5 Introduction to Genetic Algorithms

Sue Ellen Haupt

5.1 Motivation

The world is full of optimization problems. Nature constantly optimizes each of her configurations. Each ecosystem fits together to use the symbiotic nature of each element. Species have evolved to have the characteristics that are most likely to lead to survival. The wind blows in directions that best alleviate any imbalances in forces. The planets orbit in ways that best fulfill the laws of motion. In understanding the environment, we often have to discern the optimization problem to fully understand its solution.

Evolution is one of the most interesting optimization problems. Why have humans evolved to have two hands, two eyes, two legs, one head, and a large brain while other species have not? Does that make humanity the pinnacle of the optimization problem? Why do guppies evolve to have different characteristics in dissimilar environments? Can the process of evolution be codified to understand these issues better?

Many problems that we address in environmental science can be configured into an optimization problem. As an example, let's consider guppies evolving in an environment where they need to survive on the available food, attract mates for reproduction, and avoid predators. Figure 5.1 illustrates the pieces of a general optimization problem. We begin with input parameters that we wish to optimize. For our guppies, these variables might include attractiveness (to mates), disease tolerance, food requirements, appeal to their predators, and ability to hide from predators. The combination of the values of these variables makes the guppy an individual. There must be some method to rate the survivability of the guppy based on its specific variable values. There must be some way to balance its attractiveness to potential mates with its visibility to its predators. If the environment is harsh, it must be hardy. If the current is swift, it must have a long enough tail to swim fast. The objective, or cost, function codifies these considerations and weights them to rate the guppy survivability. The "most fit" guppies survive while those that do not meet the specifications of the objective function are destined to die off in a harsh environment or are eaten by a predator. The final piece of the optimization scheme is the optimization algorithm that finds some way of configuring new guppies so that they evolve into a viable species for their environment. The optimization algorithm typically minimizes some "cost," or equivalently, optimizes the objective.[1] Some common optimization algorithms include Newton's method for optimization, conjugate gradient, and Nelder-Mead downhill simplex method. Unfortunately, it is difficult to code guppy coloration, food requirements, and attractiveness to either mates or predators in a way to use these gradient seeking methods. More innovative methods are required.

A genetic algorithm (GA) is one such versatile optimization method. Figure 5.2 shows the optimization

Sue Ellen Haupt (✉)
Applied Research Laboratory and Meteorology Department, The Pennsylvania State University, P.O. Box 30, State College, PA 16802, USA
Phone: 814/863-7135; fax: 814/865-3287;
email: haupts2@asme.org

[1] In optimization terminology, an objective function could be either minimized or maximized. When the name cost function is applied, we always minimize. In contrast, a fitness function is maximized. Therefore, we concentrate on minimization problems. It is trivial to turn a maximization problem into one in minimization with a negative sign.

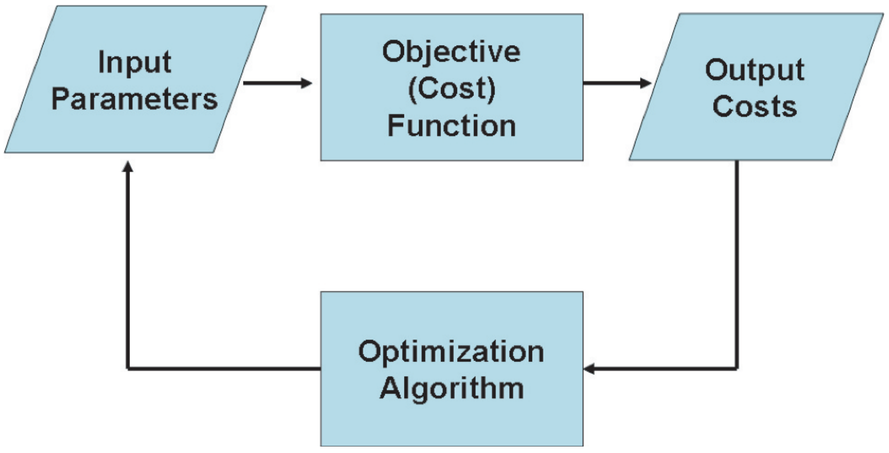


Fig. 5.1 Flowchart of the optimization process

process of a GA – the two primary operations are mating and mutation. The GA combines the best of the last generation through mating, in which parameter values are exchanged between parents to form offspring. Some of the parameters mutate. The objective function then judges the fitness of the new sets of parameters and the algorithm iterates until it converges. With these two operators, the GA is able to explore the full cost surface in order to avoid falling into local minima. At the same time, it exploits the best features of the last generation to converge to increasingly better parameter sets. GAs are remarkably robust and have been shown to solve difficult optimization problems that more traditional methods can not. Some of the advantages of GAs include:

- They are able to optimize disparate variables, whether they are inputs to analytic functions, experimental data, or numerical model output.
- They can optimize either real valued, binary variables, or integer variables.
- They can process a large number of variables.
- They can produce a list of best variables as well as the single best solution.

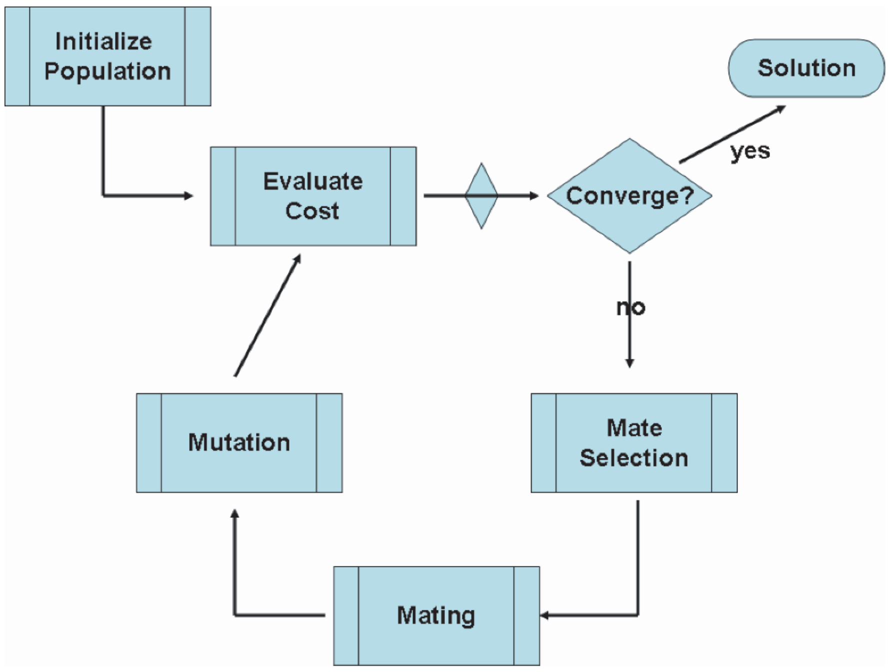


Fig. 5.2 Flowchart of optimization with a genetic algorithm

- They are good at finding a global minimum rather than local minima.
- They can simultaneously sample various portions of a cost surface.
- They are easily adapted to parallel computation.

Some disadvantages are the lack of viable convergence proofs and the fact that they are not known for their speed. As seen later in this chapter, speed can be gained by careful choice of GA parameters. Although mathematicians are concerned with convergence, often scientists and engineers are more interested in using a tool to find a better solution than obtained by other means. The GA is such a tool.

5.2 Genetic Algorithm Overview and Guppy Evolution

A genetic algorithm combines the concepts of genetics and evolution into an algorithm to optimize a function or to search a solution space. GAs were first introduced by John Holland (1975) at the University of Michigan (UM). David Goldberg (1989) popularized GAs beginning with his Ph.D. dissertation at UM where he used it to optimize a gasline piping problem, a problem that was quite difficult to solve via conventional means. De Jong (1975) demonstrated the utility of GAs for function optimization and studied how to best choose GA parameters. GAs have become a popular tool in the engineering literature but have thus far found fewer applications in the environmental sciences. More discussion of applications can be found in Chapters 14 and 18.

GAs can be configured as either binary or real valued. The GA literature began with the binary version, so we will start there too. Figure 5.2 depicts the optimization process for the GA. The basic process is the same as in Fig. 5.1, but now the GA operations of mating and mutation are specified.

The basic building block of a genetic algorithm is a gene, which represents each problem variable. For our guppy problem, the genes encode each characteristic of the guppy that we wish to consider. Table 5.1 shows the encoding of eight variables relevant to guppy evolution into eight genes. Each variable is encoded into a 2 bit gene, meaning that each variable can have up to four separate realizations. For instance, there may be four gradations of attractiveness to mates, in this case coded as:

$$\begin{aligned}11 &= \text{drop-dead gorgeous}\\ 10 &= \text{very handsome}\\ 01 &= \text{passable}\\ 00 &= \text{only if females are desperate}\end{aligned}$$

The genes are then concatenated to form a chromosome. The guppy encoded in Table 5.1 is then represented by a chromosome: 1001010001001100. The genetic algorithm is begun by creating a population of chromosomes using a random number generator. Since many computer languages generate random numbers between 0 and 1, for the binary GA the random number is simply rounded. We generate eight randomly configured guppies, each with eight 2 bit genes, and our population looks like:

$$pop = \begin{bmatrix} 1001010001001100\\ 0110001001001110\\ 0000111010101110\\ 0101001110010111\\ 0011000111001010\\ 1100011010001101\\ 0001100110100111\\ 0110011001110010 \end{bmatrix} \tag{5.1}$$

In this matrix, each row represents a chromosome, or complete guppy configuration.

The fitness of each member of the population is then evaluated via the cost function. For the guppy example, the cost function depends on the environment.

Table 5.1 Example guppy encoded into binary genes

Attractiveness (to other guppies)	Tail	Attractiveness (to predator)	Dappling (blending into environment)	Temperature tolerance	Disease tolerance	Food requirements (amount)	Feeding requirements (time between)
Very handsome 10	Short 01	Tasty 01	None 00	Little 01	Hardy 00	Bottomless pit 11	Frequent feeder 00

Each gene of the guppy is first assigned an adaptation value. For instance, the attractiveness gene, which determines the guppy's likelihood to attract a mate, is assigned adaptation values denoted in the MATLAB code below:

```
attractive = x(:,1:2); %grabs the first two bits which
   form the first gene
%likeliness to mate
%attractiveness (brightness positive)
if attractive(ind,:)==[1 1]
adapt(ind,1)=2.0; %drop dead gorgeous
elseif attractive(ind,:)==[1 0]
adapt(ind,1)=1.5; %very handsome
elseif attractive(ind,:)==[0 1]
adapt(ind,1)=1.0; %passable
else %[0 0]
adapt(ind,1)=0.5; %if the female guppies are
   desperate
end
```

In this case, the more attractive the guppy, the higher adaptation value is assigned. The second aspect of the guppy cost function weights the importance of each gene for survivability in a particular environment. Each adaptation value is weighted according to how likely that characteristic will result in the guppy (1) mating or (2) being eaten by a predator. Specifically, for a bright, well lighted pool with lots of predators, weights are assigned as:

```
%habitat 1
wts(1,:)=[1.0, 0.8, 0.0, 0.7, 0.0, 0.0, 0.0, 0.0];
   %probability of mating
wts(2,:)=[0.0, 0.0, 0.5, 0.8, 0.6, 0.2, 0.9, 0.4]
   %probability of getting eaten
```

The final step of judging the adaptability of each guppy is writing a cost function that multiplies the adaptability values for the guppy by the environment weights and weighting the importance of mating versus getting eaten for this habitat:

```
f = – ( (wts(1,:)*adapt')' + 3*(wts(2,:)*adapt')' );
```

Note that a negative sign is applied to the cost function since the GA routine is configured to look for minima. Each member of the guppy population is judged via the cost function and the costs assigned as:

$$\text{Cost}\begin{bmatrix}1001010001001100\\0110001001001110\\0000111010101110\\0101001110010111\\0011000111001010\\1100011010001101\\0001100110100111\\0110011001110010\end{bmatrix}=\begin{bmatrix}-5.4\\-4.7\\-2.1\\-4.2\\-6.7\\-7.3\\-5.9\\-1.8\end{bmatrix}\tag{5.2}$$

The next step is simply sorting the costs with the smallest cost (most fit) chromosomes put at the top:

$$\text{Cost}\begin{bmatrix}1100011010001101\\0011000111001010\\0001100110100111\\1001010001001100\\0110001001001110\\0101001110010111\\0000111010101110\\0110011001110010\end{bmatrix}=\begin{bmatrix}-7.3\\-6.7\\-5.9\\-5.4\\-4.7\\-4.2\\-2.1\\-1.8\end{bmatrix}\tag{5.3}$$

Now we are ready for the natural selection to occur. In terms of the guppies, the less fit half of the population is eaten by predators or simply dies without reproducing. We are left with only the top half (most fit four) of the population matrix above.

The GA operations of mating and mutation come into play. In mating, we select two members of the population to exchange information to produce offspring. For the guppy problem, we will use tournament selection. Here, three members of the population are randomly selected and the two most fit individuals (smallest cost function values) of that tournament will then mate. The algorithmic mating procedure mimics the genetic recombination of meiosis. In meiosis, the chromosomes line up and join at a kinetochore. When the chromosomes separate, the left portion of the mother chromosome conjoins with the right portion of the father chromosome to complete the process known as crossover. In our binary chromosomes, the process is equivalent. A random kinetochore, or crossover point, is selected and the genes to the left of this point on parent 1 are concatenated with those to the right of that point on parent 2. In this case, we form two new individuals. An example of the guppy chromosomes mating is:

$$\mathbf{cost}\begin{bmatrix}\mathbf{110001101}\mathrm{0001101}\\ \mathbf{001100011}\mathrm{1001010}\end{bmatrix}=\begin{bmatrix}-7.3\\ -6.7\end{bmatrix}$$

$$\Downarrow$$

$$\mathbf{cost}\begin{bmatrix}\mathbf{110001101}\mathrm{1001010}\\ \mathbf{001100011}\mathrm{0001101}\end{bmatrix}=\begin{bmatrix}-7.4\\ -3.7\end{bmatrix}$$

We see that one of the offspring guppies is more fit than either parent and one is less fit.

The second GA operation is mutation. Before mutating, we typically apply elitism and set aside the most fit (lowest cost) individual of the entire population and do not allow it to mutate. Then we go into the matrix and randomly change the bit value of a predetermined percentage of bits. After mating and mutation, the guppy population looks like:

$$\mathrm{cost}\begin{bmatrix}\mathit{1100011010001101}\\ \mathrm{001}\mathbf{1}\mathrm{00011100}\mathbf{1}\mathrm{010}\\ \mathrm{0001100}\mathbf{1}\mathrm{10100111}\\ \mathrm{1001010001001100}\\ \mathrm{110001}\mathbf{1}\mathrm{011001010}\\ \mathrm{001100011000}\mathbf{1}\mathrm{101}\\ \mathrm{0001100110100111}\\ \mathbf{0}\mathrm{01100011100}\mathbf{1}\mathrm{010}\end{bmatrix}=\begin{bmatrix}\mathit{-7.3}\\ -6.7\\ -7.6\\ -5.4\\ -5.2\\ -3.7\\ -7.7\\ -2.1\end{bmatrix} \tag{5.4}$$

The first row (italics) is the best "elite" guppy chromosome. The next three are the other parents that remain in the population from the last generation, but have now been subject to bit changes due to mutation (bold). The last four rows are the offspring guppy chromosomes after mutation. The first iteration has completed. The process is iterated until convergence is obtained. For the guppies, convergence means a stable population with all individuals about equally adapted (i.e. the average population is fairly stable). The lowest cost individual becomes a prototype for the guppy population with some variation around it. Figure 5.3 shows the convergence for a run of the guppy problem using a population size of 16, crossover rate of 0.5, and mutation rate of 0.2. The "best" guppy is identified after about 11 generations, but it takes a bit longer for the population to stabilize. Convergence is both problem dependent and run dependent. Since the GA relies on random numbers to generate problems and perform the mating and mutation operations, each run of the GA will produce slightly different results. Often, the primary difference is how many iterations are required to produce convergence.

This example merely serves as a basic introduction to GAs. Many variations are possible, often

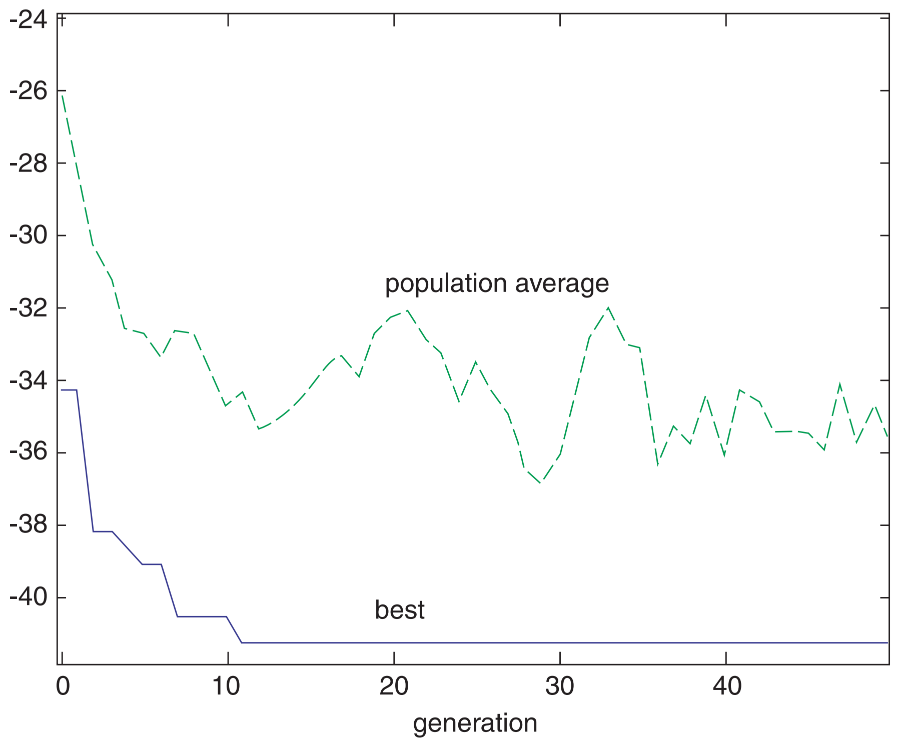


Fig. 5.3 Convergence of the GA for the guppy evolution problem

interpretable as limiting branches of the basic GA discussed here. For instance, some GAs are purely "asexual," that is, they reproduce without a mating operator, using only mutation. That scenario is equivalent to using a crossover rate of zero. Tuning the GA parameters will be discussed in Section 5.5.

5.3 Binary Genetic Algorithms and Function Optimization

We saw how GAs can be applied to a problem in evolution where the use of a GA is more "obvious." More often, we have some function we wish to optimize, subject to constraints. Here, we'll examine solving a specific function of two variables and use this example to observe the process of convergence in more detail. We wish to minimize:

$$f(x, y) = \sin(x) J_1(y) \tag{5.5}$$

where J_1 is the Bessel function of the first kind of order 1. We'll call this function, "Besin." Equation (5.5) is solved subject to the constraints:

$$0 \le x \le 10$$
$$0 \le y \le 10$$

Figure 5.4 shows plots of the equation on the given interval using both a three dimensional view and a contour plot. The exact solution, $(x, y) = (4.71, 1.81)$, is indicated on the plots.

The first step of solving our "Besin" optimization problem with a GA is initializing the population. We choose to code our binary GA with an 8 bit representation of each of the two (x, y) variables. For instance, the 16 bit chromosome with two genes representing x and y is:

$$[0\,1\,1\,1\,0\,0\,0\,0\,0\,0\,1\,0\,1\,0\,0\,0\,0] = (4.3922, 3.1373)$$

The initial population of 16 chromosomes is shown in Fig. 5.5 as asterisks on the contour plot. Costs of each (x,y) point are computed via (5.5). The minimum cost of this initial population is −0.46082 and the mean cost is 0.0096951.

The solution is evolved by the GA using a crossover rate of 0.5, population size of 16, and mutation rate of 0.2. Figure 5.6 shows the next four iterations of the GA, the sixth iteration, and the fortieth iteration. We see that the GA initially explores the entire solution space, particularly in the local minima. By the sixth iteration, the best chromosome is near the exact solution and by the fortieth iteration many of the population members are in the global solution well.

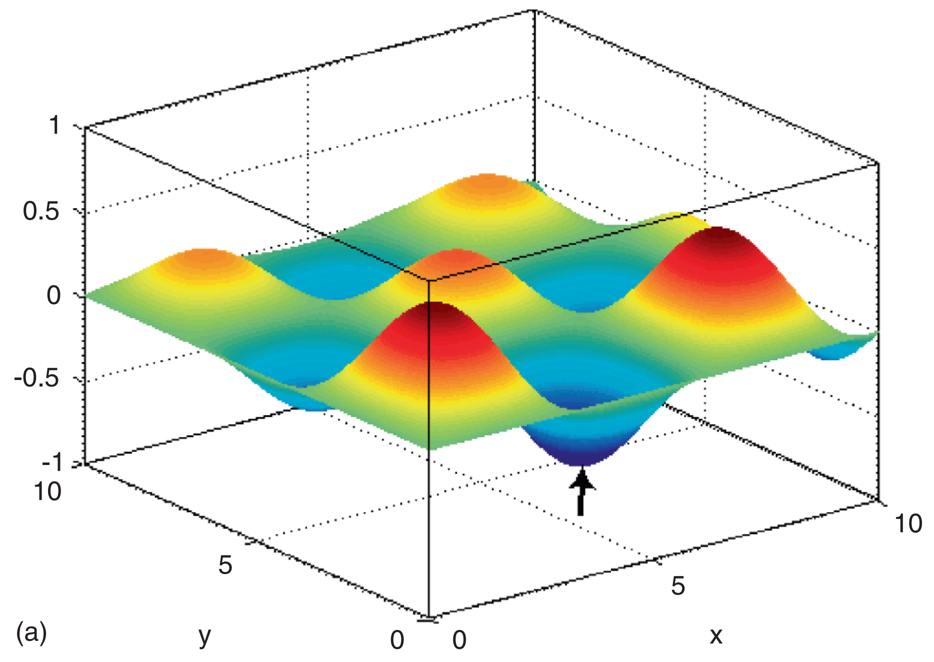
Table 5.2 Convergence of the GA solution of (5.5)

Iteration	Min cost	Mean cost	x	y
Initial	−0.46082	0.0096951	5.3333	1.5686
1	−0.46082	−0.064677	5.3333	1.5686
2	−0.46082	−0.084943	5.3333	1.5686
3	−0.50994	−0.12589	4.6275	1.2549
4	−0.50994	−0.044344	4.6275	1.2549
5	−0.57378	−0.023693	4.5490	1.8824
10	−0.58001	−0.23114	4.7843	1.8824
15	−0.58151	−0.11361	4.7059	1.8824
Exact	**−0.58186**		**4.71**	**1.81**

Table 5.2 shows the convergence for this problem. We see a rather good convergence after only 5 iterations and total convergence after 15 iterations. Notice that although the best solution converges rather quickly, the GA continues to explore the solution space. Therefore, the mean cost converges rather slowly. In fact, Fig. 5.7 plots the convergence of the solution. We see that the population average cost continues to oscillate. This behavior is due to the large mutation rate specifically chosen to force continued exploration of the solution space for this wildly oscillatory function. Thus we conclude that the choice of GA population and mutation parameters affects the performance of the algorithm. This is most certainly true and will be discussed in more detail below. But first, let's look at another way to solve this problem – directly using the real values of the (x, y) coordinates.

5.4 Continuous Variable GA – Application to Optimization

We began with the binary GA because that is where the field started. This beginning also helps explain why certain methods are used for the operations of mating and mutation that we do when working with real numbers. Plus there are problems, like the guppy evolution example, where choices are not in terms of real-valued variables. Many of the problems that we encounter, however, involve optimizing real number continuous variables, so why not work directly with continuous variables and dispense with coding in binary? In fact,



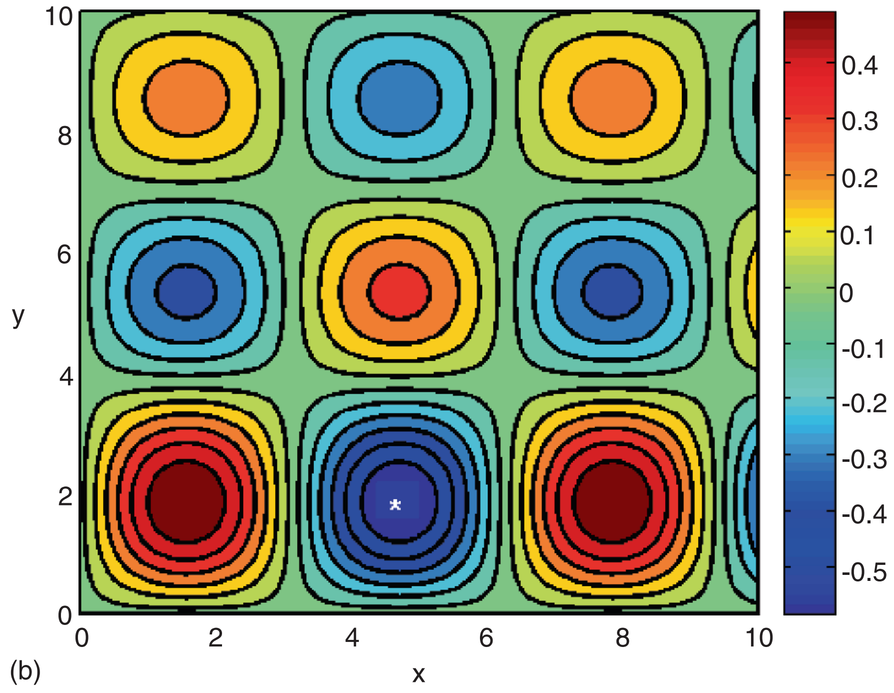


Fig. 5.4 Three dimensional (a) and contour (b) plots of the function of (5.5)

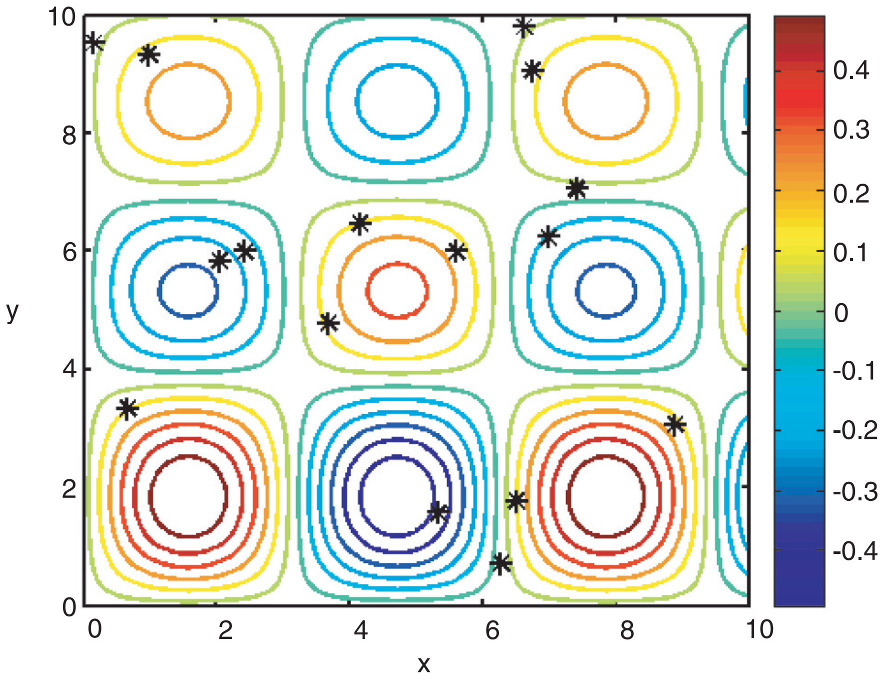


Fig. 5.5 Initial population for the binary GA solution to (5.5) superimposed on a contour plot of the function value

that is precisely what is now done with many real-valued problems.

The continuous variable GA works much the same as its binary cousin. Figure 5.8 is a flowchart of the continuous GA. It looks much the same as the flowchart for the binary GA (Fig. 5.2). The only differences are in the way certain operations are performed. In particular, chromosome definition, application of the cost function, and implementation of the operations of mating and mutation are modified to be appropriate for real values and are the only substantative changes. Let's look at each of these in more depth.

When initializing the population for continuous optimization with a GA, we must initialize the chromosome for the number of variables that we wish to find. In particular, given N_{par} variables to initialize, we define a chromosome consisting of parameters or variables each denoted by p_i as:

$$\text{chromosome} = \left[p_{1,} p_{2,} p_{3,} \cdots p_{N_{\text{par}}}\right] \tag{5.6}$$

Thus, when we define the cost function, it is in terms of a function of those variables:

$$\text{cost} = \text{F(chromosome)} = F\left[p_{1,} p_{2,} p_{3,} \cdots p_{N_{\text{par}}}\right] \tag{5.7}$$

The operations of mating and mutation are also altered to take into account these real-valued continuous variables. There are numerous methods of mating and mutation, but the ones demonstrated here are rather straightforward and most closely mimic the binary versions. For mating, the first step is to randomly choose the crossover point. The parents are then selected according to some selection criterion and labeled as m and d (mom and dad):

$$\begin{aligned} parent_m &= \left[p_{m1} p_{m2} \cdots p_{m\alpha} \cdots p_{mN_{\text{par}}}\right] \\ parent_d &= \left[p_{d1} p_{d2} \cdots p_{d\alpha} \cdots p_{dN_{\text{par}}}\right] \end{aligned} \tag{5.8}$$

The process of crossover then blends the information from the two parents. A way that most closely matches the binary GA swaps the portions of the chromosome to the right of the crossover point and blends the variable chosen as the crossover point (kinetochore):

$$\begin{aligned} p_{new1} &= p_{m\alpha} - \beta\left[p_{m\alpha} - p_{d\alpha}\right] \\ p_{new2} &= p_{d\alpha} - \beta\left[p_{m\alpha} - p_{d\alpha}\right] \end{aligned} \tag{5.9}$$

Here, β is the blending parameter between 0 and 1. The result is offspring of the form:

$$\begin{aligned} offspring_1 &= \left[p_{m1} p_{m2} \cdots p_{\text{new1}} \cdots p_{dN_{\text{par}}}\right] \\ offspring_2 &= \left[p_{d1} p_{d2} \cdots p_{\text{new2}} \cdots p_{mN_{\text{par}}}\right] \end{aligned} \tag{5.10}$$

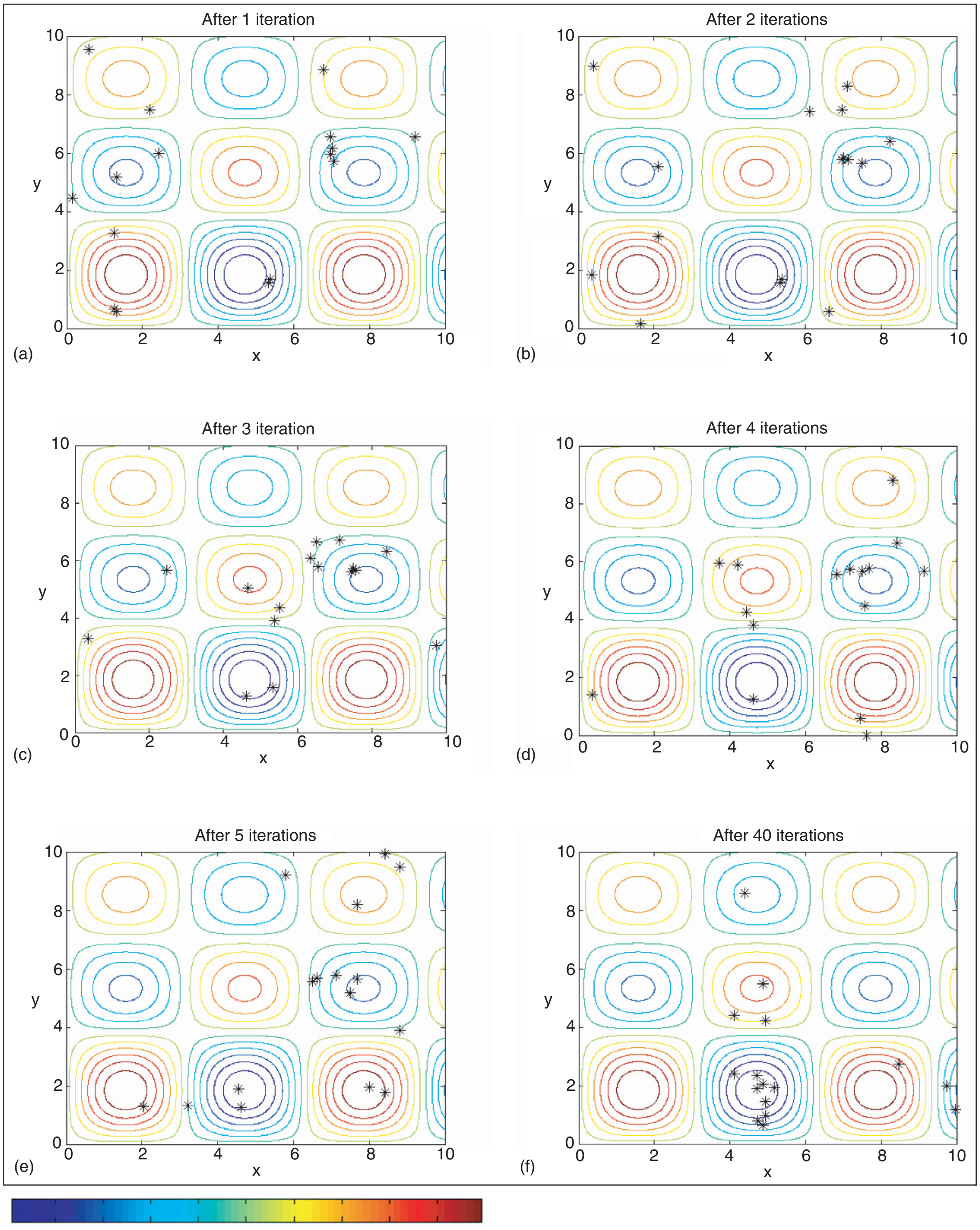


Fig. 5.6 Evolution of the GA population for solution of (5.5)

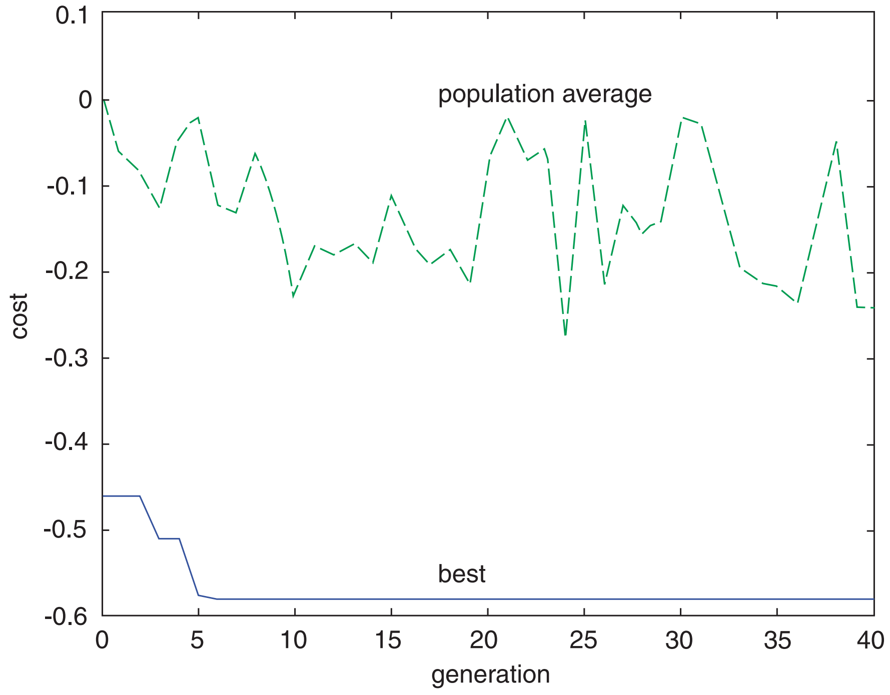


Fig. 5.7 Plot of convergence of the solution of (5.5). The solid line is the single best solution and the dotted line depicts the population mean

Altering the mutation operator is even more simple. For a continuous GA, we merely generate a new random number, $p_{i_{\mathrm{new}}}$, to replace the original value. So if the original chromosome is

$$chromosome = [p_1, p_2, p_3, p_4, \cdots p_{N\mathrm{var}}] \tag{5.11}$$

and we wish to mutate the third parameter, the new chromosome will look like

$$mutated\ chromosome = [p_1, p_2, p_{3_{\mathrm{new}}}, p_4, \cdots p_{N\mathrm{var}}] \tag{5.12}$$

Let's revisit the solution of (5.5) with the continuous GA. We choose to use a population size of 12,

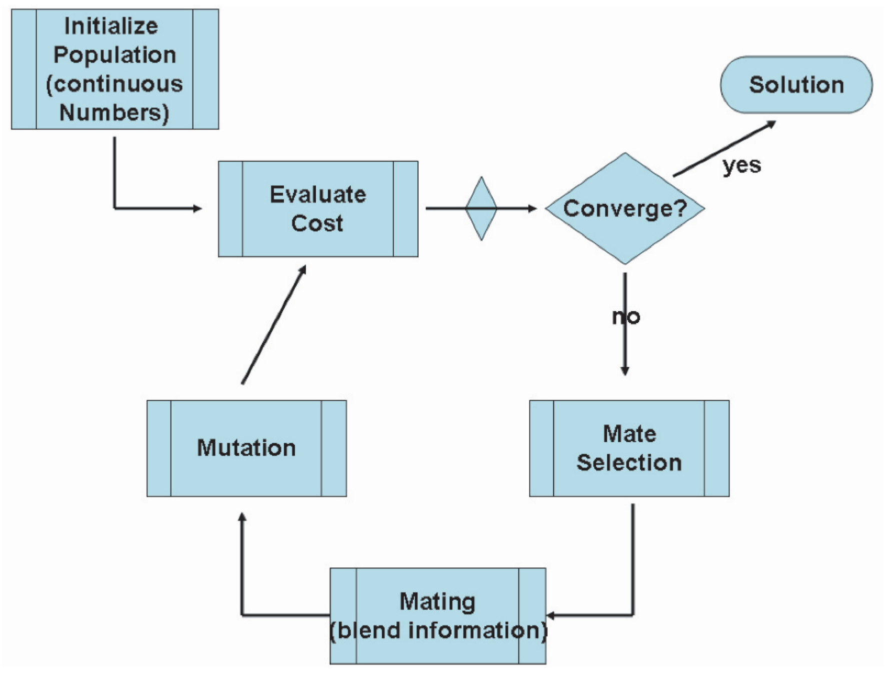

Initialize Population (continuous Numbers)
Evaluate Cost
Converge?
yes
Solution
no
Mate Selection
Mating (blend information)
Mutation

Fig. 5.8 Flow chart of a continuous parameter GA

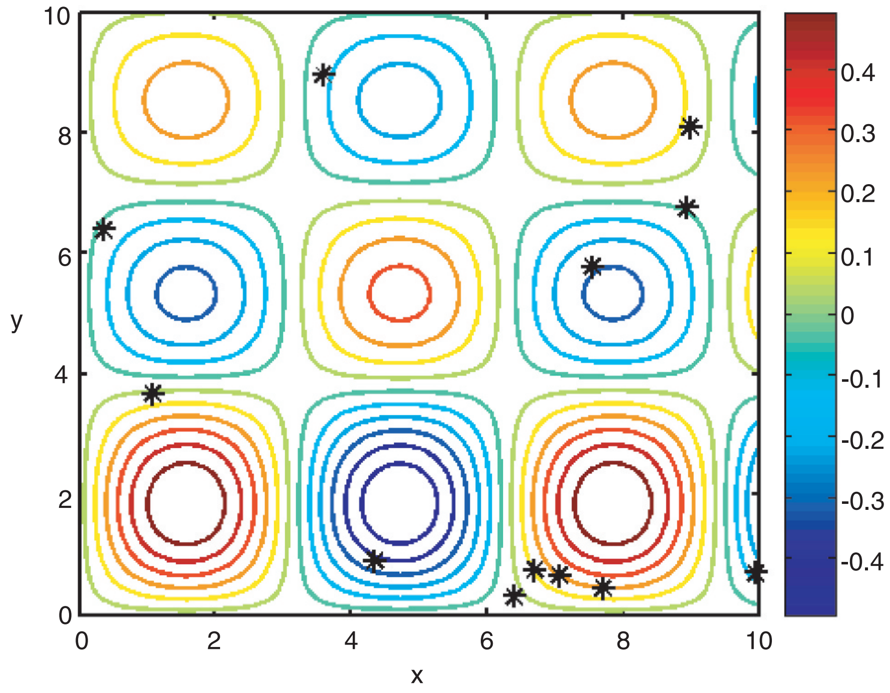


Fig. 5.9 Initial population of solutions to (5.5) for the continuous GA problem superimposed on a contour plot of the function

Table 5.3 Initial population (guesses) of solutions to (5.5) for the continuous GA and their associated costs

x	y	Cost
7.7109	0.4337	0.20967
6.7016	0.7454	0.14116
3.5830	8.9600	−0.10674
9.9596	0.6927	−0.16617
1.0709	3.6673	0.05914
8.9394	6.7496	−0.03752
4.3533	0.8984	−0.37954
6.3978	0.3043	0.01720
7.5431	5.7726	−0.29981
7.0620	0.6602	0.21947
0.3515	6.3859	−0.06386
8.9856	8.0849	0.10452

mutation rate of 0.2 and crossover rate of 0.5. Table 5.3 indicates the initial population which is also plotted on the contour plot of Fig. 5.9.

Let's follow the mating process carefully for this example. The population of Table 5.3 is sorted and the least fit chromosomes are discarded. This process leaves the following chromosomes:

4.3533	0.8985
7.5431	5.7726
9.9596	0.6928
3.5830	8.9600
0.3516	6.3859
8.9394	6.7496

These remaining chromosomes are then ranked for mating. The probability of mating is determined here by rank weighting. To do that we make a table of probabilities of mating and the cumulative probability as:

probability of mating	cum probability
0.28571	0.28571
0.2381	0.52381
0.19048	0.71429
0.14286	0.85714
0.095238	0.95238
0.0476191	1.00000

We now use a roulette wheel selection process where a random number generator determines

$$\text{pick1} = 0.26006$$
$$\text{pick2} = 0.89538$$

These correspond to chromosomes ranked 1 and 5. Now the random number generator is applied again to choose the crossover point, which is chosen here in the second variable. The value of β is randomly chosen as 0.32178. So the new offspring generated and their associated costs are:

	x	y	cost
Offspring 1:	4.3533	2.6642	−0.2689
Offspring 2:	0.3516	4.6202	−0.2496

The next step is mutation, that is, randomly changing some of the values of the parameters. Table 5.4 shows the new population. Highlighted values denote

Table 5.4 The new population at iteration 1 of solving (5.5) with a continuous parameter GA. Mating and mutation have altered the population. Highlighted values denote mutations. The original six chromosomes do not mutate, but 4 out of the 12 values in the six offspring mutate this generation

Original x	Original y	Original cost	Mutated x	Mutated y	Mutated cost
4.3533	0.8985	−0.37954	4.3533	0.8985	−0.37954
7.5431	5.7726	−0.29981	7.5431	5.7726	−0.29981
9.9596	0.6928	−0.16617	9.9596	0.6928	−0.16893
3.5830	8.9600	−0.10674	3.5830	8.9600	−0.10674
0.3516	6.3859	−0.06387	0.3516	6.3859	−0.06387
8.9394	6.7496	−0.03753	8.9394	6.7496	−0.03753
4.3533	**2.6642**	**−0.42353**	**7.2802**	**4.9394**	**−0.26890**
0.3516	4.6202	**−0.09000**	**0.8155**	4.6202	−0.24959
0.3516	5.7588	−0.10909	0.3516	5.7588	−0.10909
9.9596	1.3199	**−0.26825**	**0.3702**	1.3199	**0.19042**
4.3533	3.5033	−0.12733	4.3533	3.5033	−0.12733
3.5830	6.3552	0.08275	0.3583	6.3552	0.08275

mutations. The original six chromosomes are not mutated here, but four of the twelve values mutate in the new generation.

The continuous GA was iterated to complete the solution process. The correct solution was found after 10 iterations. Figure 5.10 shows the population clustered around the exact solution at the completion of the tenth iteration. Figure 5.11 denotes convergence. We see that the cost function value for this particular run tended to generally decrease for the entire population rather than just for the best individual.

We must be careful, however, about overinterpreting convergence. We should always remember that, because the GA uses random numbers for its initialization and operations, every time we run the GA we'll obtain a somewhat different result. If we have carefully chosen our parameters and done enough generations, we usually converge to the correct solution. Occasionally, however, our luck will be bad and we may either not get to the solution in the allowed number of iterations or convergence will be slow. Each set of initial parameters, each mating operation,

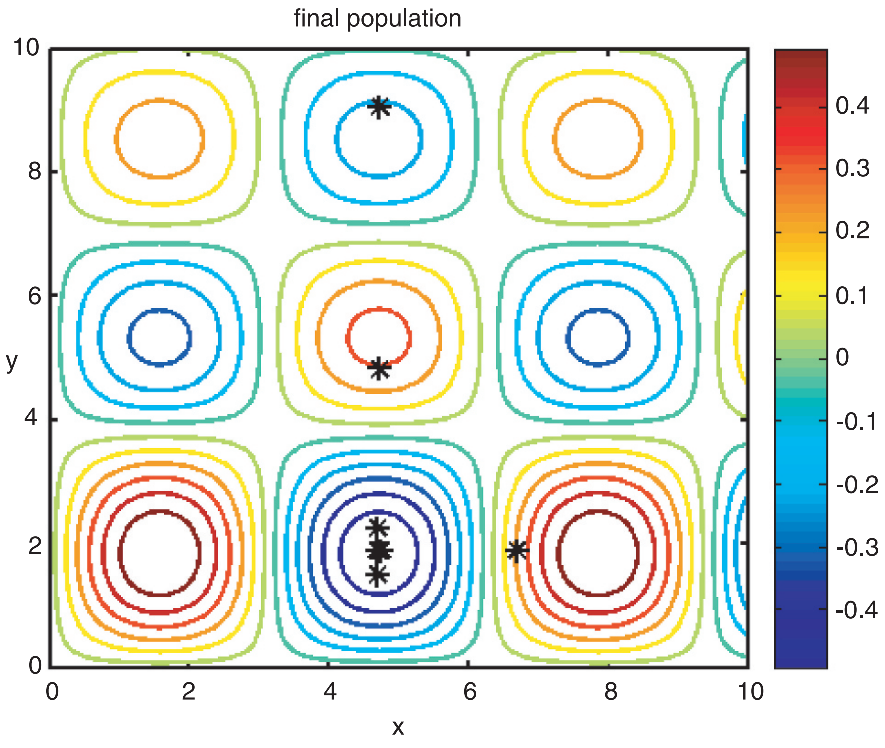


Fig. 5.10 Final population after 10 generations cluster near the exact solution to (5.5)

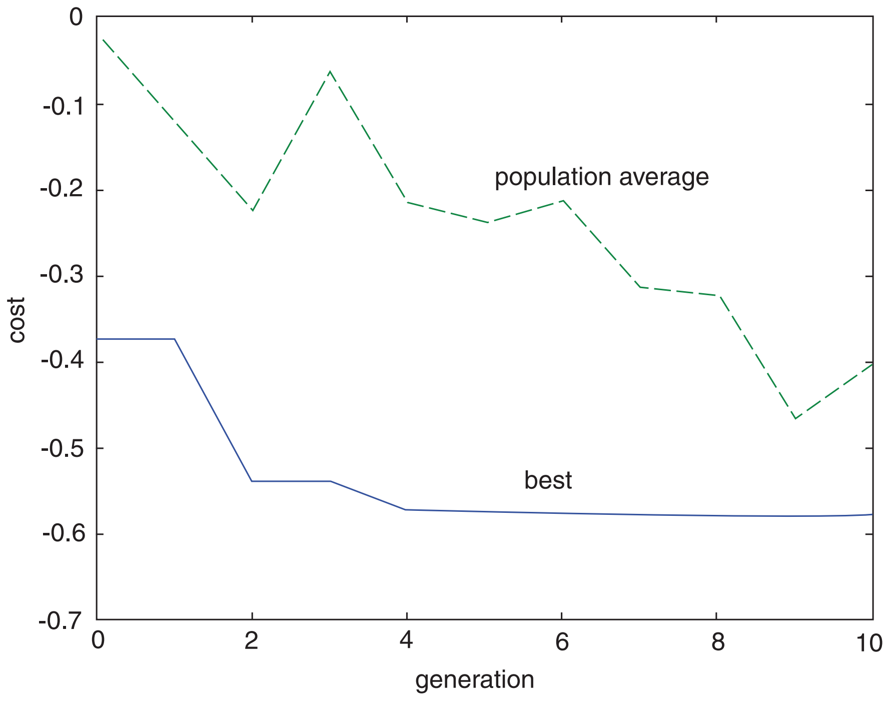


Fig. 5.11 Convergence of the continuous GA solution to the Besin problem (5.5)

and each application of mutation will have different results: therefore, the convergence plot and behavior of the solutions will differ with each run. So how do we assure ourselves that the GA is performing adequately? The answer is that we recognize these issues and deal with them directly. We never fully believe the results of a single GA run. Instead, we make a habit of always doing repeated applications. When we run the GA 10 times with essentially the same result, we are assured that the GA is behaving solidly for this problem and the solution is well optimized.

5.5 Optimizing GA Application

We have now observed several applications of a GA. So far, we have arbitrarily chosen performance parameters such crossover rate, crossover methodology, mutation rate, and population size. Let's look at these GA tuning parameters in a bit more detail.

5.5.1 Continuous vs. Binary GA

First, we have used both binary and continuous variable GAs. Which one is better? The answer, of course, depends on the problem. When we have either/or choices or choices between a small number of potential realizations (Is the stream of the guppies' environment rocky or smooth? Is the guppy dappled or plain?) then the binary GA is most appropriate. If we instead are searching for all real numbers, then one might as well use a continuous GA. Of course, one can always code continuous numbers in binary notation, but in general, the resulting GA is not as efficient as the continuous form. What if our parameters are mixed – we have some that are more conducive to binary coding and others toward continuous? We can always use a binary GA with the continuous variables coded into binary. New methods that mix the binary with the continuous have recently been developed and show promise for very efficient GA application (Haupt 2007, Haupt et al. 2008).

5.5.2 Variable Normalization

A related issue is whether and how to normalize our variables. If all of our variables are similar (for example, if we're on an (x, y) Cartesian grid from 5 to 10), normalization is irrelevant. The behavior of the GA does not depend on whether every number in the problem is divided by the same value. If however, we have variables that are wildly disparate in nature, we may want to normalize them to be on the same order of magnitude. If we're on an (x, y) Cartesian grid where x varies from 10^{-6} to 10^{-5} and y varies from 10^5 to 10^6, then we would want to divide all x's by the mean x and all y's by the mean y before the search and cost function evaluation. One common technique is to always normalize all variables to a scale from 0 to 1. This approach has several advantages: (1) it assures that the behavior of the GA is not biased to any one variable, (2) it avoids normalizing the variable each time it is sent to a cost function, and (3) it makes generating random numbers rather easy since most random number generators give numbers between 0 and 1. For some problems, it may make more sense to map a variable with some function before doing a GA application. An example from air pollution in Chapter 14 maps the monitored concentrations onto a logarithmic scale for GA solution.

5.5.3 Mate Selection

The methods of selecting mates vary rather widely. The simplest methods are random without respect to any weighting. For instance, one can just randomly choose two parents to mate. This method is known as unweighted roulette wheel pairing. The more prevalent methods are based on either a rank or cost weighting method. The most basic ranked method is to pair the sorted chromosomes in order – the first chromosome mates with the second, the third with the fourth, and so on. The more refined ranked methods compute the probability of mating according to the ordering after the chromosomes are sorted from best to worst. That was the method demonstrated in Section 5.4. Probability of mating varies according to cumulative order of the rank. Cost weighting is computed according to the actual values of the cost function – a very low cost chromosome is much more likely to be selected for mating than the one that may be next in line. The costs are merely summed, then each cost divided by the total to obtain its probability of selection.

Once the selection probabilities are computed via either ranked weighting or cost weighting, the selection can occur via any of several methods. A common method is the roulette wheel selection (demonstrated in Section 5.4) where a random number is chosen and mapped to the cumulative probability of the ranked or cost weighted chromosomes. Another common method is tournament selection (demonstrated in Section 5.2). Three (or some other number) potential parents are selected according to random number selection matched to the computed probability of mating. The two with the lowest costs then mate. One can go into much more detail on the mate selection methods and we refer the reader to any of the GA books to read more (Goldberg 1989; Mitchell 1996; Davis 1991; Haupt and Haupt 2004).

5.5.4 Mating

Once the mates are selected, the crossover technique must be specified. The first decision regards the crossover rate, that is, how many population members should be replaced with offspring at each generation. In general, although some investigators have looked at this sensitivity, it doesn't make much impact on GA performance what rate is used. Using a crossover rate (X_{rate}) of 0.5 so that the number kept (N_{keep}) is half of the population size ($N_{\text{keep}} = X_{\text{rate}} \times N_{\text{pop}}$) is as good as any and is the typical choice.

The next issue is how to perform the crossover. The examples above used single point crossover. For the binary GA, it is easy to extend the crossover to two points, three points, or use three parents in a rather straightforward way. Uniform crossover can be performed by creating a random mask of 0s and 1s to determine whether to use the value in the mother or the father chromosome at each element.

For the continuous parameter GA, equations (5.9) and (5.10) blend the information along the axes of the parent chromosomes. When the blending parameter, β, is less than or equal to 1, the values of the offspring will be between the parents. When β is allowed to be greater than 1, then the axes are extended beyond the magnitude of the parent values, which is sometimes a good choice. A more simplistic continuous GA mating scheme merely swaps values of the real-encoded genes between the parents. If the parents are denoted as in (5.8), and we choose to swap between genes 2 and 5, we would obtain

$$\begin{aligned} &\mathit{offspring}_1 = \\ &\quad [p_{m1},\ p_{m2},\ \uparrow p_{d3},\ p_{d4},\ \uparrow p_{m5},\ p_{m6}, \ldots, p_{mN_{\text{var}}}] \\ &\mathit{offspring}_2 = \\ &\quad [p_{d1},\ p_{d2},\ \uparrow\ p_{m3},\ p_{m4},\ \uparrow p_{d5},\ p_{d6}, \ldots, p_{dN_{\text{var}}}] \end{aligned} \tag{5.13}$$

The problem with this simple scheme is that no new values are ever generated. The opposite extreme is uniform random crossover where each gene is assigned a random blending parameter β_i and the resulting offspring are constructed as

$$\begin{aligned}
\mathit{offspring}_1 &= \mathit{parent}_1 - [\beta_1(p_{m1} - p_{d1}),\\
&\quad \beta_2(p_{m2} - p_{d2}), \cdots, \beta_{N_{\text{var}}}(p_{mN_{\text{var}}} - p_{dN_{\text{var}}})]\\
\mathit{offspring}_2 &= \mathit{parent}_2 + [\beta_1(p_{m1} - p_{d1}),\\
&\quad \beta_2(p_{m2} - p_{d2}), \cdots, \beta_{N_{\text{var}}}(p_{mN_{\text{var}}} - p_{dN_{\text{var}}})]
\end{aligned} \tag{5.14}$$

A scheme similar to this was used for extended runs of the air pollution source characterization problem of Chapter 14.

5.5.5 Choosing Population Size and Mutation Rate

The choice of the GA parameters of population size and mutation rate can make a large impact on algorithm performance. The performance measure that we will concentrate on here is the number of cost function evaluations required to meet a pre-specified tolerance level of accuracy of the solution. We prefer this measure because: (1) it is easy to keep track of how many times the cost function has been called, (2) as applied scientists, we often want to find the "best solution": thus, the number of calls to find that best solution is the relevant quantity in measuring required computer time, and (3) it is not dependent on the type of computer being used.[2] Of course, measuring function calls does not represent all aspects of the GA (such as population generation, mating, mutation, sorting, etc.). For large problems that are computationally intensive, however, the number of function evaluations is often the controlling factor for measuring GA speed. Since the GA begins with random numbers, each new run of the GA will take a different number of function evaluations to "solve" the problem.

[2] Note that we are assuming a serial computer for this discussion. Application on parallel machines is beyond the scope of this chapter. Application on a parallel machine will change the performance as a function of mutation rate and population size, depending on how the GA is adapted to optimize the parallelism of the type of machine being used.

We present a sensitivity analysis of the number of function evaluations necessary to solve the Besin problem (5.5) for both the binary and continuous GA. Since we know the exact solution for this construed problem, we can easily determine convergence. Here, we stop the GA when the error in the solution becomes less than 5×10^{-3} or we surpass 5,000 iterations. Ten separate sensitivity runs are completed for each combination of population size (4, 8, 12, 16, 20, 32, 40, 48, 56, 64, 72, 80, 88, and 96) and mutation rate (0.001, 0.005, 0.01, 0.05, 0.075, 0.1, 0.12, 0.125, 0.15, 0.175, 0.2, 0.225, and 0.25). The number of function evaluations for each combination of population size and mutation rate of those 10 runs are then averaged to produce plots (Fig. 5.12 for the binary GA results and Fig. 5.13 for the continuous variable GA) of average performance as a function of population size and mutation rate. Both plots contour the logarithm (base 10) of the number of function calls. We see that for both the binary and continuous GA, using too small of a mutation rate prolongs the calculation.

Figure 5.12 plots the number of function evaluations required to solve equation (5.5) using a binary GA. Note that using the smallest mutation rates requires more calls to the cost function to solve the problem. This observation implies that a sufficient number of mutations is required to push the population into the global solution basin for this highly oscillatory cost function. The fewest number of function evaluations are required when the population size is relatively small (≤ 32) and the mutation rate is moderately large (0.075 to 0.25). For this problem, mutation is a critical operator that keeps the solution from prematurely converging toward the wrong local minimum. The "best" combination for this problem using the binary GA is a population size of 8 and mutation rate of 0.15. The results for the continuous GA appear in Fig. 5.13. In this case, in addition to small mutation rates causing too many function evaluations, too small a population (≤ 20) also results in a very slow convergence. The "best" combination for the continuous variable GA applied to the Besin problem is a population size of 12 and mutation rate of 0.25.

Our prior work confirms this finding that using relatively small population sizes in combination with high mutation rates is often effective for minimizing the number of function evaluations (Haupt and Haupt 1998, 2000, 2004). One should be careful, however, to note that this conclusion is problem dependent.

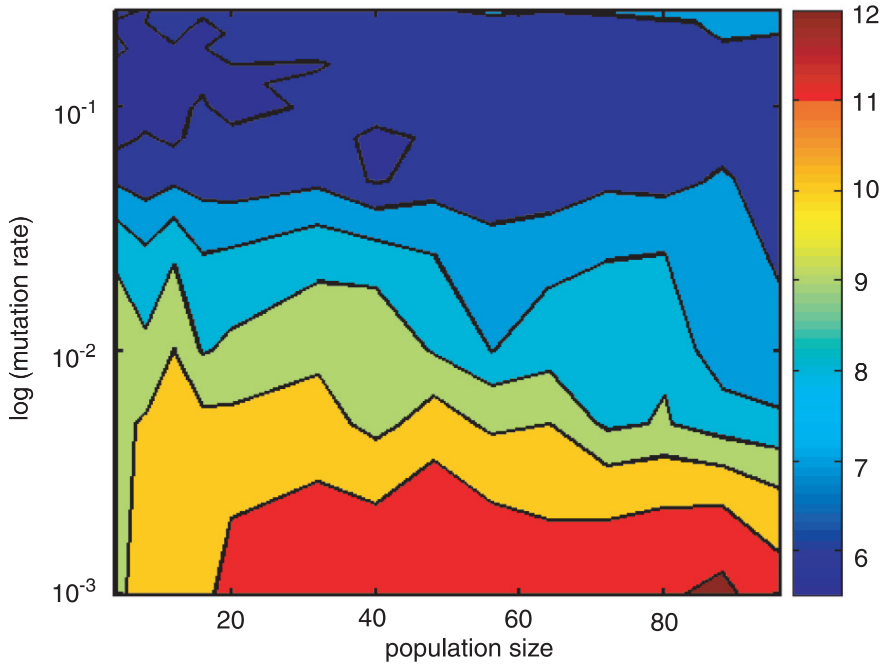


Fig. 5.12 Average number of cost function evaluations over 10 sensitivity runs required to find the solution for a binary GA (log of the number of cost function evaluations)

When there are a large number of unknowns and the cost function has fewer local minima, larger population sizes are sometimes more efficient. We still find, though, that the mutation rate must be sufficiently large.

5.5.6 When to Use a GA

How does the GA compare in speed to other methods? When do we choose to use a GA on our optimization problem rather than a more traditional technique? In

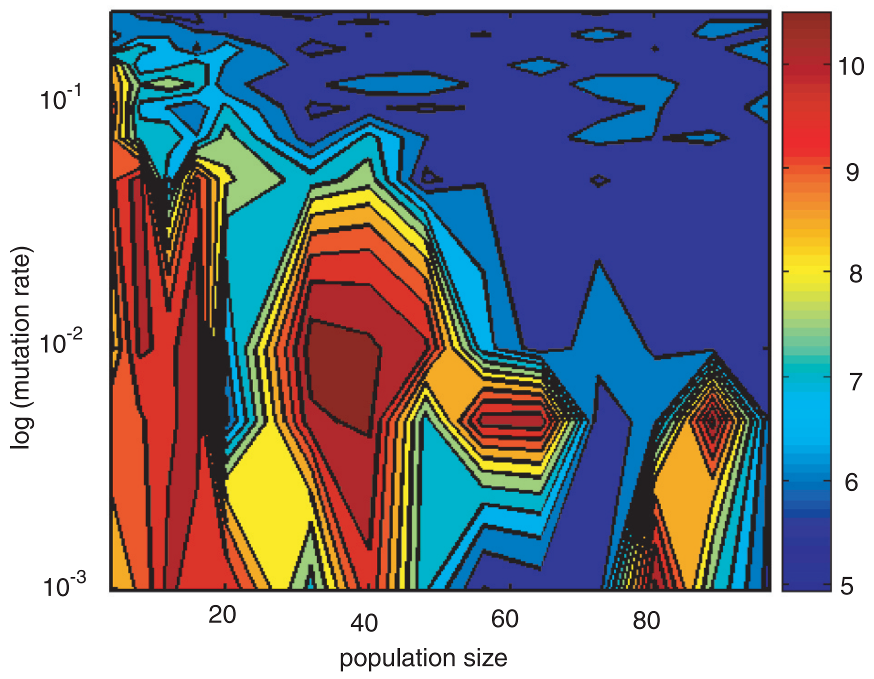


Fig. 5.13 Average number of cost function evaluations over 10 sensitivity runs required to find the solution for a continuous GA (log of the number of cost function evaluations)

general, if we have a well behaved bowl shaped continuously differentiable function and we wish to find the minimum, the gradient descent methods can't be beat. Such methods were designed for those cases and work well there. The GA will not match their speed. In contrast, if we have a very complicated function with lots of local optima, the gradient algorithms will typically find the nearest minimum, which is often local, not global. The usual situation for the practicing scientist or engineer is that he or she configures a problem in a rather large solution space and isn't quite sure what the function looks like in solution space. In that case, it is often wise to try the gradient algorithms first using various different first guesses. If the algorithm finds a different solution for each initialization, the cost function likely has local minima and that is what the algorithm is finding. Then the practitioner knows that using a more robust technique is merited. Those cases are where the GA shines.

Many optimization experts prefer to combine the strengths of the various techniques on their difficult problems. One strategy is to use a hybrid GA; that is, to begin the solution process with a GA until the correct solution basin is discovered, then switch to a fast gradient descent method. One strategy that this author uses is to employ the GA for a specified number of iterations or until a plateau in the convergence plot is reached, then switch to a descent technique. This strategy is often successful at using the GA to determine the basin of attraction then using the ability of the gradient descent method to zoom to the bottom of that basin rapidly. Chapter 14 demonstrates this strategy on a difficult air pollution problem.

5.5.7 Genetic Algorithms on a Parallel Computer

Everything we have said thus far about the speed expected of a GA assumes that we are using a serial computer. There are ways to speed GAs on a parallel computer that (1) make them competitive with other techniques in speed, (2) make efficient use of the processors, and (3) may even tune the GA to produce global minima more quickly. It is beyond our scope to go into too much detail here, but we do want to point out that for some problems, creating co-evolving populations not only makes the algorithm amenable to distributing among processors, but it also helps discover a more global minimum and speeds the convergence, even if implemented on a single processor machine. The many brands of parallel GAs can be characterized grossly into three primary categories: master-slave, island GA, and cellular GA. Figure 5.14 is a graphical depiction of these three parallel implementations.

The master-slave implementation of the GA is the most similar to the serial GAs that we have been discussing: they merely distribute the cost function evaluations to be done by slave processors that report their results to the master processor. This method is easy to implement and no subpopulations are required.

The island GA allows simultaneous evolution of independent populations. There is often a prescription for some periodic migration between these subpopulations to increase diversity. Although this method is a bit more difficult to implement, it is often successful at increasing the diversity of the overall population while speeding evolution of the individual populations through independent evolution that doesn't need to wait for communication from any master.

The cellular implementation of parallelism is ideal for architectures where communication with nearest neighbor nodes is faster than for more distant processors. In this case, each individual often evolves on its own node with options of mating only with its nearest neighbors. More detail on parallel GAs can be found in Gordon and Whitley (1993), Alga and Tomassini (2002), and Haupt and Haupt (2004).

5.5.8 Cost Function Construction

Finally, we would be negligent if we did not emphasize the importance of wise construction of the cost function. This element is, of course, unique to the problem at hand. The cost function is often the determining factor for both how much CPU time is required as well as how quickly the GA converges to the solution. A few of the typical general programming tips hold especially true for the GA since the cost function is called repeatedly – vectorize the code where possible (taking advantage of the storage order and special vectorization tools of the language being used) and

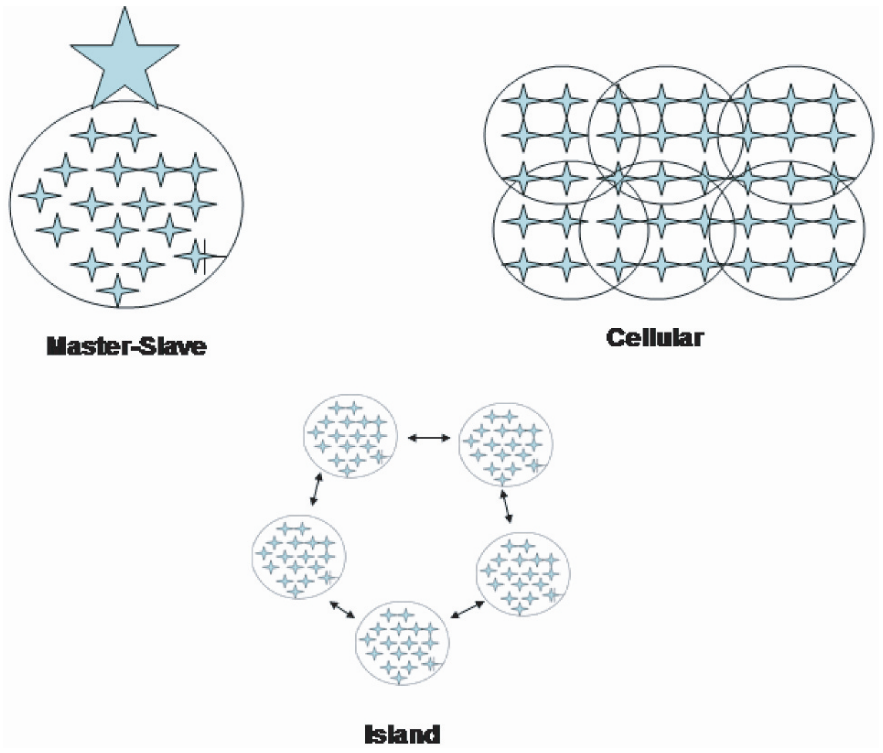


Fig. 5.14 Schematic of the three general types of parallel GA configurations

avoid using constructs known to slow down the code (such as contingencies). In addition, the GA allows more creative design of cost functions than many traditional techniques. For instance, many traditional problems minimize the L_2 norm (sum of the squares) of some difference from a known or previous value. With a GA, it is easy to explore using other powers of the difference – in some cases an L_1 norm (absolute value of the differences) is preferable while in others, higher powers are useful if we want to weight the technique to avoid outliers. In yet other problems, least squares need not play any role. For instance, in developing contingency tables for occurrences of a meteorological condition (e.g. predicting whether or not it will hail), the models are traditionally built using a least squares methodology, but then judged using more complex metrics such as Fraction Correct, Critical Success Index, or Heidke Skill Scores. The GA is capable of training the model using those same metrics to optimize agreement with past data (Marzban and Haupt 2005 and Chapter 18 of this volume). Therefore, one additional strength of using a GA to optimize a function is the greater freedom in designing the function to be optimized. In addition, it may be appropriate to weight different portions of a cost function differently. We did this for the initial guppy problem (Section 5.2) when balancing the competing parameters of the attractiveness to mates versus the likelihood of being eaten by a predator.[3] Finally, in some cases, constraints can be built directly into cost functions to avoid specific portions of parameter space. If such a constraint is necessary, one could simply add a term to the cost function to increase the cost if the function value strays too far from that expected – i.e. impose a penalty for straying too far in parameter space.

5.6 Application to a Dynamical Systems Problem

Let's revisit our initial problem of optimizing a population of guppies to survive in a particular

[3] An alternative approach would be to map the pareto front of the competing parameters. That technique is beyond the scope of this chapter but is covered in Haupt and Haupt (2004).

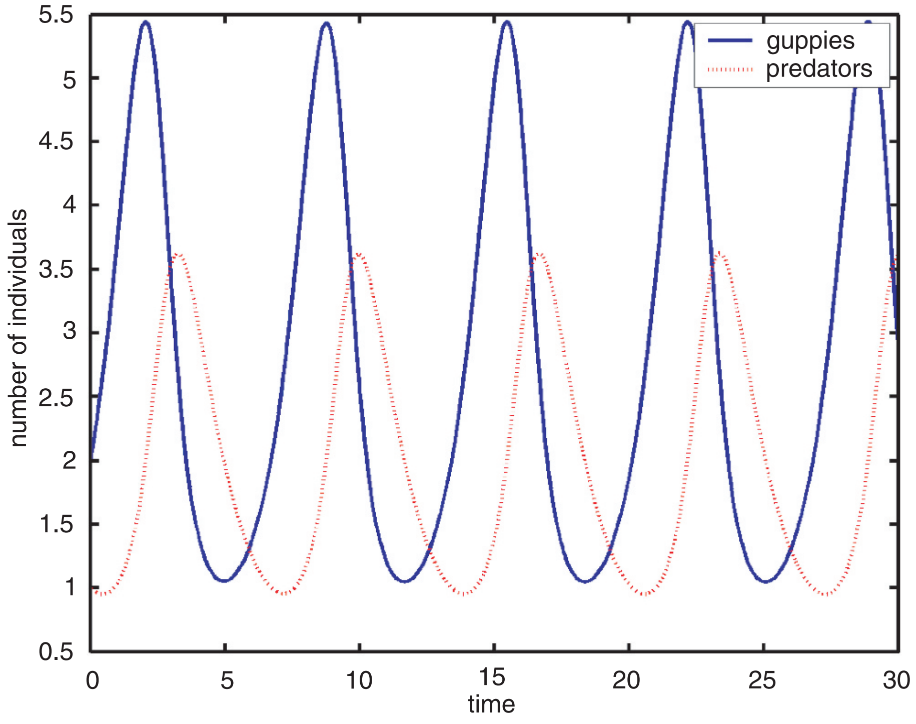


Fig. 5.15 Time evolution of the predator-prey equations (5.15)

environment populated by a predator. In Section 5.2 we directly optimized the guppy population using a binary value GA. Modern population biologists often instead model population dynamics with differential equations. For instance, the classic predator-prey problem, also known as the Lotka-Volterra equations (see Chapra and Canale 1998 for a brief discussion of these equations) is formulated as:

$$\begin{aligned}\frac{dx}{dt} &= ax - bxy \\ \frac{dy}{dt} &= -cy + dxy\end{aligned} \tag{5.15}$$

where

x = number of prey (guppies)
y = number of predators
a = guppy birth rate
c = predator death rate
b,d = interaction coefficients

This is a nonlinear system with coupling of the rate equations for the change in the number of prey and predators. It is reasonably trivial to integrate this equation in time and to characterize it in phase space. Setting the parameters $a = 1.2$, $b = 0.6$, $c = 0.8$, and $d = 0.3$ and doing a Runge-Kutta time integration with a time step of 0.01 produces a time series as shown in Fig. 5.15. Figure 5.16 is a plot in the phase space of guppies versus predators. The egg shaped pattern denotes a limit cycle that is repeated indefinitely. The nonlinear coupling of these equations is what makes this limit cycle occur.

What if we had measured data of the time evolution of the guppies and their predators but didn't know the functional form? What if we wished to come up with a model for the rate of change of guppies and predators given this data that we had obtained? For the moment, let's assume that this time series we generated through integrating equations (5.15) is that data. We wish to use these data to reproduce the time dependent behavior of the guppy/predator interaction.

The first step in solving such a problem is to generate a likely form for the solution. The simplest approach is to assume a linear model, such as:

$$\mathbf{s}_t = \mathbf{Ls} + \mathbf{C} \tag{5.16}$$

where **s** is the vector of variables (x, y), the subscript t denotes its time variation, **L** is the linear matrix

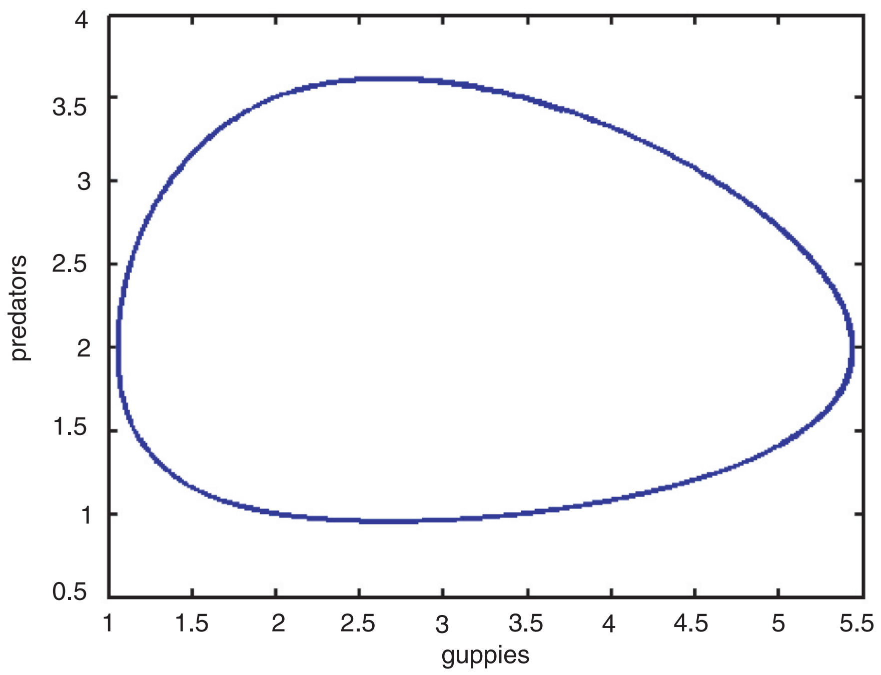


Fig. 5.16 Phase space plot of guppies vs. predators for equations (5.15)

operator, and **C** is a constant matrix. One doesn't need an optimization algorithm to fit data to this model: just use standard least squares parameter estimation to determine the unknown elements of matrices **L** and **C**. Doing that and using equation (5.16) to integrate it forward in time produces the time series observed in Fig. 5.17. We see here that this linear model finds a solution of one predator and a monotonically growing number of guppies in time. The corresponding phase space plot appears as Fig. 5.18. It is obvious that this diverging solution is not a very good match to the actual solution of Figs. 5.15 and 5.16.

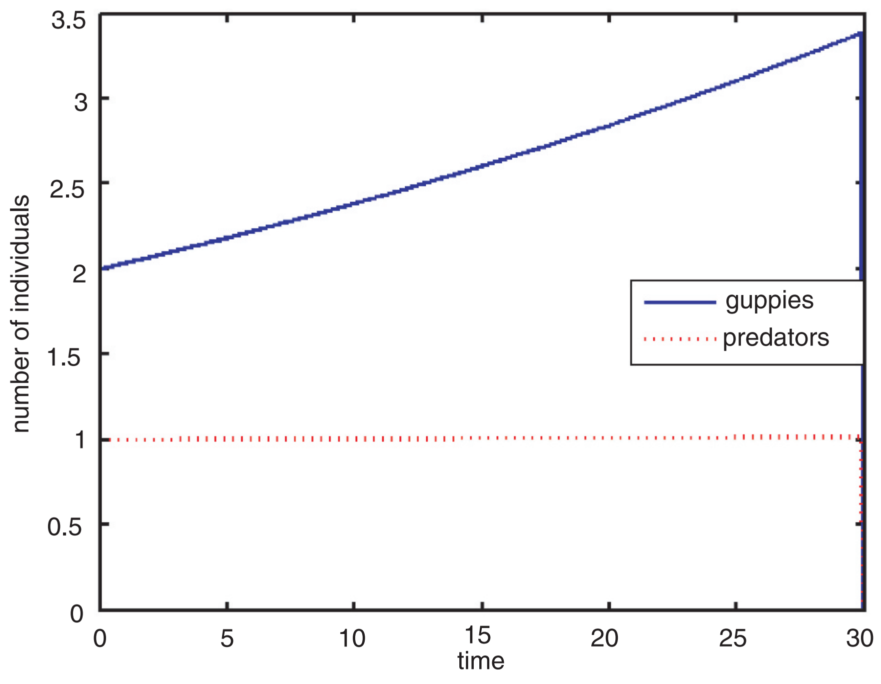


Fig. 5.17 Time series of linear empirical model fit (equation 5.16) to the Lotka-Volterra equations

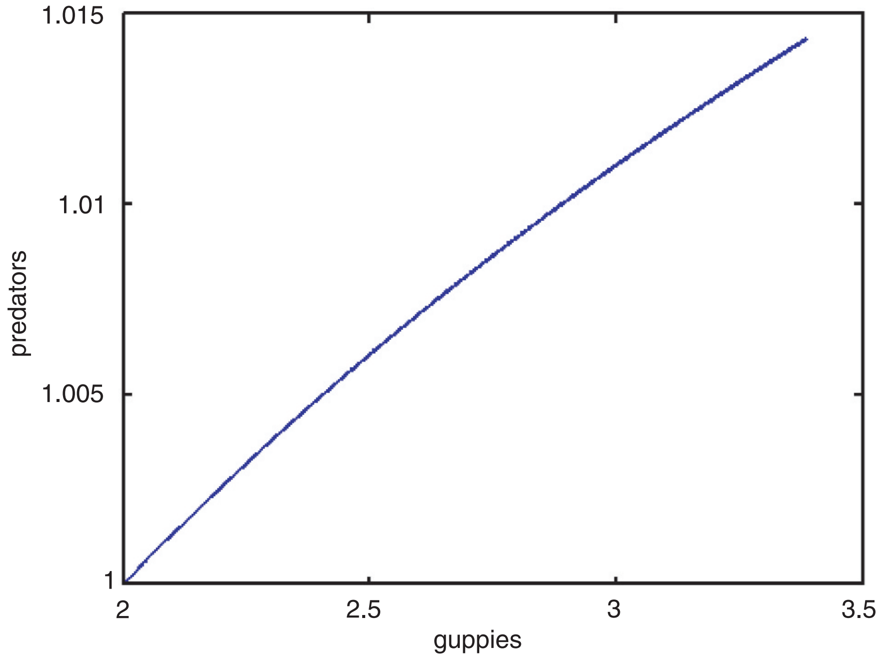


Fig. 5.18 Phase space plot of linear empirical model fit (equation 5.16) to the Lotka-Volterra equations

The next approach is to try configuring a nonlinear model of the guppy/predator time behavior. We conjecture (given that we know the solution) that the nonlinearity is quadratic. Therefore, the simplest time dependent model that includes both a linear and quadratic term is:

$$\mathbf{s_t} = \mathbf{N}\mathbf{s}^{\mathbf{T}}\mathbf{s} + \mathbf{L}\mathbf{s} + \mathbf{C} \tag{5.17}$$

This nonlinear equation can not be as simply solved using least squares techniques. An equation to do that

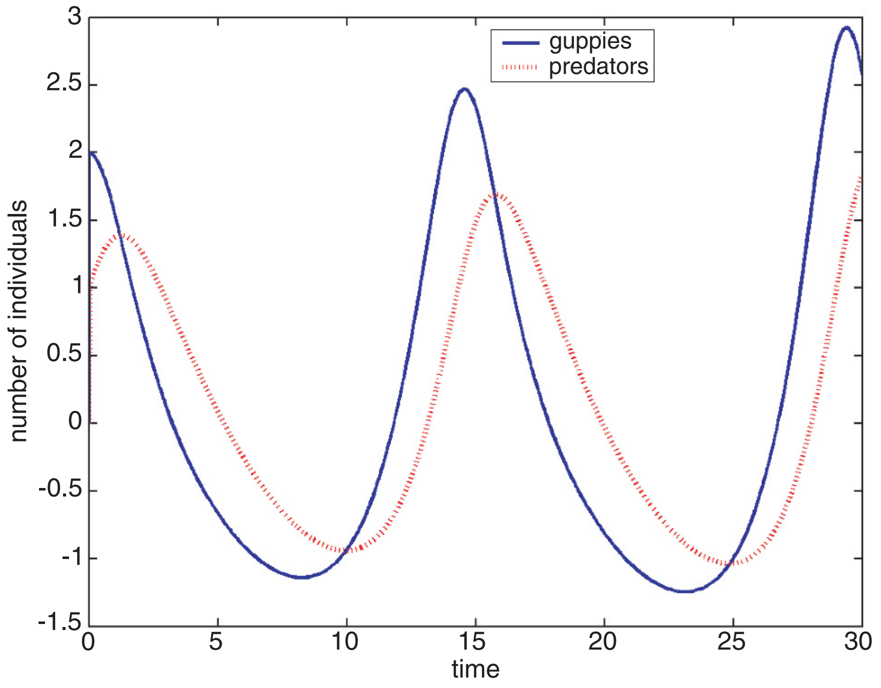


Fig. 5.19 Time series of GA solved nonlinear empirical model fit (equation 5.17) to the Lotka-Volterra equations

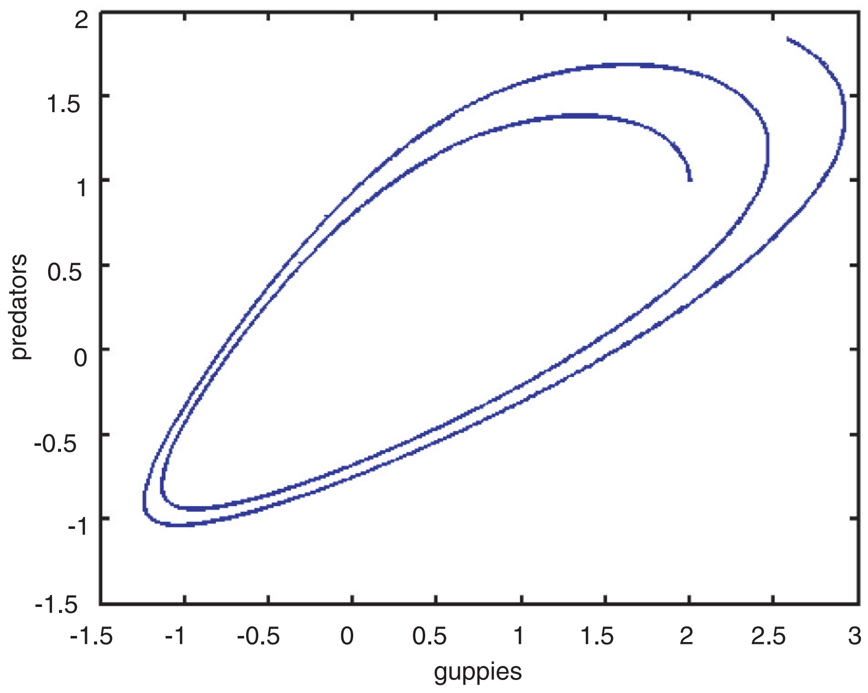


Fig. 5.20 Phase space plot of GA solved nonlinear empirical model fit (equation 5.17) to the Lotka-Volterra equations

can be found (Haupt 2006), but its solution involves inverting a fourth order tensor, which is not a trivial problem. Instead, we choose to configure this problem as one in optimization where we seek to minimize:

$$\text{cost} = \mathbf{s_t} - \left(\mathbf{Ns}^{\mathbf{T}}\mathbf{s} + \mathbf{Ls} + \mathbf{C}\right) \tag{5.18}$$

Given that we have the data of **s** and its tendency, $\mathbf{s_t}$, we can then directly solve for the elements to the matrices **N**, **L**, and **C**. In addition, we use information on the symmetries and relations expected between these matrices to minimize the number of real variables that we need to find (Haupt 2006). A GA is used to find

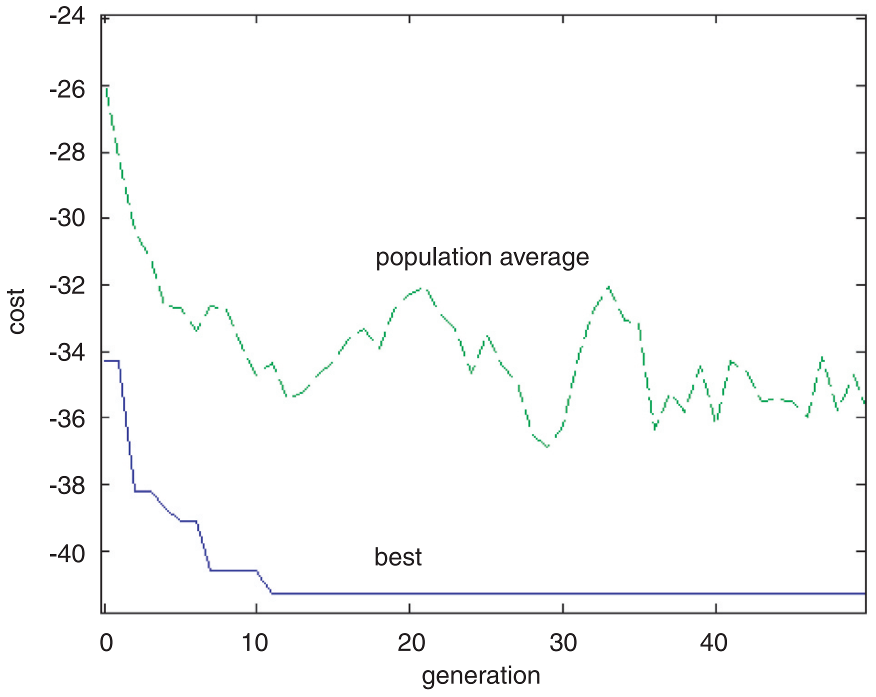


Fig. 5.21 GA convergence for the nonlinear guppy/predator empirical model

the solution – the time tendency appears as Fig. 5.19 and phase space plot as Fig. 5.20. The match to the original data (Figs. 5.15 and 5.16) is not perfect, but the general shape and oscillatory nature of the behavior is obtained. Figure 5.21 is the GA convergence curve. We see that it took very few iterations (10 for this run) to converge to the optimal solution. Note that reproducing the exact balance for a nonlinear model with an empirical model is extremely difficult and it is amazing that we have come this close with an optimization algorithm. Chapter 18 further develops this technique in the context of a chaotic dynamical system.

5.7 Conclusions

This chapter strives to give a basic introduction to genetic algorithms. We saw that GAs can be a useful technique to solve optimization problems. We also saw that they can be used for building models, whether of natural processes such as guppy evolution or of dynamical systems such as the Lotka-Volterra equations. We have only begun to scratch the surface of potential GA applications or of how to best apply the GA. There are entire books on what GAs are and how to apply them (Holland 1975; Goldberg 1989; Davis 1991; Mitchell 1996; Haupt and Haupt 1998, 2004). Chapter 18 gives a smattering of examples of how to configure a problem for a GA and reviews prior use of GAs in the environmental sciences. Chapter 14 delves into detail on how a GA was used in a specific problem in source characterization of a source of air contaminant and how that problem could be reconfigured due to GA robustness to additionally solve for meteorological variables.

The GA is a useful tool, but it is not always the first tool to try on an optimization problem. As stated in Section 5.5, traditional methods may be more time efficient on easy-to-solve problems. The strength of the GA is in optimizing difficult problems with lots of local minima. A carefully configured GA is one of the most robust tools for finding such solutions. It may not be as fast, but it is amazingly successful at identifying the basin of the global minimum. As will be demonstrated in Chapter 14, a hybrid approach of using the GA to that point, then switching to a gradient descent method is often the quickest way to a difficult solution.

The hope of this author is that we have provided the reader with enough of a view of GAs to capture his or her interest and have helped motivate the reader to apply it to his or her own problems.

Acknowledgements The GA used here was jointly authored with Randy L. Haupt. The genetics aspects of the guppy evolution problem were brought to my attention by Amy J. Haupt and her biology teacher, Beth Paterson. Bonny A. Haupt programmed the cost functions for the guppy evolution problem.

References

Alga, E., & Tomassini, M. (2002). Parallelism and evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, *6*, 443–462.

Chapra, S. C., & Canale, R. P. (1998). *Numerical methods for engineers* (3rd ed.). Boston: McGraw-Hill.

Davis, L. (Ed.) (1991). *Handbook of genetic algorithms*. New York: Van Nostrand Reinhold.

De Jong, K. A. (1975). *Analysis of the behavior of a class of genetic adaptive systems*. Ph.D. dissertation, The University of Michigan, Ann Arbor, MI.

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization, and machine learning*. New York: Addison-Wesley.

Gordon, V. S. & Whitley, D. (1993). Serial and parallel genetic algorithms as function of optimizers. In S. Forrest (Ed.), *ICGA-90: 5th international conference on genetic algorithms* (pp. 177–183). Los Altos, CA: Morgan Kaufmann.

Haupt, R. (2007). Antenna design with a mixed integer genetic algorithm. *IEEE Transactions on Antennas and Propagation*, *55*(3), 577–582.

Haupt, R. L., & Haupt, S. E. (1998). *Practical genetic algorithms* (177 pp.). New York: Wiley.

Haupt, R. L., & Haupt, S. E. (2000). Optimum population size and mutation rate for a simple real genetic algorithm that optimizes array factors. *Applied Computational Electromagnetics Society Journal*, *15*(2), 94–102.

Haupt, R. L., & Haupt, S. E. (2004). *Practical genetic algorithms*, *second edition with CD* (255 pp.). New York: Wiley.

Haupt, S. E. (2006). Nonlinear empirical models of dynamical systems. *Computers and Mathematics with Applications*, *51*, 431–440.

Haupt, S. E., Haupt, R. L., & Young, G. S. (2008). A mixed integer genetic algorithm used in chem-bio defense applications, accepted by *Journal of Soft Computing*.

Holland, J. H. (1975). *Adaptation in natural and artificial systems*. Ann Arbor, MI: The University of Michigan Press.

Marzban, C., & Haupt, S. E. (2005). On genetic algorithms and discrete performance measures. *AMS 4th Conference on Artificial Intelligence*, San Diego, CA, paper 1.1.

Mitchell, M. (1996). *An introduction to genetic algorithms*. Cambridge, MA: MIT Press.