# Chapter 14

# The Genetic Algorithm

## Chapter Contents

Darwin pioneered the idea that biological organisms develop and adapt over long periods of time via "descent with modification." For example, organism "parents," each with their own genetic makeup, mate, and their children's genetic makeup is a mixture of their parents so that often in appearance, you see characteristics of both parents (ideas originally studied by Mendel for varieties of garden peas). Sometimes, there are molecular "mutations" where an abnormal gene arises, which then affects the formation of the child. Both the mating and the mutations result in children growing up to be more or less fit to survive and mate in the environment that they live in. Children who are more fit tend to have more offspring, while children who are less fit often do not get the chance to mate, or have fewer offspring. There is then a "natural selection" that proceeds gradually over time so that populations evolve to be more fit for their environment.

The genetic algorithm (GA) is a computer simulation that incorporates ideas from Darwin's theory on natural selection, and Mendel's work in genetics on inheritance, and it tries to simulate natural evolution of biological systems. From an engineering perspective, the genetic algorithm is an optimization technique that evaluates more than one area of the search space and can discover more than one solution to a problem. (Some would call it a type of stochastic direct search method.) In particular, it provides a stochastic optimization method where, if it "gets stuck" at a local optimum, it tries, via multiple search points, to simultaneously find other parts of the search space and "jump out" of the local optimum to a global one that represents the highest fitness individuals.

Evolution is the theory and mechanism that is ubiquitous and fundamental to all of biology (bacteria, plants, and animals are all subject to the mechanisms of evolution). One would expect it to have a similar pervasive role in all of intelligent control. As discussed in this part, it applies to evolution of neural, fuzzy, expert, planning, attentional, and learning systems.

## 14.1 Biological Evolution

The basic process of biological evolution was explained in Section 2.5 in Part I on page 80 and it would be good to review that material before proceeding. Here, however, we also give a brief overview of evolution in biological systems so that you can easily form appropriate analogies to biological systems as you learn about genetic algorithms. At the basis of evolution lies selection, mating, and mutation, and each of these is outlined next.

"Survival of the fittest" refers to fitness in terms of reproductive success. Natural selection is the process where organisms with higher reproductive success generate offspring, and hence, propagate their DNA through time. Less fit individuals do not have offspring (or have fewer of them) and hence can become extinct over time. For instance, consider the cormorant (large fish-eating seabirds that catch their prey under water). There are more than one species of this bird. The one found on the coast of South America has sufficient wingspan to fly, but the only member of its family that does not fly is the "flightless"

cormorant that is found on the remote Galapagos Islands (part of Ecuador), which are quite far from mainland South America in the Pacific Ocean. There are no natural predators on the islands and a plentiful supply of fish immediately offshore. Its loss of flight does not seem to have harmed them, and in fact can be viewed as beneficial since then it does not have to use the extremely energy-expensive activity for obtaining food. Apparently, there was a selective pressure on wing length that drove the evolutionary history of the flightless cormorant. This example illustrates the powerful force that selection provides in "designing" organisms.

*It is often useful to think of the environment as the designer of the organism.*

Mating and reproduction drive evolution. Mating is a process of mixing (combining) chromosomes of the parents, and it tends to "homogenize" the gene pool of the population. Different parts of chromosomes from each parent are combined in a child. It is via this swapping of genetic material on the chromosomes that a child inherits some characteristics of each parent (and thereby, as you would imagine, characteristics of each grandparent). On *average* we may think of each individual as being composed of half of one parent and half of the other, one fourth of each grandparent, and so on. This swapping of genetic material, and hence blending of outward characteristics, is apparent in many living organisms that mate. For example, for some types of corn, pollination from a dark colored corn stalk to a corn stalk with a "white" gene bears an ear of corn that is half dark and half white.

Mutations result in variations in the offspring that result from mating. The mutation rate is dictated by the probability of error in gene replication in biological systems (and certain "mutagens," whose source can lie in the environment, can affect this rate). While we typically think of mutations as something undesirable in biological systems (e.g., some may think of "mutants" in science-fiction movies), they can also lead to more fit individuals (i.e., the mutation may represent a jump to a region of the space where the reproductive fitness increases significantly).

## 14.2 Representing the Population of Individuals

The "fitness function" measures the fitness of an individual to survive, mate, and produce offspring in a population of individuals for a given environment. The genetic algorithm will seek to *maximize* the fitness function $\bar{J}(\theta)$ by selecting the individuals that we represent with $\theta$ (note that we place the bar over the cost function to emphasize that we seek to maximize this function, where we always sought to minimize "$J$" in the studies on optimization for approximation in Part III).

### 14.2.1 Strings, Chromosomes, Genes, Alleles, and Encodings

To represent the genetic algorithm in a computer, we make $\theta$ a string. A string represents a chromosome in a biological system and one is shown in Figure 14.1.

A chromosome is a string of "genes" that can take on different "alleles." In a computer, we often use number systems to encode alleles. Here, we adopt the convention that a gene is a "digit location" that can take on different values from a number system (i.e., different types of alleles).

*We encode the parameters of the optimization problem on the chromosome, which can simply be a sequence of base-10 numbers.*

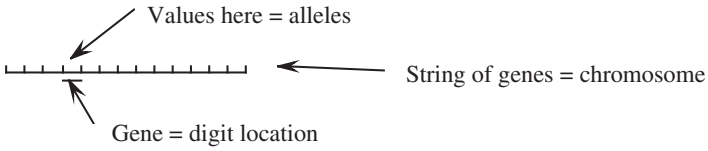


Figure 14.1: String for representing an individual.

In a base-2 number system, alleles come from the set $\{0, 1\}$, while in a base-10 number system, alleles come from the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Hence, a binary chromosome has zeros or ones in its gene locations. As an example, consider the binary chromosome

$$1011110001010$$

which is a binary string of length 13. If we are seeking to optimize parameters of a system that come in a base-10 number system, then we will need to encode the numbers into the binary number system (using the standard method for the conversion of base-10 numbers to base-2 numbers). We will also need to decode the binary strings into base-10 numbers to use them. Here, we will develop the genetic algorithm for base-2 or base-10 number systems, but we will favor the use of the base-10 representation in our examples, since encoding and decoding is simple in that case (and it can be computationally expensive for online, real-time applications if the proper representation is not used).

As an example of a base-10 chromosome, consider

$$8219345127066$$

which has 13 gene positions. For such chromosomes we add a gene for the sign of the number (either "+" or "−") and fix a position for the decimal point. For instance, for the above chromosome we could have

$$+821934.5127066$$

where there is no need to carry along the decimal point; the computer will just have to remember its position. Note that we could also use a floating point representation where we could code numbers in a fixed-length string plus the number in the exponent. The ideas developed here work just as readily for this number representation system as for standard base-2 or base-10.

### 14.2.2 Encoding Examples

It is possible to encode many different problems so that artificial evolution can be applied. We discuss a few representations here, using a base-10 number system, that are particularly relevant to the field of intelligent control.

### Proportional-Integral-Derivative Controllers

*We can encode a wide variety of structures (e.g., controllers) into a form that the genetic algorithm can operate on; hence, the genetic algorithm is a very general optimization tool (and this creates potential for its misuse).*

Suppose that you want to evolve a proportional-integral-derivative (PID) controller (e.g., using a fitness function that quantifies closed-loop performance and is evaluated by repeated simulations). In this case, suppose that we have three gains, $K_p$, $K_i$, and $K_d$, and that at some time we have

$$K_p = +5.12,\ K_i = 0.1,\ K_d = -2.137$$

then we would represent this in a chromosome as

$$+051200 + 001000 - 021370$$

which is a concatenation of the digits, where we assume that there are six digits for the representation of each parameter (two before the decimal point and four after it) plus the sign digit (this is why you see the extra padding of zeros). The computer will have to keep track of where the decimal points are. We see that each chromosome will have a certain structure (its "genotype" in biological terms, and the entire genetic structure is referred to as the "genome"). Each chromosome represents a point in the search space of the genetic algorithm. Here, we will use the term "phenotype" from biology to refer to the whole structure of the controller that is to be evolved; hence, in this case the phenotype is

$$K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}$$

where $e = r - y$ is the error input to the PID controller, $r$ is the reference input, and $y$ is the output of the plant.

Clearly, a similar approach can be used to encode lead-lag compensators, state feedback controllers, nonlinear controllers, adaptive controllers, and so on.

### Elements of Decision Making

In this case, you simply concatenate the parameters of the fuzzy or neural system that you would like to evolve. For instance, for a Takagi-Sugeno fuzzy system you may simply encode the consequent parameters, or you may want to also encode the membership function parameters (centers and spreads). Similarly, for a neural network, you could encode all the weights and biases. Hence, in this case we have $\theta$ defined similar to the case where we adjust it with least squares or gradient methods (except here, $\theta$ is often treated as a string of concatenated parameters, rather than a vector of parameters).

Another interesting, and useful, possibility to consider is the case when you encode the "structure" of the fuzzy or neural system. For instance, you may encode the number of rules, different forms of consequent functions (e.g., different nonlinearities), different membership functions (e.g., both Gaussian and triangular), different inference methods (e.g., use of product or minimum in premise quantification with fuzzy logic), or different defuzzification methods

(e.g., center-of-gravity and center-average). In a neural network you could encode the number of layers, number of neurons in each layer, existence of connections, methods to combine the inputs to the activation function (e.g., linear, polynomial functions of the inputs, other nonlinear functions), and activation function types (e.g., logistic, hyperbolic tangent, linear). Sometimes you may want to encode certain characteristics of both fuzzy and neural systems (e.g., the number of rules and neurons), and then allow them to coexist within a population.

In these cases, you can evolve the structure of the fuzzy or neural system. The genetic algorithm should be viewed as a very general optimization tool that can be used to adapt both parameters and structure of intelligent systems (of course, structure can be quantified with parameters and this is often done in practice). Indeed, many think of the genetic algorithm as having its most natural role in the adjustment (or design) of structure of systems rather than parameter tuning, which is often associated with learning over a lifetime. Next, we discuss some very general ways to encode higher-level cognitive functions.

Just like the fuzzy system, the number of rules used by an expert controller can be encoded. Moreover, all the other functional components can be encoded, such as the conflict resolution strategies. For planning systems, we could encode the look-ahead horizon length, the models used for projecting ahead, and so on. For attentional systems, we can make similar general encodings (e.g., on response times of refocusing, or intensity levels needed to evoke a response).

### 14.2.3 The Population of Individuals

*A population is a set of candidate solutions (chromosomes).*

Next, we develop a notation for representing a whole set of individuals (i.e., a population). Let $k$ denote the generation number. Let $\theta_i^j(k)$ be a single parameter at time $k$ (a fixed-length string with sign digit). Here, we number the chromosomes and the superscript $j$ refers to the $j^{th}$ chromosome. Also, we number the "traits" on each chromosome. (Note that strictly speaking, a trait in a biological system is most often thought of as a property of the phenotype, like hair or eye color, while in our systems, as in the PID controller example above, genes, which are digits of parameters, are "expressed" as traits of the phenotype; hence, we think of strings of genes as being expressed as traits in the phenotype.) With this, the $i$ subscript on $\theta_i^j(k)$ refers to the $i^{th}$ trait on the $j^{th}$ chromosome. Suppose that chromosome $j$ is composed of $p$ of these parameters (traits).

Let

$$\theta^j(k) = \left[\theta_1^j(k), \theta_2^j(k), \ldots, \theta_p^j(k)\right]^\top$$

be the $j^{th}$ chromosome. Note that earlier we had concatenated elements in a string, while here we simply take the concatenated elements and form a vector from them. We do this simply because this is probably the way that you will want to code the algorithm in the computer (e.g., in Matlab). We will at times, however, still let $\theta^j$ be a concatenated string when it is convenient to do so. It will be clear from the context which form of representation we are using.

The population of individuals at time $k$ is given by

$$P(k) = \left\{\theta^j(k) : j = 1, 2, \ldots, S\right\} \tag{14.1}$$

(not to be confused with the covariance matrix used in recursive least squares) and the number of individuals in the population is given by $S$. We want to pick $S$ to be big enough so that the population elements can cover the search space. However, we do not want $S$ to be too big, since this increases the number of computations we have to perform. In an optimization example below, we will discuss some issues in the choice of the size of $S$.

Note that while we will not do so here, you could allow the size of the population to vary with time (more like it does in nature) and the size of the population could depend on resources in the environment, competition between individuals (e.g., as measured by fitness), and physical constraints. Population size plays a fundamental role in nature. Nature often exploits the use of very high numbers of individuals and allows for many individuals to be killed so that population size may vary significantly (at least in some regions). Our simulations of evolution often do not exploit this due to a lack of computational resources.

## 14.3 Genetic Operations

The population $P(k)$ at time $k$ is often referred to as the "generation" of individuals at time $k$. Basically, according to Darwin, the most qualified individuals survive to mate and produce offspring. We quantify "most qualified" via an individual's fitness $\bar{J}(\theta^j(k))$ at time $k$. For selection, we create a "mating pool" at time $k$, something every individual would like to get into, which we denote by

$$M(k) = \left\{m^j(k) : j = 1, 2, \ldots, S\right\} \tag{14.2}$$

The mating pool is the set of chromosomes that are selected for mating. Here, we perform selection to decide who gets in the mating pool, mate the individuals via crossover, then induce mutations. After mutation we get a modified mating pool at time $k$, $M(k)$. Below, we will outline the operations involved in creating the mating pool, performing mating for individuals in the mating pool, and subsequent mutations. This will explain how the $m^j(k)$ in $M(k)$ above are created and modified.

To form the next generation for the population, we let

$$P(k+1) = M(k)$$

Evolution occurs as we go from a generation at time $k$ to the next generation at time $k+1$. Hence, in this artificial environment mating is done in parallel and is synchronized with a clock, which is far different from how it typically occurs in nature. Parallel asynchronous versions of the algorithm can, however, also be developed.

### 14.3.1 Selection

There are many ways to perform selection, but by far the most common one used in practice is fitness-proportionate selection.

#### Fitness-Proportionate Selection

In this case, we select an individual (the $i^{th}$ chromosome) for mating by letting each $m^j(k)$ be equal to $\theta^i(k) \in P(k)$ with probability

$$p_i = \frac{\bar{J}(\theta^i(k))}{\sum_{j=1}^{S} \bar{J}(\theta^j(k))} \tag{14.3}$$

*Selection dictates that the best points in the search space should be given preferential treatment in specifying the composition of the population at the next step.*

To clarify the meaning of this formula and hence the selection strategy, you can use the analogy of spinning a unit circumference roulette wheel where the wheel is divided like a pie into $S$ regions where the $i^{th}$ region is associated with the $i^{th}$ individual of $P(k)$. Each pie-shaped region has a portion of the circumference that is given by $p_i$ in Equation (14.3). You spin the wheel, and if the pointer points at region $i$ when the wheel stops, then you place $\theta^i$ into the mating pool $M(k)$. You spin the wheel $S$ times so that $S$ elements end up in the mating pool and the population size stays constant.

Clearly, individuals who are more fit will end up with more copies in the mating pool; hence, chromosomes with larger-than-average fitness will embody a greater portion of the next generation. At the same time, due to the probabilistic nature of the selection process, it is possible that some relatively unfit individuals may end up in the mating pool $M(k)$.

#### Other Selection Strategies

There are many other options that have been considered for selection besides the fitness-proportionate approach above. For instance, sometimes the individuals in the population are ranked by order of fitness and a fixed number of the least fit ones are "killed" and only the ones in the remaining set are used in the selection process, perhaps with a fitness-proportionate method (this eliminates the possibility of very unfit individuals from mating). Other times, some subset of very fit individuals are allowed to get into the mating pool without spins of the roulette wheel. Such strategies are often called "elitist" strategies since the individuals who are most fit (the elite ones) are assured to be able to get into the mating pool. Sometimes there is only one elite individual that is allowed, and at other times you could allow more than one. Often, when such elitist strategies are used, the elite individual(s) are allowed to proceed directly to the next generation, without modification via the crossover and mutation operations that are discussed next. (In this sense, we can think of the elite individuals as having an ability to clone themselves so that their "offspring" are exact copies of themselves.)

### Using Gradient Information Before or After Selection

There are many relationships between genetic algorithms and other optimization methods. For instance, there are many stochastic methods for optimization of nonlinear and nonconvex functions that bear similarities to the genetic algorithm (see the next chapter). The advantages and disadvantages of these methods relative to the genetic algorithm tend to be very application dependent, so we will not comment on relative merits of the methods and which to pick in a particular situation. There are, however, ways to use ideas from conventional gradient optimization methods in genetic algorithms and we briefly discuss this here.

*Traditional gradient methods can be integrated into genetic algorithms.*

Traditionally, it has been said that one of the key advantages of the genetic algorithm is that it does not rely on the existence and use of gradient information (as the gradient methods do). In some contexts, such as for estimation and control problems, however, it could be that there is useful gradient information available that you may not want to ignore. For instance, we know that in adjusting a nonlinear in the parameter approximator (that serves as an estimator or controller), it can be very difficult for a gradient method to find the global extremum value since it may get stuck at a local extremum. In such cases, it is possible that a genetic algorithm can help find the way out of such local extrema to find the global ones. Now, in such cases, for optimization algorithm design, you could start with a gradient method and add certain features from the genetic algorithm (e.g., evolving a population, use of random excursions by random re-initialization at various steps, etc.). In addition, for your algorithm design, you could think of the genetic algorithm as the main vehicle for optimization and interleave gradient updates. To accomplish this, you could, for instance, perform one or more gradient-based parameter update steps for every individual before (or after) selection is used to place individuals in the mating pool. In this way, it is hoped that we gain the benefits of using the directional information used in the gradient updates, and the benefits of parallel search and random excursions given by the genetic algorithm. See Section 15.5 at the end of the next chapter for more discussion on such "interleaved" and hybrid methods. See the "For Further Study" section at the end of the part for a reference that studies other approaches.

## 14.3.2 Reproduction Phase, Crossover

We think of crossover as mating in biological terms, which at a fundamental biological level involves the process of combining (mixing) chromosomes.

*Crossover adds a mechanism for both local and global search, but near fit individuals.*

For the computer simulation of evolution, the crossover operation operates on the mating pool $M(k)$ by "mating" different individuals there. First, you specify the "crossover probability" $p_c$ (usually chosen to be near 1 since, when mating occurs in biological systems, genetic material is certainly swapped between the parents). There are many types of crossover (i.e., ways to swap genetic material on chromosomes), but the simplest one is "single-point" crossover.

### Single-Point Crossover

The procedure for single-point crossover consists of the following steps:

1. Randomly pair off the individuals in the mating pool $M(k)$. There are many ways to do this. For instance, you could simply pick each individual from the mating pool and then randomly select a different individual for it to mate with. Or, you could just have all the individuals mate with the ones that are right next to each other (where "right next to each other" is defined by how you label the individuals with a number system). In this approach, if there are an odd number of individuals in $M(k)$, then, for instance, you could simply take the last individual and pair it off with another individual who has already been paired off (or you could pair it off with the individual with the highest fitness).

2. Consider the chromosome pair $\theta^j, \theta^i$ that was formed in step 1. Generate a random number $r \in [0, 1]$.

   (a) If $r < p_c$, then cross over $\theta^j$ and $\theta^i$. To cross over these chromosomes, select a "cross site" at random and exchange all the digits to the right of the cross site of one string with those of the other. This process is pictured in Figure 14.2. In this example, the cross site is position 5 on the string (be careful in how you count positions), and hence, we swap the last eight digits between the two strings. Clearly, the cross site is a random number that is greater than or equal to 1, and less than or equal to the number of digits in the string minus 1.

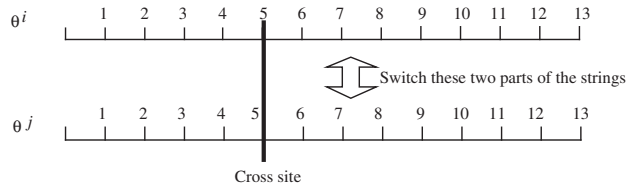


Figure 14.2: Crossover operation example.

   (b) If $r > p_c$, then we will not cross over; hence, we do not modify the strings, and we go to the mutation operation below.

3. Repeat step 2 for each pair of strings in the mating pool $M(k)$.

As an example, suppose that $S = 10$ and that in step 1 above, we randomly pair off the chromosomes. Suppose that $\theta^5$ and $\theta^9$ $(j = 5, i = 9)$ are paired off where

$$\theta^5 = +2.9845$$

and

$$\theta^9 = +1.9322$$

Suppose that $p_c = 0.9$ and that when we randomly generate $r$, we get $r = 0.34$. Hence, by step 2 we will cross over $\theta^5$ and $\theta^9$. According to step 2, we randomly pick the cross site. Suppose that it is chosen to be position 3 on the string (include the sign as a position). In this case, the strings that are produced by crossover are

$$\theta^5 = +2.9322$$

and

$$\theta^9 = +1.9845$$

Besides the fact that crossover helps to model the mating part of the evolution process, why should the genetic algorithm perform crossover? Basically, the crossover operation perturbs the parameters near good positions to try to find better solutions to the optimization problem. It tends to help perform a localized search around the more fit individuals (since on average the individuals in the mating pool $M(k)$ at time $k$ should be more fit than the ones in the population $P(k)$ at time $k$) that could be near each other on the fitness landscape. However, on a complex landscape, two relatively well-fit individuals may be on very different parts of the landscape, so that an offspring may lie "between" them (or extrapolated along a line connecting the two, but not too far away) at points that represent poor fitness. In this case, you would *not* think of crossover as producing local search. Indeed, in this situation it results in a global type of search.

### Other Crossover Methods

There are many other crossover methods that have been studied. For instance, you could use a two-point crossover, where you pick two crossover points on each chromosome and swap the elements in between the two points. Or, more generally, you could have a multi-point crossover where you have one or more crossovers per trait. Generally, when elitism is used, the elite individuals would not undergo any type of crossover.

Another option is to make the crossover rate change with time. For instance, you could start with $p_c = 1$ and then reduce it as the overall fitness of the population (as measured by, for example, the average of the fitness values of all the individuals) increases. This way, there will be fewer explorations into close-by regions when we are likely to be near a local maxima. Sometimes, however, this can cause "premature convergence" where the algorithm locks on to some values and does not properly explore other parts of the space to find the global maximum. Sometimes, for online applications, especially when $\bar{J}$ is time-varying, you want to keep the crossover rate at $p_c = 1$ for the entire time, since this will ensure good exploration of the space. Note that you can think of the crossover probability as being under genetic control so that its value could be adapted also.

In other methods, similarity measures between individuals are developed and only similar individuals are allowed to mate and hence, cross over (you can then think of the population as having multiple species, with mating only within

species). This may be a way to cope with having multiple types of structures (e.g., fuzzy and neural systems) within a population. Or, you could just allow individuals of similar fitness to mate, or you could pick the most fit individual and have everyone else mate with that one.

Sometimes, you may want to "spatially" restrict mating so that only those individuals who are "close" (e.g., with close defined in terms of the Euclidean distance between two individuals) are allowed to mate and swap genetic material (otherwise it is possible that two very different individuals mate). This would then restrict crossover to a local type search.

### 14.3.3 Reproduction Phase, Mutation

Like crossover, mutation modifies the mating pool (i.e., after selection has taken place). The operation of mutation is normally performed on the elements in the mating pool after crossover has been performed. The biological analog of our mutation operation is the random mutation of genetic material. Again, there are many ways to perform mutations. Below, we will discuss the most common methods.

*Mutation provides a mechanism to jump out of local maxima and to randomly explore local and wide areas of the search space.*

#### Gene Mutations

To perform mutation in the computer, first choose a mutation probability $p_m$. With probability $p_m$, change (mutate) each gene location on each chromosome randomly to a member of the number system being used. For instance, in a base-2 genetic algorithm, we could mutate

$$1010111$$

to

$$1011111$$

where the fourth bit was mutated to one. For a base-10 number system, you would simply pick a number at random to replace a digit, if you are going to mutate a digit location (normally we do not consider a replacement to be valid if we replace a digit with the same value).

Besides the fact that this helps to model mutation in a biological system, why should the genetic algorithm perform mutation? Basically, it provides random excursions into new parts of the search space. It is possible that we will get lucky and mutate to a good solution. It is the main mechanism (crossover can also help) that tries to make sure that we do not get stuck at a local maxima and that we seek to explore other areas of the search space to help find a global maximum for $\bar{J}(\theta)$. Usually, the mutation probability is chosen to be quite small (e.g., less than 0.01), since this will help guarantee that all the individuals in the mating pool are not mutated, so that any search progress that was made is lost (i.e., we keep it relatively low to avoid degradation to exhaustive search via a random walk in the search space).

Keep in mind that we can think of mutation as providing both a local and global search component to the genetic algorithm. If, for instance, the mutation of a gene at a particular location on the chromosome represents a small (large) magnitude change, then a random local (global, respectively) search behavior is exhibited.

#### Other Mutation Methods

Sometimes, you could mutate an entire trait (i.e., a set of genes). Other times, you may want to restrict mutation by only allowing certain genes, traits, or individuals to be mutated. There could be reasons to vary the mutation rate; indeed, in biological systems, the mutation rate is under genetic control (i.e., it could evolve to an optimal level to make sure that the population can properly adapt to its environment). As an example, in some applications you may want to start with a relatively high mutation rate and then decrease the mutation rate as the overall fitness of the population increases. Other times you may simply want to code the mutation rate in a chromosome and try to evolve it.

In applications where the fitness function is fixed in time, you often do not want to have a great reliance on mutation in generating new solutions, as it is by simple luck that mutation succeeds. However, when the fitness function changes with time (e.g., in biological "coevolution"), you may want a higher mutation rate to ensure that many options are considered. For instance, in online applications where the fitness function is time-varying, there is sometimes a need for an exceptionally high mutation rate to ensure that you do not at any point get stuck in a local maxima (since the actual maxima points can be changing) and that you actively pursue many different solution options so as to ensure active adaptation. You have to be careful, however, not to have the mutation rate too high or any search progress made by the algorithm at earlier stages can be destroyed. Also, typically in such online approaches, an "elitism" strategy is used to ensure that at least one good solution is available at all times (i.e., the elite individual is not subjected to any mutations). Even in situations where the fitness function is not time-varying, elitism has been used very effectively as a way to keep the best solution available while searching for others (this will be illustrated in Section 14.5 for an optimization problem).

## 14.4 Programming the Genetic Algorithm

In this section we briefly discuss how to code the genetic algorithm, issues you encounter in choosing the method to code it, memory and computation time requirements, and termination and initialization issues.

### 14.4.1 Pseudocode for a Simple Genetic Algorithm

To summarize the operations of the genetic algorithm, and provide some guidance on how to implement the algorithm in a computer, we provide some high-level pseudocode that could be useful in programming the genetic algorithm in

any computer language. Here, we assume for convenience that we use fitness-proportionate selection, single-point crossover with probability $p_c$ (below, `pc`), gene mutation with probability $p_m$ (below, `pm`), and we terminate after some fixed number of iterations, $N_{ga}$ (below, `Nga`).

1. Define the GA parameters (e.g., crossover and mutation probability, population size, termination parameters).
2. Define the initial population $P(0)$.
3. For $k = 1$ to $N_{ga}$ (main loop for producing generations).

   Compute the fitness function for each individual.

   Selection: From $P(k)$, form $M(k)$, the mating pool at iteration $k$, using fitness-proportionate selection.

   Reproduction: For each individual in $M(k)$, select another individual in $M(k)$, mate the two via crossover with probability $p_c$, mutate each gene position with probability $p_m$. Take the $S$ children produced by this process, and put the children in $P(k+1)$, the next generation.
4. Next $k$ (i.e., return to step 3).
5. Provide results.

Clearly this only provides a high-level view of the operation of the genetic algorithm. The details of the various steps depend on how you design your particular genetic algorithm and on the programming language you use.

### 14.4.2 Alternative Sequencing of Operations

The above pseudocode shows one common way to implement the GA. There are many possible variations on this approach. First, you could use any of the options for fitness, crossover, and mutation listed earlier. Moreover, the very way that the steps are sequenced is sometimes different from shown above.

For instance, note that one common way to implement the GA is to use selection to choose two individuals, then when the two parents are used in mating, they are allowed to form *two* children via crossover, and these children are both subjected to mutation and kept in the next generation (if the population size $S$ is odd, then one child is randomly removed). In this way, the genetic material of the two parents is not lost. Above, when a parent mates, it produces one child who will, in general, only have part of each parent's genetic material while the other part is lost (this is what generally happens in biological systems).

Next, note that with the above approach, there are two ways that we could end up with identical individuals in the mating pool at any iteration:

1. Due to the way that fitness-proportionate selection works, the same individual (e.g., the most fit one) can have more than one copy of itself in the mating pool.

2. Two individuals could have different ancestors, but just happen to end up with the same genetic makeup due to crossover and mutation operations (e.g., by random chance two individuals' genetic makeup could be the same).

With normal choices for representations and parameters, case 1 would be much more common than case 2. Using a biological system analogy, it seems satisfactory to mate two individuals with different ancestry (ignoring that in most species it takes a male and female to mate); however, if they were simply due to the multiple copies of one individual getting in the mating pool due to selection, it seems inappropriate. In either case, note that if the individual happens to mate with itself, then crossover has no effect, although mutation still does. Some researchers like to avoid this situation altogether and one way to do this, is to allow only different individuals to mate (and if they are all the same, then simply terminate). In this case, the above pseudocode would have to be modified to represent the addition of the operations to ensure no identical individuals mate. Note that if you use the approach discussed above where you do not generate the whole mating pool at one time, and just two parents, you could generate one parent, then generate the other one with selection by repeating the process until a different individual is generated. Of course, in this case that would not be a pure fitness proportionate selection approach.

Finally, note that you may want to choose the method based on your understanding of the manner in which evolution occurs in biological systems in nature. However, we would emphasize that any of the variations described here are likely only very rough approximations of what is actually happening in nature; hence, in this book we will focus more on the view of the GA as a stochastic optimization method. If one way of defining and sequencing operations works better for some application, we will accept it whether or not it models some biological system (i.e., ours is an engineering focus, not one where we are focusing on the science of computer modeling of biological systems).

### 14.4.3 Representations, Complexity, Termination, and Initialization

For programming the genetic algorithm, one issue that you will encounter is whether to use special "string operations" in a computer language (some have better features than others in this regard). Such operations allow, for instance, conversion of numbers to strings and strings to numbers (which, depending on how you code the GA, you may need, since we need the numbers in string format to cross over and mutate, and in numeric format for fitness evaluation to perform selection, and of course, to plot certain results). They may allow for concatenation, swapping, and other features that could be useful in a string-based approach to implementing genetic algorithms. Another approach to implementing the genetic algorithm is to only use standard numeric representations but ones that allow for us to perform crossover and mutation (in this case, often integer representations are used).

Next, in choosing which number system to use (e.g., base-2 or base-10), be careful in considering that the parameters will have to be encoded to, and decoded from, this representation. In some computer languages, there are special functions that perform these functions and these can be quite useful in programming the genetic algorithm.

Finally, note that there is often a need to include parameter constraints and one way to do this is to use the "projection method" used for the gradient methods.

In implementation, you are typically concerned with memory and computation time and there are certain choices that can affect these significantly. First, due to the structure of the algorithm, it is clear that increasing the size of the population $S$ will increase both memory and computation time requirements. In online applications, you typically only execute a fixed number (often one) of iterations of the genetic algorithm (an iteration in a genetic algorithm is the act of producing the next generation from the current one using the genetic operations) per sampling period. Clearly, just as in the case of the gradient methods, if you are solving, for instance, a function approximation problem, then you need to carefully consider issues involved in how big a batch of data to process at each step, and how many iterations to perform per sampling period because these can significantly affect memory and computation time requirements. Finally, the choice of the parameters of the genetic algorithm can significantly affect memory and computation time requirements, simply by how they affect performance of the algorithm and hence, how fast it finds a solution.

The discussion in the previous section showed how to produce successive generations and thereby simulate evolution. While the biological evolutionary process continues, perhaps indefinitely, there are many times when we would like to terminate our artificial one and find the following:

- The individual of the population—say, $\theta^\star(k)$—that best maximizes the fitness function (note that we do not use the notation $\theta^*$ as we reserve this for a global maximum point if it exists (they exist)). Notice that to determine this, we also need to know the generation number $k$ where the most fit individual existed (it is not necessarily in the last generation). You may want to design the computer code that implements the genetic algorithm to always keep track of the highest $\bar{J}$ value, and the generation number and individual that achieved this value of $\bar{J}$.
- The value of the fitness function $\bar{J}(\theta^\star(k))$. While for some applications this value may not be important, for others it may be useful (e.g., in many function optimization problems).
- The average of the fitness values in the population.
- Information about the way that the population has evolved, which areas of the search space were visited, and how the fitness function has evolved over time. You may want to design the code that implements the genetic algorithm to provide plots or printouts of all the relevant genetic algorithm data.

There is then the question of how to terminate the genetic algorithm. There are many ways to terminate a genetic algorithm, many of them similar to termination conditions used for conventional (gradient) optimization algorithms. To introduce a few of these, let $\epsilon > 0$ be a small number and $M_1 > 0$ and $M_2 > 0$ be integers. Consider the following options for terminating the genetic algorithm:

- Stop the algorithm after generating generation $P(M_1)$—that is, after $M_1$ generations.
- Stop the algorithm after at least $M_1$ generations have occurred and at least $M_2$ steps have occurred where the maximum (or average) value of $\bar{J}$ for all population members has increased by no more than $\epsilon$.
- Stop the algorithm once $\bar{J}$ takes on a value above some fixed value.

Of course, there are other possibilities for termination conditions (see the discussion in Section 11.1.6). The above ones are easy to implement on a computer but sometimes you may want to watch the parameters evolve and decide yourself when to manually stop the algorithm.

By "initialization" of the genetic algorithm, we mean, for instance, how to select the initial population. (Of course, to start a genetic algorithm, you need to specify other parameters.) Sometimes, the initial population is simply set to be random values. Other times, domain-specific information can be useful in establishing the initial population. Similar to the gradient methods, we generally expect better algorithm performance if we start with a better initialization. You should note, however, that unlike gradient methods, we get to initialize the parameters to $S$ *different* values if we want. So we can think of the initial population as a set of best guesses at the solution. If even one of these is close to the global maximum of $\bar{J}$, then it is possible that the performance of the genetic algorithm will be improved.

## 14.5 Example: Solving an Optimization Problem

In engineering design problems, there are many times when it is useful to solve some sort of optimization problem, since we often try to produce the "best" designs within a wide range of constraints (which include, e.g., cost). In practical engineering problems, such optimization problems can be very difficult and at times it can be useful to turn to the genetic algorithm. In this section, to illustrate the operation of the genetic algorithm, how to tune its parameters, and how to program the genetic algorithm, we study its application to a relatively simple optimization problem.

Suppose, in particular, that we want to find the maximum of the function shown in Figure 18.10 using a genetic algorithm. Such a surface is sometimes called a "fitness landscape" by analogy with mountain climbing. Notice that it

has many hills and valleys that could confuse, for instance, a gradient optimization algorithm. We will act as though we do not know an analytical expression for the underlying function. We assume, however, that we can provide candidate solutions to the function, and it will return (in finite time) the fitness of these candidate individuals. This is a necessary feature for implementing any genetic algorithm.

### 14.5.1 Genetic Algorithm Design

The genetic algorithm used to solve this problem was coded in Matlab using a base-10 encoding. We use two digits before the decimal point and four after it for a total number of six digits. (Clearly, this constrains the accuracy that we can achieve in the solution to the optimization problem.)

We will either initialize with a random population (with values uniformly distributed on the known optimization variable domains) or with all the parameters initially at zero. We assume we know the size of the domain that we want to optimize over (it is $[0, 30]$ for each dimension) and use "projection" to keep the parameters in this range.

Note that since the function goes below zero in Figure 18.10, we will shift the whole plot up by a constant (a value of 5 in this case). This will not change where the extrema occur on the landscape but it will shift the fitness values computed and displayed. Why perform this "shift"? For our selection method, we require that we have all positive fitness values since a negative one can result in a negative probability of being placed in the mating pool.

We use fitness-proportionate selection, single-point crossover, and to pair off individuals for mating, we pick each one in the mating pool and randomly select a mate for it. We use gene mutation. We will explore the use of different values for the population size $S$, the crossover probability $p_c$, and mutation probability $p_m$. We will also study the effects of using elitism, with a single elite member.

For a termination criterion, we allow no more than a fixed maximum number of iterations (here, $M_1 = 1000$). However, we also add another termination criterion that may stop the algorithm before this maximum number of iterations is achieved. In particular, we terminate the program if the best fitness in the population has not changed more than $\epsilon = 0.01$ over the last $M_2 = 100$ generations.

### 14.5.2 Algorithm Performance and Tuning

In this section, we run the genetic algorithm program under a variety of conditions to provide insights into its operation, and to provide ideas on how to tune a genetic algorithm's parameters.

#### Random Initial Population

To illustrate the operation of the genetic algorithm, we begin with $p_c = 0.8$ and $p_m = 0.05$ and a population size of $S = 20$. A random initial population

is chosen, so that each parameter is uniformly distributed on $[0, 30]$. To better view the results of the optimization process, we will plot the contour map of the function in Figure 18.10 and place points on this plot that represent individuals *at some iteration.* Figures 14.3 and 14.4 illustrate the operation of the genetic algorithm. We see that for these choices, the algorithm performs well, and it does find the best individual, but then later loses it. Note that if you run the algorithm again, it may not do as well, since it may be unlucky in its random initial choices. (This shows why you may want a big population size; if it is big, it is more likely that it will make at least one good initial choice.) The scatter pattern shown in Figure 14.3, where there are horizontal/vertical groupings (bands), is the result of our genetic operator choices (e.g., the group of points above the global maximum point results from crossover and gene mutation in *one* dimension).

*Initialization can significantly affect genetic algorithm performance.*

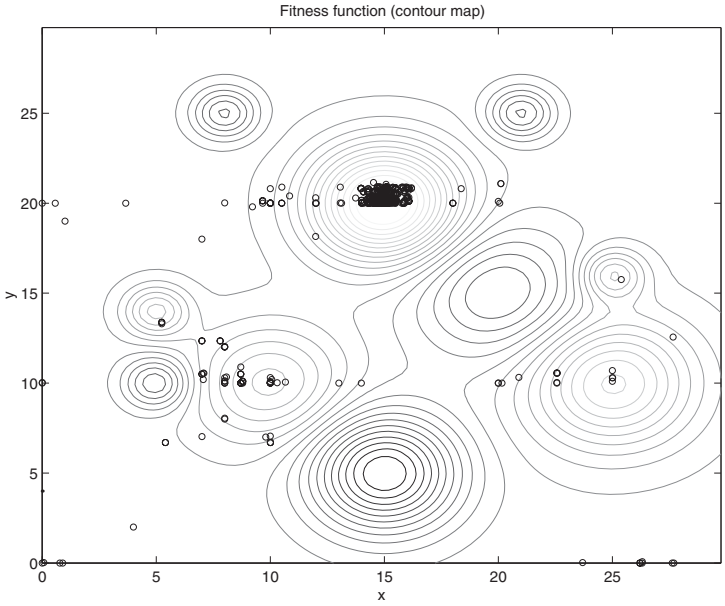


Figure 14.3: Contour plot of surface in Figure 18.10.

### Initial Population of all Zeros

Here, with all the other parameters the same, we choose an initial population with all zeros for the parameter values. With this poor initialization, it fails to find the optimum point (see Figure 14.5) by the time the algorithm terminates. If you choose a different termination criterion that allows the algorithm to evolve more generations, it may find a good solution. Also, note that if you run the algorithm again, it may be the case that it will succeed, since the algorithm may make some lucky mutations or crossovers that result in better guesses.

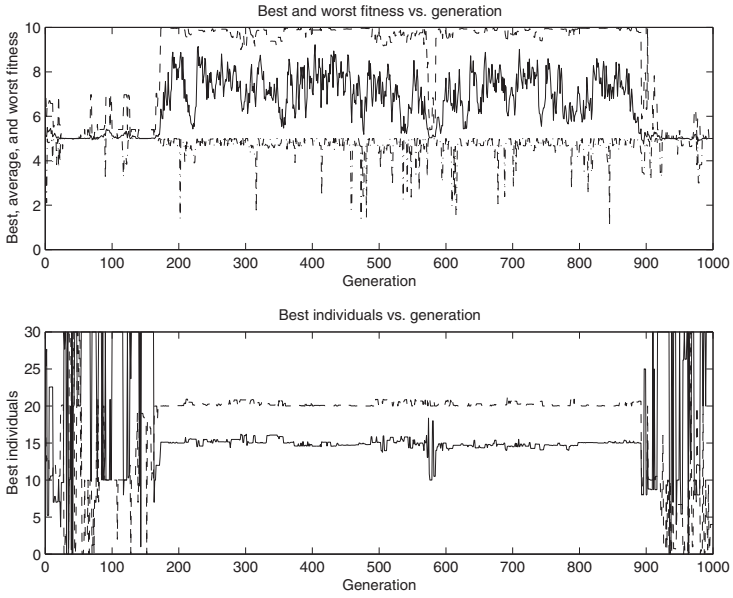


Figure 14.4: Fitness and optimization parameter evolution.

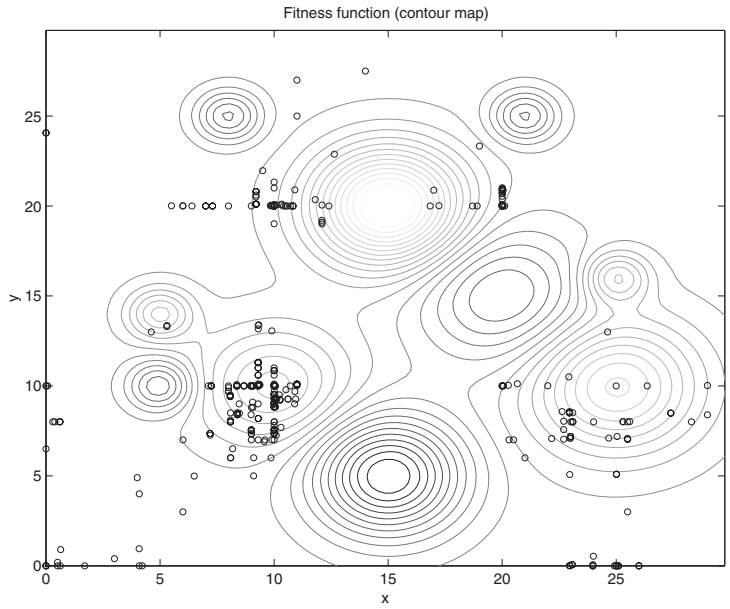


Figure 14.5: Contour plot of surface in Figure 18.10.

### Increased Mutation Probability

*Large mutation probabilities can lead to random and exhaustive search.*

Next, let $p_c = 0.8$ and $p_m = 0.1$ (a larger value than above) and consider a population size of $S = 20$. As you can see in Figures 14.6 and 14.7, with the higher value for the mutation probability, it fails to lock on to the global maximum point. Basically, this happens since mutation is destroying good solutions (i.e., it destroys the progress of the method). From this, it should be clear that if you pick the mutation probability too high, the algorithm executes what can be considered a "random walk" in the parameter space so, while it may find a good solution at some point, it may take a long time to do so and we would basically attribute its success to "dumb luck."

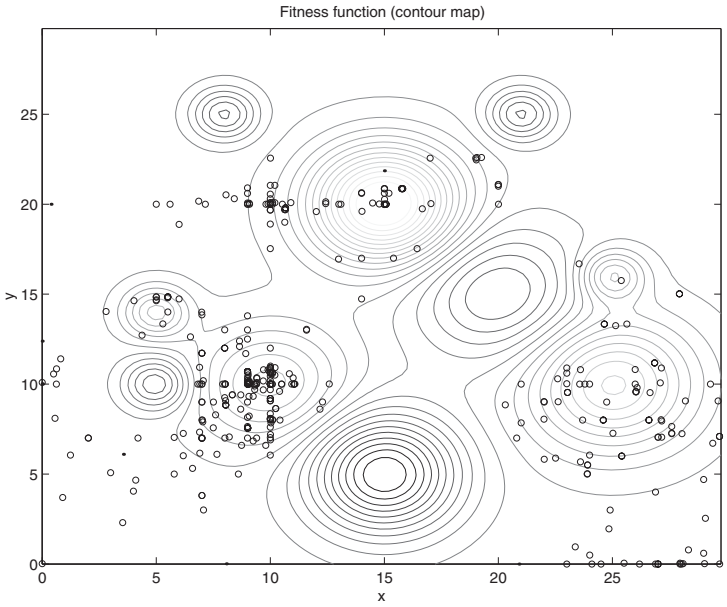


Figure 14.6: Contour plot of surface in Figure 18.10 with random initial population and higher mutation probability.

### Decreased Crossover Probability

Next, let $p_c = 0.5$ (a smaller value than above) and $p_m = 0.05$ (i.e., return it to its earlier value) and consider a population size of $S = 20$. With the lower value for the crossover probability, it does less local search between good solutions (see Figures 14.8 and 14.9). Note that if you make $p_c = 0.1$, it fails to find a local optimum (at least for one time the algorithm was run). In this case, it is passing too many individuals through the mating process without mixing genetic material; hence, it stagnates.

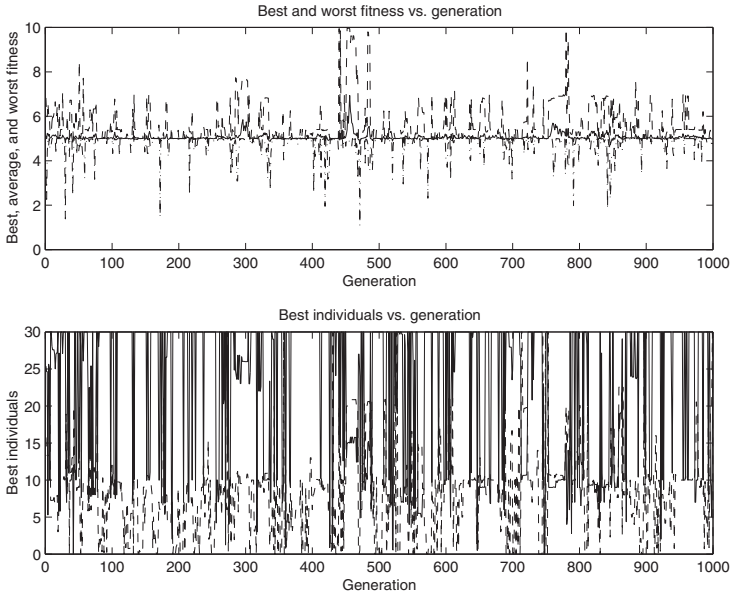


Figure 14.7: Fitness and optimization parameter evolution, with random initial population and higher mutation probability.

### Increased Population Size

Next, let $p_c = 0.8$ and $p_m = 0.05$ and consider a population size of $S = 40$ (i.e., twice as big as earlier). With a bigger population size, we still get convergence, but this shows that increasing its size is not necessarily good (see Figures 14.10 and 14.11). Of course, we have to qualify this statement by saying "for this run of the program, with these termination criteria, etc." This simulation was produced simply to make the point that bigger is not always better (even though for the population, for some applications, this may generally be true).

### Effects of Elitism

Next, let $p_c = 0.8$ and $p_m = 0.05$ and a population size of $S = 20$. Now, however, we use elitism with a single elite member. See Figures 14.12 and 14.13. With elitism we get much quicker convergence (notice that the early termination criterion was invoked) since crossover and mutation do not alter the best individual. Elitism has, in fact, been found to provide qualitatively similar results for a variety of applications. Basically, elitism ensures that there is a highly fit individual that survives in each generation. The other individuals are allowed to mate with the elite individual, so less fit individuals that do mate with this very fit individual will tend to have more fit children. This tends to accelerate convergence, while avoiding "premature convergence," since all the other individuals are allowed to explore the search space (provided that the other pa-

*In practice, elitism has often been found to be useful, especially in real-time control where you cannot afford to use anything but the best-known controller.*

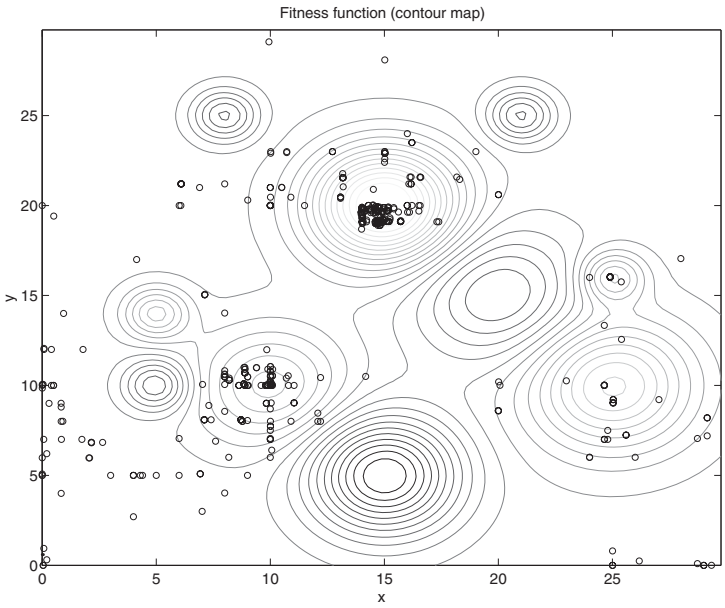


Figure 14.8: Contour plot of surface in Figure 18.10 with random initial population and lower crossover probability.

rameters, particularly the mutation rate, are set properly). It tends, for some applications, to provide a nice trade-off between focusing and exploration.

## 14.6 Approximations to Reduce Algorithm Complexity

When you start programming genetic algorithms, you often get ideas on how to modify the algorithm, either to better model how evolution works in nature, or to improve the performance of the algorithm (which may result in an algorithm that is successful, but quite unlike anything in nature). In this section, we briefly study some ways to make approximations to the genetic operations so that computational complexity of the algorithm can be reduced.

### 14.6.1 Reducing Algorithm Complexity

First, note that the encoding and decoding, even with a base-10 encoding, causes extra computations because you must convert the base-10 numbers to strings of integers and back. The only reason that we needed to do this was because we needed to perform crossover and mutation. We can, however, approximate these two operations and get reductions in complexity simply because no conversions will be necessary. Moreover, when we remove the conversions to strings, we get

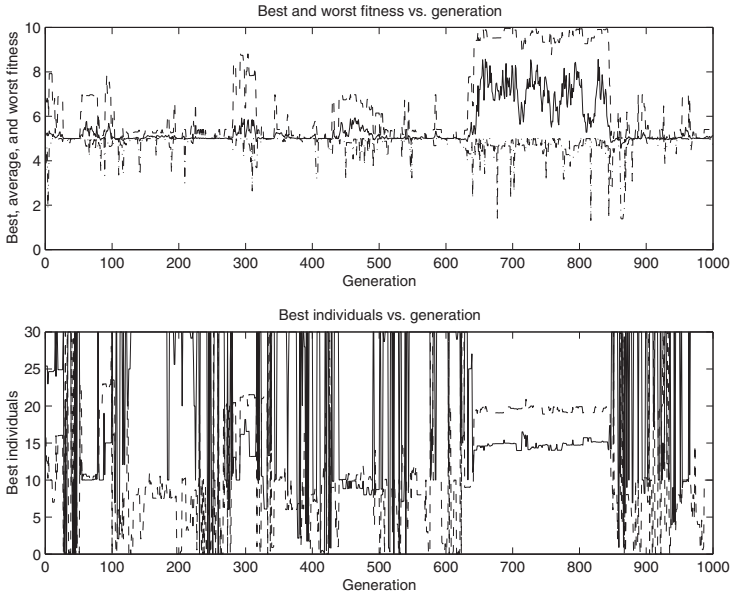


Figure 14.9: Fitness and optimization parameter evolution, with random initial population and lower crossover probability.

all standard operations on vectors so we will then be simply encoding traits with the natural representation in the computer (e.g., in Matlab, a chromosome will be a vector with elements that are traits, and the traits are simply the parameters of the problem).

*There are simple ways to modify the genetic algorithm so that it still grossly emulates evolution, but computational complexity is reduced.*

For mutation we will perform a simple "trait mutation" where, with probability $p_m$, we mutate each trait in each individual. If we mutate, we simply switch the trait to be any number on the known domain of that trait (e.g., for the optimization problem in the last section, between 0 and 30). With this approach we will know that mutation cannot generate an out-of-range value so we will not need to use "projection" to fix it. For other problems, you could consider simply adding on a random value to the trait, but then this may place the value out of range, but it can be fixed with projection.

Next, for crossover there are many possibilities. Here, we will consider two and to illustrate how they work, we will apply them to the function optimization problem studied in the last section. In both cases, we use $p_c = 0.8$ and $p_m = 0.05$ and a population size of $S = 20$. A random initial population is chosen, one so that each parameter is uniformly distributed on $[0, 30]$.

### 14.6.2 Crossover Option 1

First, we generate a method that approximates what happens when we cross over strings. In particular, consider an approach where we cross over at a random

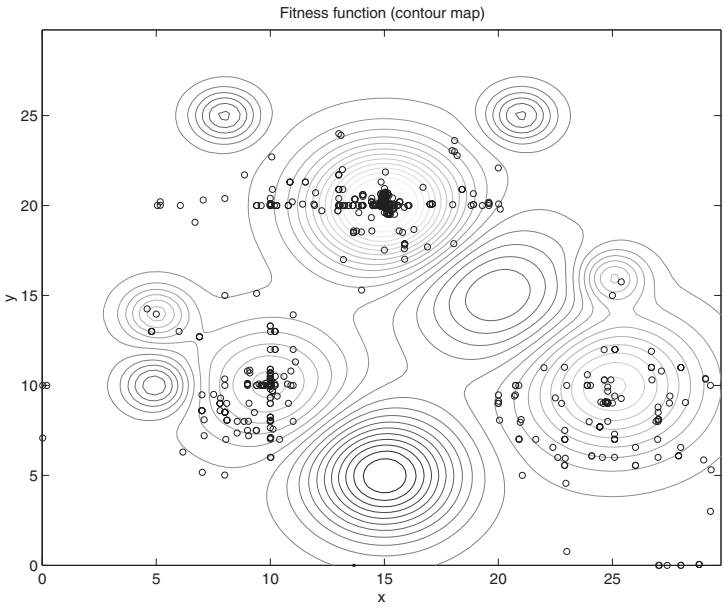


Figure 14.10: Contour plot of surface in Figure 18.10 with random initial population and increased population size.

"trait site." We use the standard crossover probability $p_c$ and suppose that we choose to cross over the vectors $\theta^i$ and $\theta^j$ where $i \neq j$ and $i, j \in \{1, 2, \ldots, S\}$. To do this, we generate a random trait site number, say $i^*$, where $1 \leq i^* \leq p$ (where $p$ is the number of traits, i.e., parameters), a random number $\alpha \in (0, 1)$, and let the child $\theta$ have

$$\theta_m = \theta_m^i$$

for $m = 1, 2, \ldots, i^* - 1$,

$$\theta_{i^*} = \alpha\theta_{i^*}^i + (1 - \alpha)\theta_{i^*}^j$$

(we think of this as "splitting" the trait at the trait site split point) and

$$\theta_m = \theta_m^j$$

for $m = i^*, \ldots, p$. When this approach is used, we get the results shown in Figures 14.14 and 14.15. Notice that the approach finds the maximum point.

### 14.6.3 Crossover Option 2

Next, we do everything the same as in "crossover option 1" except we perform crossover differently. Suppose we choose to cross over the vectors $\theta^i$ and $\theta^j$ where $i \neq j$ and $i, j \in \{1, 2, \ldots, S\}$. To do this, we generate a random number $\alpha \in (0, 1)$, and let the child $\theta$ be

$$\theta = \alpha\theta^i + (1 - \alpha)\theta^j$$

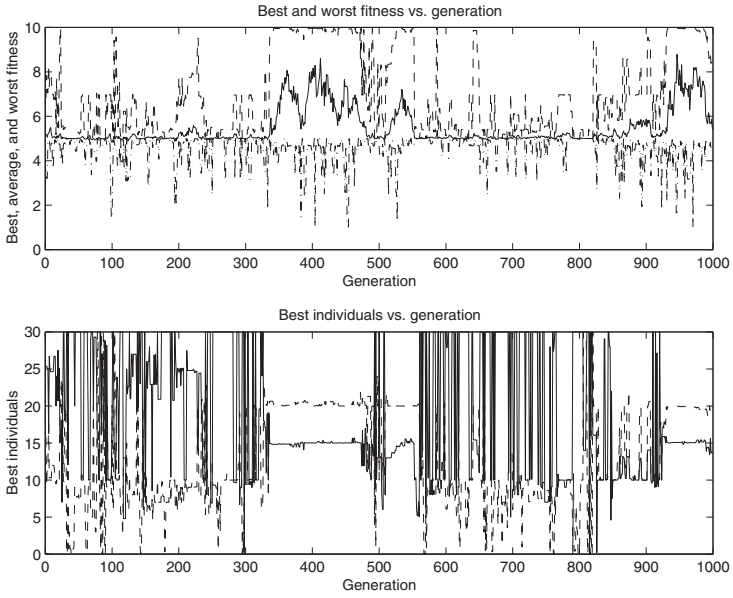


Figure 14.11: Fitness and optimization parameter evolution, with random initial population and increased population size.

This $\theta$ is a point on the line joining $\theta^i$ and $\theta^j$. It is a simple type of interpolation between $\theta^i$ and $\theta^j$. We think of this as "search in the neighborhood" of the two individuals, which is effectively what crossover tries to do. When this approach is used, we get the results shown in Figures 14.16 and 14.17. Notice that the approach finds the maximum point.

## 14.7 Exercises and Design Problems

**Exercise 14.1 (Genetic Algorithms for Optimization):** In this problem you will use the genetic algorithm to solve some simple optimization problems. You may use the code that is given at the Web site listed in the Preface.

(a) Suppose that you are given the function

$$f(x) = x \sin(10\pi x) + 1$$

which is taken from [352]. Design and implement on a computer a genetic algorithm for finding the maximum of this function over the range $x \in [-0.5, 1]$. Plot the best individual, best fitness, and average fitness against the generation. Plot the function to verify the results.

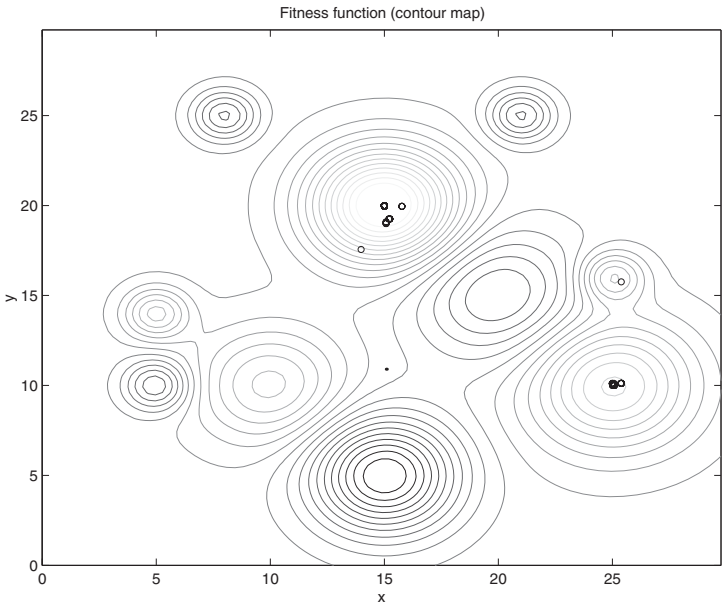


Figure 14.12: Contour plot of surface in Figure 18.10 with random initial population and elitism.

(b) Suppose that you are given the function

$$f(x) = \text{sinc}(x+2) = \frac{\sin(\pi(x+2))}{\pi(x+2)}$$

Design and implement on a computer a genetic algorithm for finding the maximum of this function over the range $x \in [-10, 10]$. Plot the best individual, best fitness, and average fitness against the generation. Plot the function to verify the results.

(c) Suppose that you are given the function

$$\begin{aligned} z &= 0.8x\exp\left(-x^2-(y+1.3)^2\right) + x\exp\left(-x^2-(y-1)^2\right) \\ &\quad +1.15x\exp\left(-x^2-(y+3.25)^2\right) \end{aligned}$$

Design and implement on a computer a genetic algorithm for finding the maximum of this function over the range $x \in [-5, 2]$, $y \in [-2, 2]$. Plot the best individual, best fitness, and average fitness against the generation. Plot the function to verify the results.

(d) Suppose that you are given the function

$$z = 1.5\text{sinc}(x) + 2\text{sinc}(y) + 3\text{sinc}(x+8) + \text{sinc}(y+8) + 2$$

Design and implement on a computer a genetic algorithm for finding the maximum of this function over the range $x \in [-12, 12]$, $y \in$

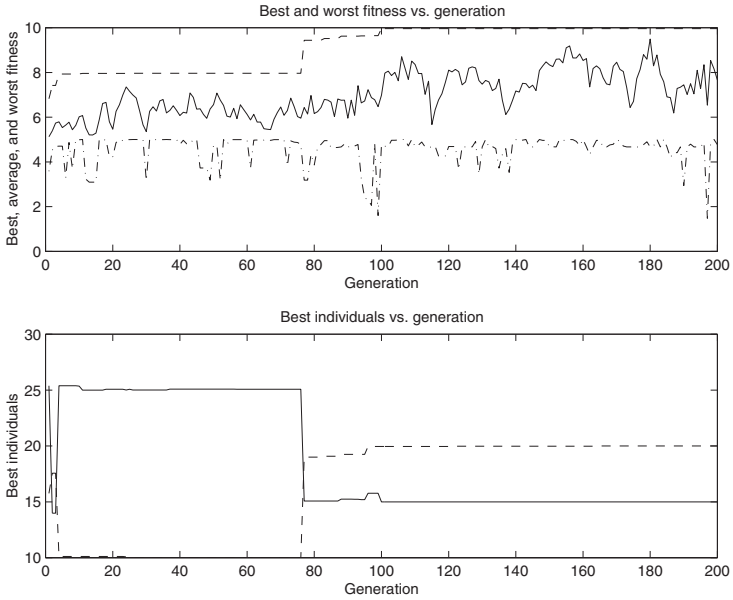


Figure 14.13: Fitness and optimization parameter evolution, with random initial population and elitism.

$[-12, 12]$. Plot the best individual, best fitness, and average fitness against the generation. Plot the function to verify the results.

**Exercise 14.2 (Approximations of Genetic Algorithms):**

(a) Repeat Exercise 14.1(c), but where you use one of the approximations discussed in Section 14.6. Code the algorithm and evaluate its performance (both complexity and ability to find the global minimum) in simulation.

(b) Repeat (a) but for Exercise 14.1(d).

**Design Problem 14.1 (Design of Genetic Operators for Genetic Algorithms):**

(a) Repeat Exercise 14.1(c), but where you use a genetic algorithm with different genetic operators. You choose which operators to use, but make the selection, crossover, and mutation operators different from those coded into the program given at the Web site. Code the algorithm and evaluate its performance (both complexity and ability to find the global minimum) in simulation.

(b) Repeat (c) but for Exercise 14.1(d).

**Design Problem 14.2 (Optimization for Finding Mountain Peaks and Coffee-Growing Regions in Topographical Data for Colombia):**

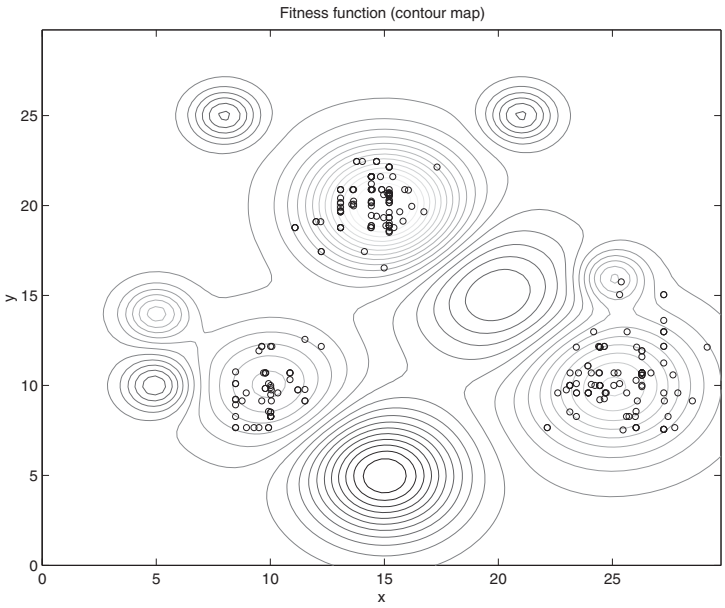


Figure 14.14: Contour plot of surface in Figure 18.10 with approximations to genetic algorithm, crossover option 1.

In this problem, you will study how to use genetic algorithms to search for the highest mountain peak in a region of the earth. To do this, you will need to go to the Web site for the book and download a topographical data set and a program that shows you how to work with the data (the topographical data were obtained from the US Dept. of Commerce, National Inst. of Geophysical Data). The topographical map for the region around Colombia is given in Figure 14.18. Notice that the data includes underwater data as negative elevations, and a black line was added at zero elevation to show roughly where the shorelines are with the Carribean Sea and Pacific Ocean.

After you download the data set and associated program, study the code to understand how to work with the data, plot it, and how to interpolate the data so that you can estimate the elevation for points that are not given in the data set. Clearly, you do not have analytical gradient information for this problem; however, you could go to the library or world atlas and find the solution to the problem for any fixed region on the earth. Hence, you should simply view the topographical data as providing an interesting cost function to search over.

(a) Design a genetic algorithm and simulate its operation on the topographical map of Colombia. As in the chapter, study choices of the population size and other genetic algorithm parameters.

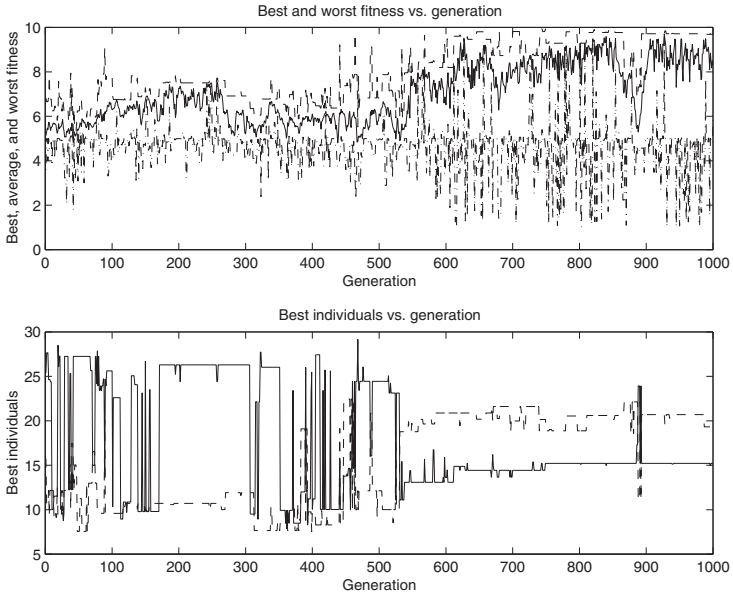


Figure 14.15: Fitness and optimization parameter evolution, with approximations to genetic algorithm, crossover option 1.

(b) What is the highest point on the map? Does this correspond to what a world atlas (or other source) tells you about the highest mountain in this region? What is the name of that mountain?

(c) In Colombia, coffee grows best at altitudes between 1000 and 2100 meters. Define a cost function that indicates where it is best to grow coffee on the region defined by the topographical map of Colombia. Formulate and solve a problem that evolves where coffee growers should live in Colombia (assuming they live near their farm). Illustrate the performance of the algorithm. Do the points where the population evolves to correspond to where coffee is actually grown in Colombia (e.g., *"la zona cafetera"*)?

**Design Problem 14.3 (Genetic Algorithms for Approximator Structure Construction)$^\star$:** Read Design Problem 11.2, where we give ideas on how to construct the structure of an approximator using gradient-type algorithms. In this problem you want to develop a genetic algorithm that can evolve the structure of an approximator for a particular function approximation problem.

(a) First, you must conduct some background research. Search the literature, evaluate existing methods, and summarize these.

(b) Using ideas from the literature, and perhaps your own ideas, design a genetic algorithm that can construct the structure of a function

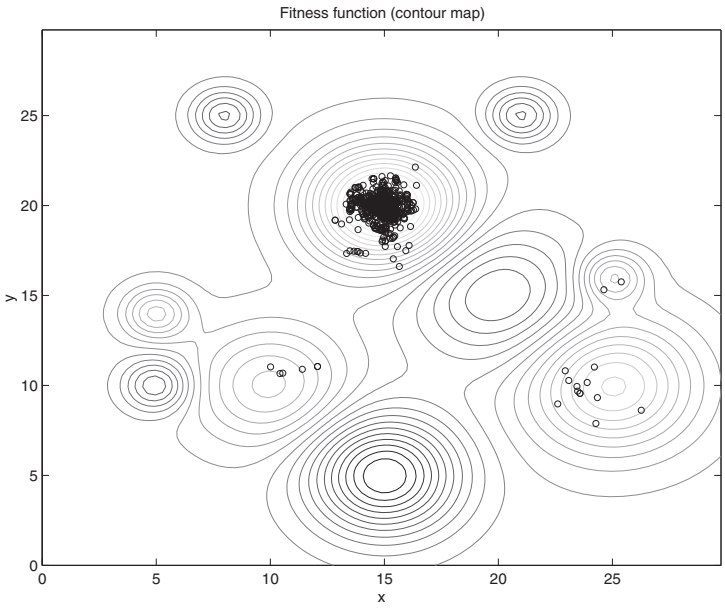


Figure 14.16: Contour plot of surface in Figure 18.10 with approximations to genetic algorithm, crossover option 2.

approximator. You must decide whether to use a genetic algorithm in conjunction with a least squares or gradient method, and all the issues associated with representation of the approximator in the genetic algorithm. Test the performance of the algorithm. Hint: Use the function approximation problem that was used throughout Part III and a fitness function that quantifies the inverse of the approximation error as measured by some test set.

**Design Problem 14.4 (Artificial Immune Systems and Evolutionary Algorithms)$^\star$:** First, see the discussion in the "For Further Study" section of this part. Second, study [125, 124] on artificial immune systems.

(a) Choose an artificial immune system and simulate it. Choose one that provides the capability for either learning or optimization, or both.

(b) Explain in detail the relationships between the algorithm you implement in (a) and the standard genetic algorithm. Be sure to identify the fitness function, selection, crossover, and mutation analogies.

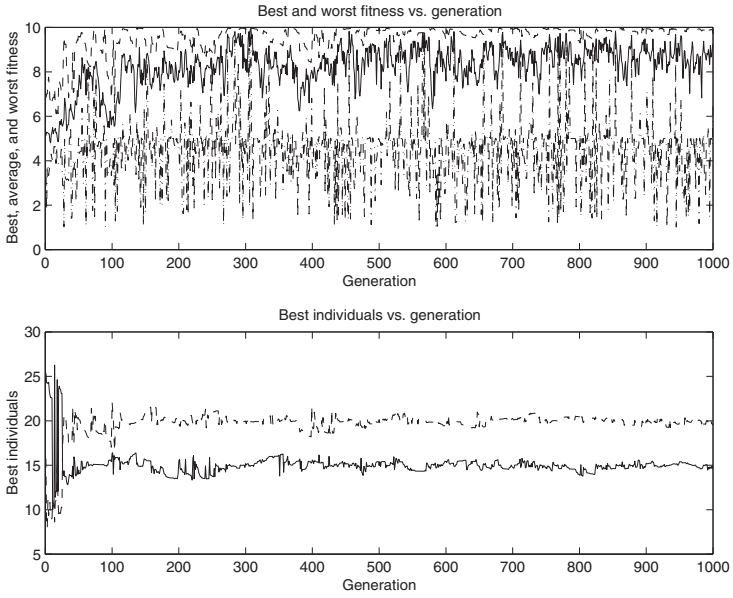


Figure 14.17: Fitness and optimization parameter evolution, with approximations to genetic algorithm, crossover option 2.

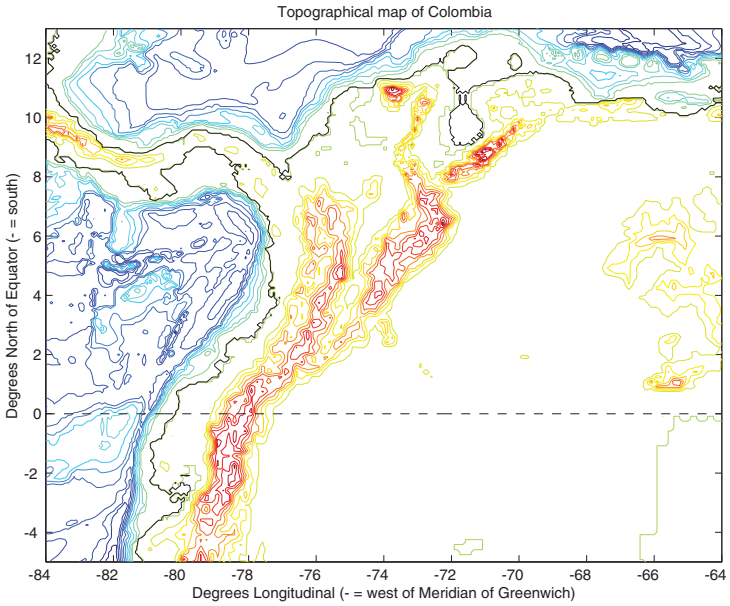


Figure 14.18: Topographical map of region around Colombia, South America.