Chapter 3

Evolutionary Computations

3. Evolutionary Computation

The design of intelligent controllers based on unconventional control techniques is undoubtedly becoming common and these developments rely heavily on the use of stochastic methods of soft computing in seeking optimum results. These methods offer a new and very exciting prospect for control engineering, leading to solutions to problems that cannot be solved by conventional analytical or numerical optimization methods.

Although stochastic methods of optimization are computer intensive, the impressive progress that has been observed in computer hardware also in past decades, which has led to the availability of extremely fast and powerful computers that make stochastic techniques very attractive for control applications and especially for real time control issues.

3.1 Evolutionary Algorithms

Evolutionary algorithms are iterative and stochastic optimization techniques inspired by the concepts of from Darwinian evolutions 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 on hand. In 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 – also referred to as *fitness function*. Figure 3.1 illustrates the initialization and the iterative cycles in Evolutionary Algorithms.

Historically, Evolutionary Algorithms were first suggested in the 1940[118]. However, the founding fathers of modern Evolutionary Algorithms are considered to be Lawrence Fogel – Evolutionary Programming [119], Ingo Rechenberg and Hans-Paul Schwefel – Evolutionary Strategies [120] and by John Holland – Genetic Algorithm [121]. Later Evolutionary Algorithms (EAs) and Evolutionary Computation (EC) were introduced as unifying terms for the forest of optimization techniques inspired by biological evolution.

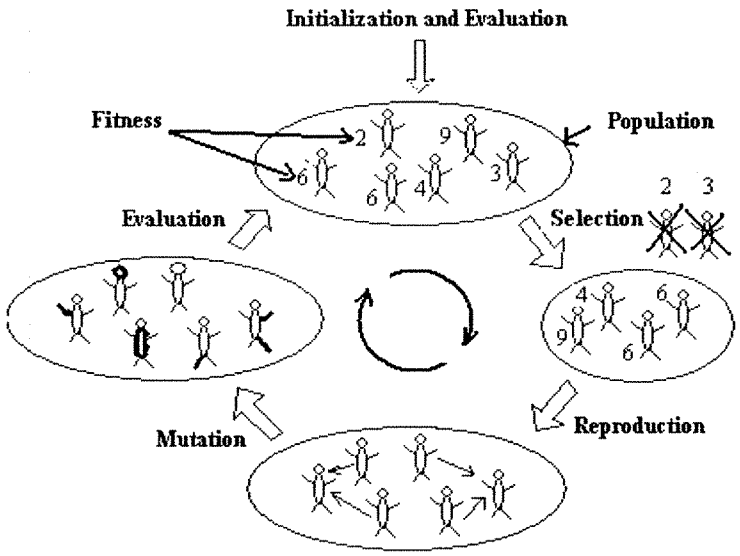

Figure 3.1: Iterative Cycles in Evolutionary Algorithms

Evolutionary computation is a generic term for computational methods that use models of biological evolutionary processes for the solution of complex engineering problems. The techniques of Evolutionary Computation have in common the emulation of the natural evolution of individual structures through process inspired from natural selection and reproduction. These processes depend on the fitness of the individuals to survive and reproduce in a hostile environment. Evolution can be viewed as an optimization process that can be emulated by a computer. Evolutionary computation is essentially a stochastic search technique with remarkable abilities for searching for global solutions.

There has been a dramatic increase in interest in the techniques of Evolutionary computation since their introduction in the mid 1970s. Many applications of the techniques have been reported, including solving problems of numerical and combinatorial optimization, the optimum placing of components in VLSI devices, the design of optimal control systems, economics, modeling ecological systems, machine learning etc.

The idea behind Evolutionary computation is best explained by the example quoted by Michalewicz in 1992 "*Do what the nature does. Let us take rabbits as an example: at any given time there is a population of rabbits. These faster, smarter rabbits are less likely to be eaten by foxes, and therefore more of them survive to do what rabbits do best: make more rabbits. Of*

course some of the slower, dumber rabbits will survive just because they are lucky. This surviving population of rabbits starts breeding. The breeding results in a good mixture of rabbit genetic material: some slow rabbits breed with faster rabbits, some fast with fast, some dumb rabbits with smart rabbits and so on. And on the top of that, nature throws in a 'wild hare' every once in a while mutating some of the rabbit genetic material. The resulting baby rabbits will on average be faster and smarter than those in the original population because faster, smarter parents survived the foxes..." By analogy, in evolutionary computation, solutions that maximize the measure of fitness will have higher probability of participating in the reproduction process for new ones. This is fundamental premise in Evolutionary Computation. Solutions of an optimization problem evolve by following the well known Darwinian principles of "survival of fittest". In following sections we are to discuss the basic principles, the principle techniques and operators of evolutionary computation. The most popular Evolutionary algorithm is Genetic Algorithm.

Genetic Algorithm derives their name from the genetic processes of natural evaluation. They were developed from Holland[121] and have been implemented successfully in a broad range of control applications, e.g. the design of neural and fuzzy controllers, for tuning of industrial controllers and also for the creation of hybrid fuzzy/ evolutionary and neural / evolutionary controllers etc. The rapid progress in the computer technology permitted the use of the evolutionary algorithms in difficult large scale optimization problems, real time control applications etc. also.

The terminology in the field of evolutionary computation is derived from biology and genetics. Although evolutionary algorithms appear to be extremely simple compared with their biological counterpart, they are, however, sufficiently complicated so as to yield solutions where conventional numerical methods have been known to fail. Evolutionary algorithms are subset of evolutionary computations and belong to the generic fields of the *simulated annealing* and *artificial life*. The search for an optimum solution is based on the natural processes of biological evolution and is accomplished in a parallel manner in the parameter search space. The terminology used in evolutionary computation is familiar. Thus, candidate solutions of an optimization problem are termed *individuals*. The *population* of solutions evolves in accordance

with the laws of natural evolution. After initialization, the population undergoes *selection*, *recombination* and *mutation* repeatedly until some termination condition is satisfied. Each iteration is termed a *generation*, while the individuals that undergo recombination and mutation are named parents that yield *offsprings*.

Selection aims at improving the average quality of the population, giving the individuals with higher quality increased chances for replication in the next generation of solutions. Selection has the feature of focusing the search in promising areas of the parameter search space. The quality of every individual is evaluated by means of a *fitness function*, which is analogous to an *objective function*. The assumption that better individuals have increased chances to reproduce even better offsprings is based on the fact that there is a strong correlation between the fitness of the parents and that of their offspring. In Genetics this correlation is termed as *heredity*. Through selection, exploitation of the numerical/ genetic information is thereby achieved.

Through *recombination*, two parents exchange their characteristics through random partial exchange of their numerical/ genetic information. The recombination of the characteristics of two parents of high fitness assumes that if a portion of the numerical/ genetic information responsible for high values of fitness recombines with an equivalent parent, then the chances that their offspring will have as high of even better or higher fitness values are correspondingly increased. Recombination is also referred to as *Crossover*. Likewise, through *mutation*, an individual undergoes random change in one of its characteristics i.e. in a specific section of the structure. Mutation aims at introducing new characteristics to the population that does not necessarily exist in the parents, leading thereby to an increase in the variance of the population. Exploration of the search space is achieved through the operators of recombination and mutation.

The cornerstone of Evolutionary Algorithm is the iterative procedure in exploring the search space while simultaneously exploiting the information that is being accumulated during the search. This is in fact, where their functionality lies. Through *exploration*, a systematic sampling of the search space is achieved, while through *exploitation* the information that has been accumulated during exploration is used to search for new areas of interest in which exploration can be continued. Unlike exploitation, exploration includes random steps. It should be emphasized that random exploration does not mean exploration without direction, since the technique focuses on the most promising directions.

3.2 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. 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 desirable for many real-world problems. Valuable alternatives are stochastic search methods such as simulated annealing, tabu search, and evolutionary algorithms. Among these techniques, Evolutionary Algorithms seem to be a particularly promising approach for several reasons. Evolutionary Algorithms are very general regarding the problem types they can be applied to (continuous, mixed-integer, combinatorial 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, Evolutionary Algorithms 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 Evolutionary Algorithm'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 Evolutionary Algorithms robustness on such problems. In multi objective problems, Evolutionary Algorithms 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, Evolutionary Algorithms 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 Evolutionary Algorithms 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. Furthermore, EAs should not be applied blind foldedly to any problem. Many simpler and faster techniques exist and they should typically be 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. 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.

3.3 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 consists 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 data structure 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 better set of solutions to the problem. Figure 3.2 illustrates the pseudo code of a simple EA.

EA Main

```
t=0
initialize population P(0)
evaluate population P(0)
while (!(termination condition))
{
   t = t +1
   select population P'(t) from P(t-1)
   create population P(t) from P'(t)
   evaluate population P(t)
}
```

Figure 3.2 Pseudo Code of Simple EA

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 consists 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 or crossover* and *mutation* of the solutions in the selected population *P'(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 improves over time and typically tends to stagnate towards the end of the run, refer figure 3.3.

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.

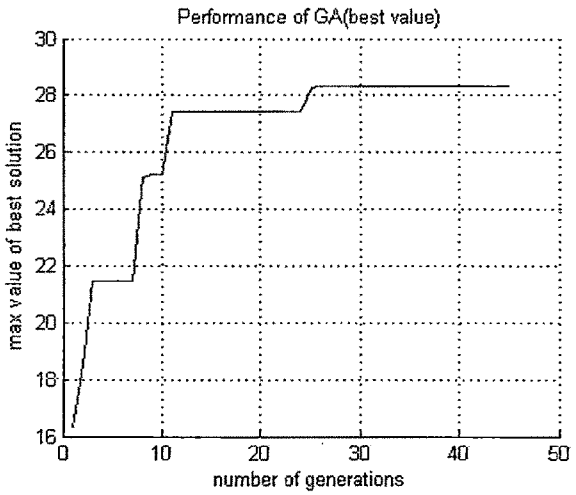

Figure 3.3 Fitness improvement during the GA run.

3.4 Terminology of Evolutionary computation

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 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 3.4 for the example.

The given problem is often described in an abstract way, top of figure 3.4. 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 for example, a domain expert may derive an *n*-dimensional parameterized model of the process that generated the data (the formalization used in figure 3.4). A completely different approach may be to use an artificial neural network to approximate the true system.

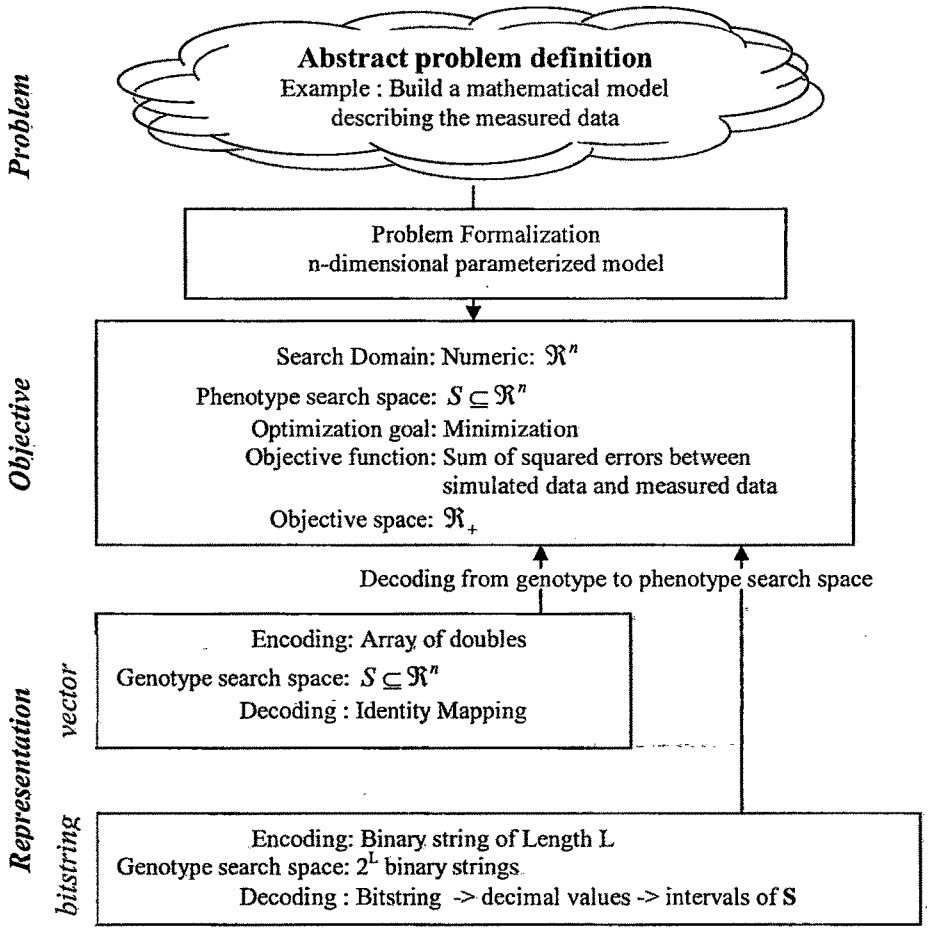

Figure 3.4: The problem, the objective and the representation

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 $\Re^n$. The actual *search space* S is usually defined by an interval for each of the n variables, i.e. $S \subseteq \Re^n$. The search space is called the *phenotype search space*[1] when there is a difference between the domain of and the search domain. 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

[1] 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.

meaningful functions for a given problem. To this end, a number of important issues arise when designing fitness functions. They are discussed in later. 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 $\Re$. For multi objective problems, the objective space is a subset of $\Re^m$, where m is the number of objectives. Settling on problem formalization and a phenotype search space narrows the number of meaningful representations. A representation consists of an *encoding* data structure and a *decoding* function. The encoding is used to store the actual solution in. The encoding defines the *genotype search space*[2] 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[3] if the search space is a natural subset of the search domain (e.g., an interval in R). In all other cases, a decoding scheme must be implemented. In the system identification example, the most straight forward 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 $\Re^n$. Finally, the choice of encoding determines the set of possible evolutionary operators. Encodings and evolutionary operators are closely connected because the operators access the data structure 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 [122]. The next sections describe the encodings and operators relevant for system identification and control problems. Furthermore, the remaining components of evolutionary algorithms are also introduced.

3.5 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

[2] 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.

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

implement specialized mutation and recombination operators for the particular encoding. The wide variety of EA-applications has 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 control applications.

3.5.1. 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 $2^{16} = 65536$. 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 $2L$ 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 3.1.

$$x_{value} = x_{\min} + \frac{x_{\max} - x_{\min}}{2^L - 1}\left(\sum_{i=0}^{L-1} x[i] \cdot 2^{L-1-i}\right) \quad \ldots(3.1)$$

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 01100111 = 103 is translated as follows:

$$x_{value} = -2 + \frac{2 - -2}{255} \cdot 103 \approx -0.3843 \quad \ldots(3.2)$$

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 00011111 =31 is not adjacent to 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 3.5 shows a Gray decoding algorithm. 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 00000001 = 1 and 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 3.5: Pseudo code for Gray decoding in linear time.

Following accepted terminology, the binary string is named a *genotype*, the decoded information the *phenotype*, while every individual solution is a *chromosome*. When the

optimization problem is multidimensional, then the partial strings are concatenated as shown in figure 3.6.

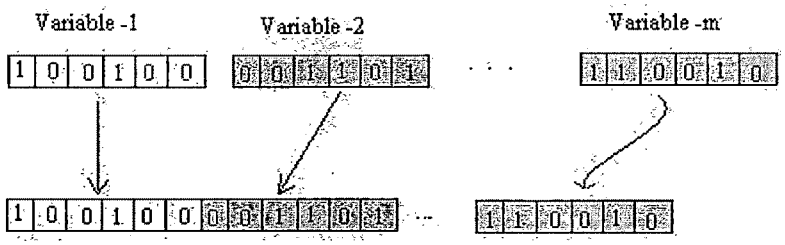

Figure 3.6: Creation of bitstring in multi dimensional problem of optimization

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.

$$t = 1 + \left\lfloor \frac{\ln(u)}{\ln(1-p_m)} \right\rfloor \qquad \ldots(3.3)$$

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 $(t'\text{-}L)^{th}$ bit. Empirical studies[123] have suggested values for $p_m \in [0.001, 0.01]$. Later Bäck[122] showed that the value p_m=1 /L is optimal for simple problems. Hence normally 1/L is used as lower bound on p_m.

N-point and uniform Crossover

A widely used crossover operator for binary and also for 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, refer figure 3.7.

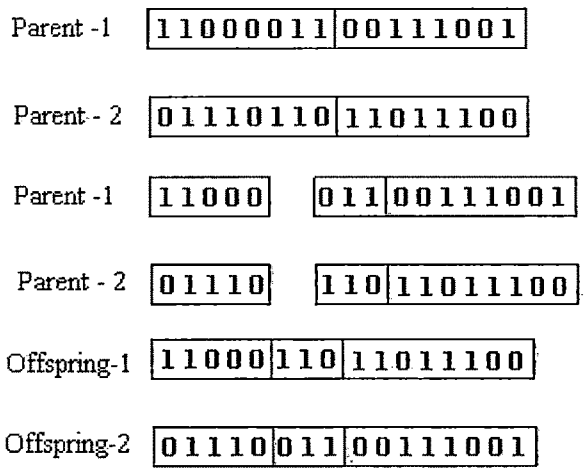

Figure 3.7: One point Crossover Operator

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 **x**, i.e., $x' = x + M$. 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 mutation. The common choice for the generation of M is the Gaussian distribution N(0,α).

A rather uncommon mutation is based on the uniform distribution $U(-\alpha, \alpha)$,where M is a value between - α and α with equal probability. A special case of the uniform mutation is 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 to too high, the algorithm has difficulties in fine-tuning the solutions while if set to 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 [120].

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[4]. Two decreasing functions are displayed in figure 3.8.

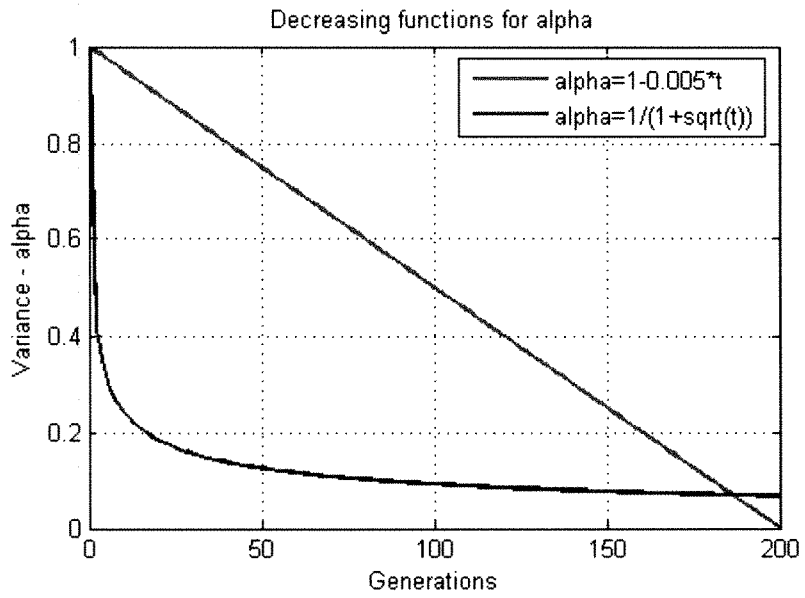

Figure 3.8 : Decreasing Functions

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.

[4] Idea also referred as annealing

$$x' = \omega \cdot X_1 + (1-\omega) \cdot X_2 \qquad \ldots(3.4)$$

where, w is the weight and x_1 and 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, refer figure 3.9(b).

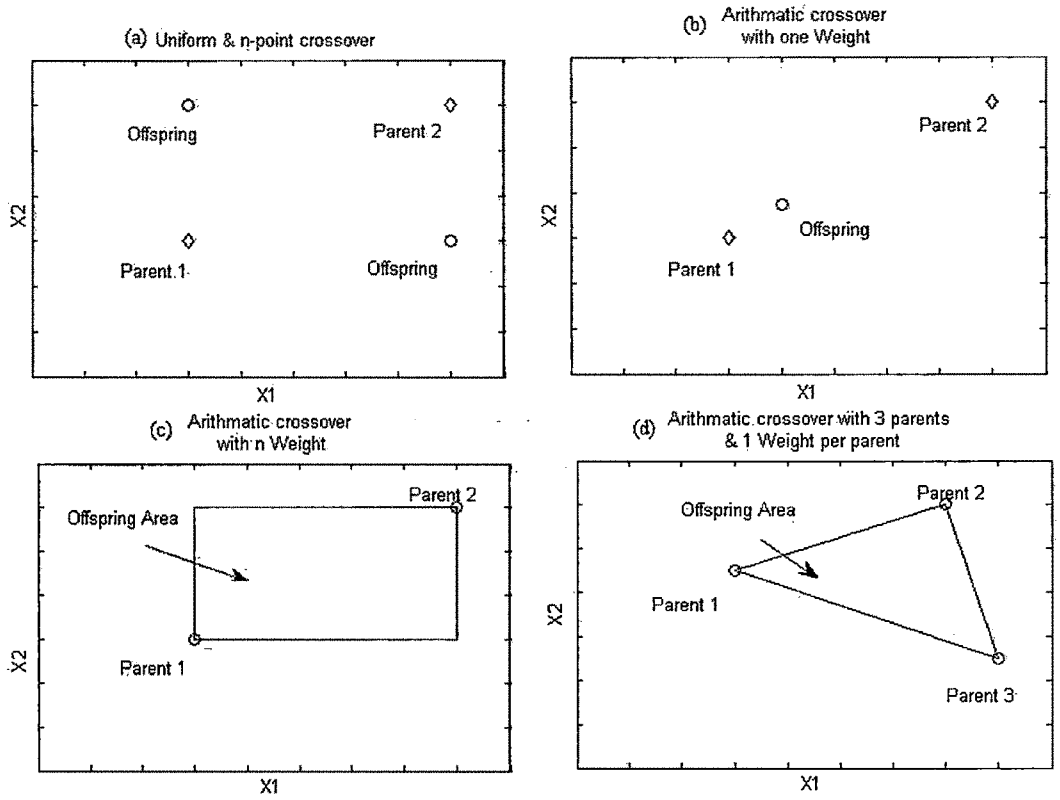

Figure 3.9 Crossover for real valued vectors

A variant of arithmetic crossover generates a specific weight w_i for each gene x_i in the genome vector $x' = (x_1', x_2', \ldots, x_n')$

$$x_i' = w_i \cdot x_{1i} + (1-w_i) \cdot x_{2i} \qquad \ldots(3.5)$$

In this variant, the offspring is placed at a random location inside the hypercube spanned by the two parents, refer figure 3.9(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 3.6.

$$x' = \sum_{j=1}^{k} w_j x_j, \text{ where } w_j \in [0,1], \sum_{j=1}^{k} w_j = 1 \qquad \ldots(3.6)$$

In this setup the offspring is created in the convex hull defined by the k parents, figure 3.9(d).

3.5.2 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 [119], 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* constructs. 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 construct 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 is 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 [124]. 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 3.10 illustrates an example of the four operators.

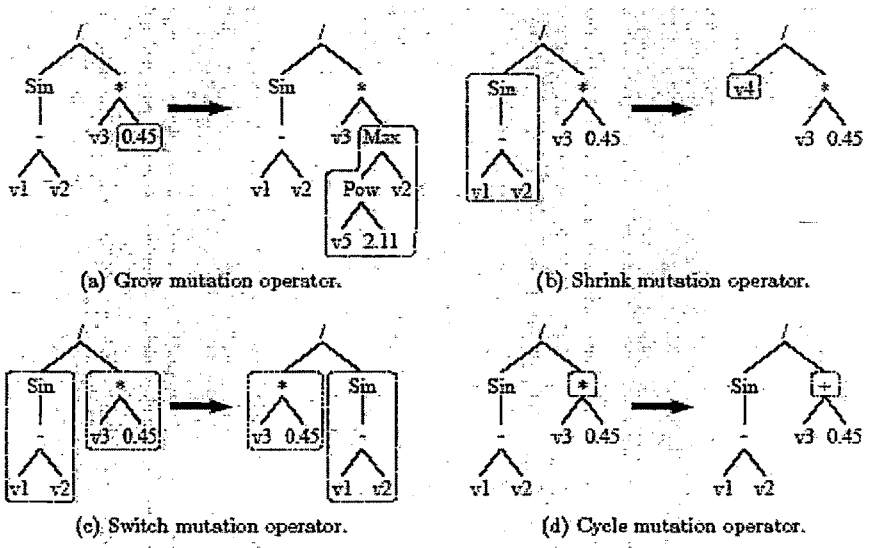

Figure 3.10: Grow, shrink, switch and cycle mutation

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, refer figure 3.11.

3.6 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, refer figure 3.12.

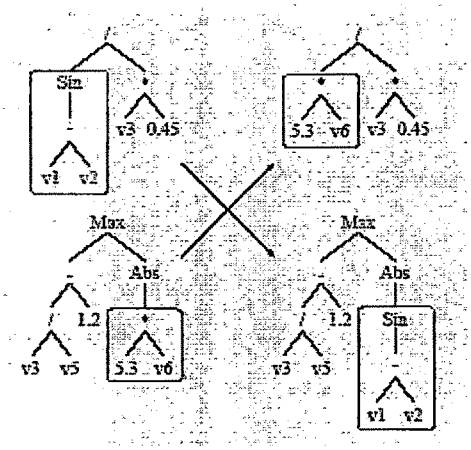

Figure 3.11: Subtree crossover

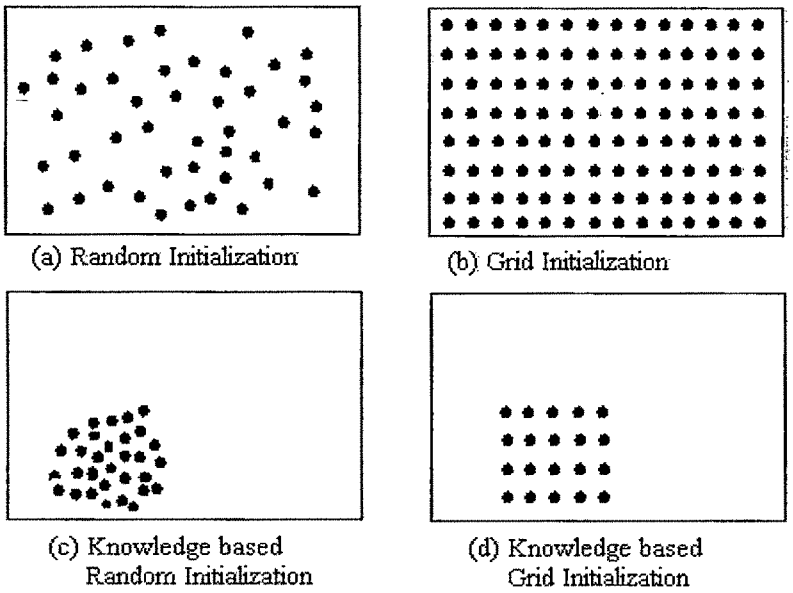

Figure 3.12 : Initialization Methods.

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 will 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 [120] study. Finally, including solutions created by other search techniques seem to be an extremely promising approach, although rarely used.

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.

3.7 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 present 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 accidentally 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 [120].

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.

3.7.1 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 3.13 shows the pseudo code 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 (i=0;i<|P(t)|;i++){
        Pick two random individuals K1 and K2 in P(t)
        Compare the fitness of K1 and K2.
        Insert a copy of the fitter individual in P(t+1) at position i.
    }
```

Figure 3.13: Pseudo code for 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 to 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.

3.7.2 Proportional selection or Roulette wheel 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.

$$p_{survival}(K) = \frac{fitness(K)}{\sum_{i=1}^{popsize} fitness(K_i)}, \quad i.e. \quad \sum_{i=1}^{popsize} p_{survival}(K_i) = 1 \qquad \ldots(3.7)$$

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, 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 much popular.

3.7.3 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. The selection is performed using the roulette wheel approach.

Table 3.1: Example of ranking scheme for population size of 20 individuals

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

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.

3.7.4 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 at every iteration. The evolutionary operators create λ potential solutions from the parent population with size μ. Afterwards the (μ + λ) individuals are sorted and λ individuals with the lowest fitness are discarded. 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.

3.7.5 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 [125], mixing of food-colors, and more experimental applications in evolutionary art [126].

3.8 Use of EA for Design of Intelligent controllers

3.8.1 Fuzzy Controllers

The traditional design of the fuzzy controllers is based primarily on heuristic techniques. The main problem in fuzzy control design is the difficulty in defining a host of parameters, such as the number and shape of fuzzy sets of the inputs and outputs, the form of inference engine and the defuzzification mechanism. Their choice has considerable influence on the overall behavior of the controller. Unfortunately, at present the theoretical foundation determining the optimum solutions does not exist and consequently only experience with similar problems can be used in

their design. Often, the result is acceptable but there is no guarantee that there do not exist a better solution.

The flexibility of Evolutionary Algorithms is the principal motivating factor for their use in determining the optimum values of the parameters of the fuzzy controllers. As we discussed in the previous chapter, the linguistic values of a fuzzy variable are defined by their membership functions. When a membership function is triangular, then the three parameters α, β and γ as shown in figure 3.14, uniquely specify the fuzzy set. The parameters of every fuzzy set are encoded into binary strings of sufficient length to give the desired precision. For n fuzzy sets and m variables and encoding with p bits, the total length of the chromosomes is clearly $n*m*p$ bits.

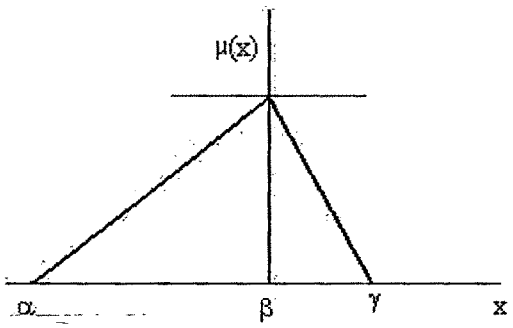

Figure 3.14: Parameters of a Triangular Fuzzy set

If the dynamic behavior of the process is known and there exists a macroscopic model of the process, then we may use the single objective function to evaluate the performance of the closed system to a step disturbance using familiar ITAE and/or ISE criteria.

3.8.2 Neural Controllers

The performance of a neural controller depends critically on the architecture of the Artificial Neural Network (ANN) used, i.e. the number of neurons in every layer, the number of the layers and the topology of the network, the form of compression function and the algorithm used to train the network.

The determination of these parameters is based on the knowledge and experience of the designer and any discussion on optimum design is of no consequence. The determination of the parameters of a neural controller can, however, be transformed into an optimization problem for which Evolutionary Algorithm, we discussed, is attractive.

Genetic Algorithms are considered very efficient in rapidly finding the approximate optimum solution of an optimization problem, but they are generally slow in finding precise solution. For better convergence, Genetic Algorithms can be combined with local search techniques such as hill climbing, which is ideally suited for finding out optimum solution in the small. For ANN with m layers and n_i neurons in the i^{th} layer, the total number of parameters in the optimization problem is...

$$N = \sum_{i=0}^{m-1} (n_i + 1) \cdot n_{i+1} \qquad \ldots(3.8)$$

If every weight is decoded into a binary string with L digits, the total length of the chromosome is clearly NL. It is obvious that the use of Genetic Algorithms in problems involving ANNs with thousands of neurons becomes difficult and extremely time consuming and hence not suitable for real time control applications. However ANNs, which are used in Neural Controllers, usually have a very simple architecture with rarely more than 30 neurons or more than one hidden layer, in which case GAs are very efficient.

3.9 Stability and Optimality in GA based control

GAs quickly locates near optimal solutions of complex problems, described by a specified fitness or objective functions. This function provides a measure of fitness of an individual and directs evolution of the population to the regions of the fittest individuals. Because the fitness value is the only information required of the application and because the GA consists of a population of many solutions, the GA performs well on nonlinear and discontinuous response surfaces, as well as surfaces corrupted with high degree of noise. These features also allow the GA locate the global optimum, where as other techniques often converge to local optima.

The control research community employ genetic algorithm on many levels. While the capabilities of the GA based controllers have been demonstrated in simulation and offline analysis using a system model, there remains lack of real on-line applications where model information is unavailable. The subject of online control is seldom addressed in past, the reasons of this are clear: GA controllers in their present state of research require robust systems that can withstand the evaluation procedure for the evolving controllers.

3.9.1 Convergence of the GA controller Population

Each member of the GA controller population corresponds to an individual controller, and each control the system for a specified number of time steps, over which the fitness of that individual is evaluated. Therefore, a necessary element to the stability analysis is a guarantee of convergence of the GA controllers to a population of stable controllers. Once stability has been established, the convergence of the GA controller's populations towards the optimal controllers is considered.

3.9.2 Stability of GA controllers

A common assumption in GA convergence analysis is that the fitness of a particular schema will remain above or below average by a constant amount through out the evolution of populations.

$$f(H,t)=(1+c)\cdot \bar{f}(t), \quad \forall t \qquad \ldots(3.9)$$

Where, $f(H,t)$ is the fitness of schema H over the generation t, $\bar{f}(t)$ is the total average fitness over the generation t. A schema represents a subset or group of individuals in the solution space and corresponds to attributes of those individuals. Goldberg [45] and Holland [121] illustrate the effect of genetic operators on schemata represented by a particular population. Let the number of schema H in the population $A(t)$ at generation t be given by $m(H,t)$. A fitness value can be associated with the schema H by averaging fitness of each string $A(t)$ representative of H at generation t. Under fitness proportionate selection pressure alone (no mutation, no crossover), the number of instances of a schema is expected to increase of decrease in accordance with that schema's relative fitness in the current population. Therefore, the expected number of schema H in population $A(t+1)$ is

$$m(H,t+1)=m(H,t)\cdot\frac{f(H)}{\bar{f}} \qquad \ldots(3.10)$$

Extrapolating equation 3.9 from equation 3.10, the effect of fitness proportionate selection on the expected number of a schema H represented in the population from generation can be modeled as

$$m(H,t+1)=(1+c)\cdot m(H,t) \qquad \ldots(3.11)$$

or in terms of the expected number of H in the initial population

$$m(H,t)=(1+c)\cdot m(H,0) \qquad \ldots(3.12)$$

neglecting the effects of crossover and mutation. For an average individual (c > 0), the number of schema H is expected to increase exponentially, while for below average solutions (c < 0) this number will decrease exponentially [45]. Also the expected number of average members of the populations (c ≈ 0) will be more or less constant. While this assumption greatly simplifies the analysis, it has received criticism for its validity and the accuracy of results obtained. An alternative approach to facilitate the GA convergence analysis is to examine the behavior of specific fitness functions that are of interest with respect to desired objective. For the GA controller, the objective may be defined in terms of stable controllers surviving and the unstable controllers perishing as quickly as possible.

At a given time, the GA controllers population may be divided into two sets, one of the proportionate $r_s(t)$ corresponding to the proportion stable individuals in the population and represented by $\boldsymbol{H_s}$, the other, $\boldsymbol{H_u}$, consisting of the unstable members of having proportionate $r_u(t)=1- r_s(t)$. Let $f_s(t)$ be the fitness of $\boldsymbol{H_s}$ at generation t calculated as the average of the stable members of $A(t)$ and likewise $f_u(t)$ be the fitness of $\boldsymbol{H_u}$. Therefore, the average fitness of the $A(t)$ is ...

$$\bar{f}(t)=r_s(t)f_s(t)+(1-r_s(t))f_u(t) \qquad ...(3.13)$$

Let the assumption hold: $f_s(t)=\alpha_s \cdot f_u(t), \forall t$ ...(3.14)

Where, the *stability fitness ratio* $\alpha_s \succ 1$. This is a more reasonable assumption then that of equation 3.9, since as the generation progress the fitness of stable individuals is more likely to remain a constant percentage above the unstable members of the population than to remain a constant percentage above the population average [127]. The average fitness in terms of stable schemata statistics becomes...

$$\bar{f}(t)=\frac{(\alpha_s-1)\cdot r_s(t)+1}{\alpha_s} f_s(t) \qquad ...(3.15)$$

Using the schema theory results to define the effect of proportionate selection on the number of stable schema from generation t to $t+1$, the following relationships is obtained ...

$$m(H_s,t+1)=m(H_s,t)\cdot\frac{f_s(t)}{\bar{f}(t)} \qquad ...(3.16)$$

$$\text{or } r_s(t+1)=r_s(t)\cdot\frac{f_s(t)}{\bar{f}(t)} \qquad ...(3.17)$$

Substituting 3.15 for the average fitness yields :

$$r_s(t+1)=\frac{\alpha_s \cdot r_s(t)}{(\alpha_s-1)\cdot r_s(t)+1} \qquad \ldots(3.18)$$

The Above equation provides the expected proportion of stable individuals in the GA population at generation $t+1$ given α_s and the proportion of stable individuals at generation t. In terms of the initial stable proportion population, the stable proportion at generation t is determined as

$$r_s(t)=\frac{\alpha_s^t \cdot r_s(0)}{(\alpha_s^t-1)\cdot r_s(0)+1} \qquad \ldots(3.19)$$

The GA controller, under pressure of fitness proportionate selection alone, is expected to converge to a population consisting entirely of stable controllers [127].

While the above results are essential to the stability analysis of the control system, the analysis should also include the effects of any genetic operators such as crossover and mutation, which are employed to evolve the population. In order to include the effects of genetic operator on the GA controller, the above analysis is required to be modified accordingly.

Let $p_s(t)$ be the probability that a stable controller selected from the GA controller population at generation t will remain stable after genetic recombination. The expected proportion of stable schema in the population is then given by...

$$r_s(t+1)=\frac{\alpha_s \cdot r_s(t)}{(\alpha_s-1)\cdot r_s(t)+1} p_s(t) \qquad \ldots(3.20)$$

If the probability of remaining stable, $p_s(t)$, is not allowed approach the one, the GA controller is not guaranteed to converge to a population of stable controllers [127]. To have guarantee for the same consider $p_s(t)$ defined in usual manner for simple crossover and mutation [45], the effects of genetic operators must decrease as the population converges in order to have $p_s(t)\to 1$. Also, the product $\alpha_s \cdot p_s(t)$ must exceed one to guarantee the convergence, which implies that the advantage gained from the stability fitness ratio outweighs the destructive nature of probability of survival. The GA controller is then expected to converge to a population of stable controllers as shown in [127].

3.9.3 Optimality of the GA controllers

Once the GA controller population has converged to stable controllers, the convergence near to optimal controllers becomes the predominant. To understand the optimality issues, let us

assume that there exists a unique optimal controller, the optimal controller provides the maximum value of fitness and the GA controller has stabilized prior to the optimality analysis.

Let $N(K^*,\sigma_f)$ to be a neighborhood of near optimal controllers about the optimal controller, K^*, where σ_f is less than 1 and represents the degree of optimality, such that...

$$f(N(K^*,\sigma_f)) \geq \sigma_f f(K^*) \qquad \ldots(3.21)$$

Therefore, the set defined by $N(K^*,\sigma_f)$ is the set of all controllers having fitness value bounded from below by $\sigma_f f(K^*)$. Also, define the average fitness of the near optimal controllers belonging to $N(K^*,\sigma_f)$ to be bounded from below by some value times the average fitness of the stable controllers.

$$\bar{f}_{N(K^*,\sigma_f)} \geq \alpha_{N(K^*,\sigma_f)} \cdot f_s \qquad \ldots(3.22)$$

Where, the $\alpha_{N(K^*,\sigma_f)}$ is the optimality fitness ratio and $\alpha_{N(K^*,\sigma_f)} > 1$

The relations defined in section 3.9.2 concerning the proportion of stable controllers in the GA controller population is required to be modified to incorporate the notion of GA controller optimality. Let the $r_{N(K^*,\sigma_f)}(t)$ represents the proportion of the population that are members of $N(K^*,\sigma_f)$ at generation t. Therefore, once the GA controller has stabilized, the following applies...

$$r_{N(\cdot)}(t+1) \geq \frac{\alpha_{N(\cdot)} \cdot r_{N(\cdot)}(t)}{(\alpha_{N(\cdot)}-1)\cdot r_{N(\cdot)}(t)+1} p_{N(\cdot)}(t), \quad where, N(\cdot) = N(K^*,\sigma_f) \qquad \ldots(3.23)$$

A set of conditions similar to those place on the stability analysis, must be met for the above relation to be converge to one. In particular, the probability of survival $p_{N(K^*,\sigma_f)}(t)$, now dependent upon the degree of convergence, must approach to one and the product $\alpha_{N(K^*,\sigma_f)} \cdot p_{N(K^*,\sigma_f)}(t) > 1$.

3.10 Simulation of PID Controller using GA

A numerical simulation for the fine tuning of the gains for the simple PID controller is implemented and results are compared with the same derived using Zeigler Nichols Criterion. Comparison of the step response for both shows how simple – GA can improve the performance

of the controllers. I have used MATLAB and MATLAB GAOT V2 GA tool box for implementing the said example.

The Transfer function of the system under consideration is ...

$$T(s)=\frac{1}{s^4+6\cdot s^3+15\cdot s^2+6\cdot s+1} \qquad \ldots(3.24)$$

Parameters for the genetic algorithm used are ...

Population Size : 100

Initialization of Population : Random

Fitness function or Evaluation function : ITAE

Termination function : Maximum Generation / Iterations

Maximum Iterations : 100/ 250;

Selection Criterion : Normal Geometric Selection

Crossover : arithmetic Crossover

Mutation : Uniform Mutation

Simulation Time[5] : 159.4393 seconds

349.8631 seconds

Proportional Gain Kp, Integral Gain Ki and Derivative Gain Kd are obtained using both Zeigler Nichols Criterion and Genetic Algorithm. Their values are

	Z-N Criterion	GA(100)	GA(250)
Proportional Gain Kp	6.00	10.97611	11.00674
Integral Gain Ki	1.91	3.3602	2.9383
Derivative Gain Kd	4.74	14.66842	13.83305

If take a look at step response of the systems under consideration in figure 3.15 & 3.17 then we can say that the response of the system is getting stable more faster with the gain obtained with the help of GA, also we can see that rise time of the system has improved, peak overshoot is reduced significantly, settling time is also reduced. Only negative observation with the usage of GA is time required to obtain the final solution.. Figure 3.16 & 3.18 shows change in different gain values at every generation. We can see the improvement in the gain factors but that is at the cost of additional time.

[5] on Intel Pentium IV 1.50GHz processor, with 450 MB of RAM

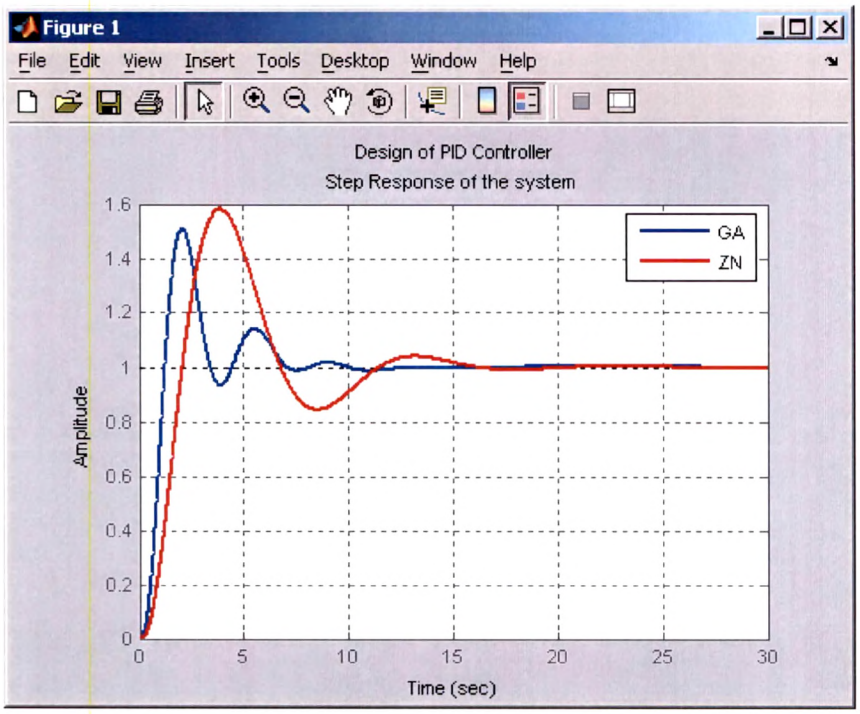

Figure 3.15: Step response of PID controller

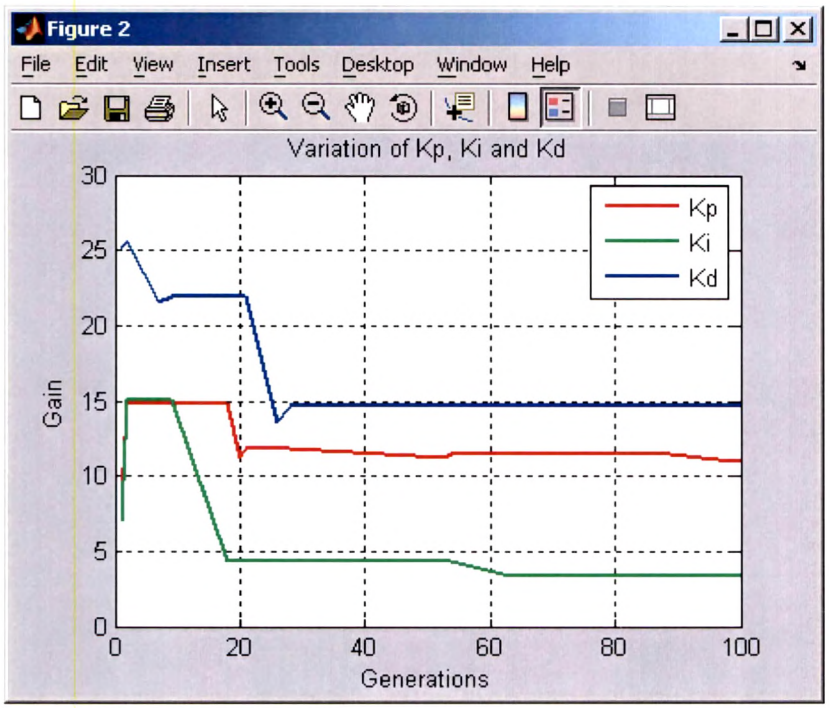

Figure 3.16 Variation of Gains of PID controller

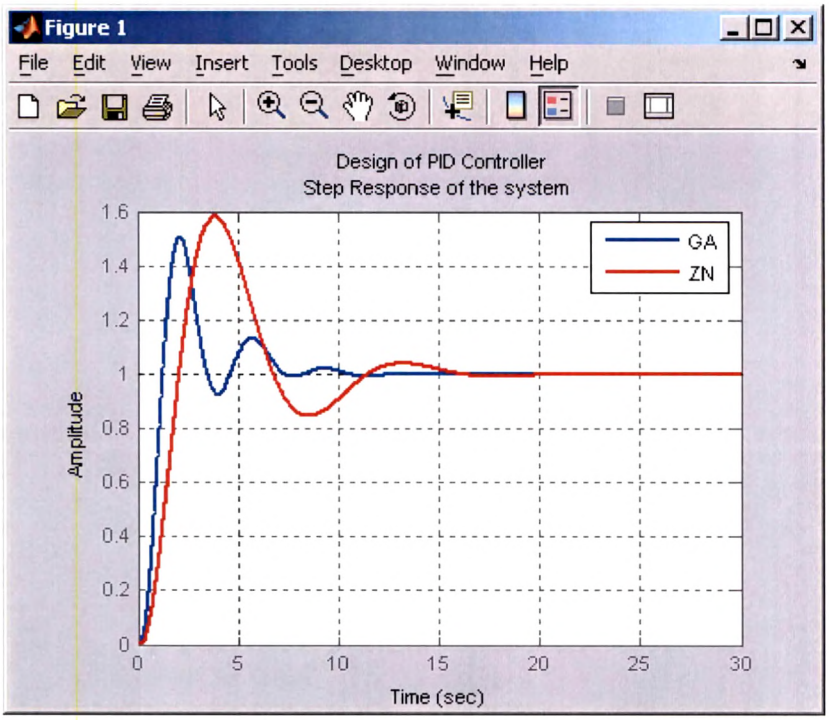

Figure 3.17: Step Response of PID Controller

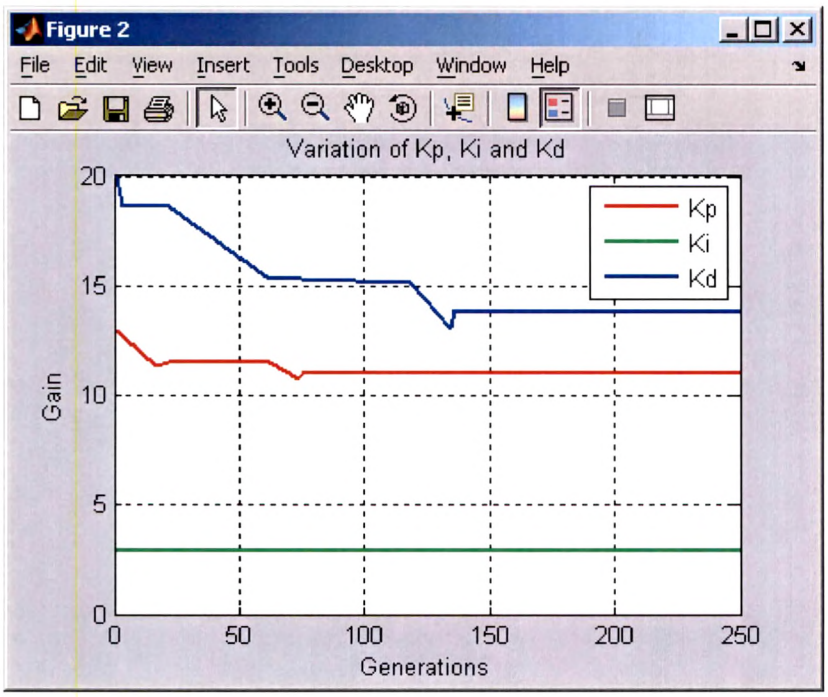

Figure 3.18: Variation of Gains of PID Controller

3.11 Summary

The main characteristic that makes Evolutionary Algorithms attractive for a broad class of optimization problems is their robustness because of

- They do not require specific knowledge or derivative information of the objective function.
- Discontinuities, noise or other unpredictable phenomena have little impact on the performance of the method.
- They perform in parallel in the solution space, exploring the search space with simultaneous exploitation of the information derived and they do not become entrapped in local optima.
- They have good performance in multidimensional large scale optimization problems.
- They can implement in many different optimization problems without big changes in their algorithmic structure.

  But the major disadvantages of the GAs are that...

- They face some difficulties in locating the precise global optimum, although it is easy for them to locate the vicinity, where the global optimum exists and
- They require a great number of evaluations of the objective function and therefore require considerable computational power.

Above discussion also suggests that if a Hybrid approach is adopted to carry out the design of real time control system then the best of Neural, fuzzy and GA can give us better control system for most of the application but again with the help of reasonably good computational power only.