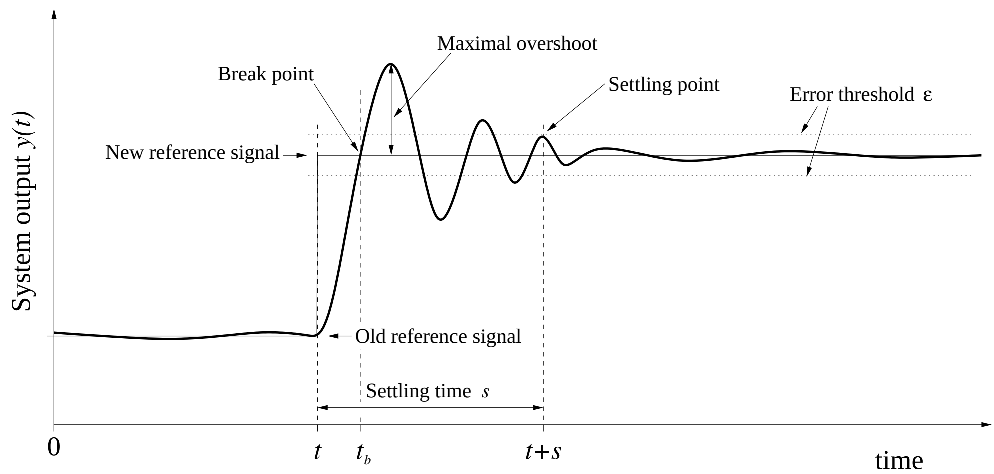


Figure 6.3: Terminology and typical behavior for reference signal control. A new reference signal is introduced at time t.

An important aspect influencing the controller's performance is whether the system is time-invariant (static) or time-varying (dynamic). Time-invariant systems are often handled by an offline training procedure prior to the actual control. The controller is typically designed on the basis of simulation during a fixed control period. In contrast, time-varying systems require online techniques, since the behavior of the system changes over time. From an optimization point-of-view, this creates a dynamic optimization problem.

As with system identification, a considerable amount of research has been conducted to develop controller techniques and parameter estimation methods for such controllers. As with system identification, the theory is well-developed for linear systems, but limited for non-linear systems (for traditional engineering techniques, see [90]). Evolutionary computation techniques have great potential for non-linear systems, because such problems typically have multimodal fitness landscapes. Furthermore, control problems are often multiobjective, there may be constraints on both control signals and system behavior, and the system may vary over time. Fortunately, EAs are easily combined with traditional engineering techniques and more AI-based approaches such as neural networks and fuzzy control. In contrast, rather few "pure" EA applications have been reported (see below). In general, EAs can be applied in three ways; for evolving the controller signals, for tuning of controller parameters, and for designing the structure of the controller. Table 6.2 offers an overview of EA-related control techniques. Most techniques can be used in both offline training and online tuning, but so far nearly all EA-research has been conducted as offline training. A checkmark in parenthesis ($\surd$) means that the control technique can be used, but no study has yet been reported (to my knowledge).

Evolutionary task	**Control technique**	**Online**	**Offline**
Control signals	Generalized predictive control	$\surd$	$\div$
	Direct control	$\surd$	$\div$
Controller parameters	Engineering controllers (PID, LQG, GPC, ...)	$\surd$	$\surd$
	Neural network control (Weights of the network)	($\surd$)	$\surd$
	Boxes control (Ranges in boxes)	($\surd$)	$\surd$
	Decision tree control (Parameters in decisions)	($\surd$)	$\surd$
Controller structure	Fuzzy control (Membership functions)	$\surd$	$\surd$
	GP-block control (Block layout)	($\surd$)	$\surd$

Table 6.2: Grouping of techniques for control of non-linear systems. Text in parentheses below techniques denote the part evolved by the EA.

In evolution of controller signals, two main methods exist; generalized predictive control and direct control. Both are online techniques and they rely on an accurate model, which is used to predict the system behavior of the tested control signal. Generalized predictive control (GPC) is an engineering approach that is based on linearization around the current control point, which allows the application of analytic methods for determining the control signal. However, GPC is sometimes applied to systems with constraints, which rules out the possibility of analytic determination of the control signal. Direct control is a pure EA approach in which the control signals are evaluated on an arbitrarily complex model. Generalized predictive control is an example of model predictive control, which covers a broad selection of online controllers for reference signal control [26]. Direct control has primarily been used for maximization or minimization control problems, but it can also be applied to reference signal control. These techniques are further discussed in chapter 8. A number of examples of direct control and model predictive control have been reported; for instance, [151; 150; 102].

Tuning of controller parameters probably represents the largest segment of real-world applications of EAs in control. Studies in this category use an EA to search for real-valued vectors representing the controller's parameters, i.e., the control approach is a hybrid between an EA and some other technique. Many traditional engineering controllers, such as Proportional-Integral-Derivative (PID) controllers, work reasonable well on non-linear systems; however, the determination of parameters may require the optimization of a multimodal function. Furthermore, the parameters may need to be re-tuned during the control, which may be done online with an EA evolving new parameters while the system is being controlled. EA-tuned PID controllers have been investigated in several offline studies, e.g., [69; 89], as well as a few online control, e.g., [2; 64]. Regarding neural network control, EAs are becoming increasingly popular as the training method, because EAs have shown to be more robust than traditional training methods like back-propagation. Several applications have been reported; for example, [73; 38]. Engineering controllers and neural network controller typically produce continuous control signals. However, some problems require on-off or bang-bang control, in which the control signals are discrete (e.g., switching a motor on or off). The so-called boxes approach partitions each of the decision variables[4] (system output, error, etc.) into a predefined number intervals, which creates a number of boxes in the space of decision variables. The EA's task is then to determine the interval boundaries and the applied control value in each box (see [148] for an example). This approach is similar to a decision tree control method. In such controllers, the decision tree contains a number of parameters to be tuned. In this context, Filipič et al. performed an interesting study combining machine learning techniques with EAs [48].

Regarding evolution of controller structures, a significant amount of work has been done on combinations of EAs and fuzzy logic controllers. The EAs task in this is to design the membership functions and the rule-base. EA-fuzzy hybrids have been applied in both offline control, e.g., [157; 77], and in online control, e.g., [76; 93]. For genetic programming, a number of studies have been reported where GP has been used to evolve so-called block-structure controllers [25; 83]. Block-structure controllers consist of a network of modules each representing a certain feature such as time-delay, integrators, and linear relationships. The controller is "wired" in a diagram that resembles electronic circuits.

[4]The variables on which the control is based.

6.2.1 Fitness function design

The fitness functions in control applications are significantly more diverse compared with those used for system identification. For reference signal control, an obvious choice is to minimize the error between the system output and the reference signal and the system output. Typically, this is done with variants of squared error measures. In offline training, the fitness of individual I can be defined as.

$$Fit(I) = \sum_{t=1}^{T} (y(t) - w_r(t))^2 \tag{6.9}$$

As mentioned in the previous section, the settling time and maximal overshoot plays an important role in many systems. For example, the settling time is crucial for fast access on a harddisk, because the desired track cannot be read or written before the head has settled over it. In such systems, the fitness can be expressed as a minimization problem.

$$Fit(I) = \min s \quad \text{where } |y(t') - w_r(t')| < \varepsilon \quad \forall t' > t + s \tag{6.10}$$

where t is the time when a new reference signal is set, s is the settling time, and ε is the error threshold (see figure 6.3). The maximal overshoot is another important aspect in many systems. For harddisk access, maximal overshoot should be as small as possible, because a too large overshoot may cause the head to crash into the boundary of its allowed range and thereby damage the mechanics. A fitness based on maximal overshoot may be defined as a minimization problem.

$$Fit(I) = \max |y(i) - w_r(i)| \quad i \in [t_b..t + s] \tag{6.11}$$

Hence, the maximal deviation between the system output y and the reference w_r after the breakpoint time t_b when a new signal is set. Typically, minimizing settling time and overshoot are conflicting objectives. A simple solution is to make a weighted sum of the two fitnesses (see section 3.3.2). However, this approach may not be the best, and more advanced multiobjective optimization techniques are used instead; for instance, the so-called Fast Elitist Non-Dominated Sorting Genetic Algorithm NSGA-II [37].

The fitness in online control is typically based on recently recorded data or the controller's predicted performance in the immediate future. GPC and other model predictive controllers base the evaluation on the difference between the reference signal. Furthermore, the change in control signal $\Delta u(t)$ is sometimes minimized as well, because the objective is to settle on the new reference signal. The fitness of an individual I at time t is then often defined as:

$$Fit(I) = \sum_{j=1}^{PH} \alpha_j (y(t+j) - w_r(t+j))^2 + \sum_{j=1}^{PH} \beta_j (\Delta u(t+j))^2 \tag{6.12}$$

where PH is the prediction horizon (see section 8.1) and α_j and β_j are weighting schemes. It should be noted that a great variety exists in this fitness definition. Some functions only use a part of the prediction horizon, some exclude the control signal, and other use different measures of the error between system output and reference signal, e.g., sum of absolute errors.

6.2.2 Constrained control

Many real-world control problems have a number of constraints that must be fulfilled during the control period. Constraints pose an additional challenge to the design process, because traditional analytical methods cannot handle constraints. For this reason, constraint optimization is relevant for both linear and non-linear systems. Furthermore, many engineering controllers are not capable of handling constraints. The most promising direction seems to be model predictive controllers, since they allow constraints to be incorporated in the objective function. Camacho and Bordons provide an extensive overview of constraint types in control [26]. Generally speaking, two types of constraints exist; constraints on the control signal $u(t)$, and constraints on the system behavior $y(t)$. In controller design with EAs, signal constraints are typically avoided by a special encoding whereas system behavior constraints are handled by constraint techniques. Typical control constraints include:

- **Signal range:** The control signal must be within certain bounds at all times $u(t) \in [u_{min}, u_{max}]$. Example: A valve cannot be more than fully open or completely closed.

- **Signal change:** The control signal cannot change more than a certain step $|\Delta u(t)| < \Delta u_{max}$. Example: A valve may have to be slowly opened not to damage the system.

- **Overshoot:** The overshoot cannot exceed a certain value when a new reference signal is set. Example: The read-write head of a harddisk must avoid collision with the boundary of the head's allowed range of positions.

- **Settling time:** The settling time cannot exceed a certain value when a new reference signal is set. Example: Systems that may be damaged if a controller settles too slowly.

- **Rise time:** The controller must rapidly bring the system state above a minimal percentage of the difference between the old and the new set point regardless of overshoot. Example: A safety system should bring the system away from the critical state as fast as possible.

- **Band:** The state of the system must follow a trajectory within a certain band. Example: Heating processes in the food industry often have to be gradual not do damage the food.

- **Monotonic behavior:** The change in the system output $y(t)$ must be monotonic when a new reference signal is imposed, i.e., no oscillations may occur. Example: The harddisk's read-write head may not allow any overshoot at all.

Constraints on the system behavior are often difficult to handle by traditional engineering optimizers. Therefore, constraint handling EAs play an increasingly important role in control. For linear constraints, variants of gradient search can typically be used to find a solution (see [26]). For non-linear constraints, a wide range of evolutionary approaches exist (see [100]).

6.2.3 Multiobjective controller design

The system behavior constraints described in the previous section are sometimes considered as multiple objectives. As described in section 3.3.2, constraints can easily be converted into multiple objectives. Controller design with multiobjective optimization techniques is often a better approach than constraint approaches, because the multiple objectives can be in conflict. For instance, minimizing the settling time and maximal overshoot are two conflicting objectives. Multiobjective optimization techniques are typically applied in offline design, since a human has to decide among the found solutions. Hence, an online multiobjective approach would need to have an automatic decision mechanism for choosing the controller to operate the real system. A number of offline applications have been reported on multiobjective controller design, e.g., [55; 18]. To my knowledge, no online applications have been reported.

6.2.4 Adaptive control

For time-varying systems, it may be necessary to adjust the controller while it is operating the system. The primary reason for this online adjustment is that the model or the controller does not express all properties of the system. For example, the efficiency of car brakes typically depends on the temperature of the brakes, humidity, and road surface, which may be impossible to model accurately [93]. Furthermore, there may be long-term effects such as wear out of the brakes, which would also be hard to take into account, since the wear out depends on the driving style. The controller may be able to eliminate or reduce the change in braking performance by constantly adjusting the controller's parameters or the model on which the control is based. So far, a rather low number of studies have been reported on adaptive control with EAs. A few examples have been published [2; 93; 64].

6.3 Summary

The fields of system identification and control are indeed very diverse and covers a broad range of problems in engineering. In this context, EAs play an increasingly important role, because these algorithms provide the means to push the limits of performance. So far, most studies have investigated single-objective static problems without constraints. Nevertheless, many real-world control problems may be better handled by multiobjective or constraint techniques. For online controllers, these issues are typically handled by constraint optimization techniques. In offline controller design, a multiobjective approach is often used because the objectives or constraints are in conflict. Naturally, a simulation based approach does not guarantee that the controller does not violate the constraints during the operation. Hence, the simulation during the design should ensure that the controller has been tested in extreme situations.

6.4 Future research

Regarding future research, relatively limited work has been performed on time-varying problems. Most, if not all, dynamic system identification and online control studies have been done with simple EAs not designed for dynamic optimization. For such applications, it would be interesting to study some of the many techniques tested on artificial dynamic problems (see [20] for a survey). Of these techniques, I would expect that memory based approaches for dynamic optimization would have an advantage over simpler techniques, because many of the system identification and control problems exhibit some kind of periodic behavior. Hence, a previously stored solution may turn out to be valuable when a similar situation arise.

Chapter 7

Case study: Parameter identification of induction motors

A fundamental part of system control is the identification of the system being controlled. In control engineering, considerable effort has been devoted to develop methods for identification of system models and their parameters. Currently, a wide range of analytical techniques exists for linear systems. An overview of the methods used in the control area for system identification can be found in Ljung's book [95]. For non-linear systems, limited progress has been made with analytical approaches. Instead, some success have been achieved with various traditional optimization methods. However, a fundamental problem of traditional optimization techniques, like least squares and local search, is their dependence on unrealistic assumptions such as unimodal[1] performance landscapes and differentiability of the performance function. Consequently, non-linear problems are often oversimplified to fulfill such assumptions. Evolutionary algorithms (EAs) and other stochastic search techniques seem to be a promising alternative to traditional techniques. First, EAs do not rely on any assumptions such as differentiability, continuity, or unimodality. Second, they are capable of handling problems with non-linear constraints, multiple objectives, and time-varying components. Third, they have shown superior performance in numerous real-world applications.

In collaboration with chief engineer Pierré Vadstrup, I investigated two induction motors used in the pumps produced by danish pump manufacturer Grundfos A/S. The work presented in this chapter is in submission to the journal "Applied Soft Computing" [147]. For several decades, induction motors have been investigated, and they are extensively described in the engineering literature. In this context, a number of non-linear models of induction motors incorporating magnetic saturation effects have been suggested, e.g., [149]. In spite of the considerable theoretical foundation of induction motors, few studies have used EAs and other stochastic search techniques to identify the model parameters. However, some investigations have been presented relying on models not accounting for non-linear effects. These methods are primarily used for control purposes and they recursively estimate the parameters by use of various methods – mainly Extended Kalman Filters. See [110], [155], and [5] for further information on these methods. Regarding

[1] Also referred to as convex problems, i.e., problems with a single optimum.

EA-investigations, Alonge et al. recently studied a 1.0 kW motor and showed that the evolutionary algorithm GENESIS was better than least squares fitting [3]. Alonge et al. determined stator resistance (R_s), stator inductance (L_s), leakage inductance (L_e), motor load (τ_r), and moment of inertia (J_m) using a state space model with scaled rotor flux. In a study from 1994, Haque et al. used a simple evolutionary algorithm to determine stator resistance (R_s), rotor resistance (R_r), and combination of stator and rotor reactance (X_{lr}) from motor data provided by the motor's manufacturer [65]. Other more distantly related work includes an investigation on determining the loads of the motor [72].

The main objective in this study is to evaluate a number of stochastic search algorithms with respect to parameter identification of two induction motors. The first motor is a 1.1 kW motor and is modeled without taking saturation into account. The second motor is a 5.5 kW motor, in which the saturation is modeled by a function of two parameters. In this problem, these two extra parameters are also determined by the search algorithms. In addition to showing the usefulness of stochastic search techniques in parameter identification, we aim at underlining the importance of carefully selecting the algorithm when approaching such practical problems. Surprisingly, in most practical applications, the focus is entirely on the problem and minimal time is invested in selecting the algorithm. This is in clear contrast to the numerous investigations on artificial numerical optimization problems showing that almost any extension of the basic algorithms will lead to a significant performance improvement. However, an important criterion for choosing a more advanced algorithm is the extra effort necessary to implement it, which should be weighted against the actual performance improvement. The advanced algorithms used in our comparison have been selected with this in mind, i.e., the extensions in these algorithms are small but have lead to improved performance for artificial and real-world problems.

7.1 Induction motor models

In this section, the fundament for calculating the performance criterion will be established. The basic idea in system identification is to compare the time dependent response of the system and a parameterized model by a norm or some performance criterion giving a measure of how well the model response fits the system response.

Normally, the dynamic response of the system is given by the solution to a vector differential equation of the form:

$$\begin{aligned}\dot{x} &= f(\theta, x, u) \\ y &= g(\theta, x)\end{aligned}$$

with the initial condition $x(0) = x_o$. In this system, u is the input signal vector, x is the state vector, y is the measurable output vector and θ is the parameter vector. Normally, the system is affected by noise in both states and measurement, which may be real noise or noise caused by unmodeled dynamics. The parameter vector θ is unknown for real systems. Hence, the objective in system identification is to determine this vector as accurately as possible. To do this, a model of the system is introduced with the same structure as the real system. The model is described by:

$$\begin{aligned}\dot{\hat{x}} &= f\left(\hat{\theta}, \hat{x}, u\right) \\ \hat{y} &= g\left(\hat{\theta}, \hat{x}\right)\end{aligned}$$

with assumed known initial condition $\hat{x}(0) = x_o$. From the system an output signal y for a given input signal u can be measured, and for a given guess of the parameter $\hat{\theta}$ the output response $\hat{y}$ of the model can be calculated, supported by the same input signal as the real system. The system response and the model response can then be compared by a performance criterion, which in the simple case can be quadratic.

$$I\left(\hat{\theta}\right) = \int_0^T (y - \hat{y})^T \cdot W \cdot (y - \hat{y}) \cdot dt$$

where W is a positive definite weight matrix. The criterion is a function of $\hat{\theta}$ and will obtain its minimum vale zero when $\hat{\theta} = \theta$. The system identification problem can now be formulated as an optimization problem, namely

$$\arg\min_{\hat{\theta}} I\left(\hat{\theta}\right)$$

Obviously, the fitness landscape of this problem type may have many local optima and a highly complex topology.

In the next two subsections, the differential equations for the induction motor will be derived together with an application specific performance criterion.

7.1.1 Model of the 1.1 kW motor without saturation

The dynamics of the 1.1 kW induction motor can be described by a set of differential equations, derived from fundamental laws of physics. Here, only the final equations and not the assumptions will be discussed. A comprehensive derivation of the model was performed by Vas [149]. The motor is supplied by three voltages supported from mains[2] or from an electronic unit that can convert the mains voltages to user specified voltages u_1, u_2, and u_3. In order to simplify the notation, we introduce complex voltages and currents. The transformation of the voltages from a three phase system to a complex system is:

$$\begin{aligned}u_s &= u_{sd} + j \cdot u_{sq} \\ &= \frac{2}{3} \cdot (u_1 + a \cdot u_2 + a^* \cdot u_3) \qquad \text{in which } a = e^{j\frac{2\pi}{3}}\end{aligned}$$

The real and imaginary voltages are then:

$$\begin{aligned}u_{sd} &= \frac{1}{3} \cdot (2 \cdot u_1 - u_2 - u_3) \\ u_{sq} &= \frac{1}{\sqrt{3}} \cdot (u_2 - u_3)\end{aligned}$$

[2] The national power grid.

The inverse transformation from the complex notation to the three phase system is for the currents given by the following set of equations.

$$\begin{aligned} i_1 &= i_{sd} \\ i_2 &= -\frac{1}{2} \cdot i_{sd} + j \cdot \frac{\sqrt{3}}{2} \cdot i_{sq} \\ i_3 &= -\frac{1}{2} \cdot i_{sd} - j \cdot \frac{\sqrt{3}}{2} \cdot i_{sq} \end{aligned} \tag{7.1}$$

where i_{sd} and i_{sq} are the complex currents.

From now on we treat the motor as a two phase motor and derive the differential equations in complex notation. The motor consists of a stator where the voltages are applied to two sets of windings perpendicular to each other. In a similar manner, the rotor is composed by two windings, but they can be revolved with respect to the stator. The voltages in the rotor are induced by the movement of the rotor with respect to the stator, the voltage equations for the stator and the rotor are given by the following equations, respectively.

$$\begin{aligned} \dot{\psi}_s &= -R_s \cdot i_s + u_s \\ \dot{\psi}_r - j \cdot \omega_r \cdot \psi_r &= -R_r \cdot i_r \end{aligned} \tag{7.2}$$

The left hand sides are the induced voltages and the right hand sides are the supplied voltages reduced by the resistive voltage drop of the windings. The rotor windings do not have any external supply at all, only the stator windings are supplied. ψ_s and ψ_r are the fluxes through the stator and the rotor windings. In the induced rotor voltage equation, ω_r is the speed of the rotor that enters the induced voltage together with the normal induction part due to the flux changes with respect to time. This extra speed dependent term evolves because the rotor terms are calculated in a stator frame of reference. To eliminate the currents in the voltage equations, the flux linkage between the stator and rotor and visa versa have to be calculated. The resulting linkages are given by the following set of equations.

$$\begin{aligned} \psi_s &= L_{sl} \cdot i_s + L_m \cdot (i_s + i_r) = L_{sl} \cdot i_s + \psi_m \\ \psi_r &= L_{rl} \cdot i_r + L_m \cdot (i_s + i_r) = L_{rl} \cdot i_r + \psi_m \end{aligned} \tag{7.3}$$

The first expression shows how the flux through the stator windings is composed of the current in the stator itself and by the currents in the rotor windings. The term ψ_m expresses the main flux shared by the stator and the rotor and the rest expresses the leakage flux in the stator and the rotor. The currents in these expressions can be isolated and applied in the differential equations in expression (7.2). Consequently, the differential equations can be solved by applying the voltage u_s, if the speed ω_r of the motor is known. However, the speed is not an input, but is governed by the following equation of motion for the rotor.

$$\dot{\omega}_r = \frac{1}{J} \cdot (M - M_L) \tag{7.4}$$

in this expression M is the developed torque of the motor and M_L is the torque load and friction. Furthermore, J is the moment of inertia of the rotor and the load.

The developed torque of the motor can be calculated from the electro-mechanical states by the expression:

$$M = \frac{3}{2} \cdot Im\left(\psi_s^* \cdot i_s\right) \tag{7.5}$$

In our experiments, we assume that the load torque M_L is zero, which eliminates the load torque as an additional input. The model can be described by a set of explicit differential equations if in the above expressions the parameters are constant. In summary, we have:

$$\begin{aligned} \dot{x} &= f\left(\theta, x, u\right) \\ y &= g\left(\theta, x\right) \end{aligned}$$

where for the actual system the input, state and input are given by the following vectors.

$$\begin{aligned} u &= \left[u_1, u_2, u_3\right]^T \\ x &= \left[\psi_{sd}, \psi_{sq}, \psi_{rd}, \psi_{rq}, \omega_r\right]^T \\ y &= \left[i_1, i_2, i_3, \omega_r\right]^T \end{aligned}$$

and the parameter vector given by:

$$\theta = \left[R_s, R_{r,} L_{sl}, L_{rl}, L_m, J\right]^T$$

This is a dynamic model of the induction motor without magnetization saturation.

7.1.2 Model of the 5.5 kW motor with saturation

Unfortunately, the real world is more complex, because not all the parameters are constant. In real motors, the iron in the motor saturates, which means that the main flux ψ_m is a function of the scalar magnetization current $i_m = |i_s + i_r|$. To account for saturation, some rewriting of the above equations is necessary. In equation (7.3), the currents can be calculated as a function of the stator, the rotor, and the main flux.

$$\begin{aligned} i_s &= \frac{1}{L_{sl}} \cdot \left(\psi_s - \psi_m\right) \\ i_r &= \frac{1}{L_{rl}} \cdot \left(\psi_r - \psi_m\right) \end{aligned} \tag{7.6}$$

These currents can be inserted into the voltage equations (7.2), and thereby eliminate the currents.

$$\begin{aligned} \dot{\psi}_s &= -\frac{R_s}{L_{sl}} \cdot \left(\psi_s - \psi_m\right) + u_s \\ \dot{\psi}_r &= -\frac{R_r}{L_{rl}} \cdot \left(\psi_r - \psi_m\right) + j \cdot \omega_r \cdot \psi_r \end{aligned} \tag{7.7}$$

However, this set of differential equations does not express the main flux ψ_m by the states ψ_s and ψ_r. The main flux ψ_m can be expressed by ψ_s and ψ_r by inserting equation (7.6) into either of following expressions.

$$\begin{aligned} \psi_s &= L_{sl} \cdot i_s + L_m \cdot (i_s + i_r) \\ \psi_r &= L_{rl} \cdot i_r + L_m \cdot (i_s + i_r) \end{aligned} \tag{7.8}$$

Isolating ψ_m, gives the explicit expression of the main flux:

$$\psi_m = \frac{\left(\frac{1}{L_{sl}} \cdot \psi_s + \frac{1}{L_{rl}} \cdot \psi_r\right)}{\left(\frac{1}{L_m(i_m)} + \frac{1}{L_{sl}} + \frac{1}{L_{rl}}\right)} \tag{7.9}$$

In this expression, the inductance L_m is a function of the magnetization current i_m, which can be calculated from the actual motor currents.

$$i_m = \sqrt{(i_{sd} + i_{rd})^2 + (i_{sq} + i_{rq})^2} \tag{7.10}$$

The varying inductance is determined by the magnetization current according to equation (7.11), which has shown to be a good approximation in practice. L_{mo} is the inductance when the iron in the motor is not saturated and i_{mo} is the current at which saturation begins. Finally, α is a factor giving the decaying curve shape of L_m at high currents.

$$L_m = \begin{cases} L_{mo} & i_m \leq i_{mo} \\ L_{mo} \cdot \left(1 + \alpha \cdot L_{mo} \cdot i_m \cdot \left(\frac{1}{i_{mo}} - \frac{1}{i_m}\right)^2\right)^{-1} & i_m > i_{mo} \end{cases} \tag{7.11}$$

The differential equation governing the dynamic behavior of the motor is given by equation (7.7), in which ψ_m is given in an implicit manner given by the expressions (7.9), (7.10), (7.6), and (7.11). The value of ψ_m cannot be explicitly calculated for given values of ψ_s and ψ_r, but has to be approximated iteratively by a fixpoint calculation. The steps are first to assume a certain value for L_m and insert this into (7.9) to calculate ψ_m. Then calculate (7.6) to determine i_s and i_r. Next, calculate the scalar magnetization current i_m. Finally, a new value of L_m can be calculated from the lower branch of equation (7.11). The exact value of ψ_m has been reached if this value is equal to the value at the beginning of the iteration cycle.

In summary, the differential equations consist of (7.7) together with (7.4), and the currents for the output vector is given by (7.6). By comparison with the model without saturation, the input vector and the output vector are the same, but the structure of the differential equation is different. Thus, the parameter vector is now given by:

$$\theta = [R_s, R_r, L_{sl}, L_{rl}, L_{mo}, i_{mo}, \alpha, J]^T$$

Hence, two extra parameters have been introduced compared with the model of the 1.1 kW motor.

7.1.3 Performance criterion

The next step is to determine the performance criterion for the induction motor system. The result of the solution of the differential equations is the states $x = [\psi_{sd}, \psi_{sq}, \psi_{rd}, \psi_{rq}, \omega_r]^T$. From these states, the stator current can be calculated using equation (7.3). This complex stator current can now be converted to three phase currents comparable with real life currents. The performance criterion then becomes.

$$I\left(\hat{\theta}\right) = \int_0^T \left((i_1 - \hat{\imath}_1)^2 + (i_2 - \hat{\imath}_2)^2 + (i_3 - \hat{\imath}_3)^2\right) dt$$

Notice that the criterion does not include the squared deviation in the rotor's motion $(\omega_r - \hat{\omega}_r)^2$. This is excluded because motion measurements on the real motor contain more noise than the measurements of the currents.

In our experiments, the non-linear differential equations are approximated using the Fourth-order Runge-Kutta method [1]. One second of the motor's startup-phase was simulated using a step-size of 0.1 millisecond, i.e., 10000 steps. Hence, we used the sum-of-squared-errors as the performance criterion (fitness).

$$I'\left(\hat{\theta}\right) = \sum_{t=1}^{10000} \left((i_1(t) - \hat{\imath}_1(t))^2 + (i_2(t) - \hat{\imath}_2(t))^2 + (i_3(t) - \hat{\imath}_3(t))^2\right)$$

7.2 Algorithms

In this study, we compare eight algorithms taken from four main groups of stochastic search algorithms. The comparison includes a simple algorithm and an advanced algorithm from each group. Selecting the algorithms to include in a comparison is difficult, especially when considering the rather large number of new algorithms suggested every year. To ensure a broad comparison, we have chosen algorithms from the following four groups:

1. Local search algorithms.
2. Evolution strategies.
3. Generational evolutionary algorithms.
4. Particle swarm optimization algorithms.

These groups represent a wide selection of stochastic algorithms currently used for numerical optimization problems. The local search algorithms are included to allow comparison with traditional single-solution optimization techniques. The comparison includes steepest decent local search and simulated annealing from this group. The second group of algorithms is the evolution strategies, which represents one of the two main design approaches to evolutionary optimization. From this group, we selected the $(\mu{+}\lambda)$-evolution strategy and the $(\mu{+}\lambda)$-evolution strategy with self-adaptation of rotation angles. The latter was chosen because it is able to automatically discover dependencies between problem variables and self-adapt to exploit this knowledge. The third group "generational evolutionary algorithms" represents the other main evolutionary approach. The comparison includes the simple evolutionary algorithm and the recently introduced diversity-guided evolutionary algorithm from this group. Finally, the recently suggested approach known as particle swarm optimization is covered by the fourth group. From this group, we included the standard particle swarm optimization algorithm and the diversity-guided particle swarm optimization algorithm. The following sections describe these algorithms in greater detail.

7.2.1 Local search algorithms

The basic idea in local search algorithms is to iteratively improve a single solution by looking in its neighborhood and choose the most promising adjacent solution as a new candidate. A local search algorithm usually starts with a randomly generated solution and iterates until no improvement occurs or until some termination criterion is met, e.g., the maximal number of evaluations is reached or the solution is sufficiently good. The advantage of local search algorithms is their simplicity. The drawback is their difficulties with multimodal performance landscapes, because they have difficulties escaping local optima (deterministic local search cannot escape).

The steepest descent local search algorithm (SDLS) is probably one of the most frequently used local search algorithms. To apply this algorithm, the continuous search space of n-dimensional numerical problems needs to be discretized. This is done by setting a step-size δ_i for each problem variable x_i. The neighborhood $\mathcal{N}_{\mathbf{x}}$ of a solution $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is then usually defined as the set of solutions reachable from $\mathbf{x}$ by adding or subtracting *one* δ_i from $\mathbf{x}$, i.e., $\mathcal{N}_{\mathbf{x}} = \{(x_1 \pm \delta_1, x_2, \ldots, x_n), (x_1, x_2 \pm \delta_2, \ldots, x_n), \ldots, (x_1, x_2, \ldots, x_n \pm \delta_n)\}$. The neighborhood size is equal to $2n$ using this definition.

Simulated annealing (SA) is a variant of local search where the convergence criterion is relaxed to allow the algorithm to escape local optima. In SA, inferior solutions are accepted with a probability depending on the so-called temperature and the difference between the current solution and the new candidate solution. The probability of accepting inferior solutions decreases as the algorithm progresses. Simulated annealing is inspired by a process in thermodynamics. A perfect crystal can be grown by heating the material to a molten state and carefully lowering the temperature to prevent irregularities. The simulated annealing equivalent of the cooling procedure is the gradual decrease of the temperature and thus the probability of accepting inferior solutions. In the experiments, we used the algorithm defined by Michalewicz and Fogel [100, p. 120] with linearly decreasing temperature from T_{max} to T_{min}. For more information on SA and other local search techniques, see [100] and [34].

The parameters for the algorithms are listed in table 7.1.

7.2.2 Evolution strategies

Evolution strategies (ES) were introduced by Rechenberg and Schwefel in 1964-65 [112; 121]. The first version of the algorithm was a so-called (1+1)-ES, in

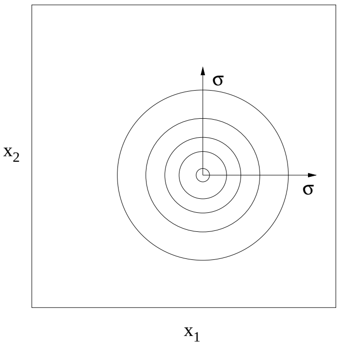
Algorithm	Parameter	Description	Value
SDLS	No parameters except step-sizes (see section 7.3)		
SA	MAX-TRIES	Max tries between change of temperature	10
	T_{max}	Start temperature	20
	T_{min}	End temperature	10

Table 7.1: Parameters for steepest descent local search and simulated annealing.

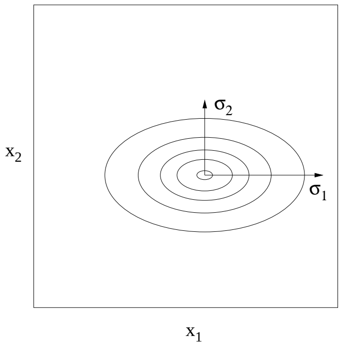
which one parent created one offspring by mutation. The offspring replaced the parent if it had a better fitness. In later studies, ES have been extended with a population of μ parents creating λ offspring using both mutation and recombination. The main difference between ES and the generational EAs is the selection procedure. Evolution strategies use deterministic selection whereas selection in generational EAs is probabilistic. Currently, two main selection strategies exist in ES; the $(\mu + \lambda)$-ES and the (μ, λ)-ES. In $(\mu + \lambda)$-ES, the μ parents create λ offspring. The next population is then formed by deterministically selecting the μ best individuals among the available $\mu + \lambda$ individuals. The number of offspring λ is usually less than the total population size, which gives an algorithm with overlapping populations. For this reason, $(\mu + \lambda)$-ES is sometimes referred to as a *Steady-state EA*. A typical setting is $\mu = 100$ and $\lambda = 15$, which will replace a maximum of 15 individuals in each iteration. The other strategy, (μ, λ)-ES, also generates λ individuals from the μ parents. However, in (μ, λ)-ES the parent population is not included in the source population in the selection procedure. The population in (μ, λ)-ES is therefore non-overlapping. Hence, λ must be larger than μ, because individuals are not cloned in ES[3]. The Gaussian mutation operator is the main component for creating new solutions in ES. In this context, most ES algorithms use self-adaptation to adjust the search process to the problem. The idea in self-adaptation is to encode algorithmic parameters in the genome and use these parameters to modify the individual. The hypothesis is that good solutions carry good parameters; hence, evolution discovers good parameters *while* solving the problem. Simple self-adaptation only encodes one variance σ for all problem variables (see Figure 7.1(a)). A more advanced self-adaptation scheme encodes one variance σ_i for each variable (Figure 7.1(b)). A third variant supports correlation between problem variables by encoding both variances σ_i and an additional set of rotation angles α_{ij} (Figure 7.1(c)). See [12, Section 6.4], for additional information on the self-adaptive mutation operator.

The comparison includes the simple self-adaptive $(\mu + \lambda)$-ES with one encoded variance σ and the complex version using self-adaptation of both variances σ_i and rotation angles α_{ij}. For comparison, all four evolutionary algorithms (the two evolution strategies and the two generational EAs) used the same crossover operator. The crossover operator had one weight for each variable, and all weights except one were randomly assigned to either 0 or 1. The remaining weight was set to a random value from the interval $[0, 1]$.

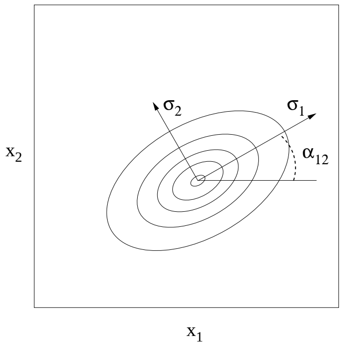
[3] A ratio of $\lambda/\mu \approx 7$ is recommended in [12, Section 6.4].



(a) Self-adaptation of one variance σ.



(b) Self-adaptation of all variances σ_i.



(c) Self-adaptation of variances σ_i and rotation angles α_{ij}.

Figure 7.1: Variants of self-adaptive mutation operator for evolution strategies. The boxes denote the search space. Ellipses are level-curves with equal mutation probability density.

The parameters for the two algorithms are listed in table 7.2.

Algorithm	Parameter	Description	Value
ES1 + ES2	μ	Parent population size	100
	λ	Offspring population size	15
	p_c	Prob. for crossover	0.5
	σ_0	Initial σ value	1.0
	τ_0	τ value for z_0 [12, Eq. 6.18]	2.0
	τ	τ value for z_i [12, Eq. 6.18]	2.0
	β	β value for z_i [12, Eq. 6.19]	0.0873
	ε_σ	Min σ value	1.0E-5

Table 7.2: Parameters for the two variants of the evolutionary strategy. The algorithms used the same values.

7.2.3 Generational evolutionary algorithms

The generational evolutionary algorithms (EAs) differ from evolution strategies on three aspects. First, the populations in the iterations are considered as a number of consecutive generations, i.e., the population at time 1 breeds the population at time 2. Hence, the populations are non-overlapping. Second, selection is stochastic rather than deterministic. Third, individuals are cloned in the selection process. Consequently, multiple copies of the better fit individuals are present in the population after selection. One of the first generational EAs was Hollands genetic algorithm [68], which used binary strings as encoding.

In this study, we included a simple evolutionary algorithm (SEA)[4] in the com-

[4]The term "standard evolutionary algorithm" is sometimes used in EA-literature. However, to the authors belief there is no such thing as a standard EA, because of the wide variety in available encodings and operators.

parison. The SEA encoded solutions as real-valued vectors and used Gaussian mutation with zero mean and annealing variance $\sigma^2 = 1/\sqrt{t+1}$ (t is the generation number). The mutation operator scaled the randomly generated numbers by 10% of the length of the search intervals to make the operator independent of the variables' ranges. Recombination was performed as described in the previous section. The next generation was selected using binary tournament selection. An elitist strategy keeping one individual was enforced to ensure that the best known solution survived to the next generation.

In addition to the SEA, we included the recently introduced diversity-guided evolutionary algorithm (DGEA) [140]. The DGEA is explained in full details in section 5.6.2. However, the DGEA was introduced with constant values for the lower and upper diversity thresholds d_{low} and d_{high}. Preliminary experimentation performed in this study showed that a simple linearly decreasing value significantly improved the performance of the algorithm. To this end, d_{lowmax} and d_{lowmin} was used to calculate d_{low} (likewise for d_{high}). Furthermore, forcing a switch to exploration if the fitness stagnated proved useful. This forced exploration is controlled by $d_{staglow}$ and $d_{stagmax}$. Generations with no fitness improvement are counted when the diversity is below $d_{staglow}$. An exploration phase is enforced if this counter reaches $d_{stagmax}$. A fitness improvement resets the internal counter.

The parameters for the SEA and the DGEA are displayed in table 7.3.

Algorithm	Parameter	Description	Value
SEA	ps	Population size	100
	p_m	Prob. for mutation	0.75
	p_c	Prob. for crossover	0.9
DGEA	ps	Population size	100
	p_m	Prob. for mutation	0.75
	p_c	Prob. for crossover	0.9
	λ	Mutation parameter	1.0
	d_{scale}	Scale factor	0.02
	d_{lowmax}	Start value for d_{low}	1.0E-5
	d_{lowmin}	End value for d_{low}	5.0E-12
	$d_{highmax}$	Start value for d_{high}	0.1
	$d_{highmin}$	End value for d_{high}	0.01
	$d_{staglow}$	Diversity for stagnation counting	0.0005
	$d_{stagmax}$	Max no. of stagnated generations	20

Table 7.3: Parameters for the simple evolutionary algorithm and the diversity-guided evolutionary algorithm.

7.2.4 Particle swarm optimization algorithms

Particle swarm optimization algorithms (PSOs) are partly inspired by the behavior of large animal swarms such as schooling fish or flocking birds. The main idea is to interpret each particle as a solution to the optimization problem and let these particles explore the search space. A good analogy is to imagine a swarm of moskitos being attracted to a lamp. The particles are the moskitos and the lamp is the best known optimum, which changes as better optima are discovered during the optimization procedure. Particle swarm optimization is a rather new technique that was introduced by Kennedy and Eberhart in 1995 [80].

In the standard particle swarm optimization algorithm (STDPSO), each particle i has a position ($\mathbf{x}_i$) in the search space, a velocity vector ($\mathbf{v}_i$), the position ($\mathbf{p}_i$) and fitness of the best point encountered by the particle, and index (g) of the best particle in the swarm. The particle's position represents the current solution to the numerical problem. The particle's next position ($\mathbf{x}'_i$) is determined by the velocity vector:

$$\mathbf{x}'_i = \mathbf{x}_i + \mathbf{v}_i \tag{7.12}$$

The velocity vector is updated according to the current velocity, the particle's own best position, and the overall best position in the swarm:

$$\mathbf{v}'_i = \chi(w\mathbf{v}_i + \overrightarrow{\varphi_{1i}}(\mathbf{p}_i - \mathbf{x}_i) + \overrightarrow{\varphi_{2i}}(\mathbf{p}_g - \mathbf{x}_i)) \tag{7.13}$$

where χ is the *constriction coefficient* [31], w is the *inertia weight* [124], and $\mathbf{p}_g$ is the position of the best particle in the swarm. The vectors $\overrightarrow{\varphi_{1i}}$ and $\overrightarrow{\varphi_{2i}}$ are randomly generated for each particle with entries uniformly distributed between 0 and φ_{1max} or φ_{2max}, respectively.

The diversity-guided particle swarm optimization algorithm (DGPSO) was introduced by Vesterstrøm and Riget [152]. It implements the same basic idea as the diversity-guided evolutionary algorithm. Like the DGEA, the DGPSO alternates between phases of exploitation and phases of exploration. Switching between the two phases is controlled by the same diversity measure as used in the DGEA (equation 5.15). The algorithm switches to exploration when the diversity drops below d_{low} and to exploitation when the diversity reaches d_{high}. Diversity is increased by using a different velocity update rule with negative sign on the attraction to the particle's own best position and best position of the entire swarm.

$$\mathbf{v}'_i = \chi(w\mathbf{v}_i - \overrightarrow{\varphi_{1i}}(\mathbf{p}_i - \mathbf{x}_i) - \overrightarrow{\varphi_{2i}}(\mathbf{p}_g - \mathbf{x}_i)) \tag{7.14}$$

Hence, equation 7.13 is used in exploitation and equation 7.14 in exploration. The thresholds for switching d_{low} and d_{high} were linearly decreased in the same way as with the DGEA, and stagnation counting was used to avoid long periods of stagnation.

The parameters for the two particle swarm optimization algorithms are shown in table 7.4.

7.3 Experiments and results

The main purpose of the experimentation was to compare the algorithms described in section 7.2 with respect to parameter identification of the two induction motors. Each algorithm was tested 20 times on the two motor identification problems. The algorithmic parameters listed in table 7.1-7.4 were found by manual trail-and-error tuning. The experiments are carried out using a simulated reference signal,

Algorithm	Parameter	Description	Value
STDPSO	ps	Population size	20
	φ_{1max}	Upper bound for φ_{1i}	2.0
	φ_{2max}	Upper bound for φ_{2i}	2.0
	χ	Constriction coefficient	1.0
	w_{max}	Start value for inertial weight w	0.7
	w_{min}	End value for inertial weight w	0.3
	v_{max}	Maximal velocity in each dimension	0.15
DGPSO	ps	Population size	20
	φ_{1max}	Upper bound for φ_{1i}	2.0
	φ_{2max}	Upper bound for φ_{2i}	2.0
	χ	Constriction coefficient	0.65
	w_{max}	Start value for inertial weight w	0.7
	w_{min}	End value for inertial weight w	0.3
	v_{max}	Maximal velocity in each dimension	0.15
	d_{lowmax}	Start value for d_{low}	1.0E-5
	d_{lowmin}	End value for d_{low}	5.0E-12
	$d_{highmax}$	Start value for d_{high}	0.1
	$d_{highmin}$	End value for d_{high}	0.01
	$d_{staglow}$	Diversity for stagnation counting	0.0005
	$d_{stagmax}$	Max no. of stagnated generations	20

Table 7.4: Parameters for the standard particle swarm optimization algorithm and the diversity-guided particle swarm optimization algorithm.

which is generated by the real motor's parameters, which have been experimentally determined by Grundfos (see below). The advantage of this setup is that the exact optimum is known, which allows us to compare the algorithms based on how close the found solutions are to the true optimum. Parameter estimation at Grundfos is considered satisfactory if the percentwise deviation is less than 5% from the true value, which is about the best precision obtainable on the real motor using traditional system identification techniques. In practice, this is done by conducting a series of experiments and calculating the motor's parameters directly. However, there is a 5-10% error in these parameters, because of the experimental setup. Hence, conducting an investigation on a simulated signal will allow us to evaluate the evolutionary method and decide if time and money should be invested in obtaining real data and performing the identification of the physical motors.

7.3.1 Identification of the 1.1 kW motor

Each algorithm was given 200000 evaluations of the fitness (equation 7.12) to find the parameters of the 1.1 kW motor. The two parameters L_{sl} and L_{rl} are linearly dependent and were therefore combined into one parameter. The 5-dimensional search space was discretized to make the search space similar for all algorithms, because the two local search algorithms require discrete search spaces. However, preliminary experiments showed that the PSOs had somewhat lower performance when using the discrete search space. To overcome this problem, the particles positions were kept as continuous variables and rounded just before calculating the fitness. Table 7.5 lists the reference values, the search intervals, and the step-sizes for the five parameters of the 1.1 kW motor.

	R_s	R_r	$L_{sl} + L_{rl}$	L_m	J
Ref. value	9.203	6.61	0.09718	1.6816	0.00077
Min	6	6	0.029	1.5	0.0001
Max	10	10	0.500	2.0	0.0100
Step	0.0001	0.0001	0.00001	0.0001	0.00001

Table 7.5: Intervals and step-sizes for the five parameters of the 1.1 kW motor.

The optimization results are shown in table 7.6. The table displays the number of runs where the exact optimum was found and the average, best, and worst fitness. As seen in the table, the local search techniques have quite poor performance indicating that the problem is highly multimodal and therefore quite difficult for local search algorithms. However, the relaxed termination criterion implemented in simulated annealing clearly improves the performance. Nevertheless, neither of the two algorithms managed to find the true optimum in any of the 20 test runs. In contrast, the two evolution strategy algorithms had very impressive performance. Even the simple algorithm found the true optimum in 16 out of 20 runs. The advanced evolution strategy with adaptation of rotation angles had even better performance. It consistently found a solution close to the true optimum (worst performance 1.20E-4), and in 17 of 20 runs this was the exact optimum. Regarding the generational EAs, the simple EA had somewhat intermediate performance. It discovered the true optimum in six runs, but was quite far from it in many cases. Interestingly, the simple EA (SEA) is considerably worse than the simple ES. However, further experimentation is necessary to determine whether it is the different selection mechanism, the self-adaptive mutation operator, or a combination of both that leads to the improved performance. Interestingly, the diversity-guided EA had the best performance of all eight algorithms. The algorithm discovered the true optimum in all 20 runs. To further test the algorithm, an additional 80 runs were performed to see if this was consistent. The true optimum was found in 99 out of the 100 runs. Conclusively, the simple extension implemented in this algorithm significantly improved the performance. Finally, the particle swarm optimization algorithms also showed good performance. The standard PSO found the true optimum as many times as the simple ES, but the algorithm is clearly not as robust. This is evident from the large average fitness, the standard error, and the worst fitness. The diversity-guided PSO had somewhat better performance compared with the standard PSO's. It has a lower average fitness and find the global optimum in 18 of 20 runs. Again, the standard error and the worst fitness indicate that the algorithm is not as robust as the evolutionary approaches.

To further analyze the performance, we calculated the average percentwise deviation from the true value of each variable. Table 7.7 lists the deviations for the eight algorithms. As shown in the table, the four evolutionary algorithms all had

Algorithm	# Exact	Avg. fitness	± std. err.	Best fitness	Worst fitness
SDLS	0	141620.61	± 24761.12	45777.88	365286.46
SA	0	34795.91	± 2069.26	13416.14	54441.81
ES1	16	3.91	± 2.13	0.00	26.06
ES2	17	1.80E-5	± 9.82E-6	0.00	1.20E-4
SEA	6	134.70	± 48.82	0.00	933.96
DGEA	20	0.00	± 0.00	0.00	0.00
STDPSO	16	226.69	± 133.87	0.00	2344.47
DGPSO	18	90.60	± 63.65	0.00	1081.88

Table 7.6: Results for parameter identification of the 1.1 kW motor. Average of 20 runs. The column denoted "# Exact" displays the number of runs where the algorithm discovered the exact optimum.

satisfying performance with respect to this criterion. In fact, the two evolution strategies and the diversity-guided EA managed to obtain an error of less than 0.2% on all parameters. The two PSOs had good performance, though somewhat worse than the evolutionary algorithms. In summary, the error percentage on the solutions obtained by the evolutionary algorithms are very encouraging because they are considerably lower than the 5% success criterion defined by Grundfos.

Algorithm	R_s	R_r	$L_{sl} + L_{rl}$	L_m	J
SDLS	14.49%	23.55%	116.46%	7.56%	83.44%
SA	8.52%	22.57%	166.74%	6.83%	50.06%
ES1	0.13%	0.08%	0.06%	0.11%	0.19%
ES2	0.0002%	0.0002%	0.00%	0.00%	0.00%
SEA	1.29%	0.91%	0.45%	1.01%	2.01%
DGEA	0.00%	0.00%	0.00%	0.00%	0.00%
STDPSO	1.16%	1.39%	0.64%	0.93%	0.97%
DGPSO	0.71%	1.07%	0.13%	0.26%	0.39%

Table 7.7: Average percentwise deviation for the parameters of the 1.1 kW motor.

The convergence speed is an important aspect of optimization because the fitness evaluation is usually the time-consuming part of real-world optimization. Figure 7.2 displays the number of evaluations versus the average fitness for the best performing algorithms. The graph is constructed from the runs where the algorithms found the exact optimum. For example, the graph for the simple EA is the average of the six runs where this algorithm found the true optimum. This approach is chosen to get a better picture of the convergence speed. The graph shows that the standard PSO clearly had the fastest convergence speed. On average, the true optimum was discovered in less than 10000 evaluations. The DGPSO had similar performance in the beginning of the runs, but a few runs with stagnation gave a somewhat slower average convergence. The evolution strategies converged faster than the generational EAs in the beginning of the optimization run, but were passed by the DGEA at approximately 60000 evaluations.

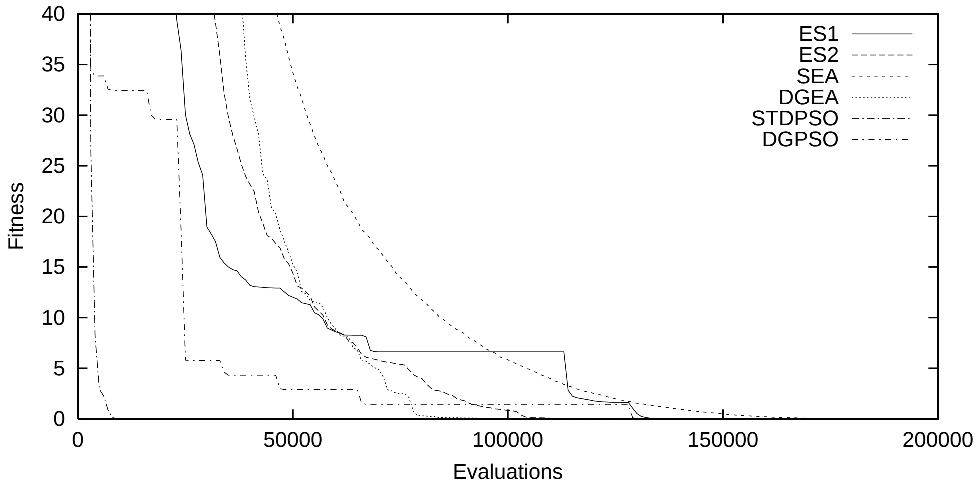


Figure 7.2: Number of evaluations versus average fitness (1.1 kW motor).

In summary, the experiments on the 1.1 kW motor show that considerable improvement in performance can be achieved by even rather simple extensions of the standard algorithms. This is the case for all four groups of algorithms. However, the very poor performance of the two local search algorithms underlines the advantage of using a population-based technique.

7.3.2 Identification of the 5.5 kW motor

In preliminary runs, the identification of the 5.5 kW motor with saturation turned out to be considerably more challenging than the simpler 1.1 kW motor. Therefore, each algorithm was given 300000 evaluations. In this problem, the two parameters L_{sl} and L_{rl} are independent and cannot be combined. Hence, eight parameters were identified. The 8-dimensional search space was again discretized. Table 7.8 displays the reference values, the search intervals, and the step-sizes for the eight parameters of the 5.5 kW motor. Furthermore, preliminary experiments on the DGEA and the DGPSO revealed that setting the parameter d_{lowmax} (table 7.3 and 7.4) to $1.0E-3$ gave better results.

	R_s	R_r	L_{sl}	L_{rl}	L_{mo}	i_{mo}	α	J
Ref. value	3.914	2.71	0.0358	0.0586	1.09	1.096	0.55	0.0084
Min	3.52	1.35	0.03	0.05	0.5	0.5	0.2	0.008
Max	4.30	4.06	0.10	0.10	2.0	2.0	1.0	0.009
Step	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001

Table 7.8: Intervals and step-sizes for the eight parameters of the 5.5 kW motor.

The optimization results are listed in table 7.9. As mentioned, the problem is significantly more challenging than the 1.1 kW motor's. None of the algorithms found the exact optimum in any run. Regarding the local search algorithms, both SDLS and SA had very poor performance except for one run where SA reached a fitness of about 35. Focusing on the evolution strategies, a surprising result emerged. Apparently, the simple ES outcompeted the advanced ES with respect to the average fitness. This is unexpected because the parameters modeling the saturation (L_{mo}, i_{mo}, and α) appeared to be highly correlated and the advanced ES is particularly designed to handle problems with correlated parameters. A possible explanation is that the correlation is difficult to discover from the fitness landscape. The simple self-adaptation scheme quickly found a mutation variance σ giving reasonable results, but the many variances and rotation angles in the advanced ES may have been too hard to find using the available number of evaluations. Despite of the higher average, the advanced ES actually managed to discover one solution close to the true optimum, which was not the case for the simple ES (best fitness near 5.66). For the generational EAs, the table shows a rather poor performance of the simple EA. The best solution was quite far from the optimum and it only obtained a fitness of about 257.80. Conversely, the diversity-guided EA had by far the best average. The best solution does not quite match the best solution of the advanced ES, but the average fitness and the fitness of the worst solution suggest that the DGEA is more robust. Finally, the particle swarm optimization algorithms had reasonable performance. The standard PSO roughly matches the simple evolution strategy with respect to best found solution. The average performance of both PSOs resemble the performance of the advanced ES. Interestingly, the diversity-guided PSO found the best solution of all algorithms. Again, the standard error and worst fitness indicate that the PSO algorithms may not be as robust as the evolutionary algorithms.

Algorithm	Avg. fitness	± std. err.	Best fitness	Worst fitness
SDLS	11342573.02	± 4908664.64	80315.85	75377366.52
SA	447887.98	± 199274.89	35.43	2867275.03
ES1	82.03	± 13.37	5.66	183.78
ES2	191.88	± 56.31	0.0256	1064.69
SEA	4224.58	± 964.28	257.80	16316.52
DGEA	12.34	± 3.39	0.1215	66.20
STDPSO	182.97	± 88.02	6.53	1441.64
DGPSO	163.25	± 39.73	0.0018	629.45

Table 7.9: Results for parameter identification of the 5.5 kW motor. Average of 20 runs.

Regarding precision of the found solutions, table 7.10 lists the percentwise deviation for the eight parameters. As seen in the table, none of the algorithms matched the 5% performance criterion on all parameters. The diversity-guided EA achieved less than 5% on six out of eight parameters and only slightly worse on the remaining two. Second best is the simple ES, which obtained less than 5% deviation on four parameters and about 5.5% on two parameters. The two PSOs obtained similar results also having four parameters with an error of less than 5% and two slightly above 5%. As with the 1.1 kW motor, the two local search techniques had rather poor performance.

Algorithm	R_s	R_r	L_{sl}	L_{rl}	L_{mo}	i_{mo}	α	J
SDLS	2.8%	20.3%	51.5%	19.3%	21.5%	21.1%	38.4%	5.8%
SA	2.9%	15.0%	77.4%	19.5%	20.3%	12.0%	33.4%	3.7%
ES1	0.3%	0.3%	10.0%	5.4%	2.6%	5.5%	13.2%	1.2%
ES2	0.1%	0.2%	17.8%	7.8%	5.9%	11.5%	26.8%	0.2%
SEA	0.7%	4.3%	32.1%	27.9%	6.5%	16.4%	42.7%	2.4%
DGEA	0.0%	0.1%	5.2%	2.4%	1.5%	3.1%	6.9%	0.0%
STDPSO	0.3%	0.3%	11.1%	6.3%	2.7%	5.8%	14.7%	1.1%
DGPSO	0.6%	0.4%	7.5%	5.9%	2.8%	5.7%	12.8%	2.1%

Table 7.10: Average percentwise deviation for the parameters of the 5.5 kW motor.

The convergence speed of the six population-based algorithms is illustrated in Figure 7.3. The graph shows the average fitness versus number of evaluations for all 20 runs. Again, the simple PSO converged rapidly, but in this case it was passed by the DGEA at about 25000 evaluations and by the simple ES near 150000 evaluations. Interestingly, the advanced ES converged faster than the simple ES. Finally, the DGPSO showed a convergence pattern similar to the one for the 1.1 kW motor.

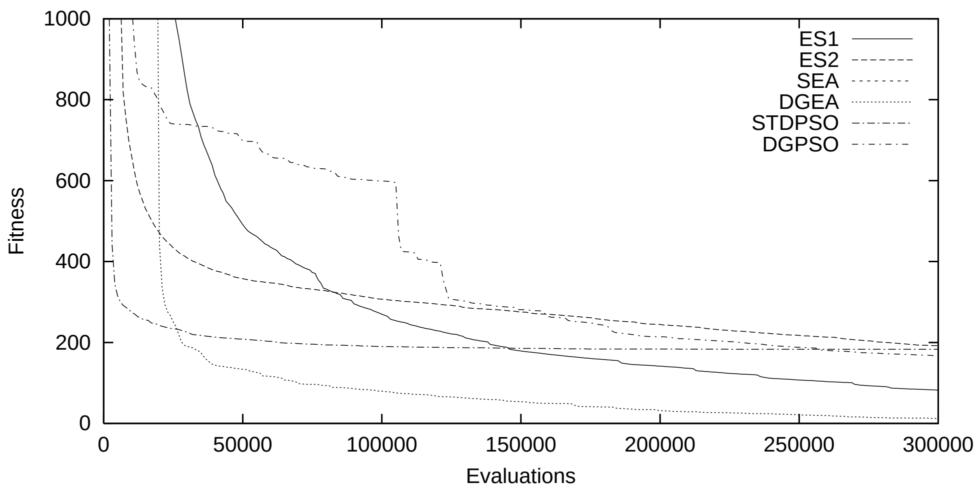


Figure 7.3: Number of evaluations versus average fitness for the best performing algorithms (5.5 kW motor).

To sum up, the experimentation on the 5.5 kW motor confirms the results on the 1.1 kW motor. First, simple extensions to the algorithms can have tremendous impact on the performance. This is the case for all four groups of algorithms, although it should be mentioned that the advanced ES had a lower average performance compared with the simple ES's. Despite of this, the advanced ES found better solutions than the simple ES. Second, the local search algorithms had very poor performance in comparison with the performance of the population-based approaches.

7.4 Summary

In this study, we have compared eight stochastic optimization algorithms with respect to parameter identification of two induction motors. The eight algorithms represent four main groups of stochastic optimization algorithms used today (local search, evolution strategies, generational EAs, and particle swarm optimization algorithms). From each group, we included a simple and an advanced algorithm. Comparing the algorithms, the diversity-guided EA clearly had the best performance. This algorithm showed the best average performance in both problems, and, for the 1.1 kW motor identification problem, it managed to locate the true optimum in 99 of 100 test runs. The local search techniques had the worst performance of all eight algorithms. This shows that both problems are highly multimodal and therefore difficult to handle with these techniques. The two evolution strategies also showed good performance on both problems. Interestingly, the simple ES had quite good average performance on both problems, and it found the true optimum for the 1.1 kW motor in 16 of 20 cases. The advanced ES showed robust performance on the 1.1 kW motor, and, with respect to the 5.5 kW motor, several runs found solutions that were better than those found by the simple ES. Regarding the PSOs, the standard PSO had very impressive convergence speed. On the 1.1 kW motor, the algorithm converged to a near-optimal solution in less than 10000 evaluations. Hence, a standard PSO may be a first choice for computationally intensive problems. Finally, the diversity-guided PSO had somewhat slower convergence, but the algorithm found good solutions to both problems. For the 1.1 kW motor, the true optimum was discovered in 18 of 20 runs, and for the 5.5 kW motor, the algorithm discovered the best solution of all eight algorithms.

In a wider context, the results show that stochastic optimization is indeed a feasible and promising approach to parameter identification of non-linear dynamic systems. Not surprisingly, the six population-based techniques clearly outcompeted the two local search techniques. Furthermore, the advanced algorithms had significantly better performance than the simple algorithms. Consequently, implementing the small extensions to the simple algorithms is indeed worth the effort. In contrast, most researchers working with practical applications only use the simplest text book variants of, e.g., evolutionary algorithms when approaching a real-world problem. In our view, this is due to three main factors. First, some experience with stochastic optimization is usually required to fully benefit from these techniques. Second, considerable time is often spent on implementing the performance criterion (e.g., a simulator) leaving limited time for the algorithm. Third, most novel algorithms are tested on rather simple and highly artificial benchmark problems that have little in common with real-world problems. This makes it difficult to find an algorithm that has shown good performance on a similar problem.

7.5 Future research

Regarding future work, a number of interesting issues can be investigated using the induction motors. A thorough examination of the results from the 5.5 kW motor revealed that the parameters in this problem are highly correlated. Somewhat surprising, the advanced ES did not appear to have a significant advantage on this problem, although this algorithm is specifically designed to handle problems with a high degree of parameter correlation. Hence, one research direction may be to investigate techniques for correlated problems such as differential evolution [127]. Additionally, the motor may be used to test novel techniques on a real-world problem.

Another important next step is to apply the best algorithms to parameter identification using real data obtained from the two motors. The impressive results obtained in this study certainly underline the strength of EAs on real-world problems, but additional tests should be carried out using real data.

Furthermore, the motors could be used to investigate different performance criteria. Typically, the sum-of-squared-errors measure is used in system identification. However, it might be the case that better prediction can be achieved with other performance criteria. Such criteria will probably not be in conflict with the traditional sum-of-squared-errors measure. Hence, it may be an advantage to have several performance criteria in play at the same time. For instance, one could use a subpopulation approach and have each subpopulation optimizing with respect to different performance criteria.

Besides parameter identification, the induction motors may be used for structural identification, which is about discovering both a model and its parameters. In this context, it would be quite interesting to investigate the prediction capabilities of genetic programming and compare with traditional techniques such as regression models.

Chapter 8

Case study: Direct control of a crop-producing greenhouse

In the evolutionary computation community, optimization of dynamic problems has been investigated over the past 15 years and numerous algorithms, operators, and extensions have been suggested to cope with such problems (for a survey, see [20]). However, most of this research has been, and still is, carried out on rather synthetic dynamic problems such as the moving peaks problem (example 3.1, section 3.3.3). To evaluate the soundness of the extensive amount of research carried out on dynamic optimization, Thiemo Krink, Mikkel T. Jensen, Zbigniew Michalewicz, and I scrutinized these artificial benchmark problems and found them to have little in common with realistic dynamic problems, especially with control problems [146] (see section 3.3.3). As a consequence, Thiemo Krink, Bogdan Filipič, and I initiated a study based on control of a crop-producing greenhouse. The main objective in this project is to examine basic issues in dynamic optimization by investigating algorithms on a realistic problem. The results presented in this chapter are described in three articles [88], [141][1] and [143].

From an optimization point-of-view, control problems appear to be particularly challenging type of dynamic problems, because of the feedback interactions between the controller and the controlled system. Furthermore, such systems may be affected by external factors; for example, some systems are relatively sensitive to changes in outdoor temperature and air humidity. This dynamic behavior poses an extra challenge to the optimization algorithm, which must be able to both find and track the optimum as the problem changes over time. Evolutionary algorithms (EAs) and particle swarm optimization (PSO) algorithms seem particularly well-suited for this task, because these algorithms keep a population of solutions instead of just one. Hence, the population will most likely contain a good solution after the problem has changed.

EAs are well-known algorithms and have successfully been applied to a wide range of problems. In contrast, PSO algorithms are rather new, and may require a brief introduction (for an overview of PSO, see [81]). PSO algorithms are partly based on the ideas of flocking behavior of large animal groups. The algorithms maintain a swarm of particles, where each particle represents a candidate solution

[1] First published as a conference paper [142].

to the given problem. Hence, the swarm in the PSO corresponds to the population in the EA and the particle to the individual. The main difference between EAs and PSO is the way new solutions are generated. EAs create new individuals by adding random noise (mutation) to an existing individual or by crossing over the chromosomes of two or more individuals (recombination). Conversely, each particle in a PSO algorithm "lives" and moves in the search space. The particle's position is updated using a velocity vector that is recalculated in every iteration. The new velocity vector is calculated using the current velocity vector, the best position encountered by the particle, and the position of the best particle in the swarm (see section 8.3.2 for details). Hence, PSO uses an explicit memory of previous best positions for each particle. The algorithm was originally developed for static optimization problems. For dynamic problems, certain modifications are necessary to ensure that the memory of the swarm is consistent with the true state of the problem. In this context, artificial dynamic problems have been used in a few investigations mainly focusing on how to update the memory [28; 39], and extensions of the update rule [17].

Regarding real-world control problems, EAs have successfully been applied to several design and tuning problems, e.g., [69; 47]. In contrast to studies on these offline controller problems, few investigations have been reported on using stochastic optimization techniques to directly control the system "online" by determining the control signal while the system is running. In online control, the simulator is repeatedly used to find the best control signals during the control period. Naturally, this approach heavily depends on the computational demand and accuracy of the simulator as well as the rate at which control signals must be provided. Hence, the approach is only feasible for rather slowly changing problems where the signal calculation may take several seconds or even minutes. An example is greenhouse control where the settings for heating, ventilation, CO_2 injection, and watering are updated every 15 minutes.

A straightforward approach to online control is to encode the control signals as a vector with one value for each control variable. This *direct control* strategy has been investigated in a few EA papers. Fogarty et al. compared the performance of a simple genetic algorithm to a PID controller with respect to control of a sugar beet press [50]. Vavak et al. combined a genetic algorithm with a variable range local search and used it to control the combustion in a multiple burner boiler [151]. Regarding PSO, only one investigation has been reported. Fukuyama et al. investigated a reactive power and voltage control problem with a standard PSO and compared it with the so-called reactive tabu search algorithm [56]. However, it should be noted that the control signal was only determined for one control point and not as a continuous series, which would be necessary to control a real system. Banga et al. investigated an EA-based control technique that encoded multi-valued strategies with M future points per control variable instead of one [15; 16]. The actual control signal at time t was then calculated by interpolating between the control points encoded in the genome. The technique was applied to a number of control problems in bio-chemistry.

The main contributions from this case study are i) knowledge of important issues in direct control, ii) a new PSO technique for dynamic problems, iii) a new local search technique, directed ascent local search (DALS), and iv) an improved understanding of the key issues in dynamic optimization. Our results are based on experimentation with direct control using a simple EA and the novel PSO and DALS algorithms.

8.1 Direct control

A control problem can be modeled by the interactions among the system being controlled, the surrounding environment, and the controller (see figure 8.1). Here, the vector $\mathbf{x}(t)$ represents the internal state of the system at time t, $\mathbf{v}(t)$ is the environment state, $\mathbf{u}(t)$ is the control signal from the controller, and $\mathbf{y}(t)$ is the output from the system. The environment state $\mathbf{v}(t)$ and the output $\mathbf{y}(t)$ from the system serve as feedback input to determine the control signal for the next time-step.

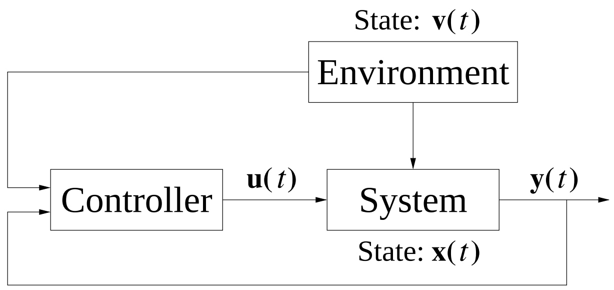


Figure 8.1: A model of the interactions among the controlled system, its environment, and the controller.

The change in system state is often modeled by a number of difference equations of the form:

$$x_i(t+h) = x_i(t) + \Delta x_i(\mathbf{u}, \mathbf{x}, \mathbf{v}, t, h) \tag{8.1}$$

where x_i is the i-th system variable in $\mathbf{x}$, $\Delta x_i(\cdot)$ is the update function for x_i, t is the time, h is the length of a time-step, and $\mathbf{u}$, $\mathbf{x}$, and $\mathbf{v}$ are the control signal, the system state, and the environment state of the previous time-step (sometimes several steps in the past). Real systems are often described by a system of non-linear differential equations. In these cases, an approximation method, such as Runge-Kutta, is used as the update function $\Delta x_i(\cdot)$.

The online control strategy used here is called "direct control" [50][2]. In direct control, each solution encodes the control signal as a vector of real-valued numbers. The best control signal for the current operating point is found by an optimization algorithm that evaluates candidate signals by simulating the real system for a number of time-steps into the future, which is called the prediction horizon (PH). The best signal from this optimization is then used to control the real system for a few time-steps, which is the control horizon (CH). The control horizon is typically set to one, i.e., $CH = 1$. Hence, a zero-order hold strategy is used [90]. For the

[2] In [50], the technique is called "direct optimal control"; however, "optimal" is a bit misleading, since optimality is not guaranteed.

determination of the control signal, only a limited amount of computation time (CT) is available, because the signal controlling the real system must be updated at certain time intervals. An important factor in this is the prediction horizon (PH), which is the number of time-steps simulated when evaluating a solution. The challenge in direct control is essentially to tune the optimization algorithm to exploit the available time in the best possible way. For EAs and PSO, this includes balancing the population size (ps) versus the number of generations (gen). The relationship between these factors is given by:

$$CT = ps \cdot gen \cdot PH \tag{8.2}$$

The product $ps \cdot gen$ is equal to the number of evaluations performed by the algorithm when the control signal for the next time-step is determined. For example, 200 evaluations can be assigned as either $ps = 200$, $gen = 1$ or $ps = 25$, $gen = 8$. The population size in directed ascent local search (DALS) is of course fixed to $ps = 1$, which implies that $gen = 200$. Thus, the only parameter to tune in DALS is essentially the neighborhood range. The pseudocode for the general direct control algorithm is listed in figure 8.2.

```
Direct control
  Initialize population of size ps
  while(control period not over) {
  Reset best control setting
    while (used_time < CT) {
      Generate new solutions
      Evaluate each solution for PH time-steps
      Store best control setting
    }
    Let best setting control for CH time-steps
  }
```

Figure 8.2: Pseudocode for the general direct control algorithm.

Figure 8.3 illustrates an abstract scenario where four control settings are evaluated with a prediction horizon of three time-steps. In each step, the best setting controls the system to the next time-step, i.e., $CH = 1$.

From a theoretical optimization perspective, direct control has the interesting property that the *search itself changes the fitness landscape*, which is because of the interactions between the controller and the system being controlled. In direct control, certain control signals may drive the system state in one direction whereas other signals may result in completely different system states. Hence, the *consecutive* fitness landscapes searched by the algorithms may be similar at time t but diverge as a result of the control signals. This introduces the possible occurrence of *optima in time* in addition to the usual *optima in space*. This concept is exemplified and further discussed in section 8.4.5.

The fundamental idea in direct control shares many properties with the engineering approach known as model predictive control (MPC), which covers a broad

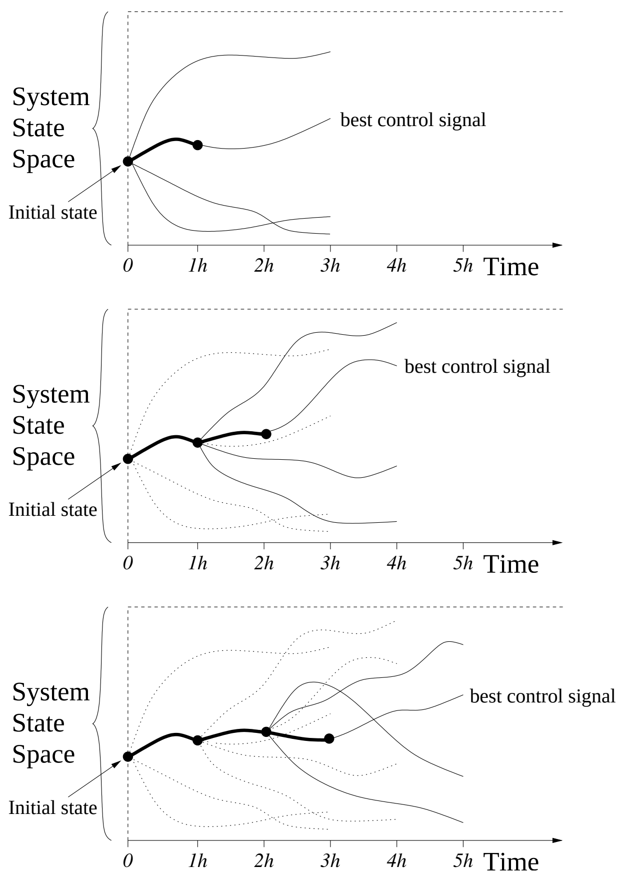


Figure 8.3: Example of state space exploration at simulation time $t = 0$, $t = 1$, and $t = 2$. Thin lines represent control settings exploration of the current time-step, thin dotted lines are previous explored control strategies, and thick lines are actual control as it was performed by the selected control setting.

selection of predictive control algorithms; for an overview, see [26]. All MPC algorithms explicitly use a predictor (a model) to calculate a future control sequence by minimizing an objective function. The various algorithms differ in the model structure and design of the objective function. Most MPC algorithms use a linear model and a quadratic objective function, since this allows the controller to find an analytic solution. However, many problems do not have linear characteristics. Non-linear systems are problematic since the objective function will typically no longer be unimodal, which rules out the use of analytic techniques. One approach is to linearize the system around the current control point. Here, the predictor is a linear combination of previous system inputs and outputs, and this linear combination is recalculated at each time-step. This approach is used in the popular generalized predictive control (GPC) method [30]. Linearization works well for many systems, but the linearization around the current operating point may not be sufficiently accurate on instable systems and on longer prediction horizons. In contrast, direct control is based on optimization of an arbitrarily complex model, which may lead to a multimodal optimization problem. Hence, direct control relies on the optimization algorithm's ability to handle problems with local optima.

Fortunately, EAs are rather good at this and, as discussed later, control problems may not have that many local optima.

8.2 Greenhouse model

The greenhouse producing crops (tomatoes) is modeled according to the outline in section 8.1. In our experiments, we used a simulator based on a description by Pohlheim and Heißner [106]. This section gives a brief overview of the rather complex simulator. A complete specification is available in appendix C and as a technical report [145]. It should be noted that our implementation differs from the original version [106] in a few minor points. In our version, the wind speed is added in the equations describing air exchange with the environment, which is done to model a non-hermetically closed greenhouse. Figure 8.4 shows the system, environment, and control variables describing the greenhouse model, and table 8.1 gives their names and units.

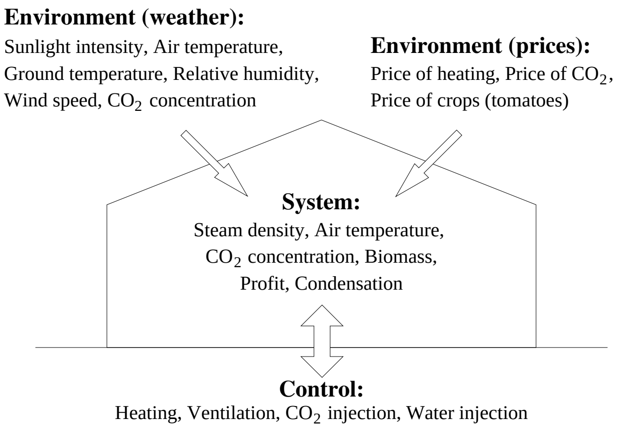


Figure 8.4: Variables in the model of a crop-producing greenhouse.

The greenhouse is controlled by heating u_{heat}, ventilation u_{vent}, injection of CO_2 u_{CO2}, and injection of water u_{water}. The ranges of these variables are: $u_{heat} \in [0, 150]$, $u_{vent} \in [0, 100]$, $u_{CO2} \in [0, 10]$, and $u_{water} \in [0, 100]$. The greenhouse state is updated according to six non-linear differential equations (one for each system variable, see appendix C for details) that are approximated with the fourth-order Runge-Kutta method. The length of a time-step h is 15 minutes and the initial state was $x_{steam} = 20$, $x_{atemp} = 20$, $x_{CO2} = 340$, $x_{biom} = 0$, $x_{profit} = 0$, and $x_{cond} = 0$.

The change in profit $\dot{x}_{profit}$ is equal to the income from the produced crops minus the expenses for heating and CO_2 (equation C.6 in appendix C).

$$\dot{x}_{profit} = \dot{x}_{biom} \cdot \mathrm{DWF} \cdot v_{Ptom} \cdot 10^{-3} - \frac{u_{CO2} \cdot v_{PCO2} \cdot 10^{-3}}{3600} - \frac{u_{heat} \cdot v_{Pheat}}{3600} \tag{8.3}$$

where DWF is the dry weight factor of the crop (the system variable x_{biom} models the dry weight of the crop).

	Description	Variable
System	Indoor steam density [g/m^3]	x_{steam}
	Indoor air temperature [°C]	x_{atemp}
	Indoor CO_2 concentration [ppm]	x_{CO2}
	Accumulated biomass [g/m^2]	x_{biom}
	Cumulative profit [DKK/m^2]	x_{profit}
	Condensation on glass [g/m^2]	x_{cond}
Environment	Outdoor sunlight intensity [W/m^2]	v_{sun}
	Outdoor air temperature [°C]	v_{atemp}
	Outdoor ground temperature [°C]	v_{gtemp}
	Relative humidity [% r.H.]	v_{RH}
	Wind speed [m/s]	v_{wind}
	Outdoor CO_2 concentration [ppm]	v_{CO2}
	Price of heating [DKK/(W·h)]	v_{Pheat}
	Price of CO_2 [DKK/kg]	v_{PCO2}
	Price of crops (tomatoes) [DKK/kg]	v_{Ptom}
Control	Heating [W/m^2]	u_{heat}
	Ventilation [m^3/(m^2 · h)]	u_{vent}
	CO_2 injection [g/(m^2 · h)]	u_{CO2}
	Water injection [g/(m^2 · h)]	u_{water}

Table 8.1: System, environment, and control variables of the simulated greenhouse. DKK denotes the currency "Danish Kroner".

The fitness of a solution $\mathbf{u} = [u_{heat}, u_{vent}, u_{CO2}, u_{water}]$ at time-step ts ($t = ts \cdot h$) is calculated as the profit achieved minus a penalty p. The penalty was introduced to avoid crop damage and to "guide" the indoor air temperature towards the optimal range for growth[3].

$$\begin{aligned}
Fit(\mathbf{u}, ts) &= \sum_{j=ts}^{ts+PH} \Delta x_{profit}(j) - p(j) \qquad (8.4)\\
\Delta x_{profit}(j) &\approx \int_{j\cdot h}^{j\cdot h+h} \dot{x}_{profit}\, dt\\
p(j) &= \begin{cases} 10 \cdot (16 - x_{atemp}(j)) & x_{atemp}(j) < 16\\ 10 \cdot (x_{atemp}(j) - 35) & x_{atemp}(j) > 35\\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

where $\approx$ is the Runge-Kutta approximation.

[3]Introducing an optimal range of growth turns the problem into a constraint problem. In this study, we use a simple penalty approach although other techniques may yield a better performance. However, the main focus in this study is on dynamic optimization and not constraint optimization.

Real weather data recorded in 2000 at the Aarslev measuring station on the Danish island Fyn was used for the environment variables sunlight intensity v_{sun}, outdoor air temperature v_{atemp}, outdoor ground temperature v_{gtemp}, relative humidity v_{RH}, and wind speed v_{wind}. The remaining environment variables were kept constant at $v_{CO2} = 340$, $v_{Pheat} = 0.0002$, $v_{PCO2} = 4.0$, and $v_{Ptom} = 12.0$. The weather records were provided by the Danish Meteorological Institute; see appendix C for further information. In this study, we simulated the greenhouse operation in the first week of May 2000. The weather data are illustrated in figure 8.5. As stated earlier, direct control determines a solution's performance by simulating a number of time-steps into the future. In practice, this includes simulating the weather, or rather, predicting the weather within the prediction horizon. Weather prediction is generally difficult, but a simple scheme is to assume that the weather does not change during the prediction horizon (few hours). This approach was used in our study.

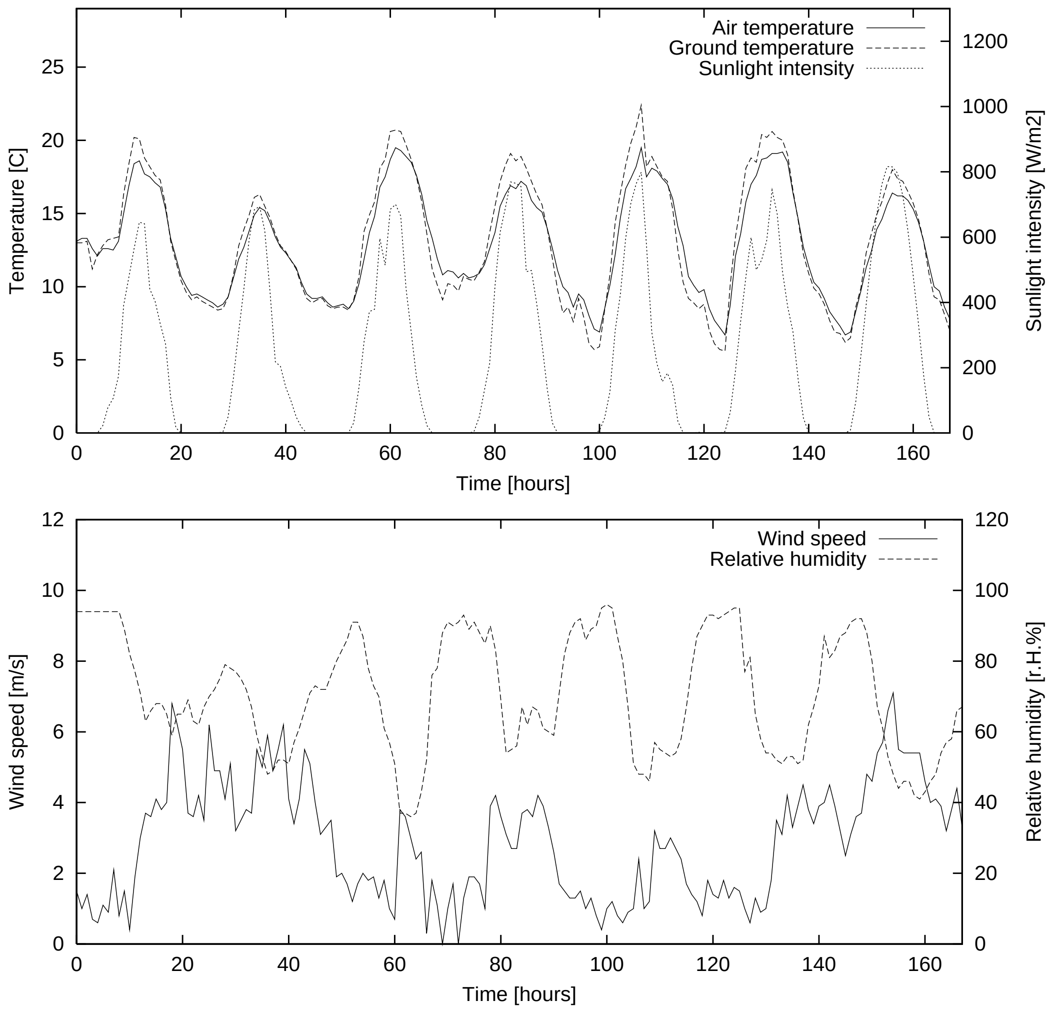


Figure 8.5: Weather data for the first week of May 2000 as used in the numerical experiments. Note that CO_2 concentration was assumed constant at 340 ppm.

8.3 Algorithms

8.3.1 Evolutionary algorithm

The EA used in this study encoded the control signals as real-valued vectors. New solutions were created using Gaussian mutation and a variant of arithmetic crossover where each variable has its own weight. All weights except one were randomly assigned 0 or 1, and the remaining weight was set to a random value from the interval $[0, 1]$. Binary tournament selection was applied to select the next generation. The algorithm used the following parameters: probability of crossover $p_c = 0.9$, probability of mutation $p_m = 0.5$, and variance $\sigma = 0.01$, which was scaled by the length of each control variable's interval. Each solution was evaluated by simulating PH time-steps using the control setting encoded in the genome.

8.3.2 Particle swarm optimization algorithm

In particle swarm optimization, each particle i has a position $(\mathbf{x}_i)$ in the search space and a velocity vector $(\mathbf{v}_i)$. Furthermore, each particle stores the best position $(\mathbf{p}_i)$ encountered and its fitness, and the index (g) of the best particle in the swarm. The next position $(\mathbf{x}'_i)$ of the particle is determined by the current position and the velocity vector:

$$\mathbf{x}'_i = \mathbf{x}_i + \mathbf{v}_i \tag{8.5}$$

The velocity vector is updated according to a weighted sum of the current velocity, the particle's own best position, and the overall best position in the swarm:

$$\mathbf{v}'_i = \chi(w\mathbf{v}_i + \overrightarrow{\varphi_{1i}}(\mathbf{p}_i - \mathbf{x}_i) + \overrightarrow{\varphi_{2i}}(\mathbf{p}_g - \mathbf{x}_i)) \tag{8.6}$$

where χ is the *constriction coefficient* [31], w is the *inertia weight* [124], and $\mathbf{p}_g$ is the position of the best particle in the swarm. The vectors $\overrightarrow{\varphi_{1i}}$ and $\overrightarrow{\varphi_{2i}}$ are randomly generated for each particle with values uniformly distributed between 0 and 1 (resampled for each dimension).

The application of PSO algorithms to dynamic problems is not as straightforward as with EAs, because the particles' memories need to be updated since the fitness of a solution may change. Modified versions of PSO for dynamic optimization have been investigated using artificial benchmark problems. Carlisle and Dozier replaced the particles own best position $(\mathbf{p}_i)$ by its current position, either at fixed intervals or when a change was detected [28]. Eberhart and Shi reevaluated the memory and used a random value between 0.5 and 1.0 as the inertia weight w every time a particle was moved [39]. Blackwell and Bentley extended the update rule for the velocity vector (equation 8.6) to introduce a repulsion effect between particles and thereby maintain diversity in the swarm [17].

In this study, we reevaluate each particle's own best position before each optimization period of the problem (inner loop in figure 8.2). However, the greenhouse control problem appeared to be particularly challenging to the PSO because the night-phase is nearly static whereas the day-phase is highly dynamic. This was problematic for the PSO because the speed of the particles at the end of the night-phase was close to zero. Hence, it was difficult for the converged swarm to spread out and track the optimum during the dynamic day-phase. To overcome this problem, we randomly initialized the velocity vectors for 80% of the particles before each optimization period (the 80% reinitialization was determined in preliminary experiments).

8.3.3 Directed ascent local search

The directed ascent local search (DALS) was specifically designed to efficiently track the optimum in a changing fitness landscape. In DALS, the direction to the previous solution determines the traversal order of the current solution's neighborhood. The first solution encountered with a better fitness becomes the new current solution. Thus, only a limited part of the immediate neighborhood is evaluated, in fact, only one solution per iteration as long as improvements occur. Figure 8.6 illustrates a situation where the current solution was obtained by a move to the right in a two-dimensional search space. The first solution examined is then the one to the right of the current one (indicated by 1 in the figure). The remaining neighboring solutions are examined in increasing order as shown in the figure.

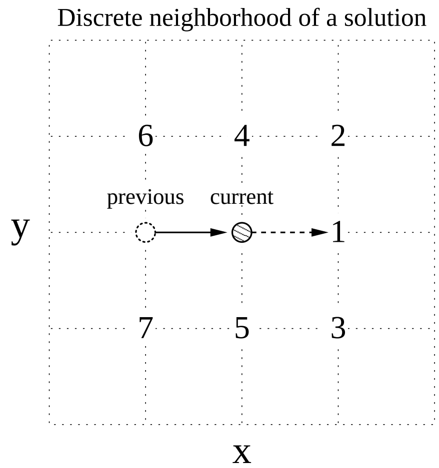


Figure 8.6: An example of the traversal order in directed ascent local search for a 2-dimensional problem.

For higher dimensions, the algorithm evaluates neighboring solutions in a slightly complex manner. The traversal algorithm is listed in figure 8.7. Here, k is the dimension of the last move, d_k is the direction of the move (± 1), s_k (or s_j) is the step-size for dimension k (or j), X is the current solution, and N is the dimensionality of the problem (in figure 8.6, $k = 1$ and $d_1 = 1$). The operator $\oplus$ denotes vector addition. The traversal algorithm stops when a better solution is found; for clarification, this is omitted in figure 8.7. The algorithm traverses the neighborhood as follows: First, the solution in the current direction is evaluated. Then the algorithm enters a nested loop. In the outer loop, it iterates along the dimension of the last move k starting at the solution in the current direction and going backwards. The inner loop examines the solutions adjacent and orthogonal to the move-dimension. For the simple example in figure 8.6, the algorithm traverses the neighborhood as follows: First, solution 1 in the current direction is evaluated.

Then the algorithm enters the outer loop with $i = 1$. In the inner loop, solutions 2 and 3 are checked. Now, $i = 0$ and solutions 4 and 5 are examined in the inner loop. Finally, with $i = -1$ solutions 6 and 7 are tested.

DALS traversal
check $X \oplus [0, \ldots, d_k \cdot s_k, 0, \ldots, 0]$
for(i=1; $-1 \leq i$; i−−) {
  for(j=1; $j \leq N$; j++) {
    if(j!= k) {
      check $X \oplus [0, \ldots, +s_j, 0, \ldots, i \cdot d_k \cdot s_k, 0, \ldots, 0]$
      check $X \oplus [0, \ldots, -s_j, 0, \ldots, i \cdot d_k \cdot s_k, 0, \ldots, 0]$
    }
  }
}

Figure 8.7: Pseudocode for the DALS traversal algorithm.

8.4 Experiments and results

The objective of the experiments was to study dynamic optimization by investigating aspects of direct control of a realistic control problem – in this case a crop-producing greenhouse. The main issue in EA and PSO-based direct control is to balance the three factors of equation 8.2. In our first study, we focused on the EA [141]. In this, the effect of varying the prediction horizon was investigated. Additionally, we examined the balance between population size (*ps*) and generations (*gen*) and tested several trade-off settings for the EA. In our second study, we extended the experiments on trade-offs to include an additional extreme setting for the EA [143]. A similar set of experiments were performed on the novel PSO algorithm. For the DALS algorithm, we investigated a number of step-sizes for the control variables. In all tests, we repeated each experiment 30 times and calculated the average profit per time-step.

8.4.1 Prediction horizon

In our first study, we investigated the effect of varying the prediction horizon. We tested six horizons having 1, 2, 3, 4, 8, and 20 time-steps. Figure 8.8 shows the cumulative profit from the 1, 2, 4, and 20 horizons[4] using the trade-off with population size $ps = 10$ and generations $gen = 20$, which was chosen because the other experiments in our first study showed that this was the best setting. A prediction horizon of 20 time-steps is the best, though only marginally better than a horizon of 8 steps. The profit achieved in the remaining four horizon cases decreases according to the look-ahead. Hence, a prediction horizon of at least 8 steps yields high profit, a horizon of 4 steps leads to a slightly lower profit, and only a few steps give rather low profit. The explanation for the significant difference between

[4]To keep the graph readable, 3 and 8 are not shown.

the worst performing setting ($PH = 1$) and the best setting ($PH = 20$) is found by examining the control signals. Figure 8.9 displays ventilation (u_{vent}) and CO_2 injection (u_{CO2}) for $PH = 1$ and $PH = 20$. The graph on ventilation shows that two general control strategies exist. The first strategy is used when $PH = 1$. Here, the EA sets ventilation high at daytime. This will require large investment in heating during night, but exploits the free CO_2 in the environment. The second strategy appears when $PH = 20$. In this strategy, ventilation is low, which saves some heating, but makes CO_2 injection necessary. The additional profit achieved by the 20-step controller is mainly related to the achieved temperature and indoor CO_2 level (equation C.4 in appendix C), which can be seen by thoroughly examining the control signals and greenhouse states of both settings. Naturally, the two different control strategies emerge as a result of the prediction horizon, but here the CO_2 level plays an important role too. The second strategy appears because the long look-ahead allows the controller to discover the long-term effect of growth, i.e., that the photosynthesis can transform more CO_2 than achievable by ventilation alone. Hence, additional growth is possible by injecting additional CO_2. The short look-ahead of the first strategy does not allow the controller to discover the long-term effects. Thus, ventilation is used because it will provide free CO_2 from the environment.

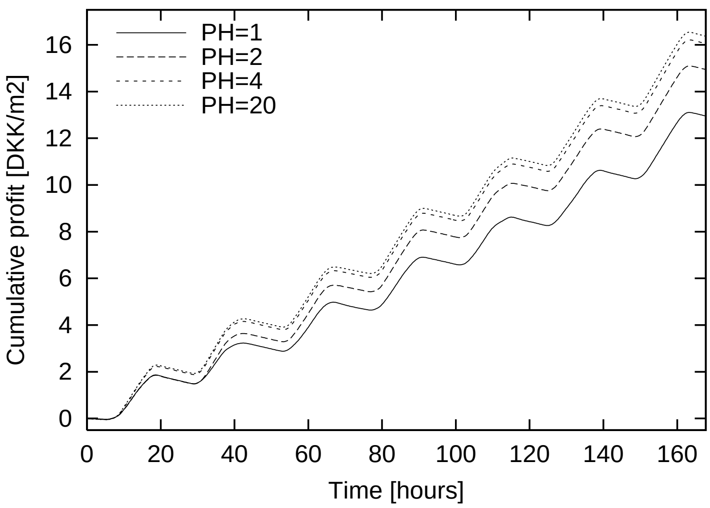


Figure 8.8: Profit per m^2 for different prediction horizons with 10 individuals and 20 generations. Average of 30 runs.

8.4.2 Population size versus generations

In the experiments on balancing the population size versus generations, we fixed the computation time to $CT = 800$ units[5]. Our investigation on the prediction horizon revealed that four time-steps ($PH = 4$), which is one hour, is sufficient to

[5] We use an abstract measure because in a real-world scenario the computation time is dependent on the available hardware of the specific greenhouse controller. A computation time of $CT = 800$ was chosen to allow a sufficient number of factorizations according to equation 8.2.

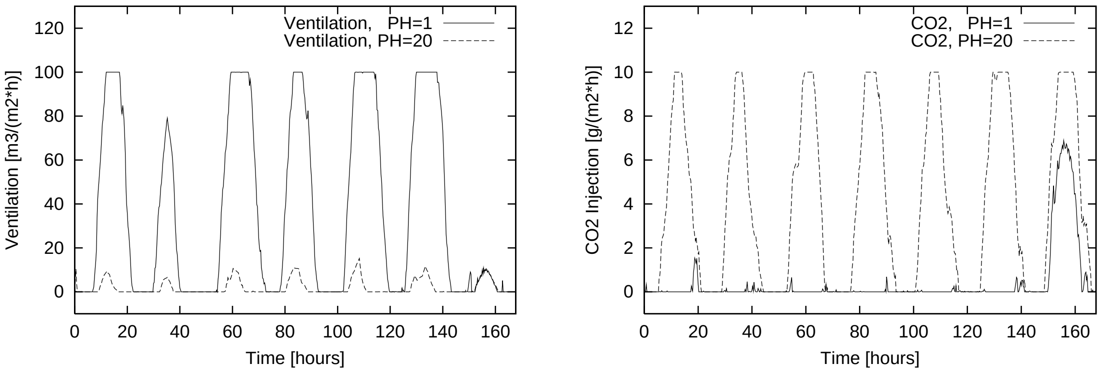


Figure 8.9: Example of ventilation and CO_2 injection for different prediction horizons (10 individuals, 20 generations).

obtain good performance[6]. Using equation 8.2, the number of evaluations ($ps \cdot gen$) is equal to $\frac{800}{4} = 200$.

In the experiments with the EA, we extended the previously investigated trade-offs [142] to include a setting with 1 individual optimizing for 200 generations. This extreme setting basically turns the algorithm into a simple (1+1)-Evolution Strategy without adaptation of the mutation variance (for reading on Evolution Strategies, see [114]). The tested trade-offs (ps,gen) were (200,1), (100,2), (50,4), (25,8), (10,20), and (1,200). Figure 8.10 illustrates the cumulative profit obtained by the tested settings. The graphs clearly show that the available computation time is best invested by having a low population size with many generations. In fact, the extreme setting (1, 200) slightly outperformed our previous best setting (10, 20), which was used in the experiments on the prediction horizon. Interestingly, the use of many generations essentially transforms the dynamic control problem into a series of related static problems.

Regarding the experiments with the PSO, we performed a set of experiments similar to those carried out with the EA. The extreme setting with 1 particle and 200 generations was not tested because the interaction between particles is a fundamental part of the PSO. Instead, we included the setting (5,40). Figure 8.11 displays the results for the tested trade-offs. Apparently, the PSO seems to be much less sensitive to the number of generations. The performance of the trade-offs is almost equivalent when the algorithm has at least four generations to find the control signal. The very poor performance of the (200,1)-setting was because the velocity vector was randomly set before each static period of the problem. In this scenario, the PSO does not have time to find a signal yielding reasonable results.

8.4.3 Step-sizes in local search

In the experiments with the DALS, we focused on comparing a number of step-sizes for the control variables. The step-size defines the neighborhood and thus

[6]Four steps were used in the experiments, because it halves the execution time compared with eight steps.

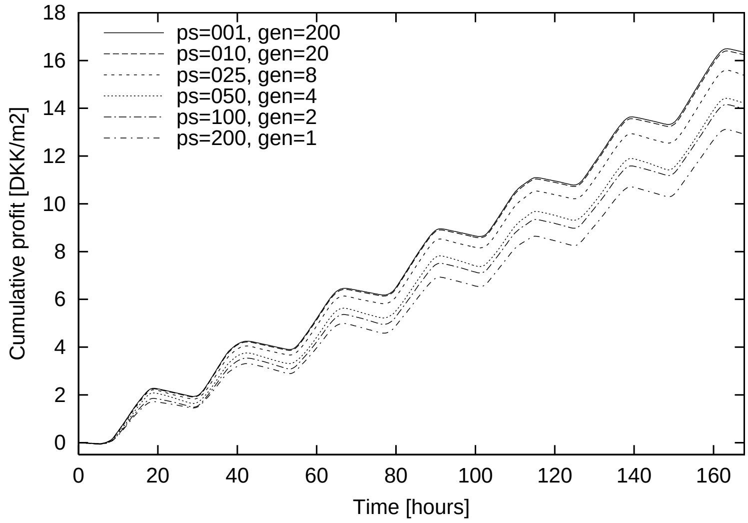


Figure 8.10: Cumulative profits for EA trade-offs between population size and number of generations. The results are averaged over 30 runs.

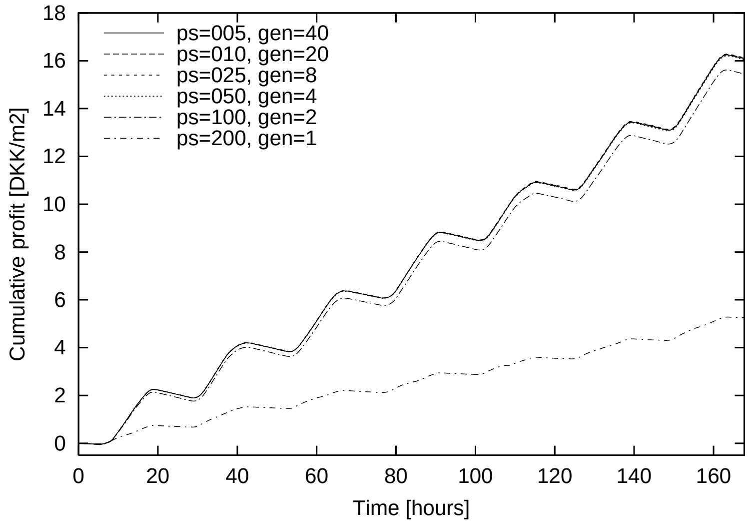


Figure 8.11: Cumulative profits for PSO trade-offs between population size and number of generations. The results are averaged over 30 runs.

how "far" the algorithm can explore the fitness landscape using 200 evaluations. The trade-off in this algorithm is between exploration distance and fine-tuning capabilities. A large step-size will allow the algorithm to obtain a solution far from the current one. On the other hand, a large step-size limits the fine-tuning capabilities. Each step-size setting consists of four values, one per control variable. The six tested step-sizes are listed in the legend of figure 8.12, in which the order of the control variables is (u_{heat}, u_{vent}, u_{CO2}, u_{water}). For instance, the step-size setting (10.0, 10.0, 1.0, 10.0) sets the step-size of u_{heat}, u_{vent}, and u_{water} to 10.0 while u_{CO2} is changed in steps of 1.0. The figure clearly shows that a large step-size is favorable. Hence, rapidly changing the value of a variable is significantly more important than fine-tuning it.

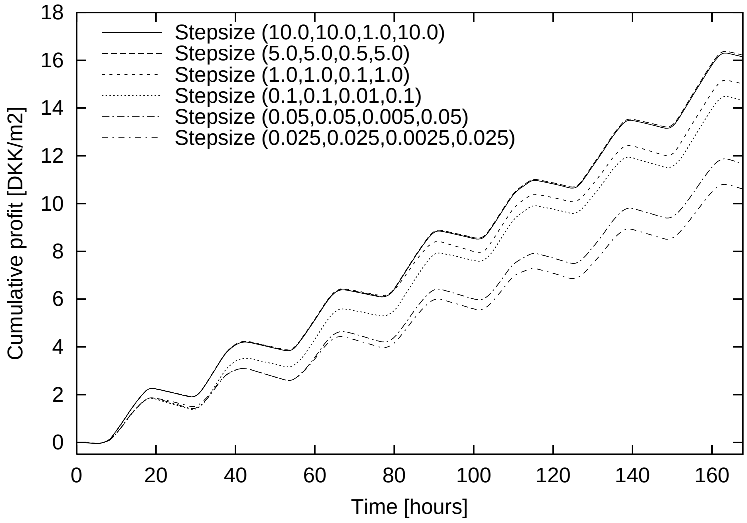


Figure 8.12: Cumulative profits for DALS trade-offs on step-size. The results are averaged over 30 runs.

8.4.4 Comparison of algorithms

For comparison, figure 8.13 shows the results obtained with the best setting for each of the three algorithms. The graph denoted 'RS, #Ev=200' illustrates the performance of a random search algorithm that, at each time-step, picks the best of 200 randomly generated solutions. This algorithm was included to give a lower bound of the performance. Unexpectedly, the three algorithms had nearly matching performance when properly tuned. A thorough examination of the data revealed that the EA was slightly better than the DALS, which was marginally better than the PSO. Somewhat surprisingly, the random search had actually a quite good performance. This and the experiments on tuning the three algorithms suggest that the problem is in fact rather easy once it is treated as a series of related static problems.

8.4.5 Analysis of control signals

The reasons for the significant differences between the various settings for each algorithm can be found by analyzing the control signals over time.

For the EA, figure 8.14 shows the signals u_{heat} and u_{CO2} for the best (1, 200) and the worst setting (200, 1). The difference in performance is closely related to these variables, because profit is easily lost by sub-optimal control of heating and CO_2 injection. At night the temperature drops, which requires heating to prevent that the crops are damaged. At daytime the sunlight permits growth, which can be augmented by injecting additional CO_2. The best control strategy (figure 8.14, left graph) properly adjusted the control to follow the day and night phases. The worst strategy that we found failed to turn off the heating at daytime, and valuable CO_2 was wasted during the night because the absence of sunlight makes growth impossible. Despite of a large population size and thus supposedly good coverage of the search space, this strategy was simply not able to follow the landscape. This is

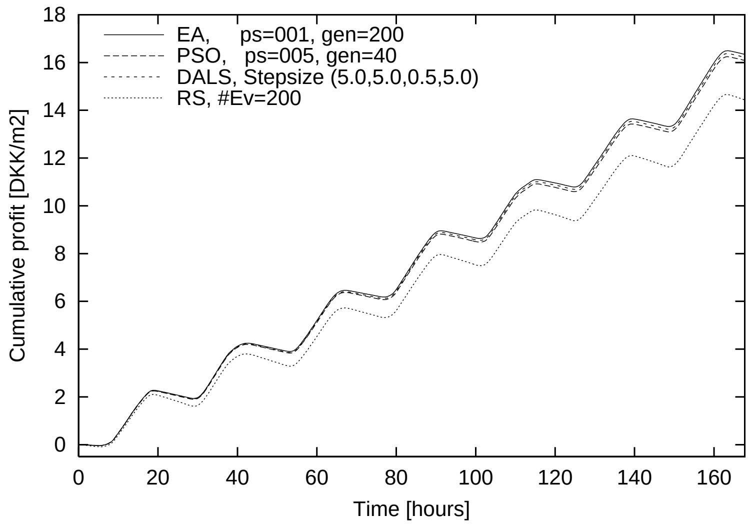
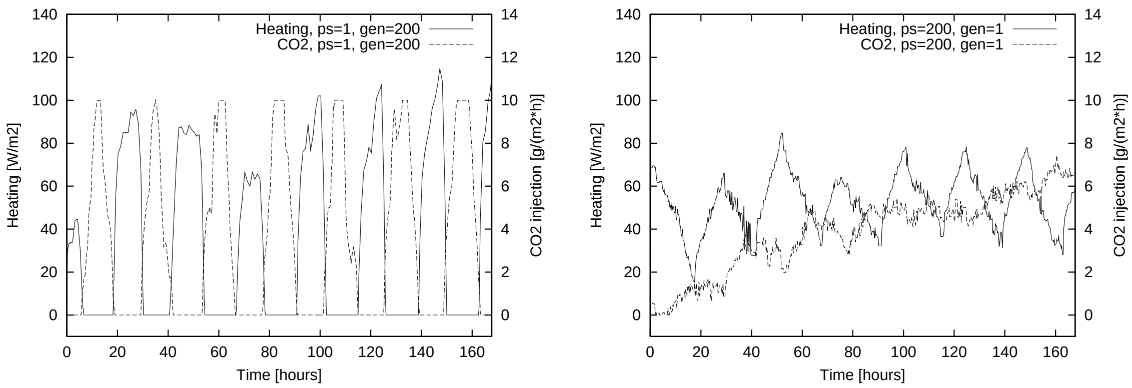


Figure 8.13: Cumulative profits for best algorithm settings. 'RS, #Ev=200' denotes random search with 200 evaluations at each time-step.

evident from the right graph in figure 8.14, where u_{heat} and u_{vent} appear noisy and are not changing as quickly as in the best strategy. Conclusively, a large population size is not an advantage versus having many generations between updates. The best and worst performing PSO setting lead to similar results regarding the control signals, and are thus not shown.



(a) Best EA control. (b) Worst EA control.

Figure 8.14: Control values for heating and CO_2 injection for best EA control (left graph) and worst EA control (right graph).

To further examine the control strategies, we plotted the partial fitness landscapes at each time-step. Note that the fitness (equation 8.4) is plotted on these graphs, and not the cumulative profit. The landscapes were obtained by fixing two of the four control variables to the values encoded in the population's best solution and then plotting the fitness for combinations of the remaining two variables. Figure 8.15 illustrates two example landscapes for u_{heat} versus u_{CO2} and u_{vent} versus u_{CO2}. The landscapes were very simple in all cases, which confirm our earlier results on rudimentary versions of the simulator [146; 88].

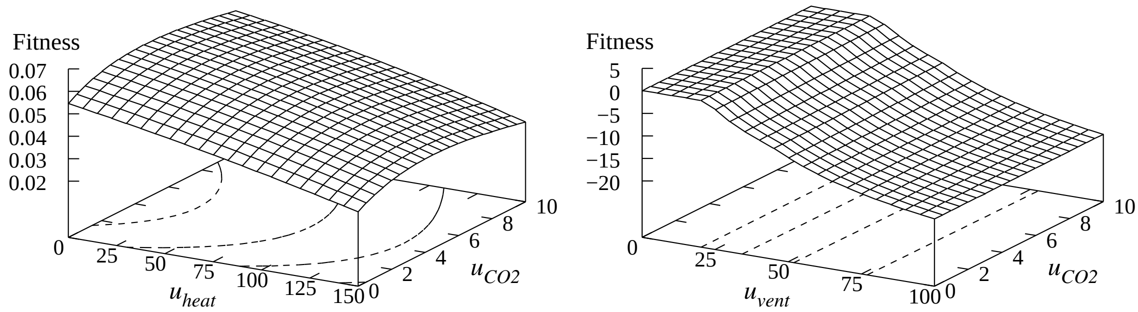
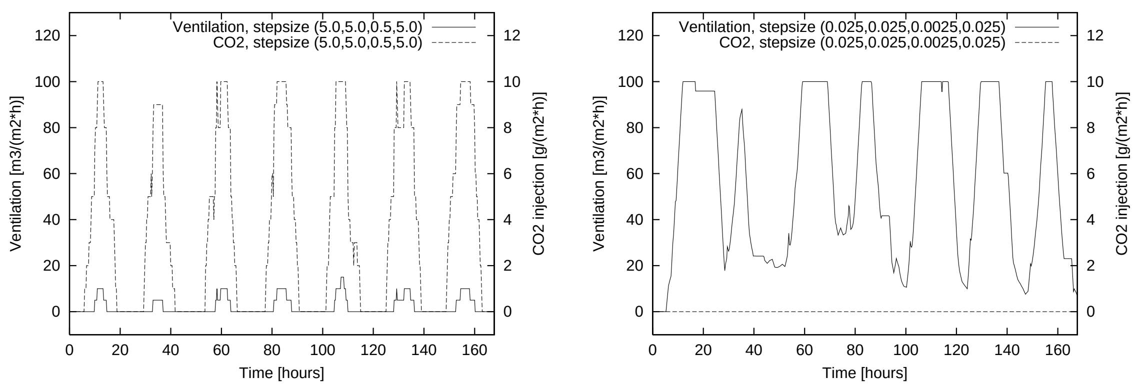


Figure 8.15: An example of partial fitness landscapes at time $t = 7.75$. The landscapes are obtained by fixing the remaining two control parameters to the current best values.

Regarding the DALS, a surprising result appeared when analyzing the control signals. Unexpectedly, two very different control strategies emerged. Figure 8.16 displays ventilation (u_{vent}) and CO_2 injection (u_{CO2}) for the best and the worst step-sizes. The worst performing strategy (small step-size) ventilated a lot but refrained from injecting expensive CO_2. The advantage of ventilating is that CO_2 is obtained for free from the environment. Conversely, high ventilation requires some heating at night, which is expensive. The best performing strategy (large step-size) was more or less the opposite. In this case, the algorithm kept ventilation at a minimum and injected much CO_2. The artificially augmented CO_2-level allowed a particularly high growth rate and thus increased profit despite of the expenses for CO_2. In addition to the increased growth, this strategy saved heating, which further improved the profit.



(a) Best DALS control.

(b) Worst DALS control.

Figure 8.16: Control values for ventilation and CO_2 injection for best DALS control (left graph) and worst DALS control (right graph).

Interestingly, the two strategies correspond to the two strategies found in our first study with an EA where the two strategies appear for two different prediction horizons (section 8.4.1). Changing the prediction horizon essentially alters the fitness definition and thus the landscapes searched by the algorithm. In the experiments on DALS, the two strategies emerged because of the *algorithm* and not as a result of the *fitness definition*. This is particularly interesting because it shows that the influence of the search procedure may actually lead to a completely different control strategy. A careful examination of the fitness landscapes (figure 8.17) revealed that the worst performing strategy induced a local optimum near $u_{vent} = 40$ and $u_{CO2} = 0$ at time $t = 8.75$ (figure 8.17, third row, left graph), which was absent for the best performing strategy at this time (figure 8.17, third row, right graph).

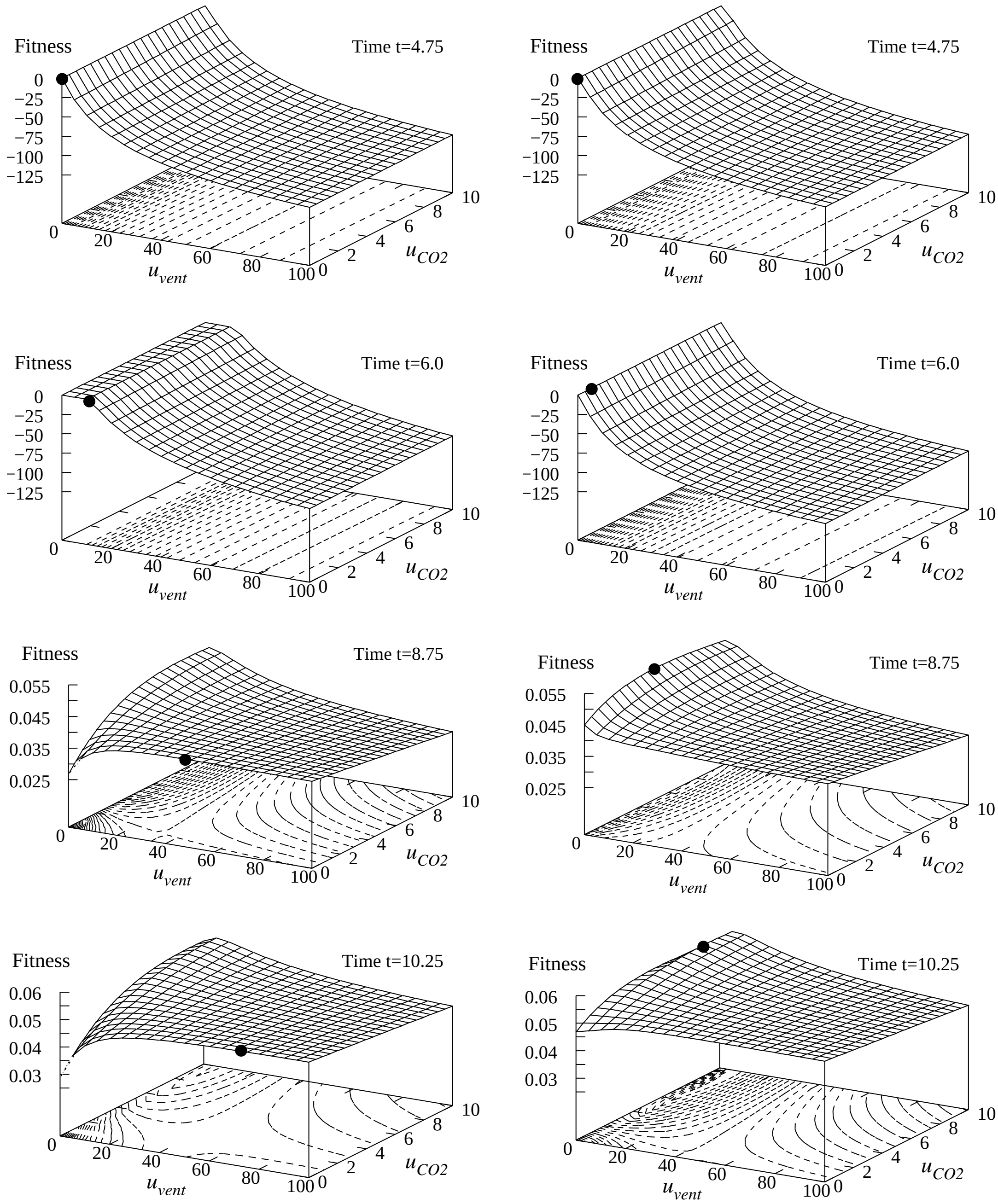


Figure 8.17: Partial landscapes of u_{vent} vs. u_{CO2} for DALS when the worst (left) and best (right) strategies diverge. The black dots denote the control signal found by the algorithm.

To further investigate this, we sampled the landscape in a four-dimensional grid and found all local optima in this grid. From time $t = 8.0$ to $t = 17.5$ (daylight) two competing control strategies existed when the worst performing strategy controlled the greenhouse. This was also the case for the best performing strategy, but in the much shorter time-span from $t = 10.25$ to $t = 12.75$. Hence, multiple optima existed in *time* from $t = 8.0$ to $t = 10.25$ and again from $t = 12.75$ to $t = 17.5$. In the period from $t = 10.25$ to $t = 12.75$, the two control strategies had two overlapping optima, although with different fitness.

8.4.6 Multi-valued control

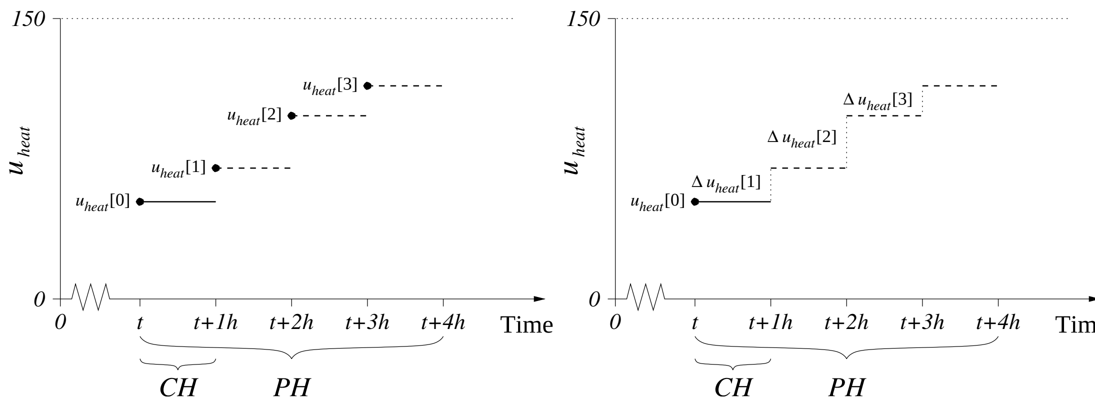
The experiments reported in the previous sections were performed using a constant control signal throughout the prediction horizon. Although this is a perfectly valid approach, it may be possible to further improve the performance by encoding a control setting for each step in the prediction horizon. Hence, the control variables are now multi-valued in the sense that each control variable is encoded with one setting per time-step, i.e.,

$$\begin{aligned}\mathbf{u} = [&u_{heat}[0], u_{vent}[0], u_{CO2}[0], u_{water}[0], \ldots, \\ &u_{heat}[PH-1], u_{vent}[PH-1], u_{CO2}[PH-1], u_{water}[PH-1]]\end{aligned} \tag{8.7}$$

An alternative representation encodes the first control setting and a number of deltas, which are added to the first signal during the prediction horizon. In this case, the multi-valued control vector is:

$$\begin{aligned}\mathbf{u} = [&u_{heat}[0], u_{vent}[0], u_{CO2}[0], u_{water}[0], \\ &\Delta u_{heat}[1], \Delta u_{vent}[1], \Delta u_{CO2}[1], \Delta u_{water}[1], \ldots, \\ &\Delta u_{heat}[PH-1], \Delta u_{vent}[PH-1], \Delta u_{CO2}[PH-1], \Delta u_{water}[PH-1]]\end{aligned} \tag{8.8}$$

The two approaches are illustrated for u_{heat} in figure 8.18.



(a) Multi-valued absolute encoding. (b) Multi-valued delta encoding.

Figure 8.18: Absolute and delta encoding for multi-valued control of heating $u_{heat} \in [0 : 150]$. Prediction horizon of four steps ($PH = 4$), control horizon of one step ($CH = 1$).

Naturally, a multi-valued encoding complicates the optimization problem, because the dimensionality of the search space is now $4PH$ instead of 4. For example, a prediction horizon of four steps gives a 16-dimensional search problem. Switching from constant to multi-valued control variables raises a number of interesting questions.

1. Does the multi-valued control variables increase performance or is the problem too complex because of the larger dimensionality?

2. Which encoding is the best? Absolute or delta?

3. Is the trade-off with minimal population size and maximal number of generations still the best for multi-valued encodings?

Furthermore, multi-valued control introduces a number of possibilities regarding variants of the encoding. To further explore the aspects of multi-valued control signals, the following encoding variants were tested on the EA (the EA was the best of the three algorithms tested in section 8.4.4).

- **Constant**: The encoding used in the previous sections, i.e., constant control signal throughout the prediction horizon.

- **Absolute encoding**: One signal per time-step (equation 8.7, figure 8.18(a)).

- **Absolute encoding + shift**: One signal per time-step (equation 8.7, figure 8.18(a)) with the signals shifted to the left after they are imposed on the system. For a prediction horizon of four steps, the left shift for u_{heat} corresponds to the assignments $u_{heat}[0] \leftarrow u_{heat}[1]$, $u_{heat}[1] \leftarrow u_{heat}[2]$, and $u_{heat}[2] \leftarrow u_{heat}[3]$. The signals are shifted one step to the left, because the control horizon is $CH = 1$.

- **Delta encoding**: First signal and one delta for each following time-step (equation 8.8, figure 8.18(b)). In delta encoding, the search ranges for the deltas must also be defined. In the experiments, these ranges were $\Delta u_{heat} \in [-10, 10]$, $\Delta u_{vent} \in [-10, 10]$, $u_{CO2} \in [-1, 1]$, and $u_{water} \in [-10, 10]$. The imposed control signal was clipped to be in the range of the control variables in case the added deltas gave a value outside of the control range. As mentioned in section 8.2, the control ranges were: $u_{heat} \in [0, 150]$, $u_{vent} \in [0, 100]$, $u_{CO2} \in [0, 10]$, and $u_{water} \in [0, 100]$.

- **Delta encoding + reset**: First signal and one delta for each following time-step (equation 8.8, figure 8.18(b)) with the deltas reset to zero before each optimization period (inner loop of in figure 8.2). This approach allows the algorithm to easily reverse the control curve created by the deltas. For example, from an increase (positive deltas) to a decrease in heating (negative deltas).

As mentioned in section 8.4.2, a prediction horizon of four steps ($PH = 4$) gave a good performance, which was nearly as good as eight steps ($PH = 8$). In multi-valued control, a doubling in prediction horizon also doubles the length of the genome, which makes the optimization problem more challenging. Hence, it may not be an advantage to double the prediction horizon if only limited computation time is available. Furthermore, the experiments on constant signals showed that the best approach was to have a population size of one individual and a maximal number of generations. For multi-valued control, the problem could be so difficult that a larger population size may give a better performance. To examine these issues, the following trade-offs (ps, gen, PH) were tested: (1,200,4), (10,20,4), (1,100,8), and (10,10,8), i.e., the total computation time was 800 units per control step. The results are displayed in table 8.2.

(ps, gen, PH)	Encoding	Dim.	Avg. profit ± SE
(1,200,4)	Constant	4	16.3415 ±0.0003
	Absolute	16	16.2406 ±0.0113
	Absolute + shift	16	15.4131 ±0.0588
	Delta	16	16.4409 ±0.0023
	Delta + reset	16	16.3539 ±0.0006
(10,20,4)	Constant	4	16.2400 ±0.0036
	Absolute	16	15.6551 ±0.0192
	Absolute + shift	16	14.1407 ±0.0782
	Delta	16	16.3546 ±0.0034
	Delta + reset	16	16.3002 ±0.0018
(1,100,8)	Constant	4	16.4954 ±0.0008
	Absolute	32	14.3333 ±0.0313
	Absolute + shift	32	14.1688 ±0.0636
	Delta	32	16.3771 ±0.0062
	Delta + reset	32	16.4907 ±0.0010
(10,10,8)	Constant	4	15.3401 ±0.0212
	Absolute	32	13.2569 ±0.0554
	Absolute + shift	32	12.6223 ±0.0656
	Delta	32	15.9792 ±0.0130
	Delta + reset	32	16.0241 ±0.0078

Table 8.2: Average profit per m^2 and standard error (SE) for the five approaches on the four trade-offs. The column denoted "Dim." shows the number of encoded variables (problem dimensionality). Average of 30 runs.

In general, the two variants of absolute encoding were clearly the worst for all trade-offs. Hence, the problem's increased complexity actually lead to a lower performance compared with the simple approach of having a constant control signal throughout the prediction horizon. This was particularly evident for the long prediction horizon. In this case, the algorithm's performance was significantly lower compared with the constant control signal approach. Surprisingly, the absolute + shift encoding had an even lower performance in all cases. This is unexpected because it should intuitively be better to shift the control signals, since this gives a better starting point for the optimization in the next time-step. A careful examination of the evolved control signals revealed that the algorithm, in nearly all control steps, started with adding a significant amount of CO_2, which was then reduced in the following steps of the prediction horizon. The shifting of control signals actually counteracts this strategy because the shifting operation constantly overwrites the first control signal. For example, a CO2-injection vector at time t, $u_{CO2}(t)[0..3] = [4.05, 1.87, 1.54, 0.67]$ becomes $u_{CO2}(t+1)[0..3] = [1.87, 1.54, 0.67, 0.67]$ after a shift. Hence, the high value of 4.05 is overwritten and has to be rediscovered by the algorithm in every control step.

In contrast to the absolute encodings, the two variants of delta encoding had good performance on all trade-offs. For the (1,200,4)-trade-off, the delta encoding (without reset) slightly improved the performance of the constant approach, whereas the delta + reset encoding had a matching performance. In the (10,20,4)-trade-off, both delta encodings had a better performance than the constant control approach. The somewhat lower performance of the delta + reset approach is because resetting distrupts strategies like the decreasing CO_2 injection example described above. For the longer prediction horizon of eight steps, the complexity of the problem clearly affected the results. For the (1,100,8)-trade-off, the constant control approach was in fact the best, but closely followed by the delta + reset approach. However, it should be noted that resetting the deltas gives a starting point of the search that corresponds to a constant control approach (first signal and deltas of zero). Interestingly, the delta approaches in the last trade-off (10,10,8) had significantly higher performance compared with the constant control approach. Hence, having multiple solutions with delta encoding is actually an advantage in this particular setup, although both variants had lower performance compared with the constant control approach in the (1,100,8)-trade-off.

Surprisingly, the best of all techniques was in fact the constant control approach with the (1,100,8)-trade-off. Besides the increased problem complexity, the reason for this is most likely that the controller used a very simple weather prediction model, i.e., the weather was assumed to be constant during the prediction horizon. Hence, equipping the controller with a better weather prediction technique may be necessary to improve the performance of multi-valued control in this particular case. Using more accurate weather prediction, the constant control approach would have to find a setting that performs well on average in the prediction horizon, whereas the multi-valued approach will be able to adapt the settings according to the changing weather.

8.5 Summary

In this study, we have investigated a number of issues regarding direct control. The main objective was to study dynamic optimization by experimenting with a realistic dynamic problem – control of a crop-producing greenhouse. First, we experimented with the prediction horizon using a simple EA. In the greenhouse problem, it turned out to be an advantage to have a rather long prediction horizon, because it allowed the algorithm to discover the long term effects of the control. Apparently, different control strategies emerge for different prediction horizons.

A short horizon lead to a strategy with high ventilation and low CO_2 injection, whereas a long horizon resulted in the opposite strategy, which turned out to be the most profitable of the two.

To get a better understanding of the greenhouse control problem, we expanded our experimentation to include a PSO algorithm and a local search technique. However, the standard PSO and simple local search require some extensions to make them capable of handling dynamic problems. Hence, we suggest an extension to the basic particle swarm optimization technique (PSO) and introduce the directed ascent local search (DALS). For all three algorithms, we investigated a number of settings of the primary parameters. Regarding the EA and the PSO, we tested various trade-offs between population size and number of generations. These experiments clearly showed that a setting with a low population size and many generations was better than the opposite setting. This confirms the Banga et al.'s results with the multi-valued control technique [15; 16]. Interestingly, a long static period (20-50 generations) between problem updates has been the preferred setting in most studies on artificial dynamic problems. Moreover, this observation confirms the results from preliminary experiments performed by Ursem in an earlier investigation on a set of artificial moving peak problems [138]. Hence, making the problem as static as possible appears to be an important key to success in dynamic optimization.

For the DALS algorithm, we compared the performance obtained with different step-sizes. Here, a large step-size proved to be the best. Somewhat surprising, the comparison of the best settings for each algorithm showed that they have nearly equivalent performance. This suggests that the problem is rather easy once it is treated as a series of related static problems, which is in fact rather surprising when considering the complexity of the simulator. However, speculating about direct control in general quickly gives a plausible explanation. In direct control, the fitness landscape expresses the quality of all possible control settings at the current operation point. Hence, a highly multimodal landscape is very unlikely because each local optimum corresponds to a distinct control strategy where neighboring strategies are suboptimal. Therefore, the EAs (and PSOs) major advantage on multimodal problems may not be that relevant for direct control, because rather few strategies exist.

To find the reason for the differences in performance, we analyzed the control signals found by the algorithms. For the EA and the PSO, this analysis showed that the worst performing setting was unable to properly track the global optimum. The two algorithms did not turn heating off during the day and valuable CO_2 was lost during the night. The analysis of the control signals for the DALS revealed an unexpected result. Here, two step-sizes lead to two very different control strategies. The worst performing strategy kept high ventilation during day and injected no CO_2, which was obtained for free from the environment. The best strategy was more or less the opposite. In this case, the algorithm operated with minimal ventilation and a high level of injected CO_2. This allowed a particularly high growth rate, which increased the profit despite the expenses to CO_2. The two strategies are similar to those reported in the first investigation on prediction horizons; however, for DALS, they emerge because of the *algorithm*, whereas two different prediction horizons were the reason in the previous investigation. Interestingly, the first strategy induced a local optimum that did not exist at the same time when the second strategy controlled the greenhouse. Hence, a local optimum in *time* appeared as a consequence of the *search*. In a wider context, optima in time may appear in any dynamic problem where the search algorithm somehow influences the future state of the problem, e.g., scheduling problems where rescheduling occurs.

Regarding multi-valued control, the additionally encoded control values allowed a more fine-grained control, but at the expense of an increased problem complexity. Surprisingly, the multi-valued control strategies did not outperform the much simpler constant control approach on the trade-off with the best performance ($ps = 1$, $gen = 100$, $PH = 8$). However, the multi-valued delta encoding did outperform the constant control approach on the four-step prediction horizon, which indicates that better performance can be achieved with multi-valued control. Besides the increased problem complexity, a likely explanation for the lower performance of the multi-valued control approaches is that the weather prediction (assuming no change) in the controller was too simple to fully exploit the potential of multi-valued control.

8.6 Future research

In a theoretical EA and PSO perspective, population diversity maintenance is important to find multiple optima in space. However, the concept of diversity is not directly transferable to algorithms dealing with optima in time. Maintaining spatial diversity does not necessarily guarantee that competing control strategies are found. Hence, it may be worth investigating algorithms that are able to locate multiple optima in time and thereby maintain alternative control strategies simultaneously. Tracking multiple spatial optima concurrently has recently been investigated in relation to artificial dynamic problems [138; 117; 22]. The main idea in these studies is to use a self-organized population structure with a variable number of subpopulations and to let each subpopulation track different optima. Thus, sub-optimal peaks are located and tracked before they may become the global optimum. These investigations show that having multiple populations allows the algorithms to track several optima concurrently. Yet, these are spatial optima and not temporal optima. Tracking temporal optima poses different challenges because they arise as a consequence of the search. To this end, algorithms capable of predicting the outcome of current control actions may have an advantage, because a currently sub-optimal control signal could generate better optima at a later stage. A straightforward way of predicting is just to simulate more time-steps. In this connection, one immediate idea may be to have multiple populations that evaluate solutions based on different prediction horizons. The signal applied to the system is then the one that gives the overall best performance on the long run.

For multi-valued control, several issues call for further attention. The complexity of the search problem should be reduced, since this appeared to be the main disadvantage of the approach. A straightforward idea would be to encode a start and a end value for each control variable, and then interpolate the control signal between these two values, i.e., the approach suggested by Banga et al. [15; 16]. Of the two encoding schemes, the delta encoding seems to be the most promising direction for future application. However, there are several open issues in this approach. First, the ranges of the deltas restricts the maximal change in a control variable, which may yield suboptimal control on some problems. Second, the experiments did not clearly indicate if it was an advantage to reset the deltas. Thus, it may be an idea to investigate inbetween reset strategies; for instance, multiplying each delta by a forgetting factor r. A complete reset then corresponds to $r = 0.0$ and no reset is $r = 1.0$. Finally, it would be very important to investigate aspects of improved weather prediction in connection with the control.

Chapter 9

Summary and conclusions

In this thesis, I have presented my research on evolutionary computation (EC). The focus in my work has been on three fundamental challenges in EC, on suggesting algorithms for dealing with these challenges, and on demonstrating the potential of the proposed algorithms on real-world problems in system identification and control. The outcome is a number of novel algorithms, improved knowledge of performance of existing techniques, and increased insight into the field of dynamic optimization. The results have been published in leading journals and conference proceedings.

Regarding basic research in EC, I concentrated on fitness function design, parameter control, and multimodal optimization, which are three of the main challenges in EC.

For fitness function design, the smoothness of the fitness landscape is of primary concern. Rugged landscapes typically arise from either imprecise fitness calculations or from the structure of the search space, i.e., that adjacent solutions in the search space have very different fitness values. In connection with the greenhouse simulator, I experienced the problem of imprecision in the fitness calculation when basing it on the Runge-Kutta-Fehlberg approximation of non-linear differential equations. The adaptive step-size for the numerical integration in this approach was the main reason for the imprecision, because different solutions in the search space required different adaptation schemes. Consequently, phantom peaks arised in the fitness landscape. Regarding the structure of the search space, Thiemo Krink and I suggested the smooth operator genetic programming (SOGP) for arithmetic expressions [144]. The SOGP approach was applied to black box structural identification. The approach had a slightly better performance than ordinary GP, but the found solutions were more robust on the test data. Our SOGP was tested on a simple identification problem, but is a general approach for arithmetic expression. Currently, the technique is being tested as a fitness function approximation approach.

In parameter control, Thiemo Krink and I suggested the terrain-based patchwork model as a self-organizing population approach to setting parameters [87]. This algorithm introduced a flocking behavior around the best parameter setting, which improved performance on some of the test problems, because more individuals exploited the best setting. In another study, I investigated the potential of self-adaptation (genetically encoded parameters) on artificial dynamic problems and found that this approach fails on even rather simple dynamic problems [138]. Self-adaptation requires many generations to find superior parameters. In conclusion, the algorithm will always be behind if the dynamic problem changes too fast and in an irregular fashion.

For multimodal optimization, I suggested the multinational EA, which is designed to find several distinct optima and thereby provide alternative solutions to a human expert [137]. The algorithm employs a self-organizing population structure that automatically divides the search space into a number of sub areas. The algorithm was compared with the sharing technique, which is another very popular method for finding multiple optima. In connection with the experiments on the multinational EA, I discovered that sharing showed a rather peculiar behavior on some problems. For this reason, I investigated sharing and found it to be very sensitive to the problem it is applied to [139]. In fact, the algorithm failed on even simple variants of the test problems used in the original study by Goldberg and Richardson [58]. This discovery is quite surprising when considering the numerous papers and theses published on variants of this technique (more than 100). In addition to these two studies, I suggested the diversity-guided EA [140]. The diversity-guided EA (DGEA) differs from most other multimodal optimization techniques, because it uses a population diversity measure to alternate between phases of exploration and phases of exploitation (fine-tuning). An unexpected result from this study is that most fitness improvements occur at surprisingly low diversity. In contrast, maintaining high diversity has long been considered the key to success in multimodal optimization. This study underlines the importance of both high and low diversity. To further test the technique, I compared it with seven other stochastic optimization algorithms on two induction motor identification problems [147]. This study was performed in collaboration with Pierré Vadstrup, Grundfos A/S [147]. The DGEA showed outstanding performance on both problems.

Regarding optimization of dynamic problems and direct control, a number of interesting results appeared. Optimization of dynamic problems has been investigated over a period of approximately 15 years. However, most of this research has been, and still is, carried out on rather artificial dynamic test problems such as the moving peaks. In an initial study, I demonstrated the usefulness of multimodal optimization techniques on such problems [138]. However, investigating artificial problems made me increasingly sceptical regarding the usefulness of these problems in research. To further examine this, and thus the foundation of a major part of EA-research in dynamic optimization, Thiemo Krink, Mikkel T. Jensen, Zbigniew Michalewicz, and I initiated a basic study on test-case generators for artificial dynamic problems [146]. As expected, we concluded that such synthetic problems have little in common with realistic dynamic problems, in particular with control problems. Consequently, Thiemo Krink, Bogdan Filipič, and I started a study on dynamic problems based on direct control of a crop-producing greenhouse. The main issue in direct control (and dynamic optimization in general) is the amount of computation time (CT) available for the optimization, which essentially determines the maximal number of evaluations before the problem changes. In EAs, the computation time can be balanced between population size (ps) and number of generations (gen) between updates, i.e., $CT = ps \cdot gen$. In direct control of the greenhouse, we found that the best control was achieved with a minimal population size (1 individual) and a maximal number of generations between updates. This was quite surprising, because it essentially turns the problem into a series of related static problems that we optimized using an EA similar to a (1+1)-evolution strategy. Conclusively, each static instance of the problem had an extremely simple fitness landscape. To further examine this, we introduced a novel particle swarm optimization algorithm and a specialized local search for dynamic problems and compared these algorithms with the simple EA [143]. The three algorithms showed similar performance. Interestingly, different step-sizes in the local search algorithm lead to two different control strategies, i.e., alternative paths of the optima in the dynamic fitness landscape. This observation is captured in the novel concept of *optima in time*, which is temporal variant of the well-known optima in search space [143]. In short, optima in time may emerge in dynamic problems where the search influence future states of the problem. As found in our study, this is the case for direct control problems. Another example is scheduling problems where rescheduling is required during the optimization. Finally, I compared the simple constant control approach with four variants of multi-valued control. Surprisingly, the simple approach turned out to be the best, which is most likely caused by the increased complexity of the search problem and that the simplistic weather predictor assumed constant weather conditions during the prediction horizon.

As mentioned, the greenhouse study was initiated to investigate dynamic optimization. Traditionally, research on dynamic optimization has been performed because real-world problems may have constraints, multiple objectives, and dynamic components. The relevance of performing research on the first two is beyond doubt, but what about dynamic optimization? In my view, our investigation on artificial dynamic problems has sort of “pulled the carpet” from under a large part of the dynamic optimization research, because nearly all publications in this field are based on synthetic benchmarks like the moving peaks problem. As mentioned earlier, the main issue in real-world dynamic optimization problems is the available computation time (CT). Now, EAs are rather demanding with respect to computation time. Hence, a small CT of only few hundred milliseconds makes EA-optimization impossible because the algorithm may not be able to evaluate enough individuals to keep up with the fast changing problem. On the other hand, a large CT will give the algorithm sufficient time to solve the problem, but during this time the problem is static. Hence, are special techniques for dynamic problems really that necessary? It is my expectation that most real-world dynamic problems can be handled by techniques for static problems, perhaps with the small extension that the best individual survives between problem instances. In addition, some performance improvement on periodic dynamic problems may be possible by using memory based techniques. However, this is unclear at the present stage.

Chapter 10

Future research

Regarding future work, several interesting topics call for further attention. For dealing with the smoothness problem in genetic programming, Thiemo Krink and I suggested the smooth operator genetic programming (SOGP). In collaboration with MSc student Kim Pedersen, I am now working on extending the technique and applying it to the problem of dealing with expensive fitness evaluations (section 3.2.3). In this connection, we will test the interpolating diviplication operator described in section 3.1.2 and possibly also a number of other smooth operators.

In parameter control, Thiemo Krink and I investigated the terrain-based patchwork model, which uses the population structure for controlling the algorithm's parameters. A related idea I have is to use so-called cooperative co-evolution for parameter control. The idea is basically to have one population of problem solutions and one population of parameter settings. The individuals in the population of problem solutions will then be mutated and recombined using the settings encoded in an individual from the parameter population. The fitness of a parameter individual is assigned according to how good solutions it produces.

For multimodal optimization, my diversity-guided EA gave very encouraging results on both artificial problems and on the two induction motor problems. For this reason, it would be interesting to further develop the algorithm. However, the algorithm should be kept as simple as possible, because this is important for practical application. The main focus should probably be on simplifying the decision process for switching the mode. The version used in the induction motor study had six parameters, which is perhaps too much. In a more theoretical context, the diversity-guided EA could be used to investigate the importance of diversity in multimodal optimization. As stated in chapter 5, diversity is believed to play a key role in multimodal optimization. However, exactly what role is rather unclear at the present stage.

In system identification, I intend to investigate several issues. The study on parameter identification of the induction motors underlined the importance of being able to handle problems with correlated parameters, although the used algorithms gave very good results. Interestingly, few algorithms are designed for this purpose. Hence, an investigation on techniques for correlated parameters would be of great value. Regarding structural identification, the smooth operator genetic programming (SOGP) appears to be a promising direction, although our first study only included one application of SOGP. An elaborate investigation of structural identification should include ordinary GP, SOGP, and other approaches such as neural networks and regression models. A broad comparison will hopefully give some insight into the performance of the genetic programming approaches and the potential of EAs in training and tuning of other techniques.

For control problems, I focused on the direct control method. Surprisingly, this study indicate that problems handled by direct control are rather simple from an optimization perspective. It would be very interesting to examine other control problems to see if this observation holds for a broader range of problems. In offline control, EAs in combination with other techniques, e.g., PID, fuzzy, and neural net controllers, have been investigated in a number of studies. However, many of these studies are based on rather simple text book EAs. Hence, a broad comparison of EA techniques would be in place. Furthermore, it would be interesting to investigate genetic programming as a "pure" EC approach to control.

Another relevant area is industrial design, because this is highly related to system identification and control. As a post.doc., I plan to initiate a study on multiobjective optimization of pump motor design. The project is a continuation of my collaboration with Pierré Vadstrup, Grundfos A/S. The practical goal in this project is to find a set of trade-off designs that are optimal with respect to two conflicting objectives; production cost and pump performance. The scientific goal is to investigate multiobjective optimization algorithms and suggest extensions of these based on my previous research. Hence, to perform basic research with focus on a real-world problem.

Bibliography

[1] Adams, R. (1995). *Calculus – a complete course.* Addison-Wesley publishers, 3rd edition.

[2] Ahmad, M., Zhang, L., and Readle, J. C. (1997). On-line Genetic Algorithm Tuning of a PI Controller for a Heating System. In Zalzala, editor, *Proceedings of the Second Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA'97).*

[3] Alonge, F., D'Ippolito, F., and Raimondi, F. M. (2001). Least squares and genetic algorithms for parameter identification of induction motors. *Control Engineering Practice*, 9(6):647–657.

[4] Angeline, P. J. (1996). Genetic Programming's Continued Evolution. In Angeline and Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 1, pages 1–20. MIT Press.

[5] Atkinson, D. J., Acarnley, P. P., and Finch, J. W. (1991). Observers for Induction Motor State and Parameter Estimation. *IEEE Transactions on Industry Applications*, 27(6):1119–1127.

[6] Babovic, V. and Keijzer, M. (2000). Genetic programming as a model induction engine. *Journal of Hydroinformatics*, 1(2):35–60.

[7] Bach, N. C. and Kjær-Larsen, R. (2001). Effect of dimensionality in the Diffusion model. In Ursem, editor, *Topics of Evolutionary Computation 2001 – Collection of Student Reports*, volume 132 of *IR*, pages 122–129. Dept. of Computer Science, University of Aarhus, Denmark.

[8] Bäck, T. (1993). Optimal Mutation Rates in Genetic Search. In Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 2–8.

[9] Bäck, T. (1996). *Evolutionary Algorithms in Theory and Practice.* Oxford University Press.

[10] Bäck et al., editors (1997). *Handbook on Evolutionary Computation.* IOP Publishing Ltd and Oxford University Press.

[11] Bäck, T. and Schütz, M. (1996). Intelligent Mutation Rate Control in Canonical Genetic Algorithms. In Ras and Michalewicz, editors, *Foundations of Intelligent Systems, 9th International Symposium (ISMIS-1996)*, pages 158–167.

[12] Bäck, T. and Schwefel, H.-P. (1995). Evolution Strategies I: Variants and their computational implementation. In Périaux and Winter, editors, *Genetic Algorithms in Engineering and Computer Science*, chapter 6, pages 111–126. John Wiley & Sons Ltd.

[13] Bagley, J. D. (1967). *The behavior of adaptive systems which employ genetic and correlation algorithms.* PhD thesis, University of Michigan.

[14] Bak, P. (1996). *How Nature Works.* Copernicus, Springer-Verlag, 1st edition.

[15] Banga, J. R., Alonso, A. A., and Singh, R. P. (1997). Stochastic Dynamic Optimization of Batch and Semi-Continuous Bioprocesses. *Biotechnol. Prog.*, 13(3):326–335.

[16] Banga, J. R., Itizarry-Rivera, R., and Seider, W. D. (1998). Stochastic Optimization for Optimal and Model-Predicitive Control. *Computers & Chemical Engineering*, 22(4):603–612.

[17] Blackwell, T. M. and Bentley, P. J. (2002). Dynamic Search With Charged Swarms. In Langdon et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*, pages 19–26.

[18] Blumel, A. L., Hughes, E. J., and White, B. A. (2001). Multi-objective Evolutionary Design of Fuzzy Autopilot Controller. In Zitzler et al., editors, *First International Conference on Evolutionary Multi-Criterion Optimization*, volume 1993 of *LNCS*, pages 668–680.

[19] Branke, J. (1999). Memory Enhanced Evolutionary Algorithms for Changing Optimization Problems. In Angeline et al., editors, *Proceedings of the Congress of Evolutionary Computation*, volume 3, pages 1875–1882.

[20] Branke, J. (2001). Evolutionary Approaches to Dynamic Optmization Problems - Updated Survey. In Spector, editor, *Genetic and Evolutionary Computation Conference Workshop Program (GECCO-2001)*, pages 27–30.

[21] Branke, J. (2001). *Evolutionary Optimization in Dynamic Environments*. Klüwer Academic Publishers.

[22] Branke, J., Kaußler, T., Schmidt, C., and Schmeck, H. (2000). A Multi-Population Approach to Dynamic Optimization Problems. In Parmee, editor, *Fourth International Conference on Adaptive Computing in Design and Manufacture (ACDM 2000)*, pages 299–308.

[23] Bremermann, H. J., Rogson, M., and Salaff, S. (1966). Global Properties of Evolution Processes. In Pattee et al., editors, *Natural Automata and Useful Simulations*, pages 3–41. Spartan Books.

[24] Brønsted, J. and From, S. (2002). Variance in Gaussian mutation. In Ursem, editor, *Topics of Evolutionary Computation 2002 – Collection of Student Reports*, volume 133 of *IR*, pages 79–84. Dept. of Computer Science, University of Aarhus, Denmark.

[25] Brucherseifer, E., Bechtel, P., Freyer, S., and Marenbach, P. (2001). An Indirect Block-Oriented Representation for Genetic Programming. In Miller et al., editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 268–279.

[26] Camacho, E. F. and Bordons, C. (1999). *Model Predictive Control*. Springer.

[27] Cantu-Paz, E. (1999). Topologies, Migration Rates, and Multi-Population Parallel Genetic Algorithms. In Banzhaf et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 91–98.

[28] Carlisle, A. and Dozier, G. (2000). Adapting Particle Swarm Optimization to Dynamic Environments. In *Proceedings of the International Conference on Artificial Intelligence 2000*, pages 429–434.

[29] Chellapilla, K. and Fogel, G. B. (1999). Multiple Sequence Alignment Using Evolutionary Programming. In Angeline et al., editors, *Proceedings of the Congress of Evolutionary Computation (CEC-1999)*, volume 1, pages 445–452.

[30] Clarke, D. W., Mohtadi, C., and Tuffs, P. S. (1987). Generalized Predictive Control – Part I. The Basic Algorithm. *Automatica*, 23(2):137–148.

[31] Clerc, M. (1999). The Swarm and the Queen: Towards a Deterministic and Adaptive Particle Swarm Optimization. In Angeline et al., editors, *Proceedings of the Congress of Evolutionary Computation (CEC-1999)*, volume 3, pages 1951–1957.

[32] Cobb, H. G. and Grefenstette, J. F. (1993). Genetic Algorithms for Tracking Changing Environments. In Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 523–530.

[33] Corne, D. W., Jerram, N. R., Knowles, J. D., and Oates, M. J. (2001). PESA-II: Region-based Selection in Evolutionary Multiobjective Optimization. In Spector et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 283–290.

[34] Davis, editor (1987). *Genetic Algorithms and Simulated Annealing*. Research Notes in Artificial Intelligence. Pitman Publishing.

[35] De Jong, K. A. (1975). *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan.

[36] Deb, K. (2001). *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley & Sons.

[37] Deb, K., Agrawal, S., Pratap, A., and Meyarivan, T. (2000). A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II. In Schoenauer et al., editors, *Parallel Problem Solving from Nature VI (PPSN-2000)*, volume 1917 of *LNCS*, pages 849–858.

[38] Doncieux, S. and Meyer, J.-A. (2003). Evolving Neural Networks for the Control of a Lenticular Blimp. In Raidl et al., editors, *Applications of Evolutionary Computing, EvoWorkshops2003: EvoBIO, EvoCOP, EvoIASP, EvoMUSART, EvoROB, EvoSTIM*, volume 2611 of *LNCS*, pages 631–642.

[39] Eberhart, R. C. and Shi, Y. (2001). Tracking and Optimizing Dynamic Systems with Particle Swarms. In *Proceedings of the Third Congress on Evolutionary Computation (CEC-2001)*, pages 94–100.

[40] Eggermont, J. and van Hemert, J. I. (2001). Adaptive Genetic Programming Applied to New and Existing Simple Regression Problems. In Miller et al., editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 23–35.

[41] Eiben, A. E., Hintering, R., and Michalewicz, Z. (1999). Parameter Control in Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141.

[42] Eiben, A. E., Nabuurs, R., and Booij, I. (2001). *The Escher evolver: Evolution to the people*, chapter 17, pages 425–439. Morgan Kaufmann publishers.

[43] El-Beltagy, M. A., Nair, P. B., and Keane, A. J. (1999). Metamodeling Techniques For Evolutionary Optimization of Computationally Expensive Problems: Promises and Limitations. In Banzhaf et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, volume 1, pages 196–203.

[44] Emmerich, M., Giotis, A., Özdemir, M., Bäck, T., and Giannakoglou, K. (2002). Metamodel-Assisted Evolution Strategies. In Merelo et al., editors, *Parallel Problem Solving from Nature VII (PPSN-2002)*, pages 361–370.

[45] Eshelman, L. J. (1991). The CHC Adaptive Search Algorithm: How to Have Safe Search When Engaging in Nontraditional Genetic Recombination. In Rawlins, editor, *Foundations of Genetic Algorithms*, pages 265–283.

[46] Fehlberg, E. (1969). Klassische RUNGE-KUTTA-Formeln fünfter und siebenter Ordnung mit Schrittweiten-Kontrolle. *Computing*, 4(2):93–106.

[47] Filipič, B. and Juričić, D. (1993). An interactive genetic algorithm for controller parameter optimization. In Albrecht et al., editors, *Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms (ANNGA-1993)*, pages 458–462.

[48] Filipič, B., Urbančič, T., and Križman, V. (1999). A combined Machine Learning and Genetic Algorithm Approach to Controller Design. *Engineering Applications of Artificial Intelligence*, 12(4):401–409.

[49] Fogarty, T. C. (1989). Varying the Probability of Mutation in the Genetic algorithm. In Schaffer, editor, *Proc. of the Third International Conference on Genetic Algorithms*, pages 104–109.

[50] Fogarty, T. C., Vavak, F., and Cheng, P. (1995). Use of the Genetic Algorithm for Load Balancing of Sugar Beet Presses. In Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 617–624.

[51] Fogel, editor (1998). *Evolutionary Computation: The Fossil Record*. IEEE Press.

[52] Fogel, D. B., Angeline, P., Porto, V., III, E. W., and Boughton, E. (1999). Using Evolutionary Computation to Learn About Detecting Breast Cancer. In Angeline et al., editors, *Proceedings of the Congress of Evolutionary Computation (CEC-1999)*, volume 3, pages 1749–1754.

[53] Fogel, L. J., Owens, A. J., and Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons.

[54] Fonseca, C. M. and Fleming, P. J. (1996). Nonlinear System Identification with Multiobjective Genetic Algorithms. In *Proceedings of the 13th World Congress of the International Federation of Automatic Control*, pages 187–192.

[55] Fonseca, C. M. and Fleming, P. J. (1998). Multiobjective Optimization and Multiple Constraint Handling with Evolutionary Algorithms—Part II: A Application Example. *IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans*, 28(1):38–47.

[56] Fukuyama, Y. and Yoshida, H. (2001). A Particle Swarm Optimization for Reactive Power and Voltage Control in Electric Power Systems. In *Proceedings of the 2001 Congress on Evolutionary Computation (CEC-2001)*, pages 87–93.

[57] Giotis, A., Emmerich, M., Naujoks, B., Giannakoglou, K., and Bäck, T. (2001). Low-Cost Stochastic Optimization for Engineering Applications. In Giannakoglou et al., editors, *Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems (EUROGEN-2001)*, pages 361–366.

[58] Goldberg, D. E. and Richardson, J. (1987). Genetic Algorithms with Sharing for Multimodal Function Optimization. In Grefenstette, editor, *Genetic Algorithms and their Applications (ICGA'87)*, pages 41–49.

[59] Gordon, V. S., Pirie, R., Wachter, A., and Sharp, S. (1999). Terrain-Based Genetic Algorithm (TBGA): Modeling Parameter Space as Terrain. In Banzhaf et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, volume 1, pages 229–235.

[60] Greenwood, G. W., Fogel, G. B., and Ciobanu, M. (1999). Emphasizing Extinction in Evolutionary Programming. In Angeline et al., editors, *Proceedings of the Congress of Evolutionary Computation (CEC-1999)*, volume 1, pages 666–671.

[61] Grefenstette, J. (1992). Genetic Algorithms for changing environments. In *Proceedings of Parallel Problem Solving From Nature II (PPSN-1992)*, pages 137–144.

[62] Grefenstette, J. J. (1986). Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):122–128.

[63] Grefenstette, J. J. (1999). Evolvability in Dynamic Fitness Landscapes: A Genetic Algorithm Approach. In Angeline et al., editors, *Proceedings of the Congress of Evolutionary Computation*, volume 3, pages 2031–2038.

[64] Gremling, J. T. and Passino, K. M. (1999). Genetic Adaptive Parameter Estimation. *International Journal of Intelligent Control and Systems*, 3(4):465–503.

[65] Haque, T., Nolan II, R., Pillay, P., and Reynaud, J. (1994). Parameter determination for induction motors. In *Proceedings of the IEEE SOUTHEASTCON'94*, pages 45–49.

[66] Herdy, M. (1996). Evolution Strategies with Subjective Selection. In Voigt et al., editors, *Parallel Problem Solving from Nature IV (PPSN-1996)*, volume 1141 of *LNCS*, pages 22–31.

[67] Hesser, J. and Männer, R. (1990). Towards an Optimal Mutation Probability for Genetic Algorithms. In Schwefel and Männer, editors, *Parallel Problem Solving from Nature I (PPSN-1990)*, pages 23–32.

[68] Holland, J. H. (1975). *Adpatation in Natural and Artificial Systems*. University of Michigan Press.

[69] Huang, W. and Lam, H. N. (1997). Using genetic algorithms to optimize controller parameters for HVAC systems. *Energy Build.*, 26(3):277–282.

[70] Janikow, C. Z. and Michalewicz, Z. (1991). An Experimental Comparison of Binary and Floating Point Representations in Genetic Algorithms. In Belew and Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 31–36.

[71] Jensen, M. T. (2001). *Robust and Flexible Scheduling with Evolutionary Computation*. PhD thesis, Department of Computer Science, University of Aarhus.

[72] Ju, P. and Handschin, E. (1995). Parameter estimation of composite induction motor loads using genetic algorithms. In *Proceedings of the Stockholm Power Technology International Symposium on Electric Power Engineering*, volume 3, pages 97–102.

[73] Kaise, N. and Fujimoto, Y. (1999). Applying the Evolutionary Neural Networks with Genetic Algorithms to Control a Rolling Inverted Pendulum. In McKay et al., editors, *Proceedings of the 2nd Asia-Pacific Conference on Simulated Evolution and Learning (SEAL-1998)*, volume 1585 of *LNAI*, pages 223–230.

[74] Kanazaki, M., Morikaw, M., Obayashi, S., and Nakahashi, K. (2002). Multiobjective Design Optimization of Merging Configuration for an Exhaust Manifold of a Car Engine. In Guervós et al., editors, *Parallel Problem Solving from Nature VII (PPSN-2002)*, volume 2439 of *LNCS*, pages 281–287.

[75] Kang, L., Cao, H., and Chen, Y. (1999). The Dynamic Evolutionary Modeling of Higher-Order Ordinary Differential Equations for Time Series Real-Time Prediction. In Angeline et al., editors, *Proceedings of the Congress of Evolutionary Computation (CEC-1999)*, volume 2, pages 1224–1229.

[76] Karr, C. L. (1991). Design of an adaptive fuzzy logic controller using a genetic algorithm. In Belew and Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 450–457.

[77] Karr, C. L. (1996). Adaptive Process Control using Biological Paradigms. *Journal of network and computer applications*, 19:21–44.

[78] Keane, A. J. and Brown, S. M. (1996). The Design of a Satellite Boom with Enhanced Vibration Performance using Genetic Algorithm Techniques. In Parmee, editor, *Proceedings of the Conference on Adaptive Computation in Engineering Design and Control*, pages 107–113.

[79] Kee, E., Airey, S., and Cyre, W. (2001). An Adaptive Genetic Algorithm. In Spector et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 391–397.

[80] Kennedy, J. and Eberhart, R. C. (1995). Particle swarm optimization. In *Proceedings of the 1995 IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948.

[81] Kennedy, J., Eberhart, R. C., and Shi, Y. (2001). *Swarm Intelligence*. Morgan Kaufmann Publishers.

[82] Kirley, M. and Green, D. G. (2000). Adaptation and Spatial Patterns: Optimization Using the Cellular Genetic Algorithm. In Cantu-Paz and Punch, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, Workshop Program (GECCO-2000)*, pages 12–16.

[83] Koza, J. R., Keane, M. A., Yu, J., Bennett III, F. H., and Mydlowec, W. (2000). Automatic Creation of Human-Competitive Programs and Controllers by Means of Genetic Programming. *Genetic Programming And Evolvable Machines*, 1(1/2):121–164.

[84] Krink, T., Mayoh, B. H., and Michalewicz, Z. (1999). A PATCHWORK Model for Evolutionary Algorithms with Structured and Variable Size Populations. In Banzhaf et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, volume 2, pages 1321–1328.

[85] Krink, T. and Thomsen, R. (2001). Self-Organized Criticality and Mass Extinction in Evolutionary Algorithms. In *Proceedings of the Third Congress on Evolutionary Computation (CEC-2001)*, volume 2, pages 1155–1161.

[86] Krink, T., Thomsen, R., and Rickers, P. (2000). Applying Self-Organised Criticality to Evolutionary Algorithms. In Schoenauer et al., editors, *Parallel Problem Solving from Nature VI (PPSN-2000)*, volume 1, pages 375–384.

[87] Krink, T. and Ursem, R. K. (2000). Parameter Control Using the Agent Based Patchwork Model. In *Proceedings of the Second Congress on Evolutionary Computation (CEC-2000)*, volume 1, pages 77–83.

[88] Krink, T., Ursem, R. K., and Filipič, B. (2001). Evolutionary Algorithms in Control Optimization: The Greenhouse Problem. In Spector et al., editors, *Proceedings of the Third Genetic and Evolutionary Computation Conference (GECCO-2001)*, volume 1, pages 440–447.

[89] Krohling, R. A. and Rey, J. P. (2001). Design of Optimal Disturbance Rejection PID Controllers Using Genetic Algorithms. *IEEE Transactions on Evolutionary Computation*, 5(1):78–82.

[90] Kuo, B. C. (1995). *Automatic Control Systems*. Prentice Hall, 7th edition.

[91] Lee, C.-H., Park, S.-H., and Kim, J.-H. (2000). Topology and Migration Policy of Fine-grained Parallel Evolutionary Algorithms for Numerical Optimization. In *Proceedings of the Second Congress on Evolutionary Computation (CEC-2000)*, pages 70–76.

[92] Lee, M. and Takagi, H. (1993). Dynamic Control of Genetic Algorithms Using Fuzzy Logic Techniques. In Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 76–83.

[93] Lennon, W. K. and Passino, K. M. (1999). Intelligent Control for Brake Systems. *IEEE Transactions on Control Systems Technology*, 7(2):188–202.

[94] Lis, J. (1996). Parallel Genetic Algorithm with Dynamic Control Parameter. In *Proceedings of the 1996 IEEE Conference on Evolutionary Computation*, pages 324–329.

[95] Ljung, L. (1999). *System identification – theory for the user.* Prentice Hall, 2nd edition.

[96] Lund, H. H., Miglino, O., Pagliarini, L., Billard, A., and Ijspeert, A. (1998). Evolutionary robotics - A childrens game. In *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation*, pages 154–158.

[97] Mahfoud, S. (1992). Crowding and preselection revisited. Technical Report 92004, Illinois Genetic Algorithms Laboratory (IlliGAL).

[98] Mengshoel, O. J. and Goldberg, D. E. (1999). Probabilistic Crowding: Deterministic Crowding with Probabilisitic Replacement. In Banzhaf et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, volume 1, pages 409–416.

[99] Michalewicz, Z. (1992). *Genetic Algorithms + Data Structures = Evolution Programs.* Springer.

[100] Michalewicz, Z. and Fogel, D. B. (2000). *How to Solve It: Modern Heuristics.* Springer.

[101] Morrison, R. W. and Jong, K. A. D. (1999). A Test Problem Generator for Non-Stationary Environments. In Angeline et al., editors, *Proceedings of the Congress of Evolutionary Computation (CEC-1999)*, volume 3, pages 2047–2053.

[102] Onnen, C., Babuska, R., Kaymak, U., Sousa, J. M., Verbruggen, H. B., and Isermann, R. (1997). Genetic Algorithms for Optimization in Predictive Control. *Control Engineering Practice*, 5(10):1363–1372.

[103] Oppacher, F. and Wineberg, M. (1999). The Shifting Balance Genetic Algorithm: Improving the GA in a Dynamic Environment. In Banzhaf et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 504–510.

[104] Oussedik, S. and Delahaye, D. (1998). Reduction of Air Traffic Congestion by Genetic Algorithms. In Eiben et al., editors, *Parallel Problem Solving from Nature V (PPSN-1998)*, pages 855–864.

[105] Petrovic, M. V., Dulikravich, G. S., and Martin, T. J. (2000). Maximizing Multistage Turbine Efficiency by Optimizing Hub and Shroud Shapes and Inlet and Exit Conditions of each Blade Row. *International Journal of Turbo and Jet-Engines*, 17:267–278.

[106] Pohlheim, H. and Heißner, A. (1996). Optimale Steuerung des Klimas im Gewächshaus mit Evolutionären Algorithmen: Grundlagen, Verfahren und Ergebnisse. Technical report, Technische Universität Ilmenau.

[107] Poli, R. and Page, J. (2000). Solving High-Order Boolean Parity Problems with Smooth Uniform Crossover, Sub-Machine Code GP and Demes. *Genetic Programming And Evolvable Machines*, 1(1/2):37–56.

[108] Poli, R., Page, J., and Langdon, W. B. (1999). Smooth Uniform Crossover, Sub-Machine Code GP and Demes: A Recipe For Solving High-Order Boolean Parity Problems. In Banzhaf et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, volume 2, pages 1162–1169.

[109] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1992). *Numerical Recipes in C*. Cambridge University Press, 2nd edition.

[110] Proca, A. B. and Keyhani, A. (2002). Identification of Variable Frequency Induction Motor Models From Operating Data. *IEEE Transactions on Energy Conversion*, 17(1):24–31.

[111] Raup, D. (1986). Biological Extinction in Earth History. *Science*, 231:1528–1533.

[112] Rechenberg, I. (1964). Kybernetische Lösungsansteuerung einer experimentellen Forschungsaufgabe. presented at the Annual Conference of the WGLR at Berlin in September 1964.

[113] Rechenberg, I. (1973). *Evolutionsstrategie*, volume 15 of *problemata*. Friedrich Frommann Verlag (Günther Holzboog KG).

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

[115] Reed, J., Toombs, R., and Barricelli, N. A. (1967). Simulation of Biological Evolution and Machine Learning. *Journal of Theoretical Biology*, 17:319–342.

[116] Rodríguez-Vázquez, K. (1999). *Multiobjective Evolutionary Algorithms in Non-Linear System Identification*. PhD thesis, Department of Automatic Control and Systems Engineering, The University of Sheffield.

[117] Ronnewinkel, C. and Martinetz, T. (2001). Explicit Speciation with few a priori Parameters for Dynamic Optimization Problems. In Branke and Bäck, editors, *Genetic and Evolutionary Computation Conference Workshop Program (GECCO-2001)*, pages 31–38.

[118] Sarma, J. and Jong, K. A. D. (1996). An Analysis of the Effects of Neighborhood Size and Shape on Local Selection Algorithms. In Voigt et al., editors, *Parallel Problem Solving from Nature IV (PPSN-1996)*, pages 236–244.

[119] Sarma, J. and Jong, K. A. D. (1997). An Analysis of Local Selection Algorithms in a Spatially Structured Evolutionary Algorithm. In Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, pages 181–186.

[120] Schaffer, J. D., Caruana, R. A., Eshelman, L., and Das, R. (1989). A Study of Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization. In Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*.

[121] Schwefel, H.-P. (1965). *Kybernetische Evolution als Strategie der experimentellen Forschung in der Strömungstechnik*. PhD Thesis, Technical University of Berlin, Hermann Föttinger–Institute for Hydrodynamics.

[122] Schwefel, H.-P. (1977). *Numerische Optimierung von Computer Modellen mittels der Evolutionsstrategie*, volume 26 of *Interdisciplinary systems research*. Birkhäuser Verlag.

[123] Sebag, M., Schoenauer, M., and Maitournam, H. (1998). Parametric and Non-Parametric Identification of Macro-Mechanical Models. In Quagliarella et al., editors, *Genetic Algorithms and Evolution Strategy in Engineering and Computer Science*, pages 327–340. John Wiley and Sons.

[124] Shi, Y. and Eberhart, R. C. (1998). Parameter Selection in Particle Swarm Optimization. In Porto et al., editors, *Evolutionary Programming VII*, pages 591–600.

[125] Shimodaira, H. (1999). A Diversity Control Oriented Genetic Algorithm (DCGA): Development and Experimental Results. In Banzhaf et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, volume 1, pages 603–611.

[126] Sorooshian, S., Bastidas, L. A., and Gupta, H. V. (1999). Application of Multi-Objective Optimization Algorithms for Hydrologic Model Identification and Parameterization. In *Proceedings of Second International Conference on Multiple Objective Decision Support Systems for Land, Water, and Environmental Management.*

[127] Storn, R. and Price, K. (1995). Differential Evolution - a Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Spaces. Technical Report TR-95-012, International Computer Science Institute, Berkley.

[128] Stroud, P. D. (2001). Kalman-Extended Genetic Algorithm for Search in Nonstationary Environments with Noisy Fitness Evaluations. *IEEE Transactions on Evolutionary Computation*, 5(1):66–77.

[129] Thomsen, R., Fogel, G. B., and Krink, T. (2002). A Clustal Alignment Improver using Evolutionary Algorithms. In Fogel et al., editors, *Proceedings of the Fourth Congress on Evolutionary Computation (CEC-2002)*, volume 1, pages 121–126.

[130] Thomsen, R. and Krink, T. (2002). Self-Adaptive Operator Scheduling using the Religion-Based EA. In *Proceedings of Parallel Problem Solving from Nature VII (PPSN-2002)*, pages 214–223.

[131] Thomsen, R. and Rickers, P. (2000). Introducing Spatial Agent-Based Models and Self-Organised Criticality to Evolutionary Algorithms. Master's thesis, University of Aarhus, Denmark.

[132] Thomsen, R., Rickers, P., and Krink, T. (2000). A Religion-Based Spatial Model For Evolutionary Algorithms. In Schoenauer et al., editors, *Parallel Problem Solving from Nature VI (PPSN-2000)*, volume 1, pages 817–826.

[133] Todd, D. S. and Sen, P. (1997). A Multiple Criteria Genetic Algorithm for Containership Loading. In Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms*, pages 674–681.

[134] Trojanowski, K. and Michalewicz, Z. (1999). Searching for Optima in Non-stationary Environments. In Angeline et al., editors, *Proceedings of the Congress of Evolutionary Computation (CEC-1999)*, volume 3, pages 1843–1850.

[135] Tsutsui, S. and Fujimoto, Y. (1993). Forking Genetic Algorithm with Blocking and Shrinking Modes (FGA). In Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 206–213.

[136] Tsutsui, S., Fujimoto, Y., and Ghosh, A. (1997). Forking Genetic Algorithms: GAs with Search Space Division Schemes. *Evolutionary Computation*, 5:61–80.

[137] Ursem, R. K. (1999). Multinational Evolutionary Algorithms. In Angeline et al., editors, *Proceedings of the Congress of Evolutionary Computation (CEC-1999)*, volume 3, pages 1633–1640.

[138] Ursem, R. K. (2000). Multinational GAs: Multimodal Optimization Techniques in Dynamic Environments. In *Proceedings of the Second Genetic and Evolutionary Computation Conference (GECCO-2000)*, volume 1, pages 19–26.

[139] Ursem, R. K. (2001). When Sharing Fails. In *Proceedings of the Third Congress on Evolutionary Computation (CEC-2001)*, pages 873–879.

[140] Ursem, R. K. (2002). Diversity-Guided Evolutionary Algorithms. In Merelo et al., editors, *Proceedings of Parallel Problem Solving from Nature VII (PPSN-2002)*, pages 462–471.

[141] Ursem, R. K., Filipič, B., and Krink, T. (2002). Exploring the Performance of an Evolutionary Algorithm for Greenhouse Control. *Journal of Computing and Information Technology*, 10(3):195–201.

[142] Ursem, R. K., Filipič, B., and Krink, T. (2002). Exploring the Performance of an Evolutionary Algorithm for Greenhouse Control. In Glavinić et al., editors, *Proceedings of the 24th International Conference on Information Technology Interfaces (ITI-2002)*, pages 429–434.

[143] Ursem, R. K., Filipič, B., and Krink, T. (in subm.). Dynamic Optimization with Stochastic Algorithms: A Comparative Study on a Control Problem. *IEEE Transactions on Evolutionary Computation*.

[144] Ursem, R. K. and Krink, T. (2002). Genetic Programming with Smooth Operators for Arithmetic Expressions: Diviplication and Subdition. In Fogel et al., editors, *Proceedings of the Fourth Congress on Evolutionary Computation (CEC-2002)*, volume 2, pages 1372–1377.

[145] Ursem, R. K., Krink, T., and Filipič, B. (2002). A Numerical Simulator for a Crop-Producing Greenhouse. Technical Report 2002-01, EVALife, Dept. of Computer Science, University of Aarhus, Denmark, `www.evalife.dk`.

[146] Ursem, R. K., Krink, T., Jensen, M. T., and Michalewicz, Z. (2002). Analysis and Modeling of Control Tasks in Dynamic Systems. *IEEE Transactions on Evolutionary Computation*, 6(4):378–389.

[147] Ursem, R. K. and Vadstrup, P. (in subm.). Parameter Identification of Induction Motors using Stochastic Search Algorithms. *Applied Soft Computing*.

[148] Varsek, A., Urbančič, T., and Filipič, B. (1993). Genetic Algorithms in Controller Design and Tuning. *IEEE transactions on Systems, Man, and Cybernetics*, 23(5):1330–1339.

[149] Vas, P. (1992). *Electrical Machines and Drives – A Space-Vector Theory Approach*. Clarendon Press – Oxford Science Publications.

[150] Vavak, F., Fogarty, T. C., and Cheng, P. (1995). Load Balancing Application of the Genetic Algorithm in a Nonstationary Environment. In Fogarty, editor, *Proceedings of the AISB workshop on Evolutionary Computation*, volume 993 of *LNCS*, pages 224–233.

[151] Vavak, F., Jukes, K., and Fogarty, T. C. (1997). Adaptive Combustion Balancing in Multiple Burner Boiling Using a Genetic Algorithm with Variable Range of Local Search. In Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms*, pages 719–726.

[152] Vesterstrøm, J. S. and Riget, J. (2002). Particle Swarms: Extensions for Improved Local, Multi-modal, and Dynamic Search in Numerical Optimization. Master's thesis, EVALife, Dept. of Computer Science, Aarhus Universitet.

[153] Wolpert, D. H. and Macready, W. G. (1995). No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute.

[154] Yamasaki, K., Kitakaze, K., and Sekiguchi, M. (2002). Dynamic Optimization by Evolutionary Algorithms Applied to Financial Time Series. In Fogel et al., editors, *Proceedings of the Fourth Congress on Evolutionary Computation (CEC-2002)*, volume 2, pages 2017–2022.

[155] Zai, L. C., DeMarco, C. L., and Lipo, T. A. (1992). An Extented Kalman Filter Approach to Rotor Time Constant Measurement in PWM Induction Motor Drives. *IEEE Transactions on Industry Applications*, 28(1):96–104.

[156] Zitzler, E., Laumanns, M., and Thiele, L. (2001). SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization. In Giannakoglou et al., editors, *Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems*, pages 95–100.

[157] Zupancic, B., Klopcic, M., and Karba, R. (1999). Tuning of Fuzzy Logic Controller with Genetic Algorithms. *Informatica*, 23:559–564.

Index

Appendix A

List of publications

Journal papers

- Ursem, R. K., Krink, T., Jensen, M., T., and Michalewicz, Z. (2002). Analysis and Modeling of Control Tasks in Dynamic Systems. In: IEEE Transactions on Evolutionary Computation, vol. 6(4):378–389.
- Ursem, R. K., Filipič, B. and Krink, T. (2002). Exploring the Performance of an Evolutionary Algorithm for Greenhouse Control. In: Journal of Computing and Information Technology, vol. 10(3):195–201.
- Ursem, R. K., Filipič, B. and Krink, T. (in subm.). Dynamic Optimization with Stochastic Algorithms: A Comparative Study on a Control Problem. Submitted to IEEE Transactions on Evolutionary Computation.
- Ursem, R. K. and Vadstrup, P. (in subm.). Parameter Identification of Induction Motors using Stochastic Optimization Algorithms. Submitted to Applied Soft Computing.

Conference papers

- Ursem, R. K. (1999). Multinational Evolutionary Algorithms. In: Proceedings of the Congress of Evolutionary Computation (CEC–1999), vol. 3, p. 1633–1640.
- Krink, T. and Ursem, R. K. (2000). Parameter Control Using the Agent Based Patchwork Model. In: Proceedings of the Second Congress on Evolutionary Computation (CEC–2000), vol. 1, p. 77–83.
- Ursem, R. K. (2000). Multinational GAs: Multimodal Optimization Techniques in Dynamic Environments. In: Proceedings of the Second Genetic and Evolutionary Computation Conference (GECCO–2000), vol. 1, p. 19–26.
- Krink, T., Ursem, R. K., and Filipič, B. (2001). Evolutionary Algorithms in Control Optimization: The Greenhouse Problem. In: Proceedings of the third Genetic and Evolutionary Computation Conference (GECCO–2001), vol. 1, p. 440–447.

- Ursem, R. K. (2001). When Sharing Fails. In: Proceedings of the Third Congress on Evolutionary Computation (CEC–2001), p. 873–879.
- Ursem, R. K. and Krink, T. (2002). Genetic Programming with Smooth Operators for Arithmetic Expressions: Diviplication and Subdition. In: Proceedings of the Fourth Congress on Evolutionary Computation (CEC–2002), vol. 2, p. 1372–1377.
- Ursem, R. K., Filipič, B. and Krink, T. (2002). Exploring the Performance of an Evolutionary Algorithm for Greenhouse Control. In: Proceedings of the 24th International Conference on Information Technology Interfaces ITI 2002, p. 429–434.
- Ursem, R. K. (2002). Diversity-Guided Evolutionary Algorithms. In: Proceedings of Parallel Problem Solving from Nature VII (PPSN–2002), p. 462–471.

Technical reports and other publications

- Ursem, R. K., Krink, T., and Filipič, B. (2002). A Numerical Simulator for a Crop–Producing Greenhouse. Technical report no. 2002-01.
- Ursem, R. K., editor (2001). Topics of Evolutionary Computation 2001 – Collection of Student Reports (booklet).
- Ursem, R. K., editor (2002). Topics of Evolutionary Computation 2002 – Collection of Student Reports (booklet).

Appendix B

Simple benchmark problems

These benchmark problems are often used in studies on evolutionary computation. For additional problems, see appendix B in [99].

1. De Jong function F1 (minimization):

   $$\sum_{i=1}^{3} x_i^2$$

   where $-5.12 \leq x_i \leq 5.12$.

2. De Jong function F2 (minimization):

   $$100(x_1^2 - x_2)^2 + (1 - x_1)^2$$

   where $-2.048 \leq x_i \leq 2.048$.

3. De Jong function F3 (minimization):

   $$\sum_{i=1}^{5} \text{integer}(x_i)$$

   where $-5.12 \leq x_i \leq 5.12$ and $\text{integer}(x_i)$ is the integer part of x_i.

4. De Jong function F4 (minimization):

   $$\sum_{i=1}^{30} i x_i^4 + N(0, 1)$$

   where $-1.28 \leq x_i \leq 1.28$ and $N(0, 1)$ is the normal distribution.

5. De Jong function F5 (minimization):

$$\frac{1}{1/K + \sum_{j=1}^{25} 1/f_j(x_1, x_2)}$$

where $-65.536 \leq x_i \leq 65.536$, $K = 500$, $f_j(x_1, x_2) = j + \sum_{i=1}^{2}(x_i - a_{ij})^6$, and

$$[a_{ij}] = \left[\begin{array}{cccccccccc} \text{-32} & \text{-16} & 0 & 16 & 32 & \text{-32} & \text{-16} & \ldots & 0 & 16 & 32 \\ \text{-32} & \text{-32} & \text{-32} & \text{-32} & \text{-32} & \text{-16} & \text{-16} & \ldots & 32 & 32 & 32 \end{array} \right]$$

6. Goldberg function F1 (maximization):

$$\sin^6(5\pi x)$$

where $0 \leq x \leq 1$.

7. Goldberg function F2 (maximization):

$$\sin^6(5\pi x) \cdot \exp\left(-2\ln(2)\left(\frac{x - 0.1}{0.8}\right)^2\right)$$

where $0 \leq x \leq 1$.

8. Ackley function F1 (minimization):

$$20 + e - 20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n} cos(2\pi \cdot x_i)\right)$$

where $-30 \leq x_i \leq 30$.

9. Griewank function F1 (minimization):

$$\frac{1}{4000}\sum_{i=1}^{n}(x_i - 100)^2 - \prod_{i=1}^{n}\cos\left(\frac{x_i - 100}{\sqrt{i}}\right) + 1$$

where $-600 \leq x_i \leq 600$.

10. Rastrigin function F1 (minimization):

$$\sum_{i=1}^{n}(x_i^2 - 10\cos(2\pi x_i) + 10)$$

where $-5.12 \leq x_i \leq 5.12$.

11. Rosenbrock function F1 (minimization):

$$\sum_{i=1}^{n-1}(100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$$

where $-100 \le x_i \le 100$.

12. Schaffer function F6 (minimization):

$$0.5 + \frac{\sin^2\left(\sqrt{x_1^2 + x_2^2}\right) - 0.5}{(1.0 + 0.001(x_1^2 + x_2^2))^2}$$

where $-100 \le x_i \le 100$.

Appendix C

Crop-producing greenhouse

This appendix describes a simulator for a crop-producing greenhouse. The simulator is implemented in Java and is, to a large extent, based on the greenhouse simulator described in [106]. The original description and MatLab code are online at `www.pohlheim.com`. The simulator described here differs on a few minor aspects:

- The description is in English. The original description is in German.
- The naming of state variables follows the control engineering terminology and the terminology presented in [146].
- The greenhouse is made more realistic by including the wind speed in the equations related to air-exchange with the surroundings. The original simulator was, to some degree, modeling a hermetically closed greenhouse.
- The approximation of the non-linear differential equations is protected to prevent unrealistic values in the variables such as realtive humidity above 100%, negative steam pressures, and temperatures below 0 Kelvin.

The interaction between the controller, the greenhouse, and the immediate surroundings of the greenhouse is illustrated in Figure C.1. Table C.1 provides an overview of the variables associated with the controller (u), the greenhouse state (x), and the environment state (v). The simulator models the profit per m^2 over time. The profit is equal to the income from the production minus the expenses to heating and CO_2 (see section C.1.5).

C.1 State equations

The greenhouse state (x) is modeled by six non-linear differential equations describing the change over time. The solution to the equations can be approximated by a fourth-order Runge-Kutta formula. The control variables are measured in hours, which is converted to seconds in the equations because the differential equations have to be approximated using a rather short step size to avoid instability. Preliminary tests showed that a step size of one minute is appropriate. The following sections describe each of the equations in detail. Real weather data is used

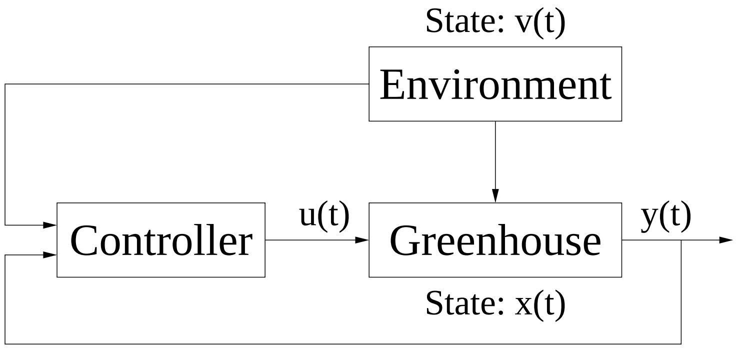


Figure C.1: The interaction between the controller, the greenhouse, and the greenhouse environment.

	Description	Variable
Greenhouse	Indoor steam density [g/m^3]	x_{steam}
	Indoor air temperature [°C]	x_{atemp}
	Indoor CO_2 concentration [ppm]	x_{CO2}
	Accumulated biomass [g/m^2]	x_{biom}
	Cumulative profit [DKK/m^2]	x_{profit}
	Condensation on glass [g/m^2]	x_{cond}
Environment	Outdoor sunlight intensity [W/m^2]	v_{sun}
	Outdoor air temperature [°C]	v_{atemp}
	Outdoor ground temperature [°C]	v_{gtemp}
	Relative humidity [% r.H.]	v_{RH}
	Wind speed [m/s]	v_{wind}
	Outdoor CO_2 concentration [ppm]	v_{CO2}
	Price of heating [DKK/(W·h)]	v_{Pheat}
	Price of CO_2 [DKK/kg]	v_{PCO2}
	Price of crops (tomatoes) [DKK/kg]	v_{Ptom}
Control	Heating [W/m^2]	u_{heat}
	Ventilation [m^3/(m^2 · h)]	u_{vent}
	CO_2 injection [g/(m^2 · h)]	u_{CO2}
	Water injection [g/(m^2 · h)]	u_{water}

Table C.1: System, environment, and control variables of the simulated greenhouse. DKK denotes the currency "Danish Kroner".

to model the environment (v). Weather data can usually be obtained from your national meteorologic institute at a reasonable price.

C.1.1 Indoor steam density x_{steam}

The indoor steam density is influenced by four factors:

1. Transpiration of the plants (TRANS).

2. Water injection (WaterInj).

3. Exchange with environment through ventilation (EnvExc).

4. Condensation and evaporation on the greenhouse hull (CondEvap).

The change in the indoor steam density $[\mathrm{g}/(\mathrm{m}^3 \cdot \mathrm{s})]$ is modeled by the following equation. Please refer to section C.3 for definition of constants, auxiliary variables, and functions.

$$\dot{x}_{steam} = \frac{1}{\mathrm{GH} \cdot 3600} \cdot (\textsc{Trans} + \textsc{WaterInj} - \textsc{EnvExc} - \textsc{CondEvap}) \tag{C.1}$$

The auxiliary variables Trans, WaterInj, EnvExc, and CondEvap are calculated as follows.

$$\begin{aligned}
\textsc{Trans} &= 100 \cdot \textsc{LeafSize}[\textsc{month}] \cdot \mathrm{PM2} \cdot \textsc{LeafTrans}[\textsc{month}] \cdot \textsc{TrGrow} \\
\textsc{TrGrow} &= (1 - b_0 \cdot (x_{CO2} - 600)) \cdot \frac{\textsc{TrCur}}{\textsc{TrStd}} \\
\textsc{TrCur} &= (b_1 + b_2 \cdot x_{sun} + b_3 \cdot (x_{sun})^2 + b_4 \cdot f_{RH}(x_{steam}, x_{atempA})) \cdot \\
&\quad f_{SD}(x_{steam}, x_{atempA}) \\
\textsc{TrStd} &= (b_1 + b_2 \cdot 300 + b_3 \cdot 300^2 + b_4 \cdot 60) \cdot 10
\end{aligned}$$

$$\textsc{WaterInj} = \mathrm{CW} \cdot u_{water} \cdot (f_{SSP}(x_{atempA}) - f_{SP}(x_{steam}, x_{atempA}))$$

$$\textsc{EnvExc} = (u_{vent} + \mathrm{VM0} + \mathrm{VM1} \cdot v_{wind}) \cdot (x_{steam} - v_{steam})$$

$$\textsc{CondEvap} = \begin{cases} \textsc{Cond} & \text{if } \textsc{Cond} > 0 \quad \text{(Condensation)} \\ \textsc{Cond} & \text{if } \textsc{Cond} < 0 \text{ and } x_{cond} > 0 \quad \text{(Evaporation)} \\ 0 & \text{if } \textsc{Cond} < 0 \text{ and } x_{cond} = 0 \quad (x_{cond} = 0 \implies \text{no evap.}) \end{cases} \tag{C.2}$$

$$\textsc{Cond} = \textsc{TrPo} \cdot \mathrm{GR} \cdot \frac{f_{SP}(x_{steam}, x_{atempA}) - f_{SSP}(x_{htempA})}{0.5 \cdot \mathrm{RWS} \cdot (x_{atempA} + x_{htempA})}$$

$$\textsc{TrPo} = \frac{1.33 \cdot 3600 \cdot |x_{atemp} - x_{htemp}|^{0.33}}{\mathrm{DA} \cdot \mathrm{HCA}}$$

$$x_{htemp} = \begin{cases} -2.71 + 0.00811 \cdot v_{sun} & \\ \quad + 0.795 \cdot x_{atemp} + 0.289 \cdot v_{atemp} & \text{if } 5 \leq \textsc{month} \leq 9 \\ \frac{1}{3} x_{atemp} + \frac{2}{3} \cdot v_{atemp} & \text{Otherwise} \end{cases} \tag{C.3}$$

where $b_0 = 5.4 \cdot 10^{-4}$, $b_1 = -2.219 \cdot 10^{-4}$, $b_2 = 5.213 \cdot 10^{-6}$, $b_3 = -6.623 \cdot 10^{-9}$, and $b_4 = 8.5 \cdot 10^{-6}$. The transpiration of the plants (Trans) is influenced by the leaf size (LeafSize[·]), the number of plants per m² (PM2), the transpiration of the leaves (LeafTrans[·]), and the current growth conditions (TrGrow). The growth conditions are modeled as the ratio between the transpiration at the current state (TrCur) and the transpiration under standard conditions (TrStd), which is at 25°C, x_{sun} = 300 W/m^2, x_{CO2}=600 ppm, and $f_{SD}(\cdot)$=10 hPa. The actual transpiration of the plants is influenced by the CO_2-level, the sunlight intensity, and the relative humidity. The steam density is also influenced by the water injected (WaterInj) by the controller (u_{water}), which, of course, increases the steam density, but is limited by the saturation steam pressure ($f_{SSP}(\cdot)$) and the actual steam pressure ($f_{SP}(\cdot)$). Furthermore, the steam density is affected by the exchange with the environment (EnvExc), which is determined by the controlled ventilation (u_{vent}), the minimal air exchange (VM0), and the wind speed (v_{wind}). Finally, condensation and evaporation (CondEvap) on the greenhouse hull (the glass) influence the steam density. The amount of water on the hull is represented by the state variable x_{cond}, which is limited between 0 and 25 g/m^2 (CHM in Table C.3). The *change* in water is modeled by one variable (CondEvap), which is negative when evaporation occurs and positive when condensation occurs. The calculation of the greenhouse hull temperature (x_{htemp}) depends on the month of the simulation. The sunlight intensity (v_{sun}) influences the hull temperature during the summer period (May-September).

Important note:
The simulator must ensure that $f_{SP}(x_{steam}, x_{atempA}) \leq f_{SSP}(x_{atempA})$. Otherwise, the greenhouse might reach an infeasible state (infinite temperature, negative pressures, temperatures below 0 K, humidity above 100%, etc.). This can easily be prevented by introducing two variables, SSP and SP, in the calculation of the derivatives, assigning the function values, and checking whether SP≤ SSP. If SP>SSP then set SP equal to SSP, i.e., SP is limited by SSP.

C.1.2 Indoor air temperature x_{atemp}

The indoor temperature is influenced by the following components:

1. Heat capacity of the air and the plants (HCap).
2. Heating through the heating system (u_{heat}).
3. Heating from the sun (HSun).
4. Heat exchange with the environment through ventilation (HExVent).
5. Heat exchange through the ground (HExGround).
6. Heat exchange through the greenhouse hull (HExHull).
7. Heat change due to condensation on the greenhouse hull (HCondEvap).
8. Heat change due to change in indoor humidity (HHum).

The change in the indoor temperature is modeled as follows.

$$\begin{aligned}\dot{x}_{atemp} &= \frac{1}{\text{HCap}} \cdot (u_{heat} + \text{HSun} - \text{HExVent} - \text{HExGround} - \\ &\quad \text{HExHull} - \text{HCondEvap} - \text{HHum})\end{aligned}$$

where

$$\begin{aligned}
\text{HC}\textsc{ap} &= \text{L}\textsc{eaf}\text{S}\textsc{ize}[\textsc{month}] \cdot \text{LSW} \cdot \text{HCW} + \\
&\quad \text{GH} \cdot \text{HCA} \cdot \text{DA} + \text{GH} \cdot \text{HCS} \cdot x_{steam}
\end{aligned}$$

$$\text{HS}\textsc{un} = \text{TS} \cdot x_{sun}$$

$$\begin{aligned}
\text{HE}\textsc{x}\text{V}\textsc{ent} &= \frac{1}{3600}(u_{vent} + \text{VM0} + \text{VM1} \cdot v_{wind}) \cdot (x_{energy} - v_{energy}) \\
x_{energy} &= \text{HCA} \cdot \text{DA} \cdot x_{atemp} + x_{steam} \cdot (\text{EEW0} + \text{HCS} \cdot x_{atemp}) \\
v_{energy} &= \text{HCA} \cdot \text{DA} \cdot v_{atemp} + v_{steam} \cdot (\text{EEW0} + \text{HCS} \cdot v_{atemp})
\end{aligned}$$

$$\text{HE}\textsc{x}\text{G}\textsc{round} = \text{HG} \cdot (x_{atempA} - v_{gtempA})$$

$$\text{HE}\textsc{x}\text{H}\textsc{ull} = \text{GR} \cdot (\text{HW0} + \text{HW1} \cdot v_{wind}) \cdot (x_{atempA} - v_{atempA})$$

$$\text{HC}\textsc{ond}\text{E}\textsc{vap} = \frac{1}{3600} \cdot \text{EEW} \cdot \text{C}\textsc{ond}\text{E}\textsc{vap} \qquad (\text{C}\textsc{ond}\text{E}\textsc{vap} \text{ from Eq. C.2})$$

$$\text{HH}\textsc{um} = \text{GH} \cdot (\text{EEW0} + \text{HCS} \cdot x_{atemp}) \cdot \dot{x}_{steam}$$

The heat capacity of the air and the plants (HCap) is determined by the heat capacity of the plants, the air, and the current amount of steam in the air. The heating from the sun (HSun) is calculated as the indoor sunlight intensity (x_{sun}) scaled by the thermic effect factor (TS). The heat exchange due to ventilation (HExVent) is influenced by the controlled ventilation (u_{vent}), the minimal air exchange (VM0), and the wind speed (v_{wind}). The difference between the energy contents of the indoor air (x_{energy}) and the outdoor air (v_{energy}) also influence this exchange. The exchange at the ground (HExGround) depends on the difference between the indoor air and the temperature of the ground. The exchange through the hull (HExHull) is determined by the difference between indoor and outdoor temperature and the wind speed (v_{wind}). Furthermore, the condensation or evaporation on the hull (HCondEvap) affects the indoor temperature. Finally, the change in humidity (HHum) affects the indoor temperature.

C.1.3 Indoor CO_2 concentration x_{CO2}

The CO_2 concentration in the greenhouse is influenced by three factors:

1. Artificially injected CO_2 (u_{CO2}).

2. CO_2 consumption by the plants through photo-synthesis and transpiration (CPhoto).

3. Exchange with the environment through ventilation (CExVent).

The change in indoor CO_2 concentration is modeled by the following equation.

$$\dot{x}_{CO2} = \frac{u_{CO2} - \text{CPhoto} - \text{CExVent}}{10^{-6} \cdot \text{DC} \cdot \text{GH} \cdot 3600}$$

where

$$\begin{aligned}
\text{CPhoto} &= 100 \cdot \text{LeafSize[month]} \cdot \text{PM2} \cdot \\
&\quad \text{LeafCO2Ex[month]} \cdot \text{CPhGrow} \\
\text{CPhGrow} &= \begin{cases} \text{CPhCur} \cdot \text{CPhDec} & \text{if CPhCur} > 0 \\ \text{CPhCur} & \text{otherwise} \end{cases} \\
\text{CPhCur} &= c_1 \cdot (1 - \exp(-c_2 \cdot 0.5 \cdot x_{sun})) \cdot (1 - \exp(-c_3 \cdot x_{CO2})) \cdot \\
&\quad (x_{atemp} + c_4 \cdot (x_{atemp})^2) - c_5 \cdot (x_{atemp} + c_6 \cdot (x_{atemp})^2) \qquad \text{(C.4)} \\
\text{CPhDec} &= \begin{cases} \exp\left(-c_7 \cdot (d_1 - f_{SD}(x_{steam}, x_{atempA}))^2\right) & \text{if } f_{SD}(x_{steam}, x_{atempA}) < d_1 \\ 1 & \text{if } d_1 \leq f_{SD}(x_{steam}, x_{atempA}) \leq d_2 \\ \exp\left(-c_8 \cdot (d_2 - f_{SD}(x_{steam}, x_{atempA}))^2\right) & \text{if } d_2 < f_{SD}(x_{steam}, x_{atempA}) \end{cases} \\
\text{CExVent} &= (u_{vent} + \text{VM0} + \text{VM1} \cdot v_{wind}) \cdot \text{DC} \cdot 10^{-6} \cdot (x_{CO2} - v_{CO2})
\end{aligned}$$

with $c_1 = 0.1381$, $c_2 = 8.687 \cdot 10^{-3}$, $c_3 = 3.697 \cdot 10^{-3}$, $c_4 = -1.9083 \cdot 10^{-2}$, $c_5 = 2.073 \cdot 10^{-3}$, $c_6 = 8.7525 \cdot 10^{-2}$, $c_7 = 0.0001$, $c_8 = 0.001$, $d_1 = 5$, and $d_2 = 10$.

The change in CO_2 concentration caused by photo-synthesis (CPhoto) is determined by the leaf size, the number of plants per m^2, the CO_2 exchange at the current month, and the growth conditions (GPhGrow). The growth depends on the indoor sunlight intensity (x_{sun}), the current CO_2 concentration (x_{CO2}), and the current indoor temperature (x_{atemp}). The growth is scaled by a factor (CPhDec) when the steam pressure is not in the optimal range between d_1 and d_2 (too dry or too moist). Note that negative growth occurs at night (x_{sun}=0) or at extreme temperatures. Negative growth is the situation where the plant's respiration is larger than its photosynthesis. The CO_2 balance is also influenced by the air exchange with the environment (CExVent), which is determined by the controlled ventilation (u_{vent}), the minimal air exchange (VM0), and the wind speed (v_{wind}). The environment CO_2-level (v_{CO2}) can be kept constant at $v_{CO2} = 340$ ppm.

C.1.4 Accumulated biomass x_{biom}

The change in biomass depends on the photo-synthesis of the plants (CPhoto). It is modeled by the following equation.

$$\dot{x}_{biom} = \text{CPhoto} \cdot \frac{30}{44 \cdot 3600} \qquad \text{(C.5)}$$

Note that x_{biom} models the dry weight of the crop, here tomatoes. Naturally, the biomass represented by x_{biom} must be positive, which is achieved by a simple constraint $x_{biom} \geq 0$ in the simluator.

C.1.5 Accumulated profit x_{profit}

The change in profit is modeled as the income minus the expenses.

$$\begin{aligned}\dot{x}_{profit} &= \dot{x}_{biom} \cdot \mathrm{DWF} \cdot v_{Ptom} \cdot 10^{-3} - \\ &\quad \frac{u_{CO2} \cdot v_{PCO2} \cdot 10^{-3}}{3600} - \frac{u_{heat} \cdot v_{Pheat}}{3600}\end{aligned} \tag{C.6}$$

In our implementation, the prices are kept constant to $v_{Ptom} = 12$ DKK/kg, $v_{PCO2} = 4$ DKK/kg, and $v_{Pheat} = 0.0002$ DKK/(W·h).

C.1.6 Condensation on greenhouse hull x_{cond}

The change in the condensation amount on the greenhouse hull is modeled by the following equation.

$$\dot{x}_{cond} = \frac{\textsc{CondEvap}}{3600} \tag{C.7}$$

Note that the condensation amount on the hull is limited between 0 and 25 g/m^2 (CHM in Table C.3). The simulator must implement this constraint.

C.2 Implementation specific details

The simulator is implemented in Java 1.3 and is integrated with EVALife's EA-library (available at `www.evalife.dk`). The code is approximately 1200 lines long including the Runge-Kutta approximation. The simulator is available upon request (`ursem@daimi.au.dk`) except for the weather data, which is copyrighted by the Danish Meteorologic Institute, DMI (`www.dmi.dk`). For comparative studies we recommend to purchase the data for the measuring station Aarslev, Denmark, 2000 from DMI, which is available to researchers for about 900 DKK (110$). Please contact Rasmus K. Ursem (`ursem@daimi.au.dk`) if you decide to purchase this weather data, since the data contains a few gaps and it would be important to use the exact same data in a comparative study.

C.3 Physical constants, auxiliary variables, and functions

Description	Variable	Unit	Value
Gas constant for steam	RWS	[J/(g·K)]	0.46152
Heat capacity for air	HCA	[J/(g·K)]	1.006
Heat capacity for steam	HCS	[J/(g·K)]	1.8631
Heat capacity for water	HCW	[J/(g·K)]	4.1868
Evaporation energy for water 20°C	EEW	[J/g]	2453
Evaporation energy for water 0°C	EEW0	[J/g]	2501
Density of air at 20°C and 760 Torr	DA	[g/m^3]	1204
Density of CO_2 at 20°C and 760 Torr	DC	[g/m^3]	1840

Table C.2: Physical constants.

Description	Variable	Unit	Value
Plants per m^2	PM2		1
Transmission degree for glass	TG		0.71
Thermic degree for sunlight	TS		0.6
Coefficient for water injection	CW	[1/Pa]	0.0005
Greenhouse height	GH	[m]	3
Ratio glass surface/ground surface	GR		1.64
Heat exchange coeff. at no wind	HW0	[W/(m^2 · K)]	3
Heat exchange gradient at nonzero wind	HW1	[(W/(m^2 · K))/(m/s)]	0.2
Heat exchange coeff. at ground	HG	[W/(m^2 · K)]	3
Maximal condensation on greenhouse hull	CHM	[g/m^2]	25
Minimal air exchange at no wind	VM0	[m^3/(m^2 · h)]	2
Minimal air exchange gradient	VM1	[(m^3/(m^2 · h))/(m/s)]	2
Leaf size to water equivalent factor	LSW	[g/m^2]	1000
Dry weight factor for crop (tomatoes)	DWF		10

Table C.3: Greenhouse constants.

The following equations are used multiple times in the differential equations describing the greenhouse state.

Saturation steam pressure over water [Pa]

Saturation steam pressure in Pa over water at temperature T in Kelvin.

$$f_{SSP}(T) = \exp\left(a_1/T + a_2 + a_3 \cdot T + a_4 \cdot T^2 + a_5 \cdot \log T\right) \qquad \text{(C.8)}$$

where T is the temperature in Kelvin, $a_1 = -6094.4642$, $a_2 = 21.1249952$, $a_3 = -0.02724555$, $a_4 = 0.0000168534$, and $a_5 = 2.4575506$.

Description	Variable	Unit	Function
Absolute temp. for x_{atemp}	x_{atempA}	[K]	$x_{atemp} + 273.15$
Absolute temp. for x_{htemp} (Eq. C.3)	x_{htempA}	[K]	$x_{htemp} + 273.15$
Absolute temp. for v_{atemp}	v_{atempA}	[K]	$v_{atemp} + 273.15$
Absolute temp. for v_{gtemp}	v_{gtempA}	[K]	$v_{gtemp} + 273.15$
Indoor sunlight intensity	x_{sun}	[W/m^2]	TG$\cdot v_{sun}$
Outdoor steam density	v_{steam}	[g/m^3]	$\dfrac{v_{RH} \cdot f_{SSP}(v_{atempA})}{100 \cdot v_{atempA} \cdot \mathrm{RWS}}$

Table C.4: Auxiliary variables.

Month	1	2	3	4	5–10	11	12
LeafSize[·]	0.5	0.5	0.8	1.5	2.0	1.0	0.5
LeafTrans[·]	0.015	0.015	0.015	0.015	0.015	0.015	0.015
LeafCO2Ex[·]	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Table C.5: Plant growth variables.

Steam pressure [Pa]

$$f_{SP}(S,T) = S \cdot T \cdot \mathrm{RWS} \tag{C.9}$$

where S is the steam density (e.g., x_{steam}), T is the temperature in Kelvin, and RWS is the gas constant (see Table C.2).

Saturation deficit [hPa]

$$f_{SD}(S,T) = \frac{f_{SSP}(T) - f_{SP}(S,T)}{100} \tag{C.10}$$

where S is the steam density (e.g., x_{steam}) and T is the temperature in Kelvin.

Relative humidity [% r.H.]

$$f_{RH}(S,T) = 100 \cdot \frac{f_{SP}(S,T)}{f_{SSP}(T)} \tag{C.11}$$

where S is the steam density (e.g., x_{steam}) and T is the temperature in Kelvin.

C.4 Translation details

The greenhouse simulator described in this report is, as mentioned, translated from the German description found in [106]. This appendix summarizes the translation of the variables and their units. The section is mainly intended for readers who want to verify the translation and seek additional information in the original description. Please, note that there might be minor differences between this description and the original description. These differences may be due to a rewriting of the equations for clarification, to remove typos, and to describe extensions suggested in the presented simulator.

English	German	Unit
u_{heat}	Q	$[\mathrm{W/m^2}]$
u_{vent}	LR	$[\mathrm{m^3/(m^2 \cdot h)}]$
u_{CO2}	W	$[\mathrm{g/(m^2 \cdot h)}]$
u_{water}	RM	$[\mathrm{g/(m^2 \cdot h)}]$
x_{steam}	DDI	$[\mathrm{g/m^3}]$
$\dot{x}_{steam}$	DDDI	$[\mathrm{g/(m^3 \cdot s)}]$
x_{atemp}	TEMI	[°C]
$\dot{x}_{atemp}$	DTEMI	[°C/s]
x_{CO2}	CI	[ppm]
$\dot{x}_{CO2}$	DCI	[ppm/s]
x_{biom}	BIOM	$[\mathrm{g/m^2}]$
x_{profit}	GEWI	$[\mathrm{DKK/m^2}]$
x_{cond}	KS	$[\mathrm{g/m^2}]$
x_{htemp}	TEMG	[°C]
x_{sun}	I	$[\mathrm{W/m^2}]$

English	German	Unit
v_{sun}	IGLOB	$[\mathrm{W/m^2}]$
v_{atemp}	TEMA	[°C]
v_{gtemp}	TEMB	[°C]
v_{RH}	FA	[% r.H.]
v_{wind}	U	[m/s]
v_{CO2}	CA	[ppm]
v_{Pheat}	PR3	[DKK/(W·h)]
v_{PCO2}	PR2	[DKK/kg]
v_{Ptom}	PR1	[DKK/kg]
v_{steam}	DDA	$[\mathrm{g/m^3}]$
x_{atempA}	TAI	[K]
x_{htempA}	TAG	[K]
v_{atempA}	-	[K]
v_{gtempA}	-	[K]

Table C.6: Translation details for control, state, and environment variables. Furthermore, the auxiliary variables calculated from other variables are listed.

English	German	Unit
RWS	RWS	[J/(g·K)]
HCA	CPL	[J/(g·K)]
HCS	CPD	[J/(g·K)]
HCW	CPW	[J/(g·K)]
EEW	VDW	[J/g]
EEW0	VDW0	[J/g]
DA	DL	$[\mathrm{g/m^3}]$
DC	DC	$[\mathrm{g/m^3}]$
LeafSize	A	
LeafCO2Ex	P0	$[\mathrm{g/(dm^2 \cdot h)}]$
LeafTrans	V0	$[\mathrm{g/(dm^2 \cdot h)}]$
DWF	-	

English	German	Unit
PM2	n	
TG	BQ	
TS	KI	
CW	KR	[1/Pa]
GH	GH	[m]
GR	GF	
HW0	KW0	$[\mathrm{W/(m^2 \cdot K)}]$
HW1	KW1	$[\mathrm{(W/(m^2 \cdot K))/(m/s)}]$
HG	KB	$[\mathrm{W/(m^2 \cdot K)}]$
CHM	KSM	$[\mathrm{g/m^2}]$
VM0	-	$[\mathrm{m^3/(m^2 \cdot h)}]$
VM1	-	$[\mathrm{(m^3/(m^2 \cdot h))/(m/s)}]$
LSW	KP	$[\mathrm{g/m^2}]$

Table C.7: Translation details for physical constants, plant constants, and greenhouse constants.

English	German	Unit
$f_{SSP}(x_{atempA})$	PSI	[Pa]
$f_{SSP}(x_{htempA})$	PSG	[Pa]
$f_{SSP}(v_{atempA})$	PSA	[Pa]
$f_{SP}(x_{steam}, x_{atempA})$	PDI	[Pa]
$f_{SD}(x_{steam}, x_{atempA})$	SDI	[hPa]
$f_{RH}(x_{steam}, x_{atempA})$	FI	[%.r.H.]

English	German	Unit
c_7	EE_1	
c_8	EE_2	
d_1	$SDIG_1$	
d_2	$SDIG_2$	

Table C.8: Translation details for auxiliary variables and functions.

English	German	Unit
TRANS	trans	$[g/(m^2 \cdot h)]$
TRGROW	VREL	
TRCUR	VRELC	
TRSTD	VSTAN	
WATERINJ	lube	$[g/(m^2 \cdot h)]$
ENVEXC	wadawe	$[g/(m^2 \cdot h)]$
CONDEVAP	kondverd	$[g/(m^2 \cdot h)]$
COND	konden	$[g/(m^2 \cdot h)]$
TRPO	trpoko	[m/h]
HCAP	CG+ (seite 6)	$[J/(K{\cdot}m^2)]$
HSUN	qglob	$[W/m^2]$
HEXVENT	qluwe	$[W/m^2]$
x_{energy}	HI	$[J/m^2]$
v_{energy}	HA	$[J/m^2]$
HEXGROUND	qboden	$[W/m^2]$
HEXHULL	qduga	$[W/m^2]$
HCONDEVAP	qkonden	$[W/m^2]$
HHUM	GH+ (seite 6)	$[W/m^2]$

Table C.9: Translation details for auxiliary variables for $\dot{x}_{steam}$ and $\dot{x}_{atemp}$.

English	German	Unit
CPHOTO	gawe	$[g/(m^2 \cdot h)]$
CPHGROW	FREL (in german simulator code)	
CPHCUR	FICT	
CPHDEC	FSD	
CEXVENT	kodiwe	$[g/(m^2 \cdot h)]$

Table C.10: Translation details for auxiliary variables for $\dot{x}_{CO2}$.