COMPUTATIONAL ECONOMICS

Genetic Algorithms and Evolutionary Games in MATLAB

David A. Kendrick
Ruben P. Mercado
Hans M. Amman

1 Introduction

Genetic algorithms are search procedures based on the logic of natural selection and genetics. The central concept of genetic algorithms is *survival*. A group of individuals—each one represented, for their computational implementation, by a string of characters, usually based on a binary code—compete with one another and the *most fit* survives to give birth to a next generation of related individuals. This process continues through a number of generations ending up with the *most fit* individual.

One application of genetic algorithms is in evolutionary game theory. In this chapter we present an example from this field, namely an iterated Prisoners' Dilemma problem.[1] First we illustrate some basic concepts of genetic algorithms and evolutionary games using very simple examples. Next we show how to work with binary representations in **MATLAB**. Then we present a basic **MATLAB** program and perform some experiments. A more sophisticated **MATLAB** program of genetic algorithms that builds on the one here is given in Chapter 12.

2 Genetic Algorithms at first glance

There are different types of genetic algorithms. Here we introduce one of the most commonly used. The algorithm starts with the generation of the initial population, and then we have a repetitive process that evaluates the fitness, selects, crosses, mutates, and replaces the old population. This process is stopped when we reach the required precision after a number of generations. In this section we present a simple example. Later we suggest ways of introducing more complex procedures.

Let us assume that we start from a given initial population of only four individuals. Each individual's characteristics are represented by a string of five binary digits (chromosomes):

Initial population

1. 00000
2. 10101

[1] The use of this example was initially motivated by the work of our student Shyam Gouri Suresh.

3. 11010

4. 11100

We are all used to dealing with decimal representation of numbers. For example, a number like 142 (one hundred and forty-two) is constructed as follows:

$$\begin{array}{l}(1)\ \ 10^2 + (4)\ \ 10^1 + (2)\ \ 10^0 = \\ 100\ +\ 4\ +\ 2\ =\ 142\end{array}$$

A binary representation works in a similar way, but with a different base: 2 instead of 10. Thus, a string of characters such as the second individual in the foregoing initial population, that is, 10101, can be interpreted as representing the number 21 (twenty-one), since

$$\begin{array}{l}(1)\ \ 2^4 + (0)\ \ 2^3 + (1)\ \ 2^2 + (0)\ \ 2^1 + (1)\ \ 2^0 = \\ 16\ \ +\ \ 0\ \ +\ \ 4\ \ +\ \ 0\ \ +\ \ 1\ \ =\ \ 21\end{array}$$

The usefulness of this kind of representation will be appreciated soon in the crossover and mutation steps.

Given the initial population, we need some fitness criteria in order to select the *best* individuals. This criterion depends on the specific problem under consideration, and it is usually represented by a mathematical function to be applied to each individual. For the sake of simplicity, let us assume here that the fittest individuals are those with the highest numerical values associated with their characteristics (their "chromosomes"). Thus, the third and fourth individuals in the foregoing initial population would be selected for reproduction, and they would form a couple. This couple has four children that replace the entire previous generation of individuals. Each new individual is generated in the following way: first there is a crossover of the last two genes in the strings of the selected couple as follows:

Couple Crossover

3)1101011000

4)1110011110

and then there is a mutation of the last gene for each of the crossover results. That is,

Crossover Mutation

1100011000

11001

1111011110

11111

Thus, from the mutation column above, the second generation ordered in an ascendant way according to numerical value is

Generation 2

1. 11000
2. 11001
3. 11110
4. 11111

We now select again the two fittest individuals, obviously the third and fourth. Then we can again apply the same crossover and mutation procedures with the following result:

Couple Crossover Mutation

11111

3)111101111111110

4)111111111011110

11111

Thus, the third generation, ordered in an ascendant way according to numerical value, is

Generation 3

1. 11110
2. 11110

3. 11111

4. 11111

Observe that we have reached the highest possible values for the third and fourth individuals, which are then selected as the parents of the next generation. Actually, if we repeat the crossover, mutation, and selection steps from now on, we find that all the next generations are identical to Generation 3. Thus, we can conclude that we have reached an optimum, that is, the fittest individual given the characteristics of our problem.

The example just presented is simple; yet it provides a basis from which we can introduce several modifications to provide an idea of what the actual practice in the field of genetic algorithms is like. For example, the size of the population could be larger or the string of characters corresponding to each individual could be longer. The initial population could be generated stochastically. The fitness criteria, as we mentioned above, could be of a different nature from the one used here and be represented by a specific fitness function. Given a larger population, more than one couple could be selected to be the parents of the next generation. To this end, a stochastic procedure could be used to form couples out of a pool, also determined with some degree of randomness, of the fittest individuals. The crossover point—the last two genes in our example—could also be randomly determined at each generation, as well as the mutating genes, which we arbitrarily chose to be only and always the last one.

Before we introduce the code for our model, we present a simple example of an evolutionary game and later introduce a number of **MATLAB** functions that can be used for manipulating binary representations.

3 A Simple Example of Evolutionary Game

An evolutionary game is one in which strategies evolve through a process of dynamic selection. As an example, we present here a simple version of the game known as iterated Prisoners' Dilemma. The game itself was introduced and analyzed in Chapter 9. Our game has the representation shown in Table 11.1, where D means defect and C means cooperate.

Player 2

Thus if the two individuals cooperate with one another they each receive a gain of 3 but if they both defect they each have a gain of only 1. In contrast, if

h

Table 1: Game Representation

	Player 1		D	C
Player 2		D	1, 1	5, 0
		C	0, 5	3, 3

Player 1 decides to cooperate but Player 2 defects, then Player 1 makes a gain of 0 and Player 2 a gain of 5.

We assume that this game is played many times by successive generations of individuals. Each individual is represented by a chromosome twenty-four bits long. Each gene of the chromosome represents an action (0 for defect, 1 for cooperate). Thus, each individual chromosome is interpreted as a strategy, which is a sequence of actions.

Within a given generation, each individual plays twenty-four times against each member of her generation following the strategy implied by her chromosomes, and her resulting payoffs are accumulated. At the end of each generation round, the two individuals with the highest accumulated payoffs are selected to be the parents of the next generation. Children are born out of the crossover and mutation of parents' chromosomes. The process is repeated for a number of generations.

Note that, given the simple formulation of this iterated game, individuals do not think and act strategically. They just follow their strategies regardless of their opponent's actions. In this sense they are very stubborn and simple-minded agents.

However, we know that the most efficient individual action for the Prisoners' Dilemma game is to defect, and that the unique Nash equilibrium is (D, D). The question for our experiments is: Would this population of simple-minded agents evolve in such a way that only defectors survive? That is, will the selective evolution of the population generate an outcome similar to the one that would be reached by rational and strategic-thinking players, which, by the way, implies that everybody will be worse off than in the case in which everybody cooperates?

4 Working with Binary Representations in MATLAB

When using binary variables to code genetic algorithms, the key concept to keep in mind is that the variables can be specified as integers but can also be thought of as strings of binary variables. Then, for example, the integer 25 would be represented in an eight-bit binary string as

$$00011001$$

that is, as

$$0\ \left(2^7\right)+0\ \left(2^6\right)+0\ \left(2^5\right)+\ 1\ \left(2^4\right)+1\ \left(2^3\right)+0\ \left(2^2\right)+0\ \left(2^1\right)+1\ \left(2^0\right)$$

$$=0\,(128)+0\,(64)+0\,(32)+1\,(16)+1\,(8)+0\,(4)+0\,(2)+1\,(1)$$

$$=25$$

Thus we can create an integer variable in the program, say genepool = 25, and use that to represent both a player's strategy (or, as we will see in Chapter 12, an economic variable such as the percentage weight of an asset in a portfolio) and at the same time manipulate it as a bit string in a genetic algorithm code.

MATLAB provides a variety of functions to manipulate the binary representations of numbers. The functions dec2bin, bitor, bitand, bitshift, and bitcmp are used in the genetic algorithm code both here and in Chapter 12.

The function dec2bin converts a decimal integer to a binary string. For example, the statement

```
x = dec2bin;
```

returns the binary string 110, which corresponds to the decimal number 6, while

```
x = dec2bin(6,8);
```

returns a binary representation of the decimal number 6 with at least eight characters, that is,

$$00000110$$

The function bitor returns the decimal bitwise OR of two nonnegative integer numbers. That is, it compares bit-to-bit each binary position of the two numbers generating a 1 whenever it finds the combination (1,1), (1,0), or (0,1) and generating a 0 when it finds the combination (0,0). For example, the four-bit representations of the numbers 7 and 9 are, respectively,

0111 and 1001,

and the bitwise OR operation on these numbers yields 1111, which corresponds to the decimal representation 15. Thus the statement

```
x = bitor(7,9);
```

returns the number 15.

The function bitand returns the bitwise AND of two nonnegative integer numbers. Comparing bit-to-bit the binary representation of two numbers generates a 0 whenever it finds the combination (0,0) or (0,1), or (1,0) and generates a 1 when it finds the combination (1,1). Thus the statement

```
x = bitand(7,9);
```

returns the decimal number 1. The function bitshift,

```
x = bitshift(A,k);
```

returns the value of the nonnegative integer A shifted by k bits (to the left when k is positive, to the right when k is negative and filled with zeroes in the new spaces). For example, when A = 14 its binary representation is 01110. Thus the statement

```
x = bitshift(14,1);
```

corresponds to the binary representation 11100 and returns the decimal value 28. If the shift causes x to overflow, the overflowing bits are dropped. Finally, the function bitcmp

```
x = bitcmp(A,n);
```

returns the bit complement of A in the form of an n-bit floating point integer, that is, each 0 is replaced with a 1 and vice versa. Thus, the statement

```
x = bitcmp(28,5);
```

where 28 is represented with five binary digits as 11100 and its complement is 00011 and thus the function returns the decimal value 3.

5 Overview of the MATLAB Code

There is a Genetic Algorithm and Direct Search Toolbox for use with **MATLAB** that includes routines for solving optimization problems using genetic algorithms and where the user can directly specify some overall characteristics of the particular problem (e.g., population size, fitness criteria, number of generations) without having to pay special attention to the workings and implementation of the genetic algorithm. However, to provide an opportunity to learn some basic concepts about genetic algorithms and to go deeper into learning the **MATLAB** software we introduced in Chapter 7, we are basing this chapter on a genetic algorithm code initially developed by one of our students, Huber Salas.

While in an earlier chapter we introduced relatively simple **MATLAB** programs, here we advance to a program that calls a number of built-in **MATLAB** functions and uses a number of M-files. **MATLAB** has two kinds of M-files that can be written by users: ?? *scripts*, which do not accept input arguments, and ?? *functions*, which do accept input arguments.[2] They contain a series of statements and can be stored in an independent **MATLAB** file. The functions receive a number of input variables, process them, and return one or more output variables. The name of the main program we present here is gagame.m, that is, it is a genetic algorithm evolutionary game problem. This program and all the functions it calls are available in the book web page.

The basic structure of the program, which follows, consists of three main parts: the first contains the initialization of counters and parameters and a function call to initialize the population; second is a for loop across generations that in turn contains several function calls; and the third contains commands to print and graph the main results.

```
% initialization of counters and parameters;
nruns = 100;   popsize = 8;
clen = 24;     pmut = 0.5;

% generation of chromosome strings of initial population
genepool= initpoprand\_gagame(popsize);
for k = 1:nruns;

% computation of fitness function and fittest individual
```

[2]For a discussion of **MATLAB** M-files see the manual "Getting Started with **MATLAB**" in the **MATLAB** Help menu options.

```
        [fit, bestind, bestfit] = fitness\_gagame(genepool,popsize,clen);
        wbest(k) = bestind;
        fbest(k) = bestfit;

% selection of parents;

        [parent0,parent1] = parentsdet(fit,genepool);

% crossover of parents chromosome strings

        [child0,child1] = crossover(clen,parent0,parent1);

% mutation of children chromosome strings

                for h = 1:2:popsize;

     child0mut = mutation(pmut,clen,child0);
     genenew(h) = child0mut;
     child1mut = mutation(pmut,clen,child1);
     genenew(h+1) = child1mut;

        end
        genepool = genenew;
 end

 % print and graph fittest individual;

 dec2bin(wbest(nruns),clen)
 fbest = fbest / (clen * (popsize - 1));
 xaxis = [1:1:nruns]';
 plot(xaxis,wbest);
 xaxis = [1:1:nruns]';
 plot(xaxis,fbest);
```

In the initialization of counters and parameters section we set the number of runs nruns and the population size popsize. We also set the length of the chromosome string clen and the probability of a child mutation pmut.

We then call the function inipoprand_gagame to initialize the vector genepool, which contains a number of individuals equal to the population size, each indi-

vidual represented by a twenty-four-bit chromosome string. So, in our example, genepool is a vector of twenty-four-bit chromosomes with an element for each of the eight individuals in the population.

Then we move on to the main for loop in the program, running from 1 to the number of runs. This loop contains a sequence of function calls. It starts with a call to the fitness_gagame function to compute the fitness function for each individual and to select the fittest individual, which at each run is stored in the kth element of the vector wbest, while the corresponding criterion value is stored in the kth element of the vector fbest. Thus, at the end of the runs, these vectors contain the sequence of optimal chromosome strings and optimal criterion values, respectively.

Next the function parentsdet, using the fitness function previously computed, selects two parents (parent0 and parent1), who form a couple. This is followed by a call to the function crossover, which generates two children (child0 and child1) as the product of the crossover of the chromosome strings of the two parents.

Next comes a for loop whose index goes from 1 to popsize in increments of two. In this loop, out of the two newborn children a new generation is created through mutations of their chromosomes. Half of the new generation comes out of mutations of the first child (child0) and the other half out of mutations of the second (child1). At every pass through the h loop the function mutation is called twice. This has the effect of generating two mutated children, and their twenty-four-bit chromosome representations are stored in subsequent cells of the genenew vector.

Once the new generation is created, the new vector genenew replaces the old vector genepool and the main loop of the program starts over again.

After the main loop goes through the established number of runs, the statement

```
dec2bin(wbest(nruns),clen)
```

prints the last element of the vector wbest, which contains the chromosome string of the fittest individual. Then the statement

```
fbest = fbest / (clen * (popsize - 1));
```

is used to compute the average value of the optimal criterion value at each run. Note here that fbest is a vector and (clen * (popsize – 1)) is a scalar so that the division operation is repeated for each element in the vector. Finally, the vector of fittest individuals, wbest, and the vector of optimal criterion values, fbest, are plotted.

This provides an overview of the program. It is important to point out that every time you run the program, particularly when changing the number of generations or the population size, you should clean out the old commands and workspace to avoid displaying spurious results. To do so, go to Edit in the top **MATLAB** menu. Then select Clear Command Window and confirm with Yes that you want to do this. Then do the same for Clear Command History and for Clear Workspace.

We now present each function in detail.

6 Functions

6.1 Initpoprand_gagame

This function is simple in that all it does is assign a random number to each individual string of chromosomes. Thus the **MATLAB** code for this initialization function is

```
function genepool= initpoprand_gagame(popsize,clean);
    for k1 = 1:popsize;
        genepool(k1) = ceil(rand * (2^clen)-1);
        % genepool(k1) =  (2^clen)-1;
        dec2bin(genepool(k1), clen)
end

function genepool = initpoprand_gagame(popsize,clen);
```

tells us that the name of the function is initpoprand_gagame, that the arguments popsize and clen are passed to the function, and that the result, genepool, is returned by the function.

Then a loop going from 1 to popsize is used to assign a random value to each element of the genepool vector. The statement

```
genepool(k1) = ceil(rand * (2\^{}clen)-1)));
```

assigns to each element of the vector the ceiling (the nearest higher integer value) of the result of multiplying the variable rand (a zero-one uniform distribution random number generator) times the number (2^clen)-1. This last number is equal to the highest possible value represented with a binary string of length equal to clen. For example, if clen equals 3, that number is equal to 23 minus one, that is, 8 minus one. Thus the number is 7, whose binary representation is 111.

The statement

```
dec2bin(genepool(k1),clen)
```

does not play an essential role in the function since it only serves to print a binary representation of genepool for debugging purposes. Since there is no semicolon at the end, this statement returns and prints the twenty-four-bit binary representation of each element of genepool. Finally, note that an end statement is not necessary at the end of a **MATLAB** function, unlike the cases of for loops or conditional if statements.

6.2 Fitness_gagame

This fitness_gagame function contains the game to be played and a procedure to select the fittest individual that is similar to the one used in Chapter 7. The first part of the function consists of three nested loops: the first one for Player 1, the second for Player 2, and the third for games. Thus, each player selected in the first loop plays against every other player selected in the second loop. These two players play twenty-four games, playing in each game the action determined by the corresponding gene in their chromosome sequence. The statements for the first part of the function are as follows:.

```
function [fit,bestind,bestfit] = fitness_gagame(genepool,popsize,clen);

payoffs(1,popsize) = 0;

% Loop for player1
for k1 = 1:popsize;
    strategyp1 = genepool(k1);

    % Loop for opponents (player2)
    for k2 = 1:popsize;
    strategyp2 = genepool(k2);
```

```
        if (k1 ~= k2)
            mask = 1;

            %Loop for games
            for k3 = 1:clen;
            actionp1 = bitand(strategyp1,mask);
            actionp2 = bitand(strategyp2,mask);
            mask = bitshift(mask,1);
                % defect, defect
                if (actionp1 == 0) & (actionp2 == 0)
                    payoffs(k1) = payoffs(k1) + 1;
                end
                % cooperate, defect
                if (actionp1 > 0) & (actionp2 == 0)
                    payoffs(k1) = payoffs(k1) + 0;
                end
                % defect, cooperate
                if (actionp1 == 0) & (actionp2 > 0)
                    payoffs(k1) = payoffs(k1) + 5;
                end
                % cooperate, cooperate
                if (actionp1 > 0) & (actionp2 > 0)
                    payoffs(k1) = payoffs(k1) + 3;
                end
            end % end loop games

        end % end if

    end % end loop opponents

end % end loop player1

fit = payoffs;
[top topi] = max(fit);
bestind = genepool(topi);
bestfit = top;
```

The function begins with the statement

```
payoffs(1,popsize) = 0;
```

which initializes to zero a vector that contains the accumulated payoffs of Player 1. This is followed by the beginning of the k1 loop for Player 1. The statement at the beginning of this loop

```
strategyp1 = genepool(k1);
```

and the statement at the beginning of the loop for Player 2

```
strategyp2 = genepool(k2);
```

assign to the temporary variables strategyp1 and strategyp2 the chromosomes of Player 1 and Player 2, respectively, that is, the strategies each player is to play.

Next, at the beginning of the loop for games, the statements

```
actionp1 = bitand(strategyp1,mask);
actionp2 = bitand(strategyp2,mask);
```

select the actions to be played in each game out the strategies of each player. The variable mask was previously initialized with the value 1. Thus, its twenty-four-bit binary representation is

000000000000000000000001

Remember that the function bitand returns the bitwise AND of two nonnegative integer numbers. Thus, comparing bit-to-bit the binary representation of two numbers, it generates a 0 whenever it finds the combination (0,0) or (0,1) or (1,0) and generates a 1 when it finds the combination (1,1). Thus the temporary variables actionp1 and actionp2 contain the first gene of each player's chromosome, that is, the first action to be played in the first game. For example, if the chromosome of Player 1 is

010101011111111100001111

the result of the bitand operation is

000000000000000000000001

and the content of the actionp1 variable is the number 1, so that the player cooperates. The statement

```
mask = bitshift(mask,1);
```

shifts the mask one position to the left, resulting in

00000000000000000000010

Thus, at each pass of the loop, the number 1 shifts one position to the left so that the next action is selected. The remaining of the loop for games, that is, the k3 loop, accumulates the payoffs for Player 1 depending on the result of the game. Each of the four possible outcomes is evaluated. For example, in the case that Player 1 plays 0 (defect) and Player 2 plays 1 (cooperate) the sentences

```
% defect, cooperate
if (actionp1 == 0) \& (actionp2 > 0)
payoffs(k1) = payoffs(k1) + 5;
```

add 5 to the element of the payoffs vector corresponding to the game being played.

Once the main loop of the function—the loop for Player 1—is completed for all players, the second part of the function selects the fittest

```
fit = payoffs;
```

to assign the variable payoffs to the temporary variable fit. The statement


```
[top topi] = max(fit);
```

then returns the value (top) and the index (topi) corresponding to the maximum value in the fitness vector fit. Finally, the topi index is used to assign to the *best individual* vector, bestind, the corresponding chromosome with the statement

```
bestind = genepool(topi);
```

while the corresponding value of the criterion function vector is assigned to the variable bestfit with the statement

```
bestfit = top;
```

With the fitness now determined we turn next to the selection of parents.

6.3 Parentsdet

The parentsdet (parents deterministic) function is a simple deterministic function that selects the two individuals that will be the parents of a new generation. The simple selection method used in the present function is to select the two individuals with the highest criterion value or fitness:

```
function [parent0,parent1] = parentsdet(fit,genepool);
[top topi] = max(fit);
parent0 = genepool(topi);
fit(topi) = 0;
[top topi] = max(fit);
parent1 = genepool(topi);
```

As in the previous function, the statement

```
[top topi] = max(fit);
```

returns the index topi corresponding to the maximum value of the fit vector, that is, the fittest individual. Using that index, the corresponding chromosome string of the first parent is stored in the variable parent0. To select the second parent, we set the fitness value of the previous maximum to zero and proceed in the same manner as before, now to select the second parent (parent1). We will see in Chapter 12, in the parentsrand function, how to implement a more sophisticated random procedure for parent selection in which more fit parents have a "higher chance" of generating offspring.

6.4 Crossover

The crossover mixes the chromosome information of the two parents to create two children. We consider here only the case of a single crossover. The function code is as follows:

```
function [child0,child1] = crossover(clen,parent0,parent1);
crossov = round(rand*(clen-1));
maska=1;
for j = 1:(crossov-1)
    maska = bitshift(maska,1);
    maska = maska + 1;
end

% add by HA because of code change in matlab wrt bitcmp
```

```
maska=int64(maska);
child0 = bitor (bitand(parent0, maska), bitand(parent1, bitcmp(maska)));
child1 = bitor (bitand(parent1, maska), bitand(parent0, bitcmp(maska)));
```

To determine the crossover point we use the statement

```
crossov = ceil(rand *(clen));
```

Remember that the clen variable contains the chromosome length of the individual (24 bits). Then, multiplying clen times the uniform zero-one random number generator function rand we are randomly choosing the crossover point. Since that point has to be an integer number, we apply the function ceil to the result, which rounds off the result to the nearest higher integer.

Next we assign the initial value 1 to the mask variable maska:

000000000000000000000001

Then we pass through a loop that goes from 1 to the crossover point. At each pass, we shift the 1 one position to the left,

000000000000000000000010

and add the value one to the result, thus switching the rightmost bit from zero to one:

000000000000000000000011

Thus if the crossover point was 12, we end up with the mask

000000000000111111111111

Next we generate the first child with the statement

```
child0 = bitor (bitand(parent0, maska),
                bitand(parent1, bitcmp(maska,clen)));
```

For example, consider the case where parent0 has the chromosome string

000100010001000100010001

In this case the statement

```
bitand(parent0, maska)
```

would apply the mask

000000000000111111111111

and create the chromosome string

000000000000000100010001

Assume as well that parent1 has the following chromosome string:

100010001000100010001000

Thus the statement

```
bitand(parent1, bitcmp(maska,clen)));
```

would apply the complement of maska, that is,

111111111111000000000000

to parent1 with the bitand operation to obtain

100010001000000000000000

Finally, the application of the bitor function to the chromosome strings

000000000000000100010001

100010001000000000000000

generates the result

100010001000000100010001

which is the chromosome string of the first child (child0). The second child is obtained in a similar fashion, reversing the position of the parents in the corresponding statement.

6.5 Mutation

The mutation function generates a random mutation in a single bit of the chromosome string of a child. The code is as follows:

```
function f = mutation(pmut,clen,child)
tt = 1;
if (rand < pmut)
    idx = round(rand*(clen-1));
    tt = bitshift(tt,idx);
    temp = bitand(child,bitcmp(tt,clen));

    if(temp==child)
        child = child + tt;
    else
        child = temp;
    end
end
f = child;
```

Recall that pmut is the probability of a child mutation. The tt variable is initially set to one and is bit shifted to create the mutation at the desired point in the chromosome. The scalar integer idx is the index of the location that is one less than where the mutation occurs. It is determined by rounding off the randomly generated location using the zero-one uniform random variable rand and the length of the chromosome clen less one.

Consider, for example, a case where the index variable is set to three, and recall that the variable tt is set to one. Thus the bitshift function call

```
tt = bitshift(tt,idx);
```

shifts the tt binary variable three positions to the left so that it becomes

1000

and the mutation is going to be done in the fourth position. Then the mutation is done with the statement

```
temp = bitand(child,bitcmp(tt,clen));
```

and the result is stored it in the temp variable. Consider first just the bitcmp part of this statement. It yields the twenty-four-bit complement to the tt variable, which is

111111111111111111110111

Then the bitand operation is applied to this bit string and the child variable to obtain the mutated string, which is stored in the temp variable. The bitand operation produces the desired mutation if the bit to be changed was a one. However, it does not produce the correct result if the bit to be changed was a zero. Therefore it is necessary to add the following lines of code:

```
if(temp==child)
    child=child+tt;
else
    child=temp;
end
```

In the case where the bit to be changed was a zero the foregoing bitand operation produced no change in the chromosome and it is necessary to accomplish that by adding an integer amount tt to the variable. In our case the binary representation of the tt variable was

1000

so its integer value is 16. Then when 16 is added to the child variable it produces the desired mutation by changing the zero bit in the fourth location to a one bit.

On the other hand, if the temp variable is not equal to the child variable—as occurs when the bit to be changed is a one—then it is only necessary to set the child variable equal to the temp variable. This completes the discussion of the mutation function and indeed the discussion of all the functions and leaves us free to turn our attention to the results obtained by using the program.

7 Results

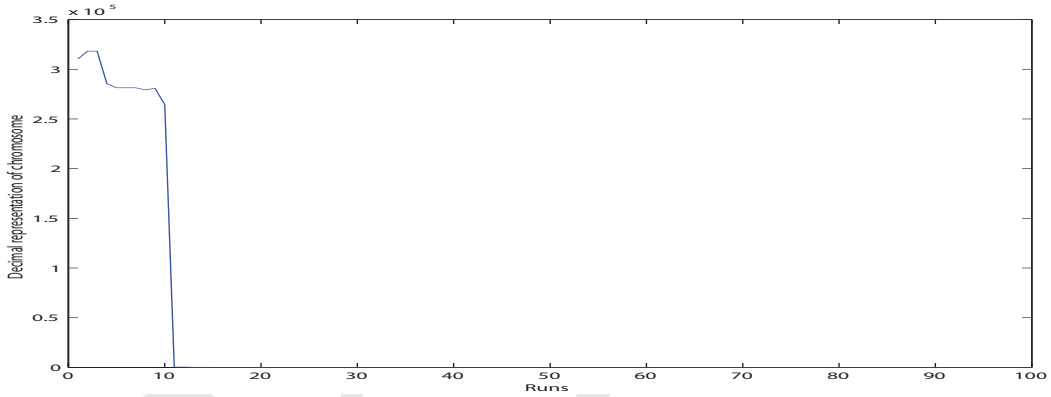
Figure 11.1 shows the results of running the main program gagame.m with a number of runs equal to 100 and a population size of eight, starting from a random initial population. The first graph shows the decimal representation of the chromosome of the fittest individual at each run. We can observe that after

about ten runs this value converges to zero and stays there. This corresponds to the chromosome

000000000000000000000000

Thus, the optimal strategy that results from the simulation is to always defect. The second graph shows the evolution of the corresponding average payoffs. We can see how these payoffs converge to a value near one, which is the value corresponding to the Nash equilibrium of the game.

Figure 1: Runs



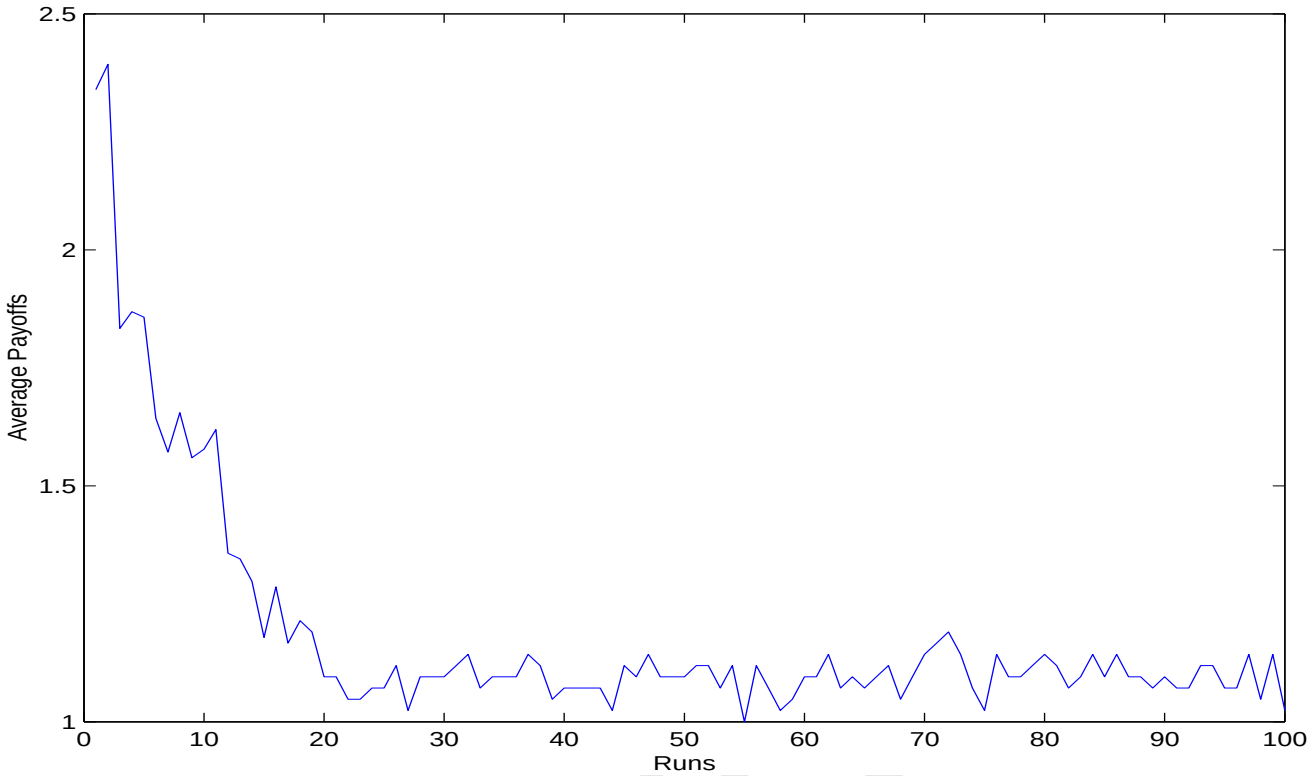
Figures 1 and 2 show the results of an experiment in which the initial population is composed entirely of cooperators. That is, individuals with a chromosome equal to

111111111111111111111111

To run this experiment, in the initpoprand.m function we have to replace the statement

```
genepool(k1) = ceil(rand * (2\^{}clen)-1);
```

Figure 2: Evolutionary game with random initial population.



with the statement

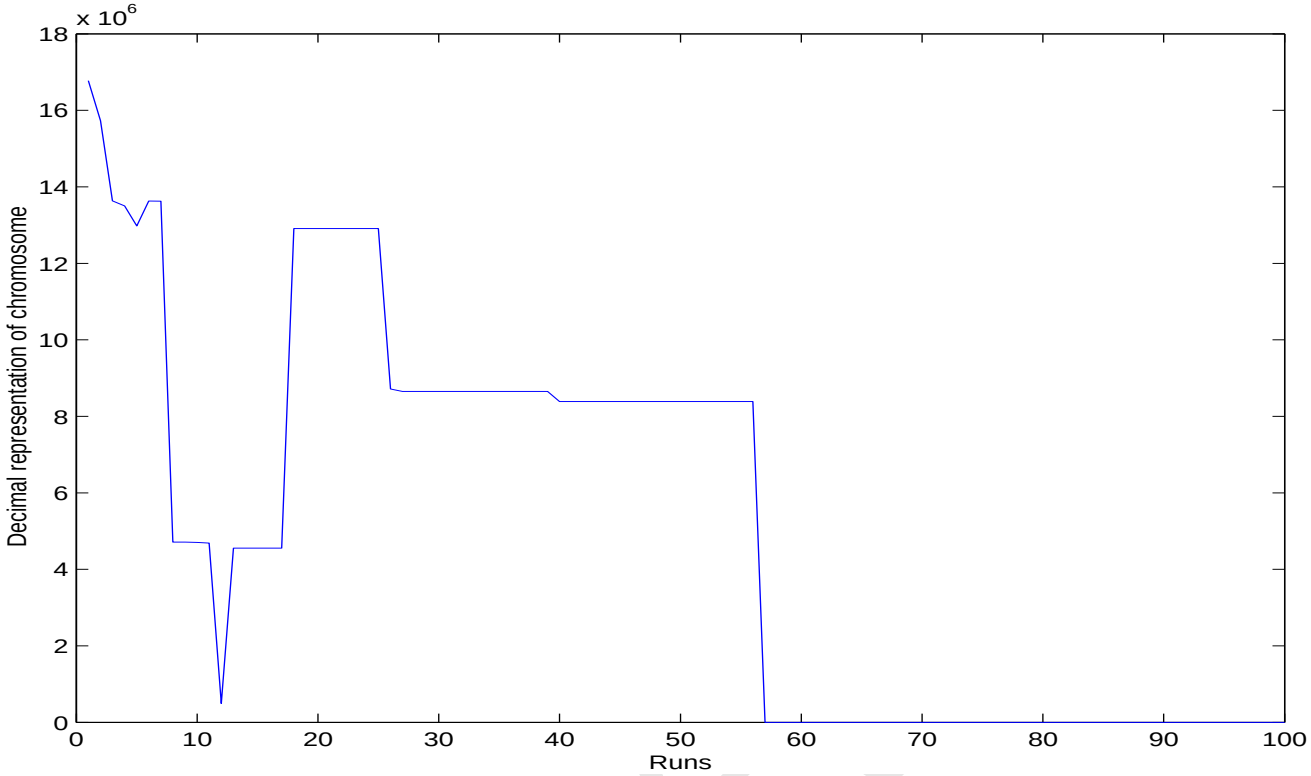
```
genepool(k1) = (2\^{}clen)-1;
```

We can see in Figures 3 and 4 that the results converge, at a slower pace than in the previous experiments, to the same outcome. The fittest individuals are defectors, born out of mutations and successive selections across generations. Interestingly, a population of all cooperators, thus achieving the higher possible payoffs, when suffering even mild mutations such as the ones implied by our MATLB code, ends up transformed into one of all defectors with an inferior standard of living.

8 Experiments

The simplest experiments with this genetic algorithm would be to change the number of model iterations and/or population size to see how this affects the

Figure 3: Runs

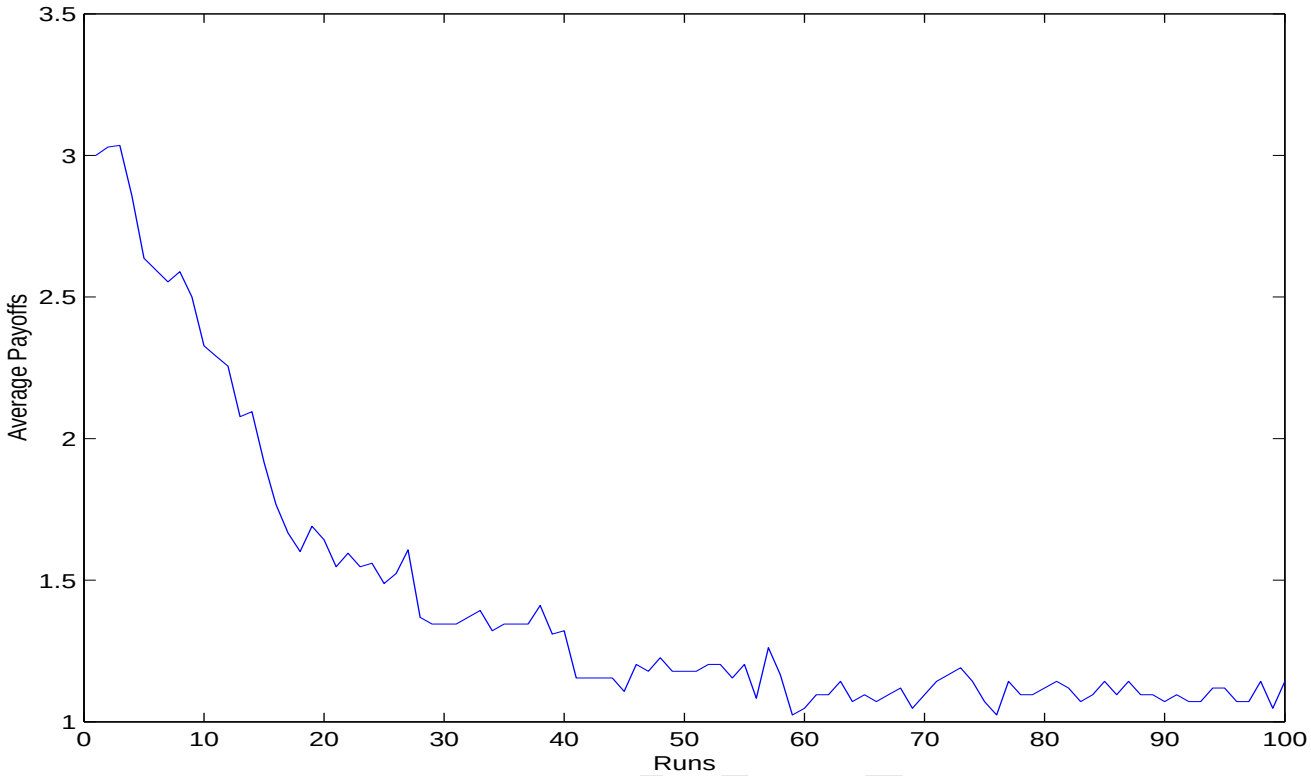


outcome. You may also want to try an experiment in which the initial population is composed of all defectors and see if they ever become all cooperators.

A more challenging set of experiments is to introduce further refinements in the code to move closer to the actual practice in the field of genetic algorithms, such as the random selection of parents and the selection of more than one couple to be the parents of the next generation. Before doing so, you are encouraged to read the Lecture on *Genetic Algorithms for solving Portfolio Models*, where these refinements are introduced.

More interesting experiments that would get you closer to the practice in the field of evolutionary games involve some sort of strategic thinking and behavior on the part of players. Instead of being taken regardless of the opponent's actions, a player's actions are determined, for example, as reactions to the opponent past behavior, Axelrod (1997). It would also be interesting to explore the evolutionary dynamics of a spatial model of local interaction in which each individual plays the Prisoners' Dilemma with her neighbors, Nowak and May

Figure 4: Evolutionary game with initial population of cooperators



(1992, 1993).

In both these cases—more sophisticated strategies or local interaction—it is found that the evolutionary behavior differs from the convergence to all defectors we found in this chapter. Indeed, it is usually the case that the evolution converges to cooperation or even displays complex patterns of cyclical behavior.

Since the **MATLAB** representation of these models may be more demanding than the one presented in this chapter, before moving in this direction you are encouraged to read the Lecture *Sugerscape Agent-Based Models* to learn about more sophisticated modeling techniques that may be useful to program problems of this nature.

9 Further Reading

A classic reference in the genetic algorithms literature is Goldberg (1989). For introductions to evolutionary games see the *Stanford Encyclopedia of Philosophy*

at http://plato.stanford.edu/ and Axelrod (1997).

References

Axelrod, R.: 1997, *The Complexity of Cooperation: Agent-Based Models of Competition and Collaboration*, Princeton University Press, Princeton, New Jersey.

Goldberg, D.: 1989, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Cambridge, Massachusetts.

Nowak, M. A. and May, R. M.: 1992, Evolutionary games and spatial chaos, *Nature* **359**, 826–829.

Nowak, M. A. and May, R. M.: 1993, Evolutionary games and spatial chaos, *International Journal of Bifurcation and Chaos* **3**, 3578.