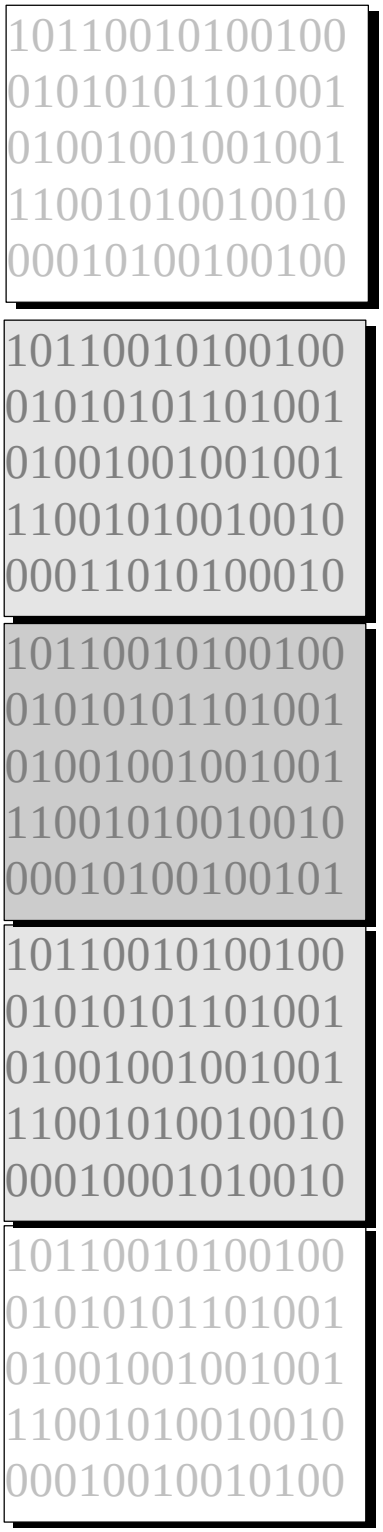
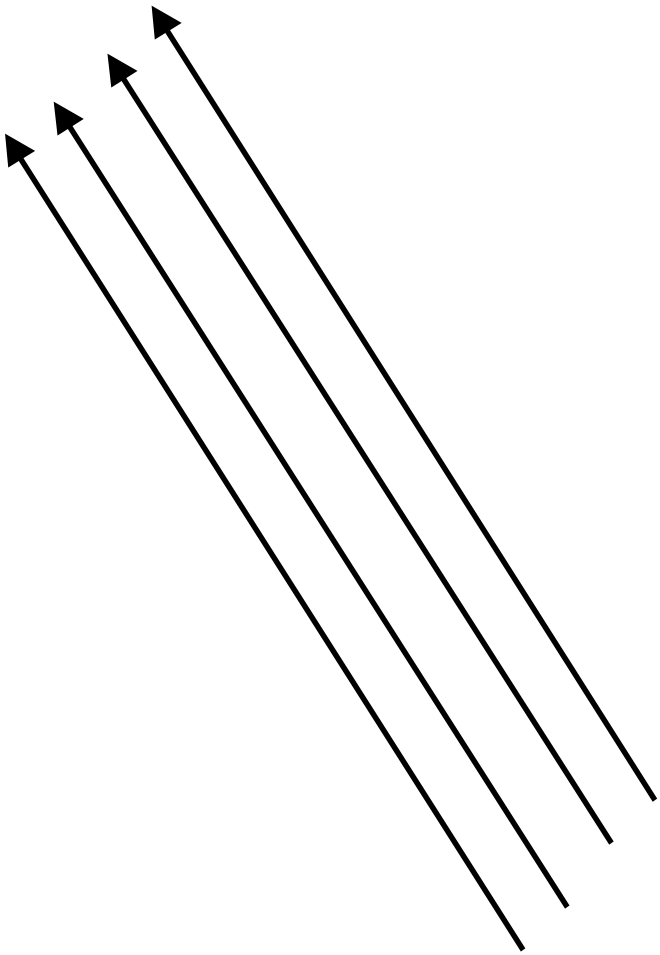
FlexGA™ Version 1.0

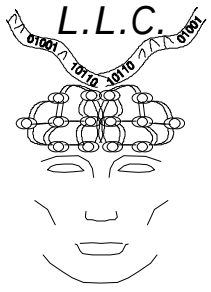
Evolutionary and Genetic Algorithms

User's Guide





Flexible Intelligence Group L.L.C.



Printing History
December 1998 First Printing

Flexible Intelligence Group, L.L.C.
P. O. Box 861477
Tuscaloosa, AL 35486-0013, USA.
e-mail: product@flextool.com
http://www.flextool.com/

Table of Contents

1. Introduction

Welcome to the **FlexGA™** User's manual. **FlexGA** implements several Evolutionary and Genetic algorithms combinations for use in the Matlab® environment. Matlab is produced and distributed by MathWorks Ltd.

Please refer to the software license agreement provide at the end of the document before using this product.

This manual consists of Five parts.

1. Introduction and up and running with FlexGA
2. Overview of Genetic Algorithms and Evolutionary Algorithms
3. Tutorial on the use of FlexGA
4. Bibliography
5. License Agreement

1.1. Up and running with FlexGA

We will discuss the following in this section.

- **FlexGA** Features
- Installation requirements
- Support Information
- Quick Start Instructions

1.1.1. FlexGA™ Features

Researchers, Industrialists, and Educators encounter innumerable problems that are difficult to solve using traditional optimization techniques. Genetic algorithms (GAs) have been shown to be robust probabilistic optimization tools for applications ranging from mathematics and engineering sciences to medicine and political sciences. GAs are parameter search procedures based upon the mechanics of natural genetics. The GA search combines a Darwinian survival-of-the-fittest strategy to eliminate unfit characteristics and uses random information exchange, with exploitation of knowledge contained in old solutions, to effect a search mechanism with surprising power and speed.

The following features makes FlexGA unique.

- **Modular**
- **User Friendly**
- Hardware and operating system **transparent**
- **GA options** include generational GA, steady state GA, micro GA
- **Coding Options** include integer, real discrete, and real continuos.
- **Selection** strategies include tournament, roulette wheel, and ranking
- **Crossover** techniques include one point, two point, and multiple point crossover
- **Niching** module to identify multiple solutions
- **Default parameter** settings for the novice
- **Statistics**, figures, and data collection

1.1.2. Installation requirements

FlexGA runs in the MATLAB environment. Thus any hardware platform that supports MATLAB is sufficient and necessary for FlexGA. Other requirements include:

1. A hard disk with at least 2.0MB free space
2. A 1.44MB 3½ inch floppy drive

1.1.3. Technical Support Information

Flexible Intelligence Group, L.L.C. (FIG) provides on-line technical support to their registered users. The following steps are advised to ensure the best utilization of FlexGA capabilities in case you encounter difficulties.

1. Ensure that you are complying with the instructions given in the tutorial section of the User's manual.
2. If the problem still remains, please note down the specific details and send **email** to **support@flextool.com** or **fax** your questions to **205-345-5095**. Please include your registration number in your email. FIG will answer your technical questions and handle any problems.

1.1.4. Quick Start Instructions

Follow these instructions step by step to install and run FlexGA™ .

INSTALLING FlexGA

1. Boot your system if so required. Ensure that you are at the command prompt.
2. Change to the appropriate directory where you want to install FlexGA. Use the **cd** command.
3. Create a directory called *flexga* on the hard disk using the command **mkdir flexga**. Change the current directory to *flexga* by typing **cd flexga**.
4. Ensure that *flexga* is the current directory on the hard disk.
   - If PC DOS based system, use the command: **copy A:\flexga*.***
   - If PC Window based system, use the same as above or use the Windows copy command.
   - If UNIX based system, use the appropriate dos copy command.
5. Remove the disk from the floppy disk drive and return it to the plastic sleeve at the back of the manual or store it in a safe place.
6. Read fmreadme.txt before starting FlexGA

RUNNING FlexGA

1. Please ensure that MATLAB is installed in your system.
2. Start MATLAB.
3. Ensure that the path to FlexGA directory is added to the MATLAB path list.
4. Type **gaex1** at the MATLAB prompt to run the first tutorial example (see Section 3).
5. Refer to the tutorial section for further help.

6. FlexGA can be executed with many combinations of choices. We have made every effort to make this software bug-free. Please let us know if you have noticed a flaw in the software execution for any one of the combinations that you might try.

1.2. FlexGA Software Modules

The following modules are included with FlexGA

Module name	Functionality
fmga_def.m	Default values for GA Parameters
flexga.m	Command line execution of GA.
fminit.m	Initialize variables for GA
fminipop.m	Creates initial random population.
fmgen.m	Runs each generation of GA.
fmcross.m	The Crossover operator for GA
fmdecode.m	Decodes the GA parameter strings (chromosomes).
fmmutate.m	Mutation function.
fmselect.m	Selects the individuals for the next generation.
fmshuffl.m	Shuffle function.
fmstats.m	Statistics.
fmnich.m	Niching function to evolve multiple solutions.
ex1.m -- ex5.m	Test PI for Examples 1 through 5
gaex1.m -- gaex5.m	Main programs for executing ex1.m -- ex5.m

2. Overview of Genetic Algorithms and Evolutionary Algorithms

Genetic algorithms (GA) are rooted in the mechanism of evolution and natural genetics. GA derive their strengths by simulating the natural search and selection process associated with natural genetics. GA accommodate all the facets of soft computing, namely uncertainty, imprecision, non-linearity, and robustness. Some of the attractive features include:

> Learning: GA are the best known and widely used global search techniques with an ability to explore and exploit a given operating space using available performance (or learning) measures. Moreover, genetic operators such as crossover, mutation, and reproduction allow express simulations of an extensive learning process of nature.
>
> Generic Code Structure: GA operate on an encoded parameter string and not directly on the parameters. This enables the user to treat any aspect of the problem as an optimizable variable.
>
> Optimality of the Solutions: In many problems, there is no guarantee of smoothness and unimodality. Traditional search techniques often fail miserably on such search spaces. GA are known to be capable of finding near optimal solutions in complex search spaces.
>
> Advanced Operators: This includes techniques such as niching (for discovering multiple solutions), combinations of Neural, Fuzzy, and chaos theory, and multiple-objective optimization.

Genetic algorithms are a numerical optimization technique. More specifically, they are parameter search procedures based upon the mechanics of natural genetics. They combine a Darwinian survival-of-the-fittest strategy with a random, yet structured information exchange among a population of artificial "chromosomes". This technique has gained popularity in recent years as a robust optimization tool for a variety of problems in engineering, science, economics, finance, etc.

Genetic algorithms in general require the parameter set of an optimization problem to be coded as a finite length string (called chromosomes, individuals, or population members). The traditional method is to use binary "bit strings", however, real numbered and integer strings have also been used successfully. For example, suppose an optimization problem has two parameters, the user might choose to represent each parameter with 10 binary bits. Thus, each chromosome would be a 20-bit binary string. A single 20 bit string represents one of the 2^{20} = 1,048,576 alternative solutions. It is also necessary to be able to assign a relative "fitness" (or "performance index") value to each chromosome. The performance index is usually calculated by decoding the chromosome, applying the parameters to the problem at hand, and evaluating the performance. Genetic algorithms then proceed by generating an (usually random) initial population of chromosomes, and applying GA "operators" to the population.

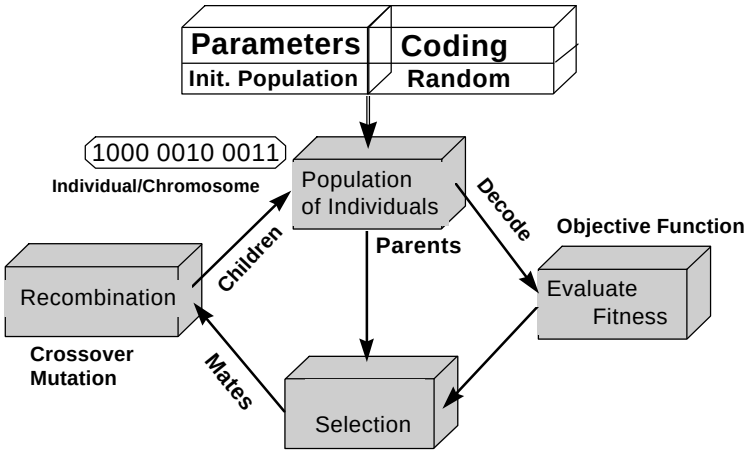


Figure 2.1. Computational Flow in Genetic Algorithms

An Genetic algorithm will typically employ three "operators": (1) selection, (2) recombination, and (3) mutation (see Figure 2.1). Typically, each of these operators is applied to the population once per "generation", and usually several generations are required to achieve satisfactory results. **Selection** is a process where an old string is carried through into a new population depending on the performance index (i.e. fitness) values. **Recombination** is usually applied after selection. The idea behind recombination is to combine pieces of two or more chromosomes, which results in new pairs of strings (chromosomes or solutions). **Mutation**, the third operator, is simply an occasional random alteration of a string position (based on a probability of mutation). In a binary code, this involves changing a 1 to a 0 and vice versa. The mutation operator helps in avoiding the possibility of mistaking a local minimum for a global minimum. Mutation is usually used sparingly. Typically, one mutation per one thousand bits is acceptable though this is a problem-dependent rule of thumb. When mutation is used in conjunction with selection and crossover, it improves the global nature of the GA search.

In general, several generations (successive applications of the operators) are applied. A new population is the result of each generation. If the GA functions the user should notice an improvement in the general fitness of the populations.

2.1. Background

Genetic Algorithms (GA) are rooted in the mechanism of evolution and natural genetics. Simulating evolution for useful purposes has been proposed and evaluated in different ways. Evolutionary Operation by Box, learning machines via evolution by Friedberg's, Evolutionary Programming by Fogel, Genetic Algorithms by Holland, and Evolutionary Strategies by Rechenberg and Schwefel were some of the activities that have made Evolutionary Algorithms a powerful approach towards finding solutions to complex problems. Genetic algorithms, as practiced today, come in different flavors: genetic algorithms; evolutionary strategies; and evolutionary programming. An offshoot of genetic algorithms is the concept of genetic programming. All of these algorithms derive their strengths by simulating the natural search and selection process associated with natural genetics. Genetic algorithms accommodate all the facets of soft computing, namely, uncertainty, imprecision, non-linearity, and robustness. Other attractive features include domain independent operation, adaptive capabilities, and inherent parallelism.

2.1.1. Genetic Evolution --- A way to solve complex optimization problems

The complexity of a problem lies in the complexity of the solution search space. This complexity arise due to: (a) size of the problem domain; (b) non-linear interactions between various elements; (c) domain constraints; (d) performance measure with dynamics and many independent and codependent elements; and (e) incomplete, uncertain, and imprecise information. Systems of nature routinely encounter and solve such problems. Good examples include genetic evolution of species and human immune system response to foreign bodies.

The genes in its chromosomes determine every organism's identity. Natural selection takes place in such a way that these characteristics are implicitly selected via the survival of the fittest criterion. Selection plays a big role in evolving fit individuals at all levels of living organisms. However, selection alone is not sufficient in an environment that is constantly changing. If selection was the only driving force, very soon the genetic makeup of all species will converge and there will no longer be any adaptation. Clearly, diversity in the gene pool is essential for successful adaptation in an evolving environment. Nature handles this problem through genetic recombination, mutation and niche formation. Genetic recombination plays the role of identifying fitter individuals by combining features from the available gene pool. This is the implicit process by which radically different individuals are brought in to compete in the environment. Mutation acts as the random search tool, jumping from one characteristic to another completely different one. Mutation provides a means for bringing diversity to a population that might need some diversity to avoid any type of convergence. Niching is nature's way of maintaining diversity by deriving maximum benefits from all aspects of the environment. Diploid and dominance are also features that are used for adapting to an evolving environment.

How does all of this help in solving complex optimization problems? The answer is simple: We could transform any optimization problem into a set of genetic characteristics (parameters to be optimized) that will survive in the best possible manner in the environment (fitness function). Examples include control (action and performance measures), game theory (strategies and reward), economic planning (mixes of goods and utility), and computational intelligence (algorithms and machine intelligent quotient or efficiency).

2.1.2. Optimization Techniques

We use different techniques today for optimizing the design space associated with various systems. Some of the popular techniques in use today includes the g*olden section* method, s*implex search* method, c*onjugate direction search* methods, *conjugate gradient search* methods. We can classify most of these techniques under calculus-based techniques, structured random techniques, and enumerative techniques. Calculus-based techniques, which rely on necessary and sufficient conditions, work well on problems where (1) analytical expressions for the necessary conditions exist and are solvable (indirect techniques); (2) numerical gradients (and in some cases the second derivatives) are easy to compute (direct techniques); and (3) the search space is either unimodal or order of the search space is small. Some of the popular direct techniques include Newton, Fibbanocci, Golden section, conjugate gradients, and simplex search technique. The major drawback of calculus-based methods is the lack of robustness over the broad spectrum of optimization functions that arise in real-world applications.

Enumeration is an approach that is guaranteed to find the global optimum. The simplest of enumerative technique is the exhaustive search. This approach is ideal for solving small problems and problems in which the time constraint is not severe. Another popular enumerative approach (the most efficient one) is the method of dynamic programming.

In the recent past, structured random searches that are problem-independent have been proposed to overcome the shortcomings of calculus-based and enumerative methods. These types, which in general require only function information, can handle non-convex and discontinuous functions. Main drawback of these schemes is that they require a trade-off between available computer time and accuracy of the optimal solution. Genetic algorithms and simulated annealing are good examples of such techniques.

Genetic algorithms (GA), have become popular among practitioners from varying fields. This popularity is due to the following:

1) Genetic algorithms are robust in the sense that they are applicable to a variety of problems with little or no modifications to the technique.

2) Genetic algorithms can handle all search spaces, including non-smooth, multimodal, and discontinuous spaces.

3) Genetic algorithms can handle multiple objectives with no explicit mixing needed to define a composite objective function.

4) Genetic algorithms can identify multiple optimal solutions.

5) Genetic algorithms can be used in dynamic optimization situations.

2.2. Genetic Algorithms Terminology

Before we present an overview of Genetic Algorithms, the terminology associated with these algorithms is addressed. In a typical application of Genetic algorithms, we transform the given problem into a set of genetic characteristics (parameters to be optimized) that will survive in the best possible manner in the environment (fitness function). To expand on the terminology further, we will examine the problem of optimizing a function of two variables. We present a step-by-step approach in setting up an optimization problem.

Step 1: Function to be Optimized

$$\min_{x_1, x_2} f(x_1, x_2) = (x_1 - 2)^2 + (x_2 - 3)^2$$

Step 2: Parameters and their limits

The parameters of the search are identified as x_1, x_2. These are called the *phenotypes* in Genetic Terminology. The limits on these parameters are:

$$-5 \le x_1 \le 5; \quad -10 \le x_2 \le 10$$

Step 3: *Phenotype* to *Genotype* conversions.

In Genetic algorithms, the *phenotypes* (parameters) are usually converted to *genotypes* by using a coding procedure. For simplicity, we will assume that the parameters can occupy one of eight values as shown in Figure 2.2 for x_1, x_2 giving rise a total of 64 possible solutions. These eight possibilities for each of the variables are represented by a 3 bit binary string (6 bits total for the two variables combined). This representation using binary coding makes the parametric space independent of the type of variables used (variables could be integer, real, or other types).

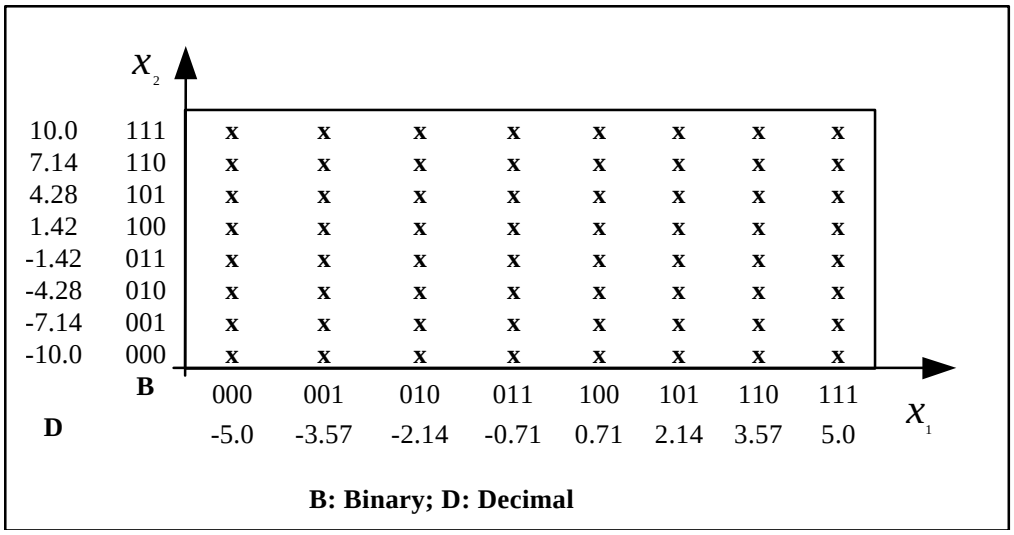


Figure 2.2.. Discretization of the search space using binary representation.

Step 4: *Chromosome* Formation

Once the genotypes are defined, the strings are concatenated to form the *chromosomes* of the function.

Step 5: Population Formation

A set of these chromosomes forms the population.

Step 6: Generations.

Next the optimization operators are applied to the population and based on certain criteria the population is altered. This iteration takes the search to the next *generation*.

The table below presents some of the terminology used through out this section. Examples are provided where appropriate.

Table 2.1. GA Terminology

Genetics Terminology	GA Equivalent	Example
String or Chromosome	A coded parametric set	010100
Population	A Collection of search points	010011 100111 ….. 100111
Generation	Next iteration in the GA sense	
Gene	Feature or character	x_1
Allele	Feature value	$101 \rightarrow 5$
Fitness function or objective function or Performance index	Function to be optimized	$f(x_1,x_2)=(x_1-2)^2+(x_2-3)^2$
Locus	String position	String: 1 0 1 0 1 1 Locus: 1 2 3 4 5 6
Genotype	String structure	101101
Phenotype	Parametric set	[5,10]

2.3. Evolutionary Algorithms

As stated earlier, evolutionary algorithms come in different flavors. The main variants include evolutionary strategies, evolutionary programming, and genetic algorithms. Genetic algorithms use a combination of Darwinistic idea of evolution (survival of the fittest) and genetic operators such as mutation and recombination. In the next sections, we present brief overviews of these techniques. We will begin by presenting a simple evolutionary algorithm with two members and proceed to examine multi-membered evolutionary strategies. Finally, we discuss genetic algorithms in which both evolutionary and genetic characteristics are utilized in finding optimal solutions.

2.3.1. Two Membered Evolutionary Algorithms

In the two membered scheme the two principles of mutation and selection, which Darwin recognized to be most important, are taken as rules for variation of the parameters and for selecting the member that survives for the next iteration. The following assumptions are made:

1) The population size is two and remains a constant

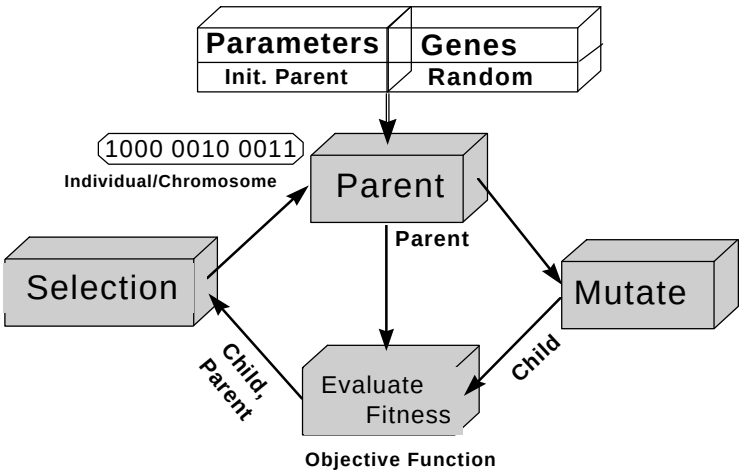
2) An individual has an infinitely long life span and capacity for producing descendants asexually.

3) No difference exists between genotype (encoding) and phenotype (appearance).

4) Only point mutations occur, independently of each other at all single parameter locations

5) The environment and thus the criterion of survival is constant over time

The iteration steps are outlined below and the Figure below provides the computational flow required toimplement this algorithm [Schw95]:



Step 0: (Initialization) A given population consists of two individuals, one parent and one descendant. They are each identified by their genotype according to a set of ***n*** genes. Only the parental genotype has to be specified as starting point.

Step 1: (Mutation) The parent ***E(g)*** of the generation **g** produces a descendant ***N(g)***, whose genotype is slightly different from that of the parent. The deviations refer to the individual genes and are random and independent of each other.

Step 2: (Selection) Because of their different genotypes, the two individuals have a different capacity for survival (in the same environment). Only one of them produce further descendants in the next generation, namely the one, which represents the higher survival value. It becomes the parent ***E(g+1)*** of the generation ***g+1***.

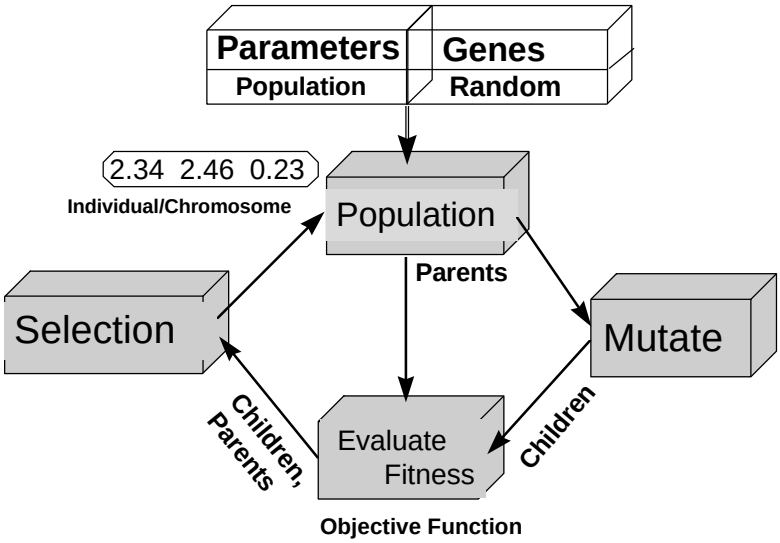
2.3.2. Multi-membered Evolutionary Strategies

In the multi membered scheme [Schw95], once again the two principles of mutation and selection are utilized. In addition to this, the population size (μ) is assumed to be more than two. The following assumptions are made:

1) The population size remains constant
2) No difference exists between genotype (encoding) and phenotype (appearance).
3) Only point mutations occur, independently of each other at all single parameter locations.
4) The environment and thus the criterion of survival is constant over time

The iteration steps outlined in the Figure below provides the computational flow required to implement this algorithm [Schw95]:

STEP 0 : (Initialization) A given population consists of μ individuals. Each is characterized by its genotype consisting of ***p*** genes, which determine the vitality, or fitness for survival.



STEP 1: (Variation) Each individual parent produces λ/μ offspring on average, so that a total of λ new individuals are available. The genotype of a descendant differs only slightly from that of its parents.
STEP 2: (Filtering) Only μ best of the λ offspring become parents of the following generation.

2.4. Simple Genetic Algorithms

Genetic algorithms differs from the multi-membered Evolutionary Algorithms in the following ways:

1) GA work with a coding of the parameter set, not the parameters themselves.

2) GA search from a large population of points, not a single point.

3) GA uses recombination operators such as crossover and reordering not just mutation.

Genetic algorithms require the natural parameter set of the optimization problem to be coded as a finite-length string. As an example, for an optimization problem with two parameters, the parameters are discretized by mapping from a smallest possible parametric set K_{min} to a largest possible parametric set K_{max}. This mapping uses a 10-bit binary unsigned integer for both K_1 and K_2. In this coding a string code 0000000000 maps to K_{min} and a 1111111111 maps to K_{max} with a linear mapping in between. Next, the two 10-bit sets are chained together to form a 20-bit string representing a particular controller design. A single 20-bit string represents one of the 2^{20} = 1,048,576 alternative solutions. Table 2.2 presents a coding example and a sample random initial population. Genetic algorithms work iteration by iteration, generating and testing a population of strings. This population-by-population approach is similar to a natural population of biological organisms where each generation successively evolves into the next generation by being born and raised until it is ready to reproduce. This approach is very different from classical search methods, where movement is from one point in the search space to another point based on some transition rule. Another important difference between GA and the classical approaches is in the selection of the transition rules. In classical methods of optimization the transition rule is deterministic. In contrast, GA uses probabilistic operators to guide their search. The algorithm for a GA follows:

Table 2.2. Coding Example

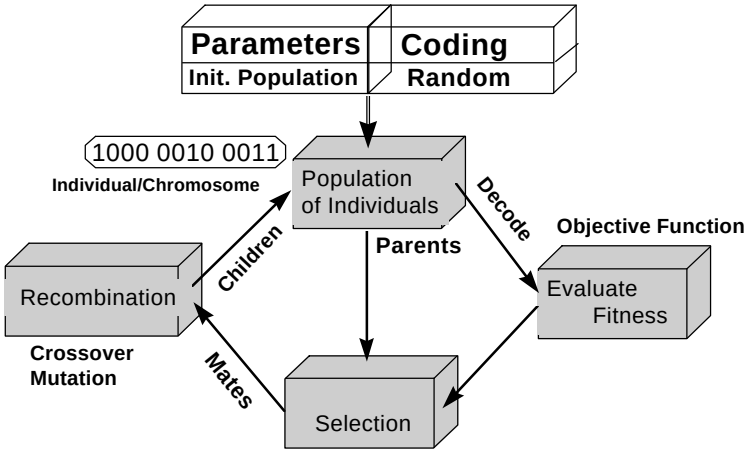
	Kmin	Kmax	Coding: Population Size=5
			1100110011 0011001100
K1:	0	25	1010101010 1010111101
K2:	-25	25	1110001110 0001101101
	0000000000	1111111111	1100111111 0000110111
			1011100011 0001101010

STEP 0: (Initialization) A given population consists of ***n*** individuals. Each is characterized by its genotype consisting of ***p*** genes, which determine the vitality, or fitness for survival. Each individual's genotype is represented by a binary bit string representing the object parameter values either directly or by means of an encoding scheme.

STEP 1: (Selection) Two parents are chosen with probabilities proportional to their relative position in the current population, either measured by their contribution to the mean objective function value of the generation (proportional selection) or by their rank (e.g., linear ranking selection).

STEP 2: (Recombination) Two different offspring are produced by recombination of two parental genotypes by means of crossover at a given recombination probability P_C. Both of these offspring are taken into further consideration. Steps 1 and 2 are repeated until ***n*** individuals represent the next generation.

STEP 3: (Mutation) The offspring eventually (with a given fixed and small probability) undergo further modification by means of point mutations working on individual bits, either by reversing a one to a zero, or vice versa; or by throwing a dice for choosing a zero or a one, independent of the original value.



Convergence of the GA search can be ascertained by either examining the variance of the population fitness (known as phenotype convergence) or by examining bit-wise convergence (known as genotype convergence). Zero population fitness variance implies absolute convergence. Since this is not always possible to achieve, the search can be terminated when the population variance is below a threshold value. In problems where computational time is a factor, the best string can be retained without any modifications (elite selection). This way, if the user decides to terminate the search, a best-so-far solution to the optimization problem is always available.

2.5. Micro-Genetic Algorithms (μGA)

Simple Genetic Algorithms (SGA) have been shown to be useful tools for many function optimization problems. One drawback of SGA is the time penalty involved in evaluating the fitness functions (performance indices) for large populations, generation after generation. In [Kris89], a small population approach (coined as Micro Genetic Algorithms--(μGA) with some very simple genetic parameters was first proposed, and it was shown that μGA implementation reaches the near-optimal region much earlier than the SGA implementation. The superior performance of the μGA in the presence of multimodality and their merits in solving non-stationary function optimization problems were demonstrated.

Just as in the simple GA, the μGA works with binary coded population. The major difference between SGA and the μGA comes in the population choice and the way new information is brought in to the evolution process. In the μGA proposed in [Kris89], the population size is fixed at five. It is a known fact that GA generally do poorly with very small populations due to insufficient information processing and early convergence to non-optimal results. The key to success with small population is in bringing in new strings at regular intervals into the population. Based on this, a step by step procedure for the μGA implementation is presented below:

1. Select a population of size 5 either randomly or 4 randomly and 1 good string from any previous search.

2. Evaluate fitness and determine the best string. Label it as string 5 and carry it to the next generation (elitist strategy). In this way there is a guarantee that the information about good schema is not lost.

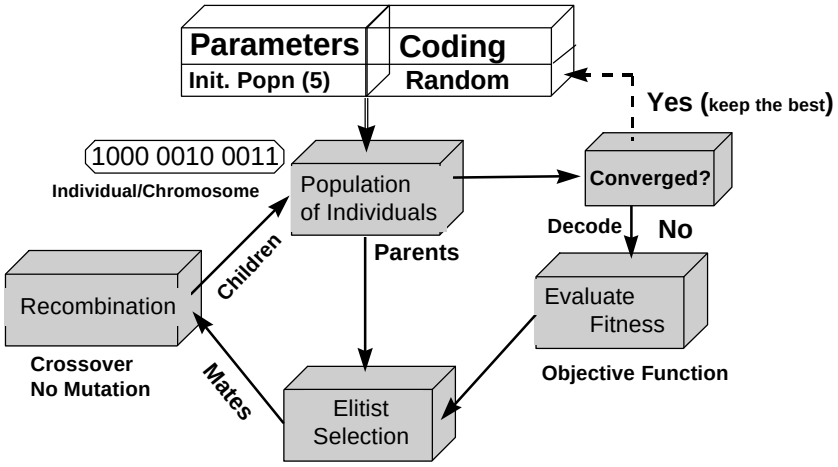
3. Choose the remaining four strings for reproduction (the best string also competes for a copy in the reproduction) based on a deterministic tournament selection strategy. Care should be taken to avoid two copies of the same string mating for the next generation.

4. Apply crossover with a probability of one. This is done to facilitate high order of schema processing. The mutation rate is kept to zero as it is clear that enough diversity is introduced after every convergence through new populations of strings.

5. Check for nominal convergence (reasonable measure based on either genotype convergence or phenotype convergence). If converged go to step 1.

6. Go to step 2.

The assumption that maximum real-time schema (information) processing yields maximum performance is supported by empirical results and this maximum schema processing is achieved in the μGA implementation by constant infusion of new schema at regular intervals. SGA is known to reach premature convergence forcing the search process to rely



entirely on the mutation operator for new information. In the case of μGA the "start and restart" procedure helps in avoiding the premature convergence and the μGA is always looking for better strings.

In implementing μGA, our interest is purely to find the optimum as quickly as possible and not in the average behavior of the population. In other words, our performance measure for μGA should be based on the best-so-far string, rather than on any average performance.

2.6. Steady State Genetic Algorithms

In steady state genetic algorithm, a percentage (user defined) of the population selected based on their fitness, is retained into the next generation. This subset of the population goes through regular selection for mating purposes but is not altered going into the next

generation. This GA variant saves time while using objective functions that require large amounts of computation time, and string lengths that require large numbers of members in the population.

2.7. Genetic Algorithms with Stochastic Coding

Studies conducted using dynamic coding, in which a search region is narrowed down as the search evolves, have usually failed. The attraction in dynamic coding is the shorter string and hence fewer function evaluations. Discarding the regions that are not promising as evaluated based on GA performance is the main reason for the failure of these techniques. Discarding regions leads to a possibility of losing a region that might actually have the global optimum. It will be desirable to have a dynamic coding scheme that does not discard any portion of the region but at the same time shifts emphasis to different regions of the search space. In this section, we detail a GA with stochastic encoding of the parameters (referred to as a Stochastic GA) that overcomes the problem of having a large search spaces that require continuous sampling. Stochastic Genetic Algorithm was first presented in [Kris94] as an approach to effectively solve problems with large number of parameters. Some of the features of the Stochastic GA as given in [Kris94b] are:

1) Each discrete possibility, as decoded from the binary string, represents a region and not a single value.

2) The definitions of the regions are dynamic.

3) Region definitions are altered based on the GA evolution.

4) No region is absolutely discarded.

5) Search Region is not explicitly constrained.

6) GA never absolutely converges.

These features can be implemented in the GA by encoding the search region as a binary string. The search region is represented by a multivariate Gaussian distribution with a mean vector (μ) and the variance matrix (Σ). The mean vector gives the expected values of the parameters in the search region, and the variance matrix gives the probability of finding an optimal parameter set in a particular area of the search region. The stochastic children are obtained by sampling this multivariate Gaussian distribution.

Before we present the algorithm for Stochastic GA, we present a step-by-step approach in setting up the optimization problem.

Step 1: Function to be Optimized

$$\min_{x_1, x_2} f(x_1, x_2) = (x_1 - 2)^2 + (x_2 - 3)^2$$

Step 2: Parameters and their limits

The parameters of the search are identified as x_1, x_2. These are called the *phenotypes* in Genetic Terminology. The starting limits on these parameters are (true limits are unknown):

$$-5 \le x_1 \le 5; \qquad -10 \le x_2 \le 10$$

Step 3: *Phenotype* to *Genotype* conversions.

In stochastic GA, the *phenotypes* (parameters) are converted to *genotypes* by using a coding procedure that is stochastic. For simplicity, we will assume that the parameters can occupy one of two stochastic regions (not values as in simple GA) as shown in Figure 2.3 for x_1, x_2 giving rise a total of 4 possible regions (not solutions as in simple GA). These two possible regions for each of the variables are represented by 1-bit binary strings (2 bits total for the two variables combined). A mean and a standard deviation for the normal distribution define the regions.

Step 4: Selection Procedure:

The selection procedure consists of a local selection and a global selection. In the local selection, a representative of a region represented by a binary string is selected by randomly drawing ***m*** children asexually from that region using the predefined normal probability distribution. The best child in the region then represents the region for the global GA based selection. The best child also becomes the mean of the region and thus provides a dynamically varying search region.

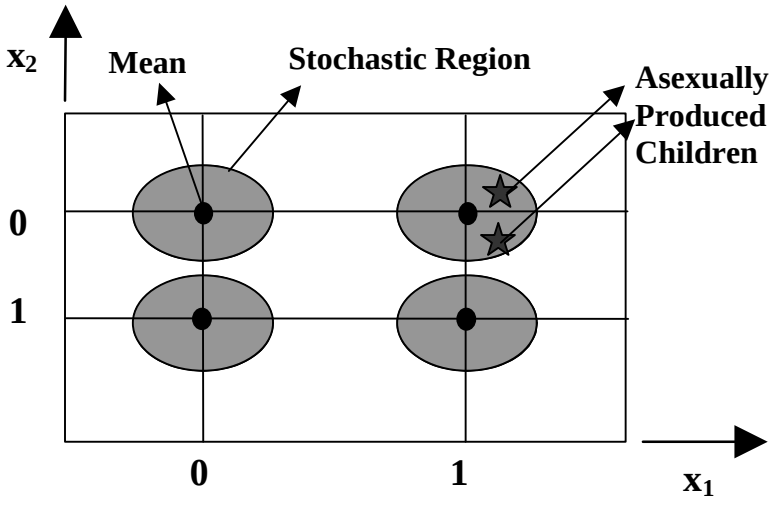


Figure 2.3. Region definition for stochastic GA.

The stochastic children are obtained by varying all the parameters in the parent's phenotype using the multivariate Gaussian distribution [Kris94b]. Here the variance matrix used in the multivariate Gaussian distribution is adapted continuously as the GA population evolves. This adaptation of the variance matrix helps to exploit the most promising regions as the GA explores the search region. The algorithm that implements the above details is presented below.

Basic Algorithm

Step 1: (Initialization) An initial population of *n* individuals, characterized by its genotype is randomly generated. Each individual's genotype is a binary string representing a search region of the parameters. The search region is encoded in the binary string by means of a multivariate Gaussian distribution with mean vector μ and a variance matrix Σ.

Step 2: (Stochastic Phenotype Variation)

Each of the *n* individuals produces *m* offspring, so that a total of *mn* new individuals are available. The search regions represented by these offspring are obtained by displacing the

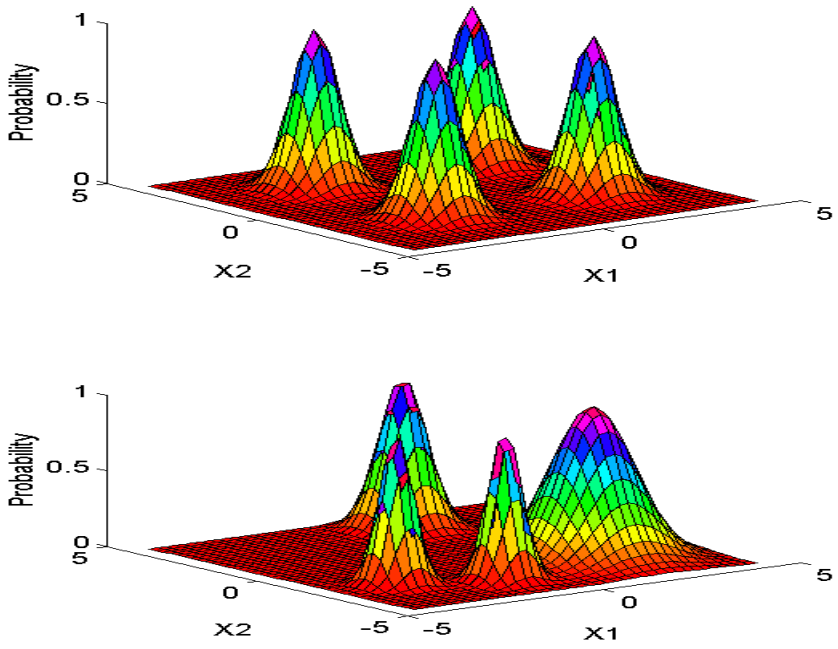


Figure 2.4. Gaussian Distribution for a 2-D problem. Top: Initial Distribution; Bottom: Evolved Distribution.

parent's mean vector ($\mathbf{m}$), with the variance matrix($\mathbf{S}$), resulting in a new mean vector for the offspring.

Step 3: (Filtering) Out of *mn* individuals in step 2, only *n* individuals become parents. The phenotypes of the chosen individuals are used to redo the coding, resulting in a modified mean vector. The variance matrix of the offspring is altered based on the $(1/5)^{th}$ success rule. According to this rule, the variance of the Gaussian distribution is decreased if at least one out of five phenotype variations (of the same genotype) in step 2 results in an improvement of the performance index. Otherwise, the variance is increased.

Step 4: (GA Selection) Two parents are chosen with probabilities proportional to their relative position in the current population, measured either by their contribution to the mean performance of the current generation (proportional selection) or by their rank (linear ranking selection). In a tournament selection procedure, the two best individuals from a random number of individuals are chosen for the next generation.

Step 5: (GA Recombination) The two parents selected in step 4 are recombined with a probability of crossover p_c, giving rise to two new parental genotypes. Steps 4 and 5 are repeated until we have *n* new individuals representing the next generation.

Step 6: (GA Mutation) The new generation obtained above undergoes a mutation operation, where the individual bits of each offspring are mutated (reversed from a one to a zero, or vice versa) with a small probability(p_m).

Initially when the search is started, a symmetric multivariate Gaussian distribution is assumed over the search region. The initial Gaussian distribution for a two-parameter optimization problem is illustrated in Figure 2.4. A two dimensional planform view of the

Gaussian distribution and the parameter choices are shown in Figure 2.5. In approach A, as the GA population evolves, the distribution shifts towards more promising regions and the variance matrix is altered to exploit the promising regions. This is shown in Figure 2.4(Bottom) for the two-parameter case, where the Gaussian distribution is shown after *N* generations. Figure 2.5 (below) shows the planform view of the Gaussian distribution after *N* generations.

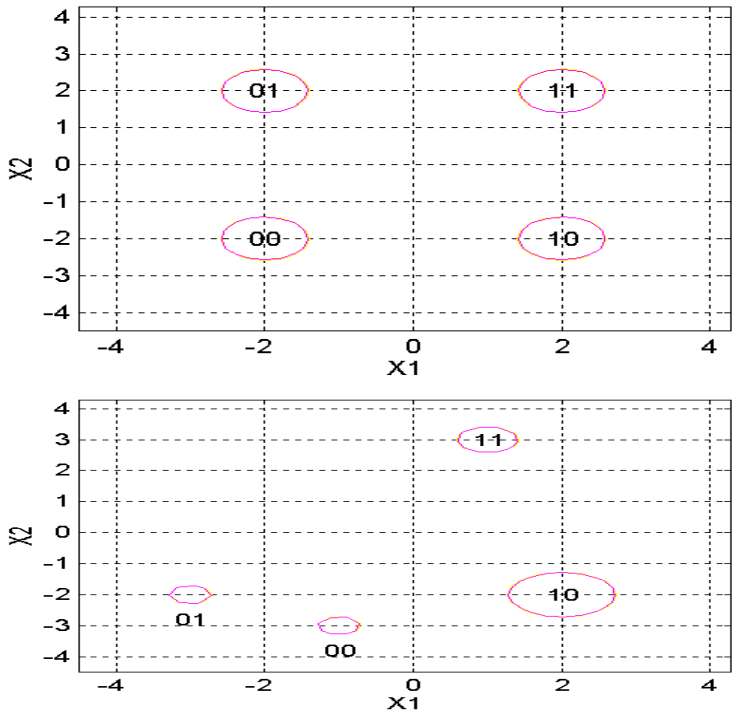


Figure 2.5. Gaussian Distribution plan form. Top: Initial Distribution; Bottom: Evolved Distribution.

2.8. Niching in Genetic Algorithms

For many optimization problems there may be multiple, equal, or unequal optimal solutions. A simple GA cannot maintain stable populations at different optima of such functions. In case of optimal solutions with equal fitness, sampling errors in GA operators cause the population to converge to a single solution. However, in the case of unequal optimal solutions, the population invariably converges to the global optimum. Figure 2.6 presents a two-parameter search space example with many optimal points. A simple GA with no niching will converge to a single optimum. Whereas a modification of the GA process with niching helps in maintaining subpopulations near global and local optima.

The availability of alternate solutions is of great practical utility. To achieve this objective, it is essential to introduce a controlled competition among different solutions near every locally optimal region. This would maintain stable subpopulations at such optimal regions. This could be accomplished by incorporating the concepts of niche and species into the GA search process.

A niche is viewed as an organism's (individual member of the population) environment (fitness function) and a species is a collection of organisms with similar features. A simple

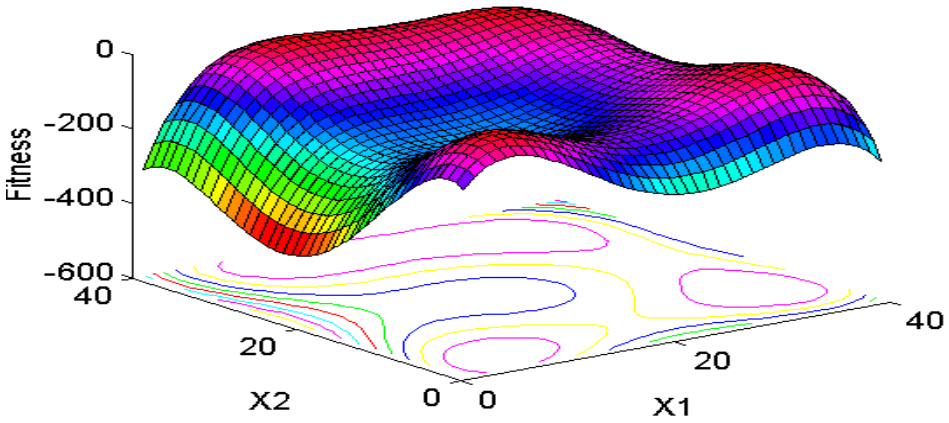


Figure 2.6. Sample Fitness Surface with Multiple Peaks.

GA with no niching converges to a single optimum even though multiple peaks of equal quality may exist. Nature address such a predicament through the formation of stable subpopulations of organisms surrounding separate niches by forcing similar individuals to share their resources. Niching helps to maintain subpopulations near global and local optima by introducing a controlled competition among different solutions near every local optimal region.

Niching in general is achieved using a sharing function. The sharing function creates subdivisions of the environment by degrading an organism's fitness proportional to the number of other members in its neighborhood. The amount of sharing contributed by each organism x_i into its neighbor x_j is determined by their proximity in the decoded parameter space (phenotypic sharing) based on a distance measure d_{ij}. Given **p** parameters of unequal boundaries over a parameter range [x_{min} - x_{max}],

Distance Measure: $$d_{ij} = \sqrt{\sum_{k=1}^{p} \frac{x_{k,i} - x_{k,j}}{(x_{k,\max} - x_{k,\min})^2}}$$

where

$x_{k,i}$ = k^{th} parameter of individual 'i'

$x_{k,j}$ = k^{th} parameter of individual 'j'

$x_{k,max}$ = Max. allowable value for k^{th} parameter

$x_{k,min}$ = Min. allowable value for k^{th} parameter

For each d_{ij}, we can apply the sharing function $s(d_{ij})$ given by the equation

Sharing Function: $$S(d_{ij}) = \begin{bmatrix} 1-(d_{ij}/\sigma_{share})^{\alpha} & if \quad d_{ij} < \sigma_{share} \\ 0 & otherwise \end{bmatrix}$$

The limiting distance between the individuals to be shared, σ_{share}, is calculated as the average distance required to identify each niche distinctly in the solution space. This value has to be set carefully based on and **q** uniformly spaced assumed peaks.

$$\sigma = 0.5q^{(-1/p)}$$

The following properties hold good for the sharing function.

1) $S(d_{ij}) \le 1.0$ for all d_{ij}. This condition imposes a fractional contribution to the effect.

2) $S(d_{ij}) = 1.0$. When both individuals are identical, they will share a full portion with each other.

3) $\lim_{d_{ij} \to \infty} S(d_{ij}) \to 0.0$ When the individuals are far apart, they produce no effect on each other).

The shared fitness of the i^{th} individual (organism) is given as

$$SharedFitness = \frac{TrueFitness}{\sum_j S(d_{ij})}$$

2.9. Handling Constraints in GA

In this section, we outline two methods for treating optimization problems with constraints. These techniques were outlined first by Michalewicz [Mich95].

Penalty Function Methods

Given:

Unconstrained objective: $\min_x J(x)$

Constraints: $$\begin{cases} g_j(x) \le 0 & for \quad 1 \le j \le q \\ h_j(x) = 0 & for \quad q+1 \le j \le m \end{cases}$$

When constraints are involved, assigned penalties usually incorporate degrees of constraint violations. Most of these methods use constraint violation measures f_j (for the j-th constraint) for the construction of the new fitness.

New Fitness Function: $\min_x J(x) + \sum_{j=1}^{m} q_j f_j^{\,2}(x)$

where

q_j = Relative Penalty weighting factor

$$f_j(x) = \begin{cases} \max\{0, g_j(x)\}, & for \quad 1 \le j \le q \\ |h_j(x)|, & for \quad q+1 \le j \le m \end{cases}$$

Some thoughts on using a Penalty Function [Mich95]

1) Penalties that are functions of the distance from feasibility are better performers than those which are merely functions of the number of violated constraints

2) For a problem with few constraints, and few full solutions, penalties that are solely functions of the number of violated constraints are not likely to find solutions

3) Penalties should be close to the expected completion cost, but should not frequently fall below it. The more accurate the penalty, the better will be the solutions found. When the penalty often underestimates the completion cost, then the search may not fit the solution.

4) The genetic algorithm with a variable penalty coefficient outperforms the fixed penalty factor algorithm, where a variability of penalty coefficient is determined by a heuristic rule.

2.10. Broad Guidelines for GA Implementation

In the next several paragraphs we give broad guidelines for a novice Genetic Algorithm user. The choices for some of the GA operators are listed in boldface. These choices have worked well consistently on various types of problems.

Encoding Structure

One advantage of GA is that the encoding structure permits diverse solution optimization. However there are some initial decisions that has to be made which mostly depends upon the number of parameters we want GA to optimize and whether we want continuous or binary representation. **The goal should be to have a minimum bit length and a population size consistent with the bit length**.

Population Size

The population size controls the possible number of solutions GA has to explore the solution space. Research indicates that the length of the string and the complexity of the problem play a critical role in determining the population size. The length of the string depends upon the encoding scheme and the number of parameters and types of parameters we want GA to optimize.

Since GA depend on information contained in the population, two important factors need to be paid close attention to. These are: (1) population size; and (2) randomness of the initial population. Diversity increases with population size, while reducing local optimum convergence. However too large a population size may result in excessive computational

time. In our experience, **a population size of 1/4th of the string length or larger** gives more reliable and consistent results. Also, the initial population needs to be randomly distributed across the search space.

Selection

Selection models nature's *survival-of-the-fittest* principle. Selection strategies include **Roulette wheel** selection (proportionate), **tournament** selection, **Ranking** selection, etc. The proportionate selection scheme allocates offspring based on the ratio of the fitness value of a string to the average fitness value of the population. Typical problems include premature convergence and weak promotion of better strings. Roulette wheel selection is intuitive and easy to implement. However, there are scaling problems, can handle only maximization problems, cause large sampling errors, and requires a large population size. In ranking selection, we rank members based on their fitness (performance index) and assign offspring as a function of rank. This avoids scaling problems and can handle both maximization and minimization problems, but may cause ad hoc allocation of offspring. Tournament selection is deterministic, works well, can handle both maximization and minimization problems, can be easily extended to multiple criteria problems, and has been shown to be superior and mathematically tractable. There are two options here. In binary, pairs of individuals are picked at random from the population. The one with higher fitness is copied into the mating pool. Larger tournaments are employed by picking n individuals instead of two (a pair). This increases the selection pressure. In probabilistic binary, the better individual wins the tournament with probability $0.5 < p < 1$. Tournament size and win probability control selection pressure. In general, **tournament (binary) or ranking selection with elitism** is a good choice. A **tournament size of two** is recommended.

Crossover

Crossover is a reproduction operator that provides random information exchange. It is aimed towards evolving better building blocks (schemata with short defining lengths and high average fitness values). Crossover points are randomly chosen. Crossover could be at a **single-point**, **two-points**, or at ***n*-points**. The frequency of crossover is governed by a user selected **crossover rate** or probability of crossover (typically around **0.75**). Increasing crossover rate increases recombination of building blocks, however, with an increasing probability of loosing good strings. The optimum value is problem specific and will have to be empirically determined. Other options include uniform crossovers and knowledge-directed crossovers. It has been observed that a crossover performs a faster search than mutation for objective functions involving high epistaticity (nonlinear interactions among the bits of the string). If the crossover rate of a bit is dependent upon its position in the string, the crossover operator is said to have positional bias. If the distribution of the number of bits exchanged by the crossover operator is non-uniform, the crossover operator has distributional bias. A single-point crossover exhibits maximum positional bias and minimum distributional bias whereas a uniform crossover exhibits the opposite. Single and two-point crossovers preserve schemata because of their low disruption rates but becomes less exploratory with homogeneous population. A uniform crossover swaps bits irrespective of their position but have high disruption rates. **A uniform crossover is recommended for small populations where the disruptiveness helps to sustain a highly explorative search. However, in large populations, the inherent diversity reduces the need for exploration. A two-point crossover is more suitable in such cases. Based on empirical research results, a small population size should have a relatively large crossover rate and vice versa**.

Mutation

Mutation is traditionally seen as a background or secondary reproduction operator which restore inadvertently lost gene values (alleles), prevent genetic drift, and provide a small element of random search in the vicinity of the population when it has largely converged. A user specified **mutation rate** or probability of mutation (typically around **0.001**) determines whether to mutate or not. Usually as the population converges, mutation becomes more productive than crossover. It may be noted that too high a mutation rate would make the search too random. Research has shown that a small population size should have a relatively large mutation rate and vice versa. Also, towards the end of the search, mutation rates can be increased to help in avoiding premature convergence.

Replacement Strategy over Generations

Options include generational GA (the entire population is replaced every generation) and steady-state GA (only a small fraction of the strings are replaced every generation). Steady-state GA use populational elitism, large population sizes, and high mutation and crossover rates. For problems with heavy computational burden, steady-state GA with elitism is a good choice.

Objective Function

The objective function should reflect the desired characteristics of the system being optimized. It is very important that the designer use an appropriate objective function since this drives the evolution process.

2.11. Genetic Algorithms Building Blocks

Synthesis and analysis of complex systems can be viewed as a puzzle that needs to be solved using several individual pieces. These individual pieces can be studied, tested, understood, and built better than the complete system. These pieces, labeled in this book as *Building Blocks*, when put together in the required forms result in many different complex systems. In this section, we will identify these building blocks pertinent to Genetic Algorithms.

In the next several pages we present several building blocks along with their software file names under the broad categories of:

1) Coding
2) Population
3) Recombination
4) Mutation
5) Selection

2.11.1. Coding

One advantage of GA is that the encoding structure permits diverse solution optimization. However there are some initial decisions that has to be made which mostly depends upon the number of parameters we want GA to optimize and whether we want continuous or binary representation. Encoding schemes include Binary (example: 1001010), K-ary (example:

1ff0000c0), real (example: 43.76), stochastic, permutation, lisp, etc. . The most common method is the binary integer representation. Here, each variable (parameter) is first linearly mapped to an integer defined in a specific range and then encoded using a fixed number of binary bits. The population members are formed by concatenation of these binary bits.

Binary to Decimal (see *fmdecode.m*)

Given a binary representation, we convert to the required decimal equivalent in two steps. In step 1, we convert the binary to an integer representation and in step 2, we convert the integer representation to decimal equivalent or any other preferred mapping chosen by the user. It is emphasized here that step 2 is problem dependent and it is up to the user to arrive at the necessary definition.

Step 1 Algorithm:

$$I_i = \sum_{j=1}^{n\_bits(i)} a_{i,j} 2^{j-1}$$

where $a_{i,j}$ for $j = 1,\ldots,n\_bits(i)$ represents the bit string segment of length *n_bits(i)* for encoding the i[th] parameter integer value.

Example:

$$a_i = \begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} a_{i,1} & a_{i,2} & a_{i,3} & a_{i,4} \end{bmatrix}$$

$$I_i = a_{i,i} 2^0 + a_{i,2} 2^1 + a_{i,3} 2^2 + a_{i,4} 2^3 = 1*2^0 + 0*2^1 + 1*2^2 + 0*2^3 = 5$$

Step 2 Algorithm:

Explicit upper and lower bounds are given as $L_i \le x_i \le U_i$ and from L_i computed in step 1, the parameter value x_i can be computed as

$$x_i = L_i + \frac{U_i - L_i}{2^{n\_bits(i)} - 1} I_i$$

Examples:

$I_1 = 5; \quad U_1 = 10; \quad L_1 = -5; \qquad n\_bits(1) = 4; \qquad \Rightarrow \quad x_1 =$

$I_2 = 2; \quad U_2 = 15; \quad L_2 = -5; \qquad n\_bits(2) = 5; \qquad \Rightarrow \quad x_2 =$

2.11.2. Population

In GA, each possible solution is an encoded string of 1s and 0s of a specific length (a member of the population). A set of such strings (members) forms the population and the number of strings is the size of the population. The population size controls the possible number of solutions GA has to explore in the solution space. Research indicates that the length of the string and the complexity of the problem play a critical role in determining the population size. The length of the string depends upon the encoding scheme and the number

of parameters and types of parameters we want GA to optimize. Since GA depend on information contained in the population, two important factors need to be paid close attention to. These are: (1) population size; and (2) randomness of the initial population. Too large a population size may result in excessive computational time.

Uniform random initialization (see *fminipop.m*)

To ensure that the initial population is an unbiased random distribution, a uniform probability density function can be used to distribute the population. This function could be used either for Global binary population or for Local binary/non-binary population.

Example:

Binary Coding; Population Size = 5; Chromosome Length =10;

0	0	1	0	1	1	0	0	0	1
0	1	1	1	0	1	1	1	1	0
1	1	0	1	1	0	1	1	1	0
1	0	0	1	0	0	1	1	0	0
1	0	0	1	1	1	0	1	0	0

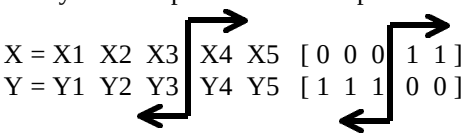
2.11.3. Recombination

Recombination is a reproduction operator which provides random information exchange. It is aimed towards evolving better building blocks (schemata with short defining lengths and high average fitness values). Most popular recombination operator is the Crossover operator. Crossover could be at a **single-point**, **two-points**, or ***n*-points.** Another type of crossover is **uniform crossover.**

N-point crossover (see *fmcross.m*)

Crossover is applied to a new generation of population strings. Pairs of strings are selected at random from the population. Crossover points are randomly chosen. The information present in the two strings beyond these crossover points are exchanged to form new strings. In two-point crossover, the information between just the two crossover points will be exchanged. Two-point crossover eliminates the single-point crossover bias towards end-of-string bits. The frequency of crossover is governed by a user selected **crossover rate** or probability of crossover (typically around **0.75**). Increasing crossover rate increases recombination of building blocks, however, with an increasing probability of loosing good strings.

As an example of a single point crossover, consider two strings X and Y of length 5 mated at random from the mating pool of the new generation (numbers in brackets show a binary coded representation of a possible combination):



If the random draw chooses position 3, the resulting crossover yields two new strings X^*, Y^* after the crossover.

X^* = Y1 Y2 Y3 X4 X5 [1 1 1 1 1]
Y^* = X1 X2 X3 Y4 Y5 [0 0 0 0 0]

2.11.4. Mutation

Mutation is traditionally seen as a background or secondary reproduction operator which restore inadvertently lost gene values (alleles), prevent genetic drift, and provide a small element of random search in the vicinity of the population when it has largely converged. Mutation improves the global search. Usually as the population converges, mutation becomes more productive than crossover. It may be noted that too high a mutation rate would make the search too random. This would disallow exploitation of gradient information in the fitness function. Research has shown that a small population size should have a relatively large mutation rate and vice versa.

Bit-wise Mutation (see *fmmutate.m*)

An user specified **mutation rate** or probability of mutation (typically around **0.001**) determines whether to mutate or not. Usually each bit in the string (chromosome) is flipped (if 1 then 0, if 0 then 1). Other option includes random replacement (replace the value with a random one).

Example:
Before Mutation: X = [0 0 0 **1** 1]; After Mutation: X = [0 0 0 **0** 1]

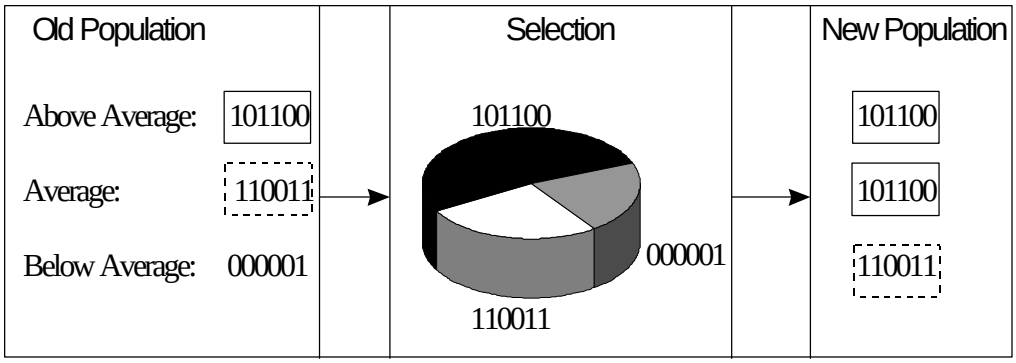
2.11.5. Selection

Selection models nature's *survival-of-the-fittest* principle. The aim is to ensure that fitter strings (solutions) receive a higher number of offspring, thereby getting a higher chance of surviving in the new generation. Selection strategies include **Roulette wheel** selection (proportionate), **tournament** selection, **Ranking** selection, **Boltzmann** selection, etc. **.**

Roulette Wheel Selection (see *fmselect.m*)

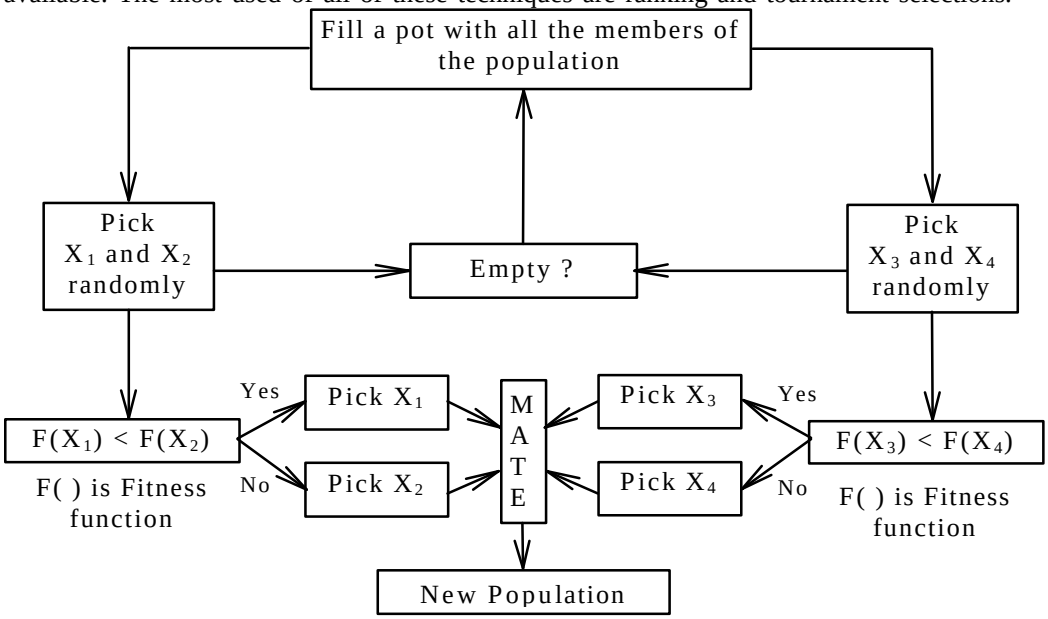
The proportionate selection scheme allocates offspring based on the ratio of the fitness value of a string to the average fitness value of the population. Typical problems include premature convergence and weak promotion of better strings. Efficient implementations of the proportionate selection scheme includes stochastic remainder technique and the stochastic universal sampling technique. These implementations reduce the sampling error which creeps in due to the bias in real-integer conversion. Roulette wheel selection is intuitive and easy to implement. However, it has scaling problems, handles only *maximization* problems, causes large sampling errors, and requires a large population size. Each string is allocated a slot (sector angle = $\frac{2\pi \ fitness}{average\ fitness}$ of a roulette wheel). A string is allocated an offspring

if a random number between 0 and 2π falls in the sector corresponding to the string. Proportionate selection allocates approximately equal numbers of offspring to all strings. This may hinder the promotion of better strings. In Ranking selection, we rank members based on their fitness (performance index) and assign offspring as a function of rank. This avoids scaling problems and can handle both max. and min. problems, but may cause ad hoc allocation of offspring.



Tournament Selection (see *fmselect.m*)

Roulette wheel selection defined earlier suffers from two problems: (1) it can handle only maximization problems; and (2) scaling of the fitness become very important near convergence. Essentially, members with very poor fitness die out very early causing premature convergence. To overcome these problems, few other selection techniques are available. The most used of all of these techniques are ranking and tournament selections.



In the tournament selection strategy, the strings are grouped randomly and adjacent pairs are made to compete for the selection process. There are two options here. In binary, pairs of individuals are picked at random from the population. The one with higher fitness is copied into the mating pool. Larger tournaments are employed by picking n individuals instead of two (a pair). This increases the selection pressure. In probabilistic binary, the better individual wins the tournament with probability $0.5 < p < 1$. Tournament size and win probability controls selection pressure. The fitter member in the pool gets selected for mating. (Care should be taken to avoid two copies of the same string mating for the next generation). There are several advantages in using the tournament selection. These are:

1) Deterministic, works well and shown to be superior to roulette wheel.
2) Tournament size can vary.
3) Can handle maximization or minimization problems.
4) Can be easily extended to multiple criteria problems.

3. FlexGA Tutorial on Software Usage

Step 1: Define objective function

The first step in using the software is to develop an objective function that looks like

PI=flperf(p)
code
PI=....

where *PI* is the performance index (fitness) returned by the function.

An Example is shown on the right.

File Name: ex1.m

```
function PI=ex1(x)

npx(1)=size(x,1);
npx(2)=size(x,2);
np=max(npx);

PI=0.0;
for i=1:np
    PI=PI+(x(i)-0)*(x(i)-0);
end
```

Step 2: Define function and GA parameters

Once an objective function is written, the following vectors that describe the problem to be optimized have to be created.

(Noofp = Number of parameters).

P_type(1:Noofp) = A vector that describes the type of parameter used.

The options are:

1 = Integer type.
2 = Real type that takes discrete values.
> 2 is a Real type that takes continuous values (this number also defines the number of bits that are used to represent each parameter using stochastic GA (see Section 2.6).

P_min(1:Noofp) = A vector of minimum values for the parameter vector.

- If P_type (i) <= 2, P_min(i) represents the minimum value for P(i).
- If P_type (i) > 2, P_min(i) represents an approximate value for the minimum of P(i).

P_max(1:Noofp) = A vector of maximum values for the parameter vector.

- If P_type (i) <= 2, P_max(i) represents the maximum value for P(i).
- If P_type(i) > 2, P_max(i) represents an approximate value for the maximum of P(i).

P_res(1:Noofp) = A vector of resolution values for the parameter vector.

- If P_type (i) <= 2, P_res(i) represents the resolution value for P(i).

- If P_type(i) > 2, P_res(i) is ignored.

gap(1) = Type of GA
- 1= Regular GA
- 2= Micro-GA
- 3=Steady State GA

gap(2) = # of generations for evolution
gap(3) = population size
gap(4) = No of peaks;
gap(5) = Type of selection desired
- 1= Roulette Wheel Selection
- 2= Tournament Selection
- 3= Ranking Selection

gap(6) = Tournament size (> 1 if Tournament selection is chosen).
gap(7) = Steady State population size (>1 if Steady State GA is chosen)
gap(8) = Probability of Crossover (0 - 1)
gap(9) = Number of crossover points (> 1)
gap(10) = Probability of Mutation (0 - 1)
gap(11) = Probability of Gaussian Mutation (0 - 1) [OPTIONAL]
gap(12) = Micro-GA inner loop # of Generations [OPTIONAL]
gap(13) = # of asexual children for stochastic coding [OPTIONAL]

G_disp = Gap Display options [OPTIONAL]
- = 0 (show no plots, no display)
- = 1 (Default, show plots of evolution)
- = 2 (show only Generation count)
- = 3 (show data on screen and no plots. Data includes Generation #, Max, Min, Avg Values)

Several default gap vectors can be created using *fmga_def*.

The command is as follows: **gap=*fmga_def*(gap_choice)**

where,

gap_choice
- =1; Small problem; 10 parameters or less; Regular GA
- =2; Medium problem; 30 parameters or less; Regular GA
- =3; Small problem; 10 parameters or less; Micro GA
- =4; Medium problem; 30 parameters or less; Micro GA
- =5; Small problem; 10 parameters or less; Steady-state GA
- =6; Medium problem; 30 parameters or less; Steady-state GA
- =7; Big problem; 30 parameters or more; Steady-state GA

Step 3: **Write main GA code using the template given below.**

```
%#call ex1  %A necessary statement for declaring the file
            %name where the objective function resides.

fname='ex1'; %Objective function is programmed in ex1.m

%Next lines describe the required inputs to GA.
size=3; % # of parameters in the objective function that need
        % to be optimized

P_min = -3*ones(size,1);
      %Minimum for the parameters (a column vector)
P_max =  3*ones(size,1);
      %Maximum for the parameters (a column vector)
P_res =  0.01*ones(size,1);
      %Resolution for the parameters (a column vector)
P_type = 2*ones(size,1); %Type is real discrete
      %Type of the parameter (a column vector)

% The next line is used to generate default values for flexga
software

gap=fmga_def(1);

G_disp=1; %Display option

%.........................................................
% The program returns
%  Jmax(1:Ngen)=  A vector of max performance index values
%  Jmin(1:Ngen)=  A vector of min performance index values
%  Javg(1:Ngen)=  A vector of average PI values
%  bestP(1:Noofpeaks,1:Noofp) =  A matrix of best Parameters
%  PI(1:popsize)= A vector of  fitness functions for the last
%                 generation
% where
%  Ngen        =  # of generations
%  Noofp       =  # of parameters
%  Noofpeaks   =  # of desired sub-optimal solutions
%  popsize     =  population Size
%.........................................................

[max_PI,min_PI,avg_PI,bestP,PI]=flexga(fname,P_min,P_max,P_re
s,P_type,gap,G_disp);
```

Step 4: **Execute GA software**

Once the main GA code is created, the GA software can be executed at the Matlab command prompt as follows (assuming ***gaex1.m*** is the file name):

gaex1

3.1. Example 1: A Simple Function

We will use a simple function (***ex1.m***) to illustrate the ease of using FlexGA. The function is defined as

$$\min \quad f(x)=x(1)*x(1)+x(2)*x(2)+x(3)*x(3)$$

The function is coded in ***ex1.m.***

The user needs to define:

P_max = [3 3 3]
P_min = [-3 -3 -3]
P_res = [0.01 0.01 0.01]
P_type = [2 2 2]

All three parameters are real types with discrete possibilities separated by 0.01.

Next the gap vector needs to be defined. The Table below gives the values used here.

Gap vector	Values	Description
gap(1)	1	Type of GA (1= Regular GA)
gap(2)	30	# of generations for evolution
gap(3)	31	population size
gap(4)	1	No of peaks;
gap(5)	2	Type of selection desired (2= Tournament Selection)
gap(6)	2	Tournament size (> 1 if Tournament selection is chosen).
gap(7)	1	Steady State population size (>1 if Steady State GA is chosen)
gap(8)	0.77	Probability of Crossover (0 - 1)
gap(9)	2	Number of crossover points (> 1)
gap(10)	0.0077	Probability of Mutation (0 - 1)

Execute GA using the ***gaex1.m*** (shown below).

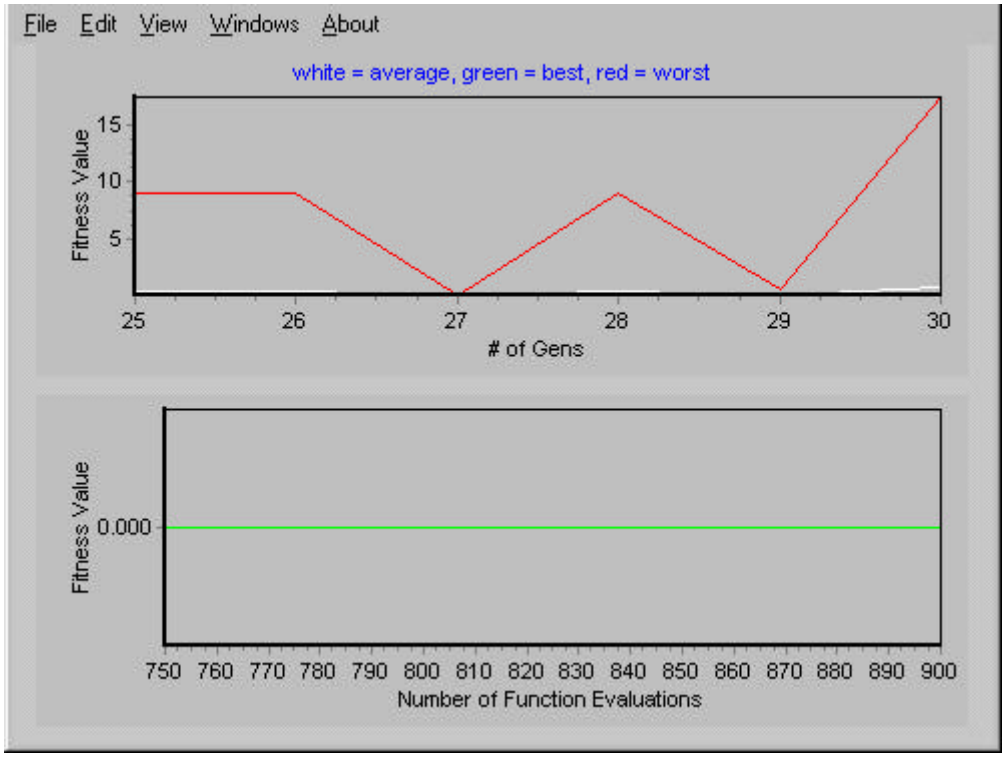
```
%gaex1.m % Main program

%#call ex1
fname='ex1';
size=3;
P_min = -3*ones(size,1);
P_max =  3*ones(size,1);
P_res =  0.01*ones(size,1);
P_type = 2*ones(size,1); %Type is real discrete
gap=fmga_def(1);
G_disp=1;

[max_PI,min_PI,avg_PI,bestP,PI]=flexga(fname,P_min,P_max,P_res,
P_type,gap,G_disp);
```

FlexGA will evolve and show a plot of evolution. An example is shown below. Once the evolution is complete, the following variables can be viewed in the workspace.

max_PI, min_PI, avg_PI, bestP, PI



3.2. Example 2: A Difficult Function.

This function has many optima. For a given dimension *d*, the total number of local minima for (3< x < 13) is 2^d.

$$\max f(x) = \sum_{i=1}^{d} (\sin(x_i) + \sin(\frac{2}{3} x_i))$$

The function is coded in ***ex2.m.***

Please note that a maximization objective can be achieved by minimizing a negated function (as shown below).

$$\min f(x) = -\sum_{i=1}^{d} (\sin(x_i) + \sin(\frac{2}{3} x_i))$$

```
%gaex2.m % Main program

%#call ex2
fname='ex2';
P_min = [3 3 3 3 3 3 3 3 3 3]';
P_max = [13 13 13 13 13 13 13 13 13 13]';
P_type = [2 2 2 2 2 2 2 2 2 2]';
P_res = P_min*0.01;
gap=fmga_def(6);  %Bigger problem
G_disp=1;

[max_PI,min_PI,avg_PI,bestP,PI]=flexga(fname,P_min,P_max,P_res,
P_type,gap,G_disp);
```

3.3. Example 3: Multiple Peaks

The next function used is the Himmelblau function with four equal peaks located at [3,-2], [-3,-2], [3,2], and [-3,2]. This function is defined as:

$$\min f(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$$

We will execute FlexGA with Niching option. this option is automatically invoked when the number of peaks is defined to be greater than 1 (gap(4) defines the number of peaks as shown below).

The function is coded in ***ex3.m.***

Note: When the *number of peaks > 1*, ***bestP*** shows all of the solutions of the last generation sorted in best to worst order.

```
%gaex3.m % Main program

%#call ex3
fname='ex3';
P_min = [-10 -10]';
P_max = [10 10]';
P_res = [0.01 0.01]';
P_type = [2 2]';
gap=fmga_def(1);
gap(4)=2; % number of desired optimal solutions = 2;
G_disp=1;

[max_PI,min_PI,avg_PI,bestP,PI]=flexga(fname,P_min,P_max,P_res,
P_type,gap,G_disp);
```

3.4. Example 4: Non-convex optimization

The function used is presented to the right:

$$\min f(x) = \sum_{i=1}^{10} ix^4$$

The function is coded in ***ex4.m.***

```
%gaex4.m % Main program

%#call ex4
fname='ex4';
size=10;
P_min = -3*ones(size,1);
P_max =  3*ones(size,1);
P_res = 0.01*ones(size,1);
P_type = 3*ones(size,1); %real and continuous with 3 bits to
                         %represent the region
gap=fmga_def(5);
gap(4)=2; % number of desired optimal solutions = 2;
G_disp=3;

[max_PI,min_PI,avg_PI,bestP,PI]=flexga(fname,P_min,P_max,P_res,
P_type,gap,G_disp);
```

3.5. Example 5: Control Optimization

One of the most used products of the modern control theory is the theory of the Linear Quadratic Regulator. In this section an example problem presented by Bryson and Ho [Brys65], a lateral autopilot design, is optimized using the GA (This example was chosen to show the inner workings of a GA and not to suggest GA as an alternative to classical LQR design). Specifically, the GA designs lateral feedback gains based on a quadratic performance index to maintain heading and roll attitude. The perturbation equations of lateral motion are given by a fifth-order system:

$$\dot{X} = AX + BU$$

$X = [\ ß\ r\ p\ \Phi\ \ \Psi\ ]^T$; $U = [\delta_r\ \delta_a\ ]^T$;

where ß = sideslip angle, Ψ = yaw angle,
r = yaw rate, Φ = roll angle,
p = roll rate,
δ_r = rudder deflection,
δ_a = aileron deflection.

The all state feedback system and the Quadratic Performance Index are defined, respectively, as

$$\begin{bmatrix} \delta_r \\ \delta_a \end{bmatrix} = -\begin{bmatrix} K11 & K12 & K13 & K14 & K15 \\ K21 & K22 & K23 & K24 & K25 \end{bmatrix} \begin{bmatrix} ß \\ r \\ p \\ \Phi \\ \Psi \end{bmatrix}$$

Minimize: $J[u] = \lim_{t \to \infty} 1/2 \int_0^t [\ \delta_a^2 + \delta_r^2 + (ß+\Psi)^2 + \Phi^2\ ]\ dt$

To solve this problem, each gain in the gain set $[K_{ij}]$, can be represented as a seven-bit string. The strings then are concatenated by the software to produce a 70-bit string, each string representing one feedback design. As an example, a probable 70-bit string, with spaces added for emphasis, is shown below:

0100101 0110100 1010110 0101011 1011101 1100101 1011100 1100110 1101110 1011000
K11 K12 K13 K14 K15 K21 K22 K23 K24 K25

This string represents one possible solution out of 2^{70} solutions. In the above representation 0000000 represents a predetermined minimum value for the gains and 1111111 represents a predetermined maximum value for the gains. Note here that the mapping of the gain set on to the 7-bit string can be different for different gains depending on the desired range and resolution of the gain values.

The fitness function for the GA implementation is defined as

Minimize: $F(K_{ij}) = \sum_{i=1}^{3} 1/2 \sum_{t=0}^{t=60} [\ \delta_a^2 + \delta_r^2 + (ß+\Psi)^2 + \Phi^2\ ]\ dt$

where l = 1 to3 represents three random initial condition responses of the lateral system. Three initial conditions are chosen to ensure excitation of all lateral modes. For this problem, K_{min} and K_{max} are set at -1 and 1 respectively, and the resolution is chosen to be at least 0.01

The function is coded in ***ex5.m.***

```
%gaex5.m % Main program

%#call ex5
fname='ex5';
P_min  = -1*ones(10,1);
P_max  =  ones(10,1);
P_res  = 0.01*ones(10,1);
P_type = 3*ones(10,1) %real and continuous with 3 bits to
                      %represent the region
gap=fmga_def(5);
G_disp=1;

[max_PI,min_PI,avg_PI,bestP,PI]=flexga(fname,P_min,P_max,P_res,
P_type,gap,G_disp);
```

4. Bibliography

[Adel94] Adeli, H., and Cheng, N. (1994), Augmented Lagrangian genetic algorithm for structural optimization, *Journal of Aerospace Engineering*, 7(1), 104-118.

[Adel94] Adeli, H., and Cheng, N. (1994), Concurrent genetic algorithms for optimization of large structures, *Journal of Aerospace Engineering*, 7(3), 276-296.

[Adle94] Adleman, H. E. (1994), Molecular Computation of Solutions to Combinatorial Problems, *Science*, vol. 266, no. 11, 1021-1024.

[Alli93] Alliot, J., Gruber, H., Joly, G., and Schoenauer, M. (1993), Genetic Algorithms for Solving Air Traffic Control Conflicts, in *Proceedings of the 9th International Conference on Artificial Intelligence for Applications*, 338-344.

[Axel93] Axelsson, J., Menth, S., and Semmler, K.. (1993), Genetic Algorithms in Industrial Design, in *IEEE Proceedings of the International Conference on Tools with Artificial Intelligence*, 64-67.

[Back97] Bäck, T. (1997), Evolutionary Algorithms in Theory and Practice, Oxford University Press.

[Back96] Bäck, T. (1996), Handbook of Evolutionary Computation, Oxford University Press.

[Box57] Box, G. E. P. (1957), Evolutionary Operation: A Method for Increasing Industrial Productivity, *Applied Statistics*, vol. 6, 81-101.

[Bram89] Bramlette, M., and Cusic, R. (1989), A Comparative Evaluation of Search Methods Applied to Parametric Design of Aircraft, Lockheed Aeronautical Systems Co., *Proceedings of the 3rd International Conference on Genetic Algorithms*, 213-218.

[Brys65] Bryson, A. E., and Ho, Y. (1965), *Applied Optimal Control.* Hemisphere Publishing Corporation, New York.

[Conl81] Conley, W. (1981), Optimization: A Simplified Approach, Petrocelli Books, Inc.

[Cord95] Cordon, O., and Herrera, F. (1995), A General Study on Genetic Fuzzy Systems, in: J. Periaux and G. Winter, Ed., *Genetic Algorithms in Engineering and Computer Science*, John Wiley & Sons, Ltd, England.

[Davi91] Davis, L. (1991), Handbook of Genetic Algorithms, NY: Van Nostrand Reinhold.

[Dela94] Delahaye, D., Alliot, J. M., Schoenoauer, M., and Farges, J. L.. (1994), Genetic algorithms for partitioning air space, in *Proceedings of the Conference on Artificial Intelligence Applications*, 291-297.

[Flex95] FlexTool(EFM) M2.1: Evolutionary Fuzzy Modeling Tool for MATLAB (software and manual), Flexible Intelligence Group, L.L.C., Tuscaloosa.

[Foge62] Fogel, L. J. (1962), Autonomous Automata, *Industrial Research*, vol. 4, 14-19.

[Foge95] Fogel, D. B. (1995), Evolutionary Computation, IEEE Press .

[Frie58] Friedberg, R. M. (1958), A Learning Machine: Part I, *IBM J. of Research and Development*, vol.2, 2-13.

[Giff94] Gifford, D. K. (1994), On the Path to Computation with DNA, *Science*, vol. 266, no. 11, 993-994.

[Gold89] Goldberg, D. (1989), Genetic Algorithms in Search, Optimization, and Machine Learning, Addison- Wesley, .

[Gold94] Golddberg, D. E. (1994), "Genetic and Evolutionary Algorithms come of Age," *Comm. ACM*, vol. 37, no. 3.

[Haft90] Haftka, R. T., Gurdal, Z., and Karmat, M. P. (1990), Elements of Structural Optimization, Kluwer Academic Publishers.

[Haje90] Hajela, P. (1990), Genetic Search - an approach to the non convex optimization problem, *AIAA Journal*, vol. 28, no. 7, 1205-1210.

[Holl75] Holland, J. H. (1975), Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor.

[Homa94] Homaifer, A., Lai, H. Y., and McCormick, E. (1994), System optimization of turbofan engines using genetic algorithms, *Applied Mathematical Modeling*, vol. 18, no. 2, 72-83.

[Juri94] Juric, M. (1994), Optimizing Genetic Algorithm Parameters for Multiple Fault Diagnosis Applications, in *IEEE Proceedings of the 10th International Conference on AI for Applications*, 434-440.

[Kauf93] Kauffman, S. A. (1993), The Origin of Order, Oxford University Press, New York.

[Kirb94] Kirby, G., Matic, P., and Lindner, D. K. (1994), Optimal actuator size and location using genetic algorithms for multi variable control, in *Adaptive Structures and Composite Materials: Analysis and Application*, vol. 45, 325-335.

[Koza92] Koza, J. K. (1992), Genetic Programming, MIT Press, 1992.

[Kris89] Krishnakumar, K. (1989), Micro-Genetic Algorithms for Stationary and Non-stationary Function Optimization, *Proceedings of the SPIE's Intelligent Control and Adaptive Systems Conference*, 289-296.

[Kris92] Krishnakumar, K., and Goldberg, D. E. (1992), Control system optimization using genetic algorithms, *Journal of Guidance, Control, and Dynamics*, vol. 15, no. 3, 737-740.

[Kris94a] Krishnakumar, K., Swaminathan, R., and Montgomery, L. (1994), Multiple optimal solutions for structural control sing genetic algorithms with niching, *Journal of Guidance,Control, and Dynamics*, vol. 17, no. 6, 1374-1377.

[Kris94b] Krishnakumar, K., Swaminathan, R., and Garg, S. (1995), Solving Large Parameter Optimization Problems Using a Genetic Algorithm with Stochastic Coding, in: J. Periaux and G. Winter, Ed., *Genetic Algorithms in Engineering and Computer Science*, John Wiley & Sons, Ltd.

[Kris95] Krishnakumar, K., and Satyadas, A. (1995), Evolving Multiple Fuzzy Models and its application to an aircraft control problem, in: J. Periaux and G. Winter, Ed., *Genetic Algorithms in Engineering and Computer Science*, John Wiley & Sons, Ltd, .

[Kris95] Krishnakumar, K., Gonsalves, P., Satyadas, A., and Zacharias, G. (1995), Hybrid fuzzy logic controller synthesis via pilot modeling, *AIAA Journal of Guidance, Control, and Dynamics*, vol. 18, no. 5, 1098-1105.

[Leri93] Leriche, R., and Haftka, R. T. (1993), Optimization of laminate stacking sequence for buckling load maximization by genetic algorithm, *AIAA Journal*, vol. 31, no. 5, 951-956.

[Mich92] Michalewicz, Z. (1992), Genetic Algorithms + Data Structures = Evolution Programs, Springer-Verlag, .

[Mich95] Michalewicz, Z., and Michalewicz, M. (1995), Pro-life Versus Pro-choice Strategies in Evolutionary Computation Techniques, Computational Intelligence, A Dynamic System Perspective, IEEE Press, .

[Mitc96] Mitchell, M. (1996), Introduction to Genetic Algorithms, MIT Press, .

[Noto95] Noton, M. (1995), Orbital strategies around a comet by means of a genetic algorithm, *Journal of Guidance, Control, and Dynamics*, vol. 18, no. 5, 1217-1220.

[Pell94] Pellazar, M. (1994), Vehicle route planning with constraints using genetic algorithms, in *IEEE Proceedings of the National Aerospace and Electronics Conference*, vol. 1, 111-118.

[Quag95] Quagliarella, D., and Cioppa, A. (1995), Genetic algorithms applied to the aerodynamic design of transonic airfoils, *Journal of Aircraft*, vol. 32, no. 4, 889-891.

[Rao93] Rao, S. S. (1993), Genetic algorithmic approach for multi objective optimization of structures, in *Structures and controls optimization*, vol. 38, 29-38.

[Rech65] Rechenberg, I. (1965), Cybernetic Solution Path of an experimental problem, *Royal Aircraft Establishment*, Library Translation No. 1122.

[Russ90] Russell, P. J. (1990), Genetics, Harper Collins Publishers, .

[Samu97] Samuelson, L. (1997), Evolutionary Games and Equilibrium Selection, MIT Press.

[Saty95] Satyadas, A., and Krishnakumar, K. (1995), Evolving Lean Fuzzy Controllers using Evolutionary Fuzzy Modeling, *Proc. of VIth International Fuzzy Systems Association (IFSA) World Congress*, I, Sao Paulo, Brazil, 253-256.

[Schw95] Schwefel, H. (1995), Evolution and Optimum Seeking, Wiley Inter-Science.

[Scot93] K. A. Scott, Five ways to a smart Genetic Algorithms, *AI Expert*.

[Seyw95] Seywald, H., Kumar, R. R., and Deshpande, S. M. (1995), Genetic algorithm approach for optimal control problems with linearly appearing controls, *Journal of Guidance, Control, and Dynamics*, vol. 18, no. 1, 177-182.

[Tucc95] Tuccillo, R., and Senatore, A. (1995), A Genetic Algorithm Based Approach to Radial Flow Impeller Design, *Computational Fluid Dynamics in Aeropropulsion* ASME, vol. 49, 51-62.

[Twar94] Twardowski, K. (1994), "An Associative Architecture for Genetic Algorithm-Based Machine Learning," *IEEE Computer*, 27-38.

[Zuo95] Zuo, W. (1995), Multi variable adaptive control for a space station using genetic algorithms, *IEE Proceedings: Control Theory and Applications*, vol. 142, no. 2, 81-87.

FLEXIBLE INTELLIGENCE GROUP, L.L.C. SOFTWARE LICENSE AGREEMENT

READ THE TERMS AND CONDITIONS OF THIS LICENSE CAREFULLY BEFORE OPENING THIS PACKAGE. THIS LICENSE AGREEMENT REPRESENTS THE ENTIRE AGREEMENT BETWEEN YOU (the **"Licensee" - either an individual or an entity)** AND FLEXIBLE INTELLIGENCE GROUP, L.L.C. ("FIG") CONCERNING THE COMPUTER SOFTWARE CONTAINED HEREIN ("FlexGA") THE ACCOMPANYING USER DOCUMENTATION.

BY OPENING THIS PACKAGE, YOU ACCEPT THE TERMS OF THIS AGREEMENT. IF YOU ARE NOT WILLING TO DO SO RETURN THE UNOPENED PACKAGE IMMEDIATELY FOR A REFUND. YOU MAY ALSO RECEIVE A FULL REFUND IF TERM OF AGREEMENT.

LICENSE GRANT: FIG hereby grants to Licensee a nonexclusive license to install and use the FlexGA and documentation as provided herein.

INSTALLATION AND USE: This license permits Licensee to install and use one copy of the FlexGA on a single computer. "Use" means that a copy is loaded into temporary memory or installed into the permanent memory of a computer, except that a copy installed on a network server for the sole purpose of distribution to other computers is not in "use". Licensee is responsible for limiting the number of possible concurrent users to the number licensed. Each copy of the FlexGA may be used on a backup computer (if the original is not functional) or a replacement computer. The documentation provided with the FlexGA may not be copied.
Licensee shall use the FlexGA only for its internal operations. "Internal operations" shall include use of the FlexGA by Licensee's own employees or those of its subsidiaries or parent company, and in the performance of consulting or research for third parties who engage Licensee as an employee or independent contractor. Licensee may allow use of the FlexGA by employees, consultants, students and/or (in the case of individual licensees) colleagues, but Licensee may not make the FlexGA available for use by third parties generally on a "time sharing" basis.
Licensee may make copies of the FlexGA only for backup or archival purposes. All copies of the FlexGA and Documentation shall contain all copyright and proprietary notices in the originals.
FlexGA licensed to degree-granting educational institutions are further restricted to use in connection with on-campus computing facilities that are used solely in support of classroom instruction and research activities of students and faculty. FIG excludes the right for the Licensee to use the FlexGA for commercial purposes.
Licensee shall take appropriate action by agreement, instruction, or otherwise with all persons permitted access to the FlexGA, to enable Licensee to satisfy this Agreement obligations.

TERMS OF AGREEMENT: This Agreement shall continue until terminated by FIG or Licensee as provided below.

TERMINATION: FIG may terminate this license by written notice to Licensee if Licensee (a) breaches any material term of this Agreement, (b) ceases conducting business in the normal course, becomes insolvent or bankrupt, or avails itself of or becomes subject to any proceedings pertaining to insolvency or protection of creditors. Licensee may terminate this Agreement at any time by written notice to FIG. Licensee shall not be entitled to any refund if this Agreement is terminated. Upon termination, Licensee shall promptly return all copies of the FlexGA and Documentation in Licensee's possession or control, or promptly provide written certification of their destruction.

LIMITED WARRANTY: FIG warrants that FlexGA will conform in all material respects to the description of its operation in the Documentation for a period of 30 days from delivery. In the event that FlexGA does not materially operate as warranted, Licensee's exclusive remedy and FIG's sole liability under this warranty shall be (a) FIG shall correct or work around major defects within a reasonable time, or (b) should such correction or work around prove neither satisfactory or practical, termination of the License and refund of the license fee paid to FIG for the FlexGA. THE FOREGOING WARRANTY IS IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. FIG SHALL NOT BE LIABLE FOR ANY SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING WITHOUT LIMITATION LOST PROFITS. Licensee accepts responsibility for its use of the FlexGA and the results obtained therefrom.

LIMITATION OF LIABILITY: FIG SHALL NOT BE LIABLE TO LICENSEE FOR MORE THAN THE AMOUNT PAID BY LICENSEE TO FIG FOR THE FlexGA WITH RESPECT TO WHICH THE LIABILITY IN QUESTION ARISES, AS INSTALLED ON THE SINGLE DESIGNATED COMPUTER FOR WHICH USE OF THE PROGRAM IS LICENSED HEREUNDER.

GENERAL PROVISIONS: Licensee may not assign this License without written consent of FIG, except to a parent or subsidiary company of Licensee. Should an act of Licensee purport to create claim, lien, or encumbrance on any FlexGA, such claim, lien, or encumbrance shall be void. All provisions regarding liability, warranty, and limits thereon, and protection of proprietary rights, shall survive termination of this Agreement, as shall all provisions regarding payment of amounts due at the time of termination. This Agreement shall be governed by the internal laws of Alabama. Should Licensee install the FlexGA outside the United States, Licensee shall comply fully with all applicable laws and regulations relating to export of technical data. This Agreement contains the entire understanding of the parties and may be modified only by written instrument signed by both parties.