Contents

hawk or a dove. The game is frequency-dependent because although a hawk interacting with a dove has a higher fitness than the dove, a hawk interacting with another hawk suffers a decrement in fitness. The equilibrium frequency of hawks and doves in the population depends upon the relative values in the payoff matrix and is called an ESS. It is obtained simply by equating the payoff to hawks with the payoff to doves: at equilibrium the two must be equal. In simple terms an ESS is one that cannot be invaded by a mutant playing an alternate strategy (see Hammerstein [1998] for a more formal definition). Game theoretic models are discussed in detail in Chapter 6.

1.3 Some general principles of model building

Models are not replicas of nature: If they were they would be just as complicated and equally hard to understand. The purpose of a model is to extract the essential elements that define the problem under study. Having done this we investigate the impact of the model components and compare the predictions of the model with nature. Should there be an obvious discrepancy we return to the model and examine the underlying assumptions: A model is simply the logical outcome of the assumptions and thus any failure to fit reality is a failure of the assumptions. Having modified the model we again compare predictions and observations, repeating the process until a satisfactory fit is obtained.

In constructing a model the following should be kept very much to the fore:

1. Keep the model as simple as possible and focus upon the problem. Modeling the mechanism for telling time provides an instructive example of this process. The modern digital watch is a highly complex affair and seemingly vastly different from the earliest mechanical clocks. Further, when one looks at the history of clocks and watches one sees an enormous variety of mechanisms. Yet under all this complexity and variety, all mechanical or electrical clocks have five elements in common that determine how time is monitored: "(1) a source of energy (spring or battery); (2) an oscillating controller (balance or quartz crystal); (3) a counting device (escapement or solid state circuit); (4) transmission (wheelwork or electric current); (5) display (hands or liquid crystal segments)" (Landes 1983, p. 377). All mechanical or electrical clocks must satisfy these requirements. Thus to find out how a clock works one must strip away the extraneous details such as the size of the clock, whether it gives the date or altitude or compass direction and look for these five preceding elements.
2. Make assumptions explicit. Verbal models are frequently "preferred" because they seem less confined than a mathematical model but in reality verbal models are generally full of "hidden" assumptions that may well result in any conclusions to come crashing down once these assumptions are noted. In this book I adopt the policy of beginning with a general conceptual model and then move to a mathematical construct based on the general assumptions. For example, we might assume that there is a negative relationship between the size and number of offspring that a female produces. This statement is very

general and might be sufficient in some analyses but most cases an analysis will require a more detailed specification such as that the number of offspring is proportional to the reproductive biomass divided by offspring size.

3. This book is primarily concerned with numerical analysis of models: If an analytical solution is possible, then it is to be preferred. Such solutions may be possible only on very simplified versions of the model and numerical analysis of more complex scenarios may reveal inadequacies in the simple analytical solution.
4. While simplicity is desirable it is important to maintain a reasonable level of realism. In this regard it is important to provide operational definitions of all parameters and variables in the model. If a variable cannot be measured, then it is not useful and an alternate approach should be sought.
5. As much as possible, write the model incrementally and as a series of modules that can be examined and debugged separately.

To illustrate these points the next section constructs a model of the evolution of migration in a spatially and temporally heterogeneous environment.

1.4 An introduction to modeling in R and MATLAB

The purpose of this section is twofold: First, it is to outline, by using a simple example, the process of creating a model to address an evolutionary question, and second to illustrate the most important R and MATLAB codes used in the remainder of the book.

The problem we shall consider is that of the evolution of migration in a heterogeneous environment. As used in all the scenarios throughout this book we begin first by outlining a conceptual model and then convert this model into one that can be programmed.

1.4.1 General assumptions

1. The environment is heterogeneous in time and space.
2. This heterogeneity affects population dynamics by causing variation in the vital statistics of the population (e.g., fecundity and survival) and the carrying capacity of the environment.

These two assumptions are too general to be programmed as such and must be converted into a suitable form by addressing the underlying mathematical assumptions, which will necessarily restrict the model to some extent. While we could pose a mathematical model that included the processes outlined above it would include factors, such as age structure, that may not be important to the central issue but could complicate the analysis. Thus to start we begin with a very simple model and ask if in this case spatial and temporal heterogeneity could be an important selective agent. This does not prove that such variation is an

important selective agent but does demonstrate that an empirical investigation is warranted.

Our first objective is to examine the hypothesis that environmental variation is plausibly a significant factor in population persistence: If we find this to be the case then it would seem reasonable to suppose that such variation will favor particular life histories, the next step being then to examine what trait might be favored. As noted earlier, we build the computer program incrementally, ensuring that at each step the model is performing as specified by the mathematical assumptions. We begin with the simplest possible model, assuming no environmental variation and then add temporal variation. Our initial model assumes the following.

1.4.2 Mathematical assumptions of model 1

1. There is no age structure.

2. Generations do not overlap.

3. The environment is constant in space and time.

4. Growth per generation is a constant.

An appropriate mathematical model given the above is

$$N_{t+1} = \lambda N_t \tag{1.31}$$

where N_t is the population size at time t and λ is the per generation rate of increase. The above equation is called a **recursive equation**. To program this in R or MATLAB we proceed as follows.

Step 1: Clearing memory

One of the advantages of R and MATLAB is that values are retained in memory even after the program has finished. This can be very useful in that it allows programs to be run sequentially, where one program utilizes the output of the preceding program (e.g., one program might generate values and the second program display them graphically). On the other hand, it can cause problems if one runs another unrelated program that contains parameters with the same name but which have not, due to error, been assigned values (e.g., suppose one ran a program that contained the parameter `Afit` and then a second program that also contained `Afit` but this parameter was inadvertently not assigned a value). In this case the program will pick up the wrong parameter values, most probably leading to incorrect solutions. Unless one wishes to retain values in memory, the best practice is to wipe the memory at the start of each program by having the first line of coding read:

R CODE:
```
rm(list=ls())
```

MATLAB CODE:
```
clear all
```

Step 2: Annotating programs

At the time of writing a computer program the structure and logic might (should) appear clear. However, upon returning to the code after a week or so it is a common experience that the lines of coding have reached a level of obscurity that may necessitate considerable time and effort in clarifying. It is thus very important to annotate the program to a degree that may well seem absurd while constructing the original code. In general, every line of code should have an annotation. Blocks of code that carry out a particular operation should also be annotated at the beginning with a description of the process. In both R and MATLAB remarks can either be on their own line or on the same line as but following a coding instruction. Remarks in R are designated by # and in MATLAB by %. I also like to try to align the text in the coding for ease of reading. Thus for the above two codes clearing memory one should type

R CODE:
```
rm(list=ls())     #  Clear memory
```

MATLAB CODE:
```
clear all         % Clear memory
```

Step 3: Assigning values to parameters and variables

A parameter is defined by the Oxford dictionary as a "quantity constant in case considered, but varying in different cases" whereas a variable is "able to assume different values." Thus in equation (1.31), λ is a parameter but N is a variable. However, variables are considered as parameters when passed to a function (discussed in Step 8), which makes the definitions somewhat murky. The assignment of values to parameters and variables is the basic operation in any program. Consider the task of assigning the value 3 to a variable X. In the usual mathematical notation we write $X = 3$. This is the method used in MATLAB but in R and S-PLUS the "=" sign is replaced by an arrow "<−". (The "=" sign can be used in R but it has a more restricted definition than "<−", as described in the R help dialogue: "The operators <− and = assign into the environment in which they are evaluated. The operator <− can be used anywhere, whereas the operator = is only allowed at the top level [e.g., in the complete expression typed at the Code prompt] or as one of the subexpressions in a braced list of expressions.")

Thus in R we write `X <- 3`. In like manner any operation on the right is assigned to the variable on the left: for example, $X = a + b$, where a and b are previously assigned parameter values of, say, 1 and 4, respectively, is written as follows:

R CODE:
```
a  <- 1          # Assign the value of 1 to a
b  <- 4          # Assign the value of 4 to b
X  <- a + b      # Assign the sum of a and b to X
```

MATLAB CODE:
```
a = 1;      % Assign the value of 1 to a
b = 4;      % Assign the value of 4 to b
X = a + b;  % Assign the sum of a and b to X
```

Notice that in the MATLAB statements each line before the comment statement is ended with the symbol ";". If this symbol is not appended to the line MATLAB echoes the result of the assignment statement. While this can be a simple and convenient method to print results, it can give very messy output when there are a lot of lines of coding and iterations.

It is good practice to make the names of parameters and variables meaningful so that the code is not too obscure. In the present case we need to assign the number of generations the model will run, the rate of increase, and the initial population size. Now it is possible to insert the first two values in all the relevant locations in the program, but a better approach is to assign the values to parameters, which means that we need only change a single line when changing either value. This is not only easier than altering all lines but eliminates the problem of missing a line and having different values in different parts of the program.

R CODE:

```
MAXGEN  <-  100     # Set maximum number of generations
N.init  <-  20      # Initial population size
LAMBDA  <- 1.1      # Rate of increase
```

MATLAB CODE:

```
MAXGEN  = 100;      % Set maximum number of generations
N.init  = 20;       % Initial population size
LAMBDA  = 1.1;      % Rate of increase
```

Step 4: Creating space to store the output: c(...), vectors, matrices, etc.

For any model there will be information that is generated by the program that we will want to analyze at the end of the simulation. While it is possible to dynamically allocate space, a better method is to preassign the space at the start of the simulation. Information can be stored in a matrix, a vector, an array, a data frame, or a list.

A **matrix** is a two-dimensional (2-D) structure that contains only information of the same type (e.g., only numerical information). A **vector** is simply a matrix with a single column or row. Examples of a vector and a matrix are as follows:

$$\texttt{A.vector} = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \qquad \texttt{A.matrix} = \begin{bmatrix} 1 & 6 & 0 \\ 2 & 4 & 2 \\ 4 & 8 & 1 \end{bmatrix}$$

To assign 1, 3, 5 to the vector `A.vector` we can use the concatenate code c(...) in R and square brackets in MATLAB

R CODE:

```
A.vector <- c(1, 3, 5)     # Assign values
A.vector                   # print result
```

MATLAB CODE:

```
A.vector = c[1, 3, 5]  % Assign values and print result
```

which will produce the row vector `1 3 5`, or we can use the R matrix code

```
A.vector <- matrix(c(1,3,5), nrow=1, ncol=3)
```

which will produce the same output. The designators `nrow=` and `ncol=` can be omitted as R uses the position to determine which are the row and column counts (putting `nrow=` and `ncol=` in the code does make reading easier). To produce a column vector we can simply switch row and column counts

```
A.vector <- matrix(c(1,3,5), nrow=3, ncol=1); A.vector
```

Note that in the above construct the two commands are entered not on separate lines but separated by a ";": this can be convenient in compressing code. To create the matrix `A.matrix` we first note that in R the default for filling in a matrix is to fill by columns and hence the sequence of entries is given column-wise

```
A.matrix <- matrix( c(1,2,4,6,4,8,0,2,1),3,3); A.matrix
```

which produces the output

```
     [,1] [,2] [,3]
[1,]    1    6    0
[2,]    2    4    2
[3,]    4    8    1
```

An **array** is an extension of the matrix in that there can be more than two dimensions. A **data frame** is like a matrix except that it can contain data of different modes: For example, one column might contain character data such as population names and another column could contain numeric data. Data frames are used extensively in statistical analysis but most of the programs in this book use matrices, because the output is typically numeric only. Finally, a **list** is a construction that concatenates a variety of information. Most statistical output in R comes as a list which can be deconstructed to obtain the relevant pieces of information: for more on lists, see Steps 11 and 12.

In the present case we want to store the population size at each generation. There are several possible ways to do this: we shall consider two.

Approach 1: Two vectors

We create two vectors, one that holds the generation number and the second that holds the population size. We know that the generations will run from 1 to `MAXGEN` and hence we can use the following codes:

R CODE:

```
Generation  <- seq(from=1, to=MAXGEN) # Generation vector
```

MATLAB CODE:

```
Generation = 1:MAXGEN;            % Generation vector
```

To create the vector for population size we first create a matrix with 1 column filled with zeros and then insert our initial population size in the first space.

R CODE:

```
Npop    <- matrix(0,MAXGEN,1) # Generation vector
Npop[1] <- N.init             # Store initial population size
```

MATLAB CODE:

```
Npop    = zeros(MAXGEN);  % Generation vector
Npop(1) = N_init;         % Store initial population size
```

Approach 2: One matrix
An alternate approach is to create a matrix, which I shall call `OUTPUT`, that has `MAXGEN` rows and two columns, the first holding the generation number and the second the population size. This can be done in a single call but for clarity I prefer splitting the process

R CODE:

```
OUTPUT     <- matrix(0,MAXGEN,2)   # Pre-assign output space
OUTPUT[,1] <- seq(from=1, to=MAXGEN)# Assign gen nos to col 1
OUTPUT[1,2]<- N.INIT               # Assign initial popn size
```

MATLAB CODE:

```
OUTPUT      = zeros(MAXGEN,2);  % Pre-assign output space
OUTPUT(:,1) = 1: MAXGEN);       % Assign gen nos to col 1
OUTPUT(1,2) = N_INIT;           % Assign initial popn size
```

Step 5: Iterating over generations: loops

The use of loops is discouraged in any programming language: This is not because loops are intrinsically bad (in fact, they are frequently the most obvious way of writing code) but because no one has come up with a method of making them efficient in terms of speed. R and MATLAB are object-oriented languages and hence in many cases loops can be replaced with an object-oriented approach: For example, suppose we have a vector, `X`, of `N` values to which we wish to add the value 3. Using a loop we can write

R CODE:

```
for ( i in 1: N) {X[i] <- X[i]+3}      # Add 3 to X
```

MATLAB CODE:

```
for i  = 1:N          % ; not required here
X(i)   = X(i) + 3;    % Add 3 to X
end                   % end loop
```

In both R and MATLAB the above construct can be replaced by

R CODE:

```
X <- X + 3
```

MATLAB CODE:

```
X = X + 3;
```

However, recursive equations are best dealt with using a loop structure. In the present case, we wish to iterate from 1 to MAXGEN applying the recursive formula of equation (1.31). I have omitted the remark statement.

R CODE:

```
    for (i in 2:MAXGEN){Npop[i]      <- LAMBDA*Npop[i-1]}
OR  for (i in 2:MAXGEN){OUTPUT[i,2] <- LAMBDA*OUTPUT[i-1,2]}
```

MATLAB CODE:

```
for i    = 2:MAXGEN
Npop(i) = LAMBDA*Npop(i-1);
end
```

```
OR  for i = 2:MAXGEN
    OUTPUT(i,2) = LAMBDA*OUTPUT(i-1,2);
    end
```

Step 6: Plotting the results: 2-D graphs

In general, a graphical output is desirable to see if there is anything obviously wrong with the program. There are many "bells and whistles" that can be added to the graph. The default is a graph that plots the x, y data as points. Neither R nor MATLAB is as convenient as a dedicated graphical package such as SigmaPlot and my own preference is to plot "working graphs" in R and then dump the data into a text file to create better quality plots using SigmaPlot. The graphs given in this book are such "working graphs" and while perfectly satisfactory for visual analysis are not of publishable quality: these are used here to keep the coding simple and to show the reader what the actual output will look like. In the present program, we want (*a*) a line plot and (*b*) specified labels on the axes. The appropriate coding is

R CODE:

```
plot(Generation, Npop, xlab='Generation', ylab='Population
size', type='l')
```

OR

```
plot(OUTPUT[,1],OUTPUT[,2], xlab='Generation', ylab='Popula-
tion size', type='l')
```

MATLAB CODE:

```
plot(Generation, Npop);
xlabel('Generation');
ylabel('Population size');
```

OR

```
plot(OUTPUT(:,1),OUTPUT(:,2));
xlabel('Generation');
ylabel('Population size');
```

Putting all of this together gives the R code

```
rm(list=ls())                          # Clear memory
MAXGEN     <- 100                      # Set maximum number of
                                         generations
N.init     <- 20                       # Initial population
                                         size
LAMBDA     <- 1.1                      # Rate of increase
Generation <- seq(from=1, to=MAXGEN) # Generation vector
Npop       <- matrix(0,MAXGEN,1)       # Generation vector
Npop[1]    <- N.init                   # Store initial
                                         population size
# Iterate over generations
   for (i in 2: MAXGEN){ Npop[i]  <- LAMBDA*Npop[i-1]}
plot(Generation, Npop, xlab='Generation', ylab='Population
size', type='l')
   print(Npop[MAXGEN])             # Print last population size
```

Note that I have added a print statement to print out the last population size. In this instance the word `print` is not required and the same result would be obtained if I had written `Npop[MAXGEN]`. However, the print function is required in some instances, such as within a loop, and so, as a general rule, I prefer to use it. The graphical output is shown in Figure 1.1. As expected, population growth is exponential with the printout showing that the population has expanded to 250,556.6 individuals. We now move on to the next step and add temporal heterogeneity in model 2.

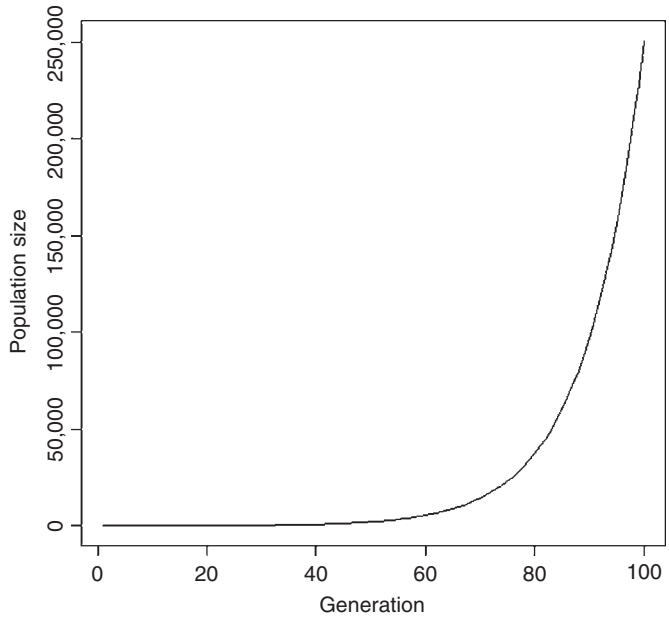

Figure 1.1 Output from model 1 showing exponential increase in population size.

1.4.3 Mathematical assumptions of model 2

1. Assumptions 1 and 2 of model 1 remain the same.
2. There is temporal heterogeneity in the rate of increase λ. For the present pedagogical purpose, I shall assume that λ is a random uniform variate from 0 to `MAX.LAMBDA`. The mean value of λ, $\bar{\lambda}$, under this scenario is `LAMBDA/2`.

If `MAX.LAMBDA=2.2`, then $\bar{\lambda} = 1.1$, the same value as in the constant environment. As the mean growth rate exceeds unity we might, naively, expect that the population would still grow without bound. The expected population size after `MAXGEN` generations is `N.init*LAMBDA^(MAXGEN-1)`, which in the present case would be the same as in model 1, namely 250,556.6. However, as the numerical analysis will show this is not a correct assessment.

Step 7: Seeding a random number generator

To add temporal variation to the rate increase we use a uniform random number generator (functions `runif` in R and `rand` in MATLAB). All random number generators are pseudorandom numbers in that they are based on a formula that generates numbers that are random for at least a subset of numbers (typically, the generators cycle such that the same sequence is generated after a large number [e.g., 63,000] of generations). Unless and otherwise specified, the generator takes its initial value from some varying component such as the computer clock. For the purposes of debugging a program, it is useful to be able to recreate the same

sequence of random numbers: To do this we "seed" the random number generator, which means that it always starts at the same point and generates the same sequence.

R CODE:

```
set.seed(100) # set seed
```

MATLAB CODE:

```
rand('twister', 100);  % set seed
```

In the above code, the integer 100 is arbitrary and set by the user (see the "help" menus in each language for further details): the important point is that changing the integer will change the random number sequence generated.

Step 8: Adding a random element: functions `runif` and `rand`
According to the earlier assumptions λ varies between 0 and `MAX.LAMBDA`. This means that we must change the variable `LAMBDA` from a constant to a vector of random uniform elements. To do this in R we replace

```
LAMBDA      <- 1.1   # Rate of increase
```

with

```
MAX.LAMBDA <- 2.2                 # Maximum rate of increase
LAMBDA     <- runif(MAXGEN, min=0, max=MAX.LAMBDA) # Random
                                                    lambdas
```

In MATLAB we use

```
MAX_LAMBDA = 2.2;                   % Maximum rate of increase
LAMBDA     = Max_LAMBDA*rand(MAXGEN, 1); % Random lambdas
```

The new R coding is

```
rm(list=ls())               # Clear memory
set.seed(100)               # set seed
MAXGEN      <- 100          # Set maximum number of generations
N.init      <- 20           # Initial population size
MAX.LAMBDA  <- 2.2          # Maximum rate of increase
LAMBDA      <- runif(MAXGEN, min=0, max= MAX.LAMBDA) # Random
                                                     lambdas
Generation <- seq(from=1, to=MAXGEN) # Generation vector
Npop    <- matrix(0,MAXGEN,1)   # Generation vector
Npop[1] <- N.init               # Store initial population size
for (i in 2: MAXGEN){ Npop[i] <- LAMBDA[i-1]*Npop[i-1]}
plot(Generation,  Npop,  xlab='Generation',  ylab='Population
size', type='l')
print(Npop[MAXGEN])             # Print last population size
```

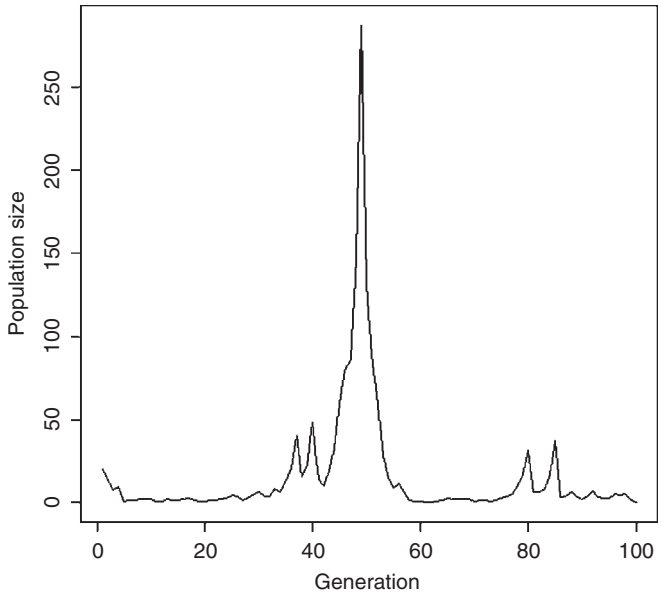

Figure 1.2 Output from a single run of model 2.

Contrary to our naive expectation, the population has a peak at less than 300 and finishes the simulation at only a population size of 0.09446408, much less than the expected value of 250,556.6 (Figure 1.2). The question that immediately arises is whether this is just a fluke of the random number seed we chose: by varying this seed it is easy to see that this is not the case. It is perhaps unreasonable to allow the population size to drop below a single individual and we should assume that the population is extinct at this point.

Step 9: Adding a conditional statement: the `while` loop

One approach to stop the simulation if the population falls below 1 individual is to change the loop to a `while` loop (an alternative possibility is the use of an "`if`" statement. In the present case this is slower). The `while` construct cycles through the instructions enclosed by {...} until a specified condition is met. We could replace the `for` loop in the model by a `while` loop (ignoring for the present the issue of population sizes less than 1):

R CODE:

```
  Gen       <- 1        # Set the generation counter to 1
  while (Gen<MAXGEN)
{
  Gen       <- Gen+1    # Increment the generation counter
  Npop[Gen] <- LAMBDA[Gen-1]*Npop[Gen-1] # new population size
}                       # End of while loop
```

MATLAB CODE:

```
Gen       = 1;           % Set the generation counter to 1
while (Gen<MAXGEN);
Gen       = Gen+1;       % Increment the generation counter
Npop(Gen) = LAMBDA(Gen-1)*Npop(Gen-1); % new population size
end;                     % End of while loop
```

This gives exactly the same output as previously (i.e., Figure 1.3). To add the population size condition we change the `while` statement to

```
while  (Gen<MAXGEN  &&  Npop[Gen]>1)     # R code
while  (Gen<MAXGEN  &&  Npop(Gen)>1);    %  MATLAB R code
```

The cycle continues so long as `Gen` is less than `MAXGEN` and `Npop` of the previous cycle is greater than 1. Because we have preassigned zeros to the population numbers, if the simulation stops before the maximum number of generations is reached the plot still shows the population at zero for the remaining generations. An alternative method would be to plot only the data from 1 to `Gen`, which is the last generation of the simulation:

```
plot(Generation[1:Gen],   Npop[1:Gen],   xlab='Generation',
ylab='Population size', type='l')
```

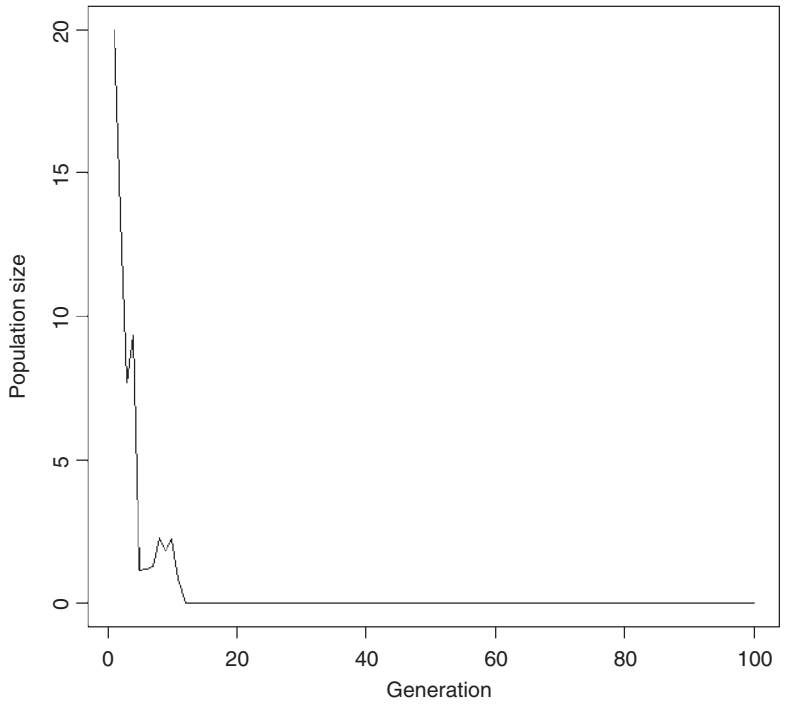

Figure 1.3 Output from model 2 using a while loop to stop the simulation if the number of individuals falls below 1.

We now have two types of output from the model, the final population size and the persistence time (`Gen` or `MAXGEN`) for a given run. What we want to do now is to determine the frequency distribution of population sizes and minimum persistence times. Thus, we need to run many replicates.

Step 10: Running multiple simulations: functions

While we could enclose the above coding in an outside loop that ran through the iterations we need, a faster method is to write the model as a function and then use a function such as `apply` in R (for simplicity I use a loop for MATLAB code). Functions have already been introduced in terms of those supplied by the program (e.g., in R we have used `seq`, `runif`, `print`, `set.seed`). Here we create our own function. As in Los Vegas, what happens in a function stays in the function, unless passed back to the main program. A function has a name and a set of variables that can be passed to it. If a variable occurs in the function but is not passed to the function the R program uses the value set elsewhere. For example, consider a function called `TEST` that adds numbers `a` and `b`

```
  rm(list=ls())
  TEST  <- function(x,y) {x+y}   # Function to add two numbers
# Main program
  a   <- 5; b   <- 3              # Define numbers
  TEST(a,b)                       # Call function
  print(c(a, b, a+b))             # Print a, b and their sum
```

Running this program gives the output

```
> TEST(b)
[1] 8
> print(c(a,b, a+b))
[1] 5 3 8
```

Now suppose we change the function so that only `b` is passed and `a` is incremented within the function

```
  rm(list=ls())
  TEST <- function(y) {a <- a+1; a+y}  # Function to add two numbers
# Main program
  a <- 5; b <- 3                       # Define numbers
  TEST(b)                              # Call function
  print(c(a, b, a+b))                  # Print a, b and their sum
```

This program gives

```
> TEST(b) # Call function
[1] 9
> print(c(a, b, a+b))# Print a, b and their sum
[1] 5 3 8
```

The function correctly sets the initial value of a to 5 and increments it to give the summed value of 9 but the value of a is not changed in the main program. This property means that it is not actually necessary to name all variables that are used in the function: However, it is a good practice and avoids possible errors to pass all variables in the function call.

In the present model we want the final population size. Thus we do not need to store the intermediate values. I shall call the function POP (in this book I use all capitals for "user-supplied" functions to distinguish them from the R-supplied functions, which are generally in lower case) and pass to it all the initial parameters (MAXGEN, Npop, MAX.LAMBDA), and receive back the final population size and minimum persistence time. Note that the names in the function call are arbitrary, but it is useful to at least have them similar for readability: thus I have used (Maxgen, Npop, MAX.Lambda). I could equally have used (MAXGEN, Npop, MAX.LAMBDA). The important point is that parameter names in the function match by order the parameter names in the function declaration. The population size and generation are concatenated into a vector for return to the main program. The function should be placed above the main program otherwise R will not find it. However, the clear code (rm(list=ls())) needs to be the first line or it will delete the function from the workspace. In MATLAB, the function is placed in a separate file and has a different opening structure.

R CODE:

```
  POP <- function(Maxgen,Npop,MAX.Lambda) # Populationfunction
{
    Gen <- 1              # Set the generation counter to 1
# Generate Maxgen random lambdas
    LAMBDA <- runif(Maxgen, min=0, max= MAX.Lambda)
# Cycle through until MAXGEN or extinction
    while (Gen<Maxgen && Npop > 1)
  {
    Gen <- Gen+1                # Increment the generation counter
    Npop <- LAMBDA[Gen-1]*Npop # New population size
  }                             # End of while loop
# Concatenate and return the final population size and persistence
  time
  return (c(Npop, Gen))
} # End of function
```

MATLAB CODE:

```
function [Npop, Gen] = POP( Maxgen, Npop, MAX_Lambda ) % Population
                                                          function
    Gen = 1                     % Set the generation counter to 1
% Generate Maxgen random lambdas
    LAMBDA = rand(Maxgen, 1)*MAX.Lambda;
```

```
# Cycle through until MAXGEN or extinction
    while (Gen<Maxgen && Npop > 1)
    Gen = Gen+1;                % Increment the generation counter
    Npop = LAMBDA(Gen-1)*Npop; % New population size
    end                         % End of while loop
% End of function
```

To access this function we simply call it with the appropriate parameters.

R CODE:

```
  set.seed(100)               # set seed
  MAXGEN      <- 100          # Set maximum number of generations
  N.init      <- 20           # Initial population size
  MAX.LAMBDA  <- 2.2          # Maximum rate of increase
  POP(MAXGEN, N.init, MAX.LAMBDA )     # Call function POP
```

MATLAB CODE:

```
  rand('twister',100)    % set seed
  MAXGEN      = 100;     % Set maximum number of generations
  N_init      = 20;      % Initial population size
  MAX_LAMBDA  = 2.2;     % Maximum rate of increase
 [Npop, Gen]  = POP(MAXGEN, N_init, MAX_LAMBDA) % Call function POP
```

which gives the output `0.8325039 11.0000000`. In this case the population size should actually be set to zero (and one could argue that the persistence time is 10 not 11): we shall deal with these issues below.

Step 11: Running multiple simulations: the `apply` function

Suppose we wish to run `NREP` replicate runs: one way would be to use a loop.

R CODE:

```
    Nrep  <- 10 # Set the number of replicates
# Pre-assign space for the final popn values and generation values
# Column 1 will hold the population and column 2 the generation
    Output  <- matrix(0,Nrep,2)
  for (Irep in 1: Nrep)           # Iterate over replicates
{
  Output[Irep,] <- POP(MAXGEN, N.init, MAX.LAMBDA)  # Call POP
}                                 # End of replicate loop
```

MATLAB CODE:

```
    Nrep - 10 % Set the number of replicates
% Pre-assign space for the final popn values and generation values
% Column 1 will hold the population and column 2 the generation
    Output = zeros(Nrep,2);
```

```
    for (Irep = 1: Nrep)              %  Iterate over replicates
    [Npop, Gen] = POP(MAXGEN, N_init, MAX_LAMBDA);  %  Call POP
    Output[Irep,1:2] = [Npop, Gen]; %  Store output
    end                               %  End of replicate loop
```

The full coding in R (lines omitted from `POP`) is

```
    rm(list=ls())                      # Clear memory
    POP  <- function(Maxgen, Npop, MAX.Lambda) {enter lines as above}
#####################  MAIN PROGRAM  ####################
    set.seed(100)             # set seed
    MAXGEN         <- 100     # Set maximum number of generations
    N.init         <- 20      # Initial population size
    MAX.LAMBDA     <- 2.2     # Maximum rate of increase
    Nreps          <- 10      # Set the number of replicates
# Pre-assign space for the final popn values and generation values
# Column 1 will hold the population and column 2 the generation
    Output <- matrix(0,Nreps,2)
    for (Irep in 1: Nreps)    #  Iterate over replicates
{
    Output[Irep,] <- POP(MAXGEN, N.init, MAX.LAMBDA)  #  Call POP
}  #  End of replicate loop
Output                 #  print out matrix called Output
```

The data are stored in the matrix called `Output`, the first column holding the population sizes and the second column the generation times. The last line prints out the matrix `Output`:

```
> Output
             [,1]  [,2]
 [1,]  0.8325039    11
 [2,]  0.8863995     5
 [3,]  0.4853632    29
 [4,]  0.5199308    65
 [5,]  0.1201204    18
 [6,]  0.4594043    14
 [7,]  0.6047426     4
 [8,]  0.1101742    17
 [9,]  0.4876619     7
[10,]  0.7386976     3
```

An alternate approach in R that is quicker is the use of the R function `apply`. Its use in this instance is somewhat unusual in that it is used simply to generate replication whereas it is more typically used to apply a function to the rows or columns of a matrix. The general structure of the function `apply` is `apply(X, MARGIN, FUN, ...)`, where `X` is the array to be used, `MARGIN` is a vector giving the subscripts which the function will be applied over (`1` indicates rows, `2` indicates

columns, and `c(1,2)` indicates rows and columns), `FUN` is the function to be applied, and . . . denotes optional arguments to `FUN`.

Before showing how the apply function can be used to run multiple replicates I shall give an example of its more typical use: suppose we wished to examine the effect of different maximum rates of increase, specifically, `MAX.LAMBDA` = 2.1, 2.2, 2.3, and 2.4. First, we create a matrix holding these values:

```
Maximum.Lambdas <- matrix(c(2.1,2.2,2.3,2.4))
```

This matrix is the matrix `X` to be supplied to the `apply` function. The matrix is a 4 × 1 matrix and hence `MARGIN`=1 (i.e., use rows). The function to be supplied is `POP` but we have to make a change to the function declaration sequence because `apply` expects the first component of the sequence to be the value supplied by `X`: thus we rewrite `POP` as

```
POP <- function(MAX.Lambda, Maxgen, Npop )
```

and supply `Maxgen` and `Npop` as optional arguments in `apply`

```
Output <- apply(Maximum.Lambdas, 1, POP, MAXGEN, N.init )
```

The function cycles through the matrix `Maximum.Lambdas` and applies the function `POP`, storing the results in `Output`. Printing `Output` gives

```
> Output
          [,1]        [,2]         [,3]       [,4]
[1,]  0.96059   0.8863995    0.9807614   732.6482
[2,]  5.00000   5.0000000   31.0000000   100.0000
```

R is "intelligent" enough to place the two values returned on each cycle in separate rows, meaning that population size occupies the first row and generation number the second row. We can extract these separately by

```
Npops<- Output[1,1:4];Gens<- Output[2,1:4]# Separate output
```

Returning now to the issue of using `apply` to run multiple replicates: We do not actually wish to supply different values of the three variables in the declaration sequence but simply to call the function multiple times. Therefore, to do `Nreps` replications we create an `Nreps` × 1 (called, say, `MaxL`) matrix with the same value of the first parameter in the function declaration in each cell: so assuming that, as above, `MAX.Lambda` is the first parameter we create `MaxL` using the replicate function `rep`

```
Nreps <- 10            # Set the number of replications
MaxL  <- matrix(rep(MAX.LAMBDA, times = Nreps)) # Create
                                                matrix
                                                for
                                                apply
```

We can now call `apply` using `MaxL` as the `X` array

```
  Output <- apply(MaxL, 1, POP, MAXGEN, N.init )   # Call apply
  Npops  <- Output[1,1:Nreps]; Npops    # Extract populations
  Gens   <- Output[2,1:Nreps]; Gens     # Extract generations
```

The full coding reads

```
    rm(list=ls())    # Clear memory
    POP  <-  function(MAX.Lambda, Maxgen, Npop, ) {enter lines as
above}
#################### MAIN PROGRAM ####################
  set.seed(100)          # set seed
  MAXGEN       <- 100    # Set maximum number of generations
  N.init       <-  20    # Initial population size
  MAX.LAMBDA   <- 2.2    # Maximum rate of increase
  Nreps        <- 10     # Set the number of replicates
  MaxL   <- matrix(rep(MAX.LAMBDA, times = Nreps)) # Create matrix
                                                     for apply
# We can now call apply using MaxL as the X array
  Output     <- apply(MaxL, 1, POP, MAXGEN, N.init ) #  Call apply
  Npops   <- Output[1,1:Nreps]; Npops      # Extract populations
  Gens       <- Output[2,1:Nreps]; Gens    # Extract generations
which generates the output
>  Npops     <- Output[1,1:Nreps]; Npops
  [1] 0.8325039 0.8863995 0.4853632 0.5199308 0.1201204 0.4594043
  [7] 0.6047426 0.1101742 0.4876619 0.7386976
>  Gens        <- Output[2,1:Nreps]; Gens
[1] 11 5 29 65 18 14 4 17 7 3
```

Despite the large expected value, the ten replicates become extinct within less than 100 generations. To better examine the statistical distribution of these two types of output we must run many more simulations, 1,000 being a reasonable number for a first analysis. Before doing this we shall return to the issue of setting population sizes of those populations that go extinct to zero and decreasing the displayed generation by 1 to reflect the actual generation at which extinction occurred.

Step 12: Matrix-wide comparisons

Our object is to change all population values less than 1 to zero and subtract 1 from the generations if the population goes extinct before the end of the simulation. While the former could be done using a loop and an "`if`" statement, a much better approach is to use the following object-oriented construction:

```
    Npops[Npops<1] <- 0  # R Set all pop sizes < 1 to 0
    Npops(Npops<1) = 0;  % MATLAB Set all pop sizes < 1 to 0
```

This statement can be read as "set all values of `Npops` less than 1 in the matrix `Npops` to zero." We could do the same for the 2-D matrix `Output`. In this case we want to examine only the entries in the first column as it is this column that contains the population sizes. The appropriate coding is

```
Output[Output[,1]<1,]   <- 0  #  Examine all entries in the
                                 first column
Output(Output[:,1]<1,:] = 0; %  Examine all entries in the
                                 first column
```

which would give the output

```
> Output
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    0    0    0    0    0    0    0    0    0     0
[2,]   11    5   29   65   18   14    4   17    7     3
```

A value of 1 can be subtracted from each element by the matrix-wide operation:

```
Gens <- Gens-1 # R Subtract 1 from each element of Gens
Gens = Gens-1; % MATLAB Subtract 1 from each element of Gens
```

OR

```
Output[2,] <- Output[2,] −1   # R Subtract 1 from each element of
                                 row 2
Output(2,:) = Output[2,:] −1 % MATLAB Subtract 1 from each ele-
                                 ment of row 2
```

However, this operation would also subtract 1 from those runs in which the population persisted. To exclude these cases we can write a two-step operation:

R CODE:

```
Gens <- Gens-1              # Subtract 1 from all generations
Gens[Gens==MAXGEN-1] <- MAXGEN # If generation = MAXGEN-1
                                 set to MAXgEN
```

MATLAB CODE:

```
Gens = Gens-1;                % Subtract 1 from all genera-
                                tions
Gens(Gens=MAXGEN-1) = MAXGEN; % If generation = MAXGEN-1
                                set to MAXGEN
```

The first line subtracts 1 from all generations, while the second line restores this value if the subtraction gives `MAXGEN−1`, meaning that the simulation had run its full course in this replicate.

Step 13: Summarizing and plotting the results: functions `hist`, `summary`, `length`

To obtain sufficient replicates to accurately depict the distributions requires at least 1,000 replicates: here I use 10,000 (i.e., `Nreps <- 10000`). A simple graphical display is produced by the histogram function `hist` in both R and MATLAB. There are three graphs that are worth producing: (*a*) the distribution of population sizes for the entire data set, (*b*) the distribution of persistence times, and (*c*) the distribution of population sizes for those populations that persisted the full length of the simulation. To obtain the third group we extract the data from the full set of population sizes:

```
Pops.not.extinct <- Npops[Npops>0] # Extract all populations that
                                     persisted
```

For reasons that will become clear I also plot the log of population size of those populations that persisted. To display all four graphs on the same page we split the graphics page into four sections using

```
par(mcol=c(2,2))          # Split the graphics page into quadrats
```

which tells R to plot the graphs by columns (thus the sequence of plotting will be top left, bottom left, top right, and bottom right. To plot across rows use `par (mfrow=c(2,2))`. The four histograms are plotted with

```
hist(Npops)                    # Histogram of population sizes
hist(Gens)                     # Histogram of persistence times
hist(Pops.not.extinct)         # Histogram of surviving pop sizes
hist(log10(Pops.not.extinct))# Histogram of log surviving pops
```

It is clear from visual inspection of the histograms (Figure 1.4) that the vast majority of populations become extinct before the end of the simulation and that persistence times are generally less than 20 generations. To get a better idea of the numerical results we use the R function `summary` (a similar function is available in the statistics toolbox of MATLAB). This function is a generic function in that it supplies information depending on the R object supplied to it: Thus supplying it with a set of numbers, as in this case, causes it to send back a set of standard summary statistics, whereas supplying it with the object obtained from an analysis of variance causes it to send back an analysis of variance table and associated information. In most cases the information is stored as a list but in this particular case the mode is numeric (to determine the mode of an object use the code `mode(Object name)`. The result of being a numeric rather than a list mode is that the way one extracts information is different. To illustrate the method in the present case we call `summary` and store it as an object:

```
# Get summary data
  Data.Npops             <- summary(Npops)
  Data.Pops.not.extinct  <- summary(Pops.not.extinct)
  Data.Gens              <- summary(Gens)
```

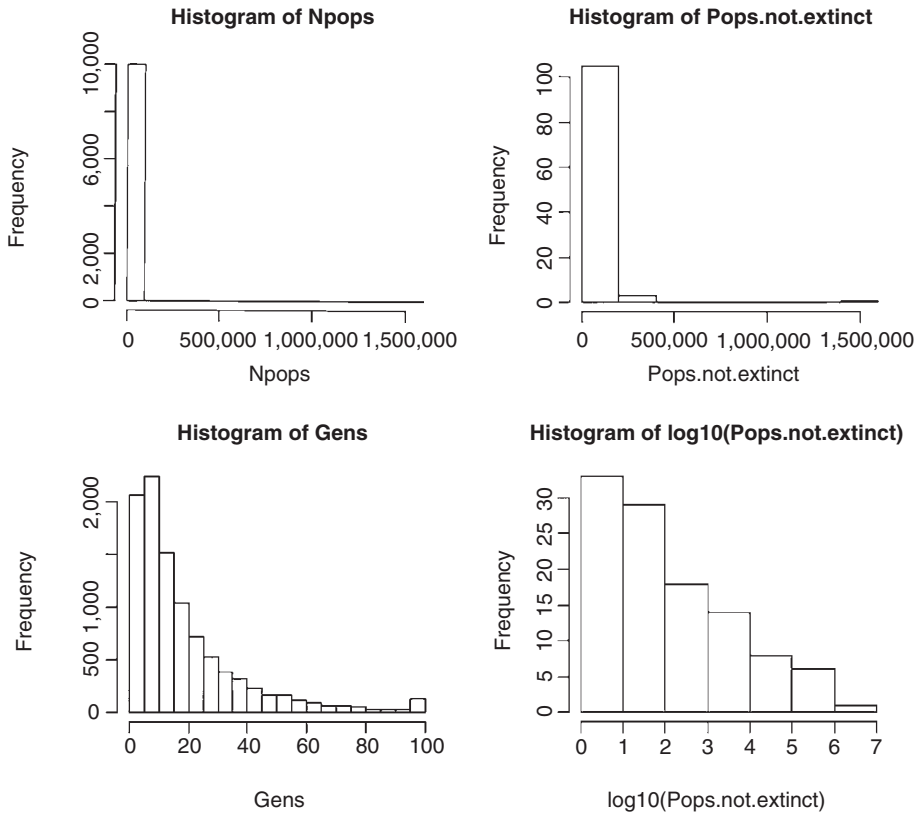

Figure 1.4 Histograms of population sizes and persistence for model 2.

To print out the entire summary information we simply type the object name (I have inserted print statements to make the output more readable):

```
print("Summary data for population sizes") ; Data.Npops
print("Summary data for Pops.not.extinct"); Data.Pops.not.extinct
print("Summary data for persistence times") ; Data.Gens
```

which generates

```
> print("Summary data for population sizes");Data.Npops
[1] "Summary data for population sizes"
   Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
    0.0       0.0       0.0     315.7       0.0 1574000.0
> print("Summary data for Pops.not.extinct"); Data.Pops.not.ex-
tinct
[1] "Summary data for Pops.not.extinct"
   Min.    1st Qu.  Median      Mean     3rd Qu.      Max.
1.498e+00 5.556e+00 5.884e+01 2.896e+04 1.072e+03 1.574e+06
```

```
> print("Summary data for persistence times");Data.Gens
[1] "Summary data for persistence times"
   Min.    1st Qu.    Median     Mean    3rd Qu.     Max.
   1.00       6.00      13.00    19.09     25.00   100.00
```

Before discussing this output let us return to the summary objects. The summary object in the present case contains six pieces of information: the minimum value, the first quantile, the median, the mean, the third quantile, and the maximum. These components can be accessed separately: suppose, for example, we only wanted the mean value of `Npops`, which is the fourth entry in `Data.Npops`: to get this we simply use `Data.Npops[4]`.

Returning to the above output: the mean population size is only 315.7, which is far below the expected value of 250,556.6, though the maximum population size is 1,570,0000, which is far above the expected population size. Unfortunately, the `summary` function does not give the sample sizes and so it is not possible from this information to assess how many populations persisted through the entire simulation. To extract this information we can use the R function `length`, which gives the number of elements in an object:

```
length(Pops.not.extinct) # number of populations that persisted
```

which gives 109: so out of 10,000 replications only 1.09% persisted for 100 generations.

Step 14: Further model analysis: more on lists

It is clear that the predicted population size from the deterministic model (model 1) does not match the result from the stochastic model with the same mean rate of increase (model 2). To further illustrate the list construct let us statistically compare the population sizes from model 2 with the size predicted by model 1. For simplicity I shall use a single sample *t*-test, recognizing that the extreme skew in the distribution makes such a test suspect (but this is for illustration of lists not statistical rigor). This test is available in the statistics toolbox of MATLAB. The R code, saving the output as an object called `T.results`, is

```
Data       <- Npops-250556.6   # Subtract predicted value
T.results  <-  t.test(Data)    # T test with null of zero
T.results                      # Print out results of t test
```

The result is

```
      One Sample t-test
data: Data
t = -1501.757, df = 9999, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
-250567.6 -249914.3
sample estimates:
mean of x
-250240.9
```

Obviously the difference is highly significant, even given the skew in the data. The object `T.results` consists of a list of items that can be accessed individually in a number of ways. First, to determine what items are in the list we issue the code

```
names(T.results) # Names of the items in the object T.results
```

with the result

```
[1]"statistic"  "parameter"  "p.value"  "conf.int"  "estimate"
[6]"null.value"  "alternative"  "method"  "data.name"
```

The estimate is located in position 5 and can be accessed in the following ways:

```
T.results[5]
T.results[[5]]
T.results$"estimate"
```

which give three slightly different outputs (but not different values):

```
> T.results[5]
$estimate
mean of x
−250240.9
- - - - - - - - - - - - -
> T.results[[5]]
mean of x
−250240.9
- - - - - - - - - - - - -
> T.results$estimate
mean of x
−250240.9
- - - - - - - - - - - -
```

If one wishes to use the resulting variable the second two methods are preferred since, for example, `T.results[5]^2` results in the error message "`Error in T.results[5]^2 : non-numeric argument to binary operator`". If one wished to convert the value to a simple numerical value and eliminate the accompanying label "mean of x" use the function `as.numeric`, as in, for example,

```
a <- as.numeric(T.results$estimate) # Convert value to numeric
```

Step 15: An analytical aside: what is going on?

The present model results indicate that the arithmetic mean growth rate does not appear to be a good index of population persistence in a temporally stochastic environment. If this is the case then perhaps the arithmetic mean growth rate is also not an appropriate measure of fitness in a stochastic environment. Haldane and Jayakar (1963) and Cohen (1966) showed that the appropriate measure is the geometric rate of increase. The reason for this resides in the difference between the geometric and arithmetic means (Lewontin and Cohen 1969). In our model population size at time t is given by

$$N_{t+1} = N_0\lambda_1\lambda_2\lambda_3\ldots\lambda_t = N_0\prod_{i=1}^{t}\lambda_i \tag{1.32}$$

We assumed that λ_i is a random, uncorrelated variable with mean $\bar{\lambda}$. The expected population size at time t is then given by the product of the initial population size, N_0, times the expectation of the product $\lambda_1\lambda_2\lambda_3\ldots\lambda_t$. Because the λ's are uncorrelated, the expected value of the product is equal to the product of the expected values, giving

$$E\{N_t\} = N_0E\{\prod_{i=1}^{t}\lambda_i\} = N_0\prod_{i=1}^{t}E\{\lambda_i\} = N_0\bar{\lambda}^t \tag{1.33}$$

At first glance the above result suggests that an appropriate measure of fitness is $\bar{\lambda}$, which is the arithmetic mean of the finite rates of increase (i.e., $\bar{\lambda} = \sum_{i=1}^{t}\lambda_i$) not the geometric, which is given as $\bar{\lambda}_g = (\Pi_{i=1}^{t}\lambda_i)^{\frac{1}{t}}$. However, the behavior of populations in a temporally randomly varying environment has the curious property that the expectation of population size will grow without bound whenever $\bar{\lambda} > 0$ but the probability of extinction within a few generations can be virtually certain (Lewontin and Cohen 1969; Levins 1969; May 1971, 1973; Turelli 1977). This paradoxical behavior can be illustrated with a simple example: suppose that λ can take two values, 0 or 3, with equal frequency. The expected value of λ is $(0 + 3)/2 = 1.5$, and hence the expected population size increases without bound as t increases. For example, starting from a single female, after 10 generations $E\{N_{10}\} = 1.50^{10} = 57.7$ but either $N_{10} = 59{,}049$ or $N_{10} = 0$ and the probability that the population persists for the 10 generations is $(0.5)^{10} = 0.00098$, a very small probability indeed! The geometric mean is always smaller than the arithmetic and the two are related by the approximation $E(\ln\ \lambda) \approx \ln\bar{\lambda} - \frac{\sigma_\lambda^2}{2\bar{\lambda}^2}$ (Lewontin and Cohen 1969), where $E(\ln\lambda)$ is the geometric mean, $\bar{\lambda}$ is the arithmetic mean, and σ_λ^2 is the variance. Selection should operate to increase the arithmetic mean and decrease its variance, which will increase the geometric mean.

Step 16: Adding spatial heterogeneity

The important conclusion from the model so far is that temporal heterogeneity could be an important selective agent favoring particular types of life history. Our impetus for this analysis was the hypothesis that migration is an important evolutionary response to environmental heterogeneity. Thus the next step of the analysis is to introduce spatial variation, initially keeping all subpopulations isolated.

1.4.4 Mathematical assumptions of model 3

1. The habitat is divided into a number of discrete patches.
2. Rates of increase are stochastically variable and uncorrelated among patches.
3. There is no migration of animals among patches.

There is no conceptual difference between this model and the previous model but there are two avenues by which it could be programmed:

1. Each population is run over its entire simulation within the function as previously done, with the main program iterating over patches.

2. The function could compute the single generation change for all populations and the function called iteratively for each generation of the simulation.

In the subsequent model we wish to introduce migration between habitats at each generation which could not be done under the first approach. Thus the second approach is appropriate in this instance. The coding is a straightforward extension of the previous approach with the following changes:

1. Within the `POP` function the number of random `LAMBDA` values generated is equal to the number of patches, not the number of generations.
2. The mean population size (called `Npop.Sizes`) is followed, which is obtained using the R and MATLAB function `mean`:

```
Npop.Sizes[Igen] <- mean(Npop)   # R code
Npop_Sizes(Igen) = mean(Npop);   % MATLAB code
```

where `Npop` is the vector of population sizes at generation `Igen`.

3. The simulation is stopped when either the maximum number of generations is reached or the mean population size is zero (i.e., all populations are extinct).
4. In addition to mean population size the program also keeps track of the number of extinct populations over time, `N.extinct[Igen]` (another example of matrix-wide comparison). The R and MATLAB codes are, respectively,

```
N.extinct[Igen] <- length(Npop[Npop==0]) # Store number of
                                           extinct popns
N.extinct(Igen) = length(Npop(Npop=0)); % Store number of
                                          extinct popns
```

The full coding in R is

```
rm(list=ls())    # Clear memory
POP  <- function( MAX.Lambda, Npop, N.patches )  # Population
                                                   function
{
LAMBDA  <- runif(N.patches, min=0, max= MAX.Lambda)# Generate
                                                     lambdas
Npop <- Npop*LAMBDA # Generate new population size for all patches
Npop[Npop<1] <- 0  # Check for extinction
return (Npop)         # Return the vector of new population sizes
} # End of function
#################### MAIN PROGRAM ####################
   set.seed(100)             # set seed
   MAXGEN       <- 100       # Set maximum number of generations
   N.init       <- 20        # Initial population size
   MAX.LAMBDA   <- 2.2       # Maximum value of lambda
   N.patches    <- 10        # Number of patches
   Npop   <- matrix(N.init, N.patches, 1) # Initialise populations
   Npop.Sizes  <- matrix(0,MAXGEN)     # Pre-assign storage for
                                         mean popn size
```

```
    Npop.Sizes[1] <-  mean(Npop) # Store first generation mean
                                    popn size
    N.extinct     <- matrix(0,N.patches,1)   # Storage for nos of
                                               extinct popns
    Igen          <- 1           # Initialize generation counter
    while ( Igen<MAXGEN && Npop.Sizes[Igen]>0) # Start while loop
{
    Igen          <- Igen+1                # Increment generation
    Npop <- POP(MAX.LAMBDA, Npop, N.patches) # New population sizes
    Npop.Sizes[Igen] <- mean(Npop)  # Store mean population size
    N.extinct[Igen]  <- length(Npop[Npop==0]) # Store number of
                                                extinct popns
  }                                          # End of while loop
    par(mfcol=c(1,2))            # Divide graphics page into two
#  Plot Mean population size over generations and nos extinct per
generation
    plot(seq(1, Igen), Npop.Sizes[1:Igen], xlab="Generation",
ylab="Mean population size", type="l")
    plot(seq(1, Igen), N.extinct[1:Igen], xlab="Generation",
ylab="Mean population size", type="l", ylim=c(0,N.patches))
```

OUTPUT: (Figure 1.5)

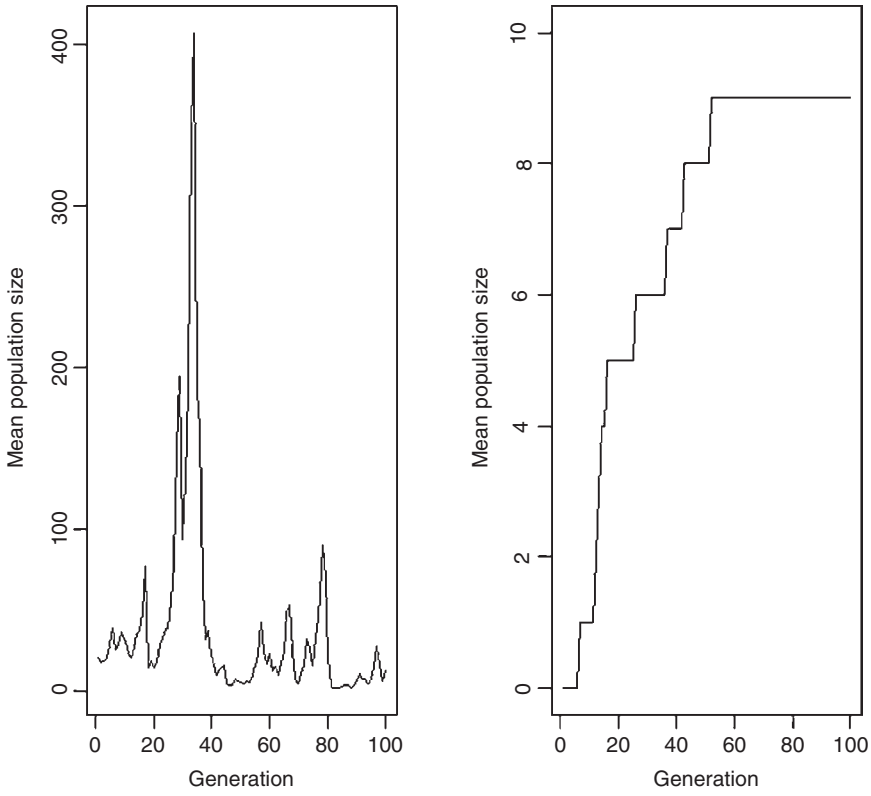

Figure 1.5 Population size and number of extinct populations for model 3.

The output from this model is the same as simply running 10 replicates of model 2 and thus although persistence time is increased the increase is of little evolutionary consequence, most populations becoming extinct within 50 generations. The importance of this model is as a stepping stone to the next model in which we introduce migration among patches.

Step 17: Adding migration
We shall make the simplest possible assumption concerning migration, namely that migrants are distributed among the patches in equal numbers. More complex models are possible (e.g., random assignment, distance-related, etc.) but one should always begin with the simplest model that is biologically not unreasonable.

1.4.5 Mathematical assumptions of model 4

1. All the assumptions of model 3 remain the same in model 4 except that there is now migration among patches.
2. Migrants enter a "common" pool and are then distributed in equal number among the patches.
3. A fixed proportion, P_{mig} (= 0.8 in the present simulation) of the population migrates.
4. Migrants survive migration at the fixed rate of P_{surv} (= 0.95 in the present simulation).
5. Reproduction occurs after migration.

From the above assumptions we get the following recursive equation for the population size in the ith patch at generation $t+1$:

$$N_{t+1,i} = \lambda_{t,i}\left[N_{t,i}(1 - P_{\text{mig}}) + \frac{P_{\text{surv}}P_{\text{mig}}\sum_{j=1}^{n}N_{t,i}}{n}\right] \tag{1.34}$$

where n is the number of patches (`N.patches` in the coding). The R code for this is

```
  Emigrants  <- Npop*P.mig                          # Nos leaving
  Immigrants <- sum(Emigrants)*P.surv/N.patches  # Immigrants
per patch
  Npop  <- Npop  -  Emigrants + Immigrants #  Distribute migrants
  Npop  <- Npop*LAMBDA                      #  new population sizes
```

The full coding in R is

```
   rm(list=ls())                          #  Clear memory
   POP  <- function(MAX.Lambda, Npop, N.patches,P.mig, P.surv)
                                                        #  Pop func
{
LAMBDA  <- runif(N.patches, min=0, max= MAX.Lambda)   #  n random
                                                          lambdas
```

```
    Emigrants  <-  Npop*P.mig                        # Nos leaving
    Immigrants <-  sum(Emigrants)*P.surv/N.patches # Immigrants
per patch
    Npop  <- Npop -  Emigrants + Immigrants  # Distribute migrants
    Npop  <-  Npop*LAMBDA                    # new population sizes
    Npop[Npop<1] <-  0                 #  Check for extinction
    return (Npop)                      #  Return the vector of new
population sizes
}  #  End of function
####################  MAIN PROGRAM  ####################
    set.seed(100)                          #  set seed
    MAXGEN           <-  1000              # Set maximum number of
                                             generations
    N.init           <-  20              #  Initial population size
    MAX.LAMBDA       <-  2.2             #  Maximum value of lambda
    N.patches        <-  10               #  Number of patches
    P.mig            <-  0.5              #  Proportion migrating
    P.surv           <-  0.95           #  Survival rate of migrants
    Npop  <- matrix(N.init,N.patches,1) # Initialisepopulations
    Npop.Sizes       <-  matrix(0,MAXGEN)   #  Pre-assign storage
    Npop.Sizes[1]    <- mean(Npop)       #  Store first generation
                                             mean population size
    N.extinct <- matrix(0,N.patches,1)  #  Storage for nos extinct
                                             popns
    Igen             <-  1               #  Initial generation
  while ( Igen<MAXGEN && Npop.Sizes[Igen]>0)  #  Start while loop
{
    Igen             <-  Igen+1          #  Increment generation
    Npop      <- POP(MAX.LAMBDA, Npop, N.patches, P.mig, P.surv)
                                         #  New popn sizes
    Npop.Sizes[Igen] <-  mean(Npop)      #  Store mean population
                                             size
    N.extinct[Igen]  <- length(Npop[Npop==0]) # Number of extinct
                                                 populations
}                                              # End of while loop
  par(mfcol=c(1,2))                         #  Split page into two
  plot(seq(1, Igen), log10(Npop.Sizes[1:Igen]), xlab="Genera-
tion", ylab="Mean population size", type="l")
  plot(seq(1,  Igen),  N.extinct[1:Igen],  xlab="Generation",
ylab="Number of pops extinct", type="l", ylim=c(0,N.patches))
```

OUTPUT: (Figure (1.6)

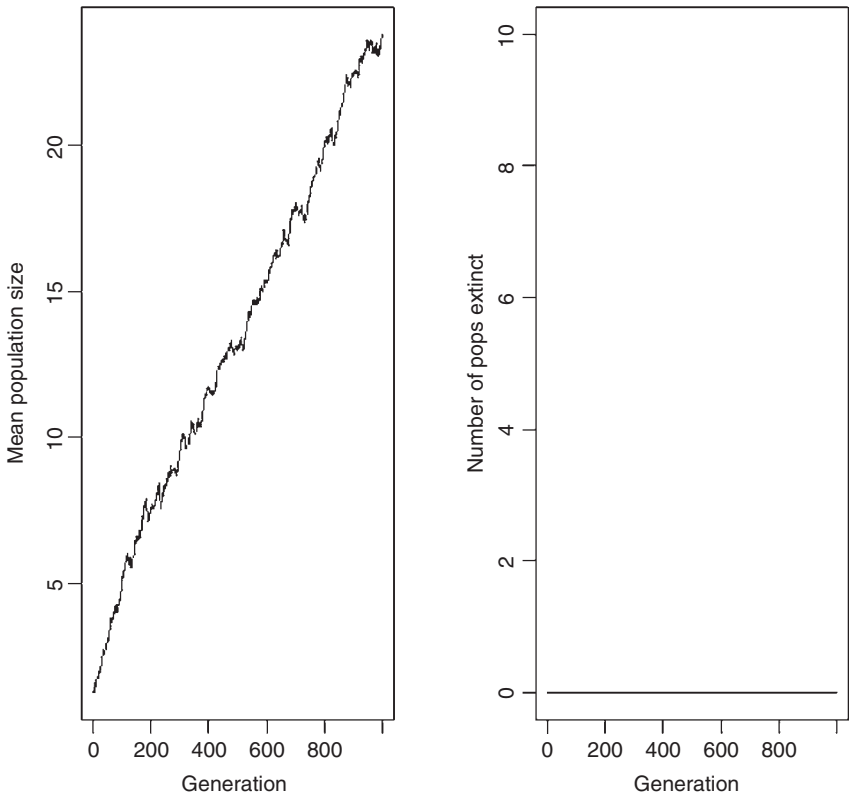

Figure 1.6 Population size and number of extinct populations for model 4.

The output is shown in Figure 1.6. The effect of migration is so great that the plot of population size is best shown on a log scale. Whereas the population crashed quickly in the absence of migration, the presence of migration prevents extinction and the population size increases to the unreasonably large value of 10^{15}.

Step 18: Controlling population growth: model 5

The mean population size reached after the addition of migration is unrealistic but demonstrates the potential evolutionary importance of migration. Before considering a model that allows migration rate to evolve we first add a carrying capacity to constrain population size.

1.4.6 Mathematical assumptions of model 5

1. All assumptions of model 4 apply to model 5 except that there is now a limitation to population size.
2. Population size is limited to a maximum of 1,000 individuals.

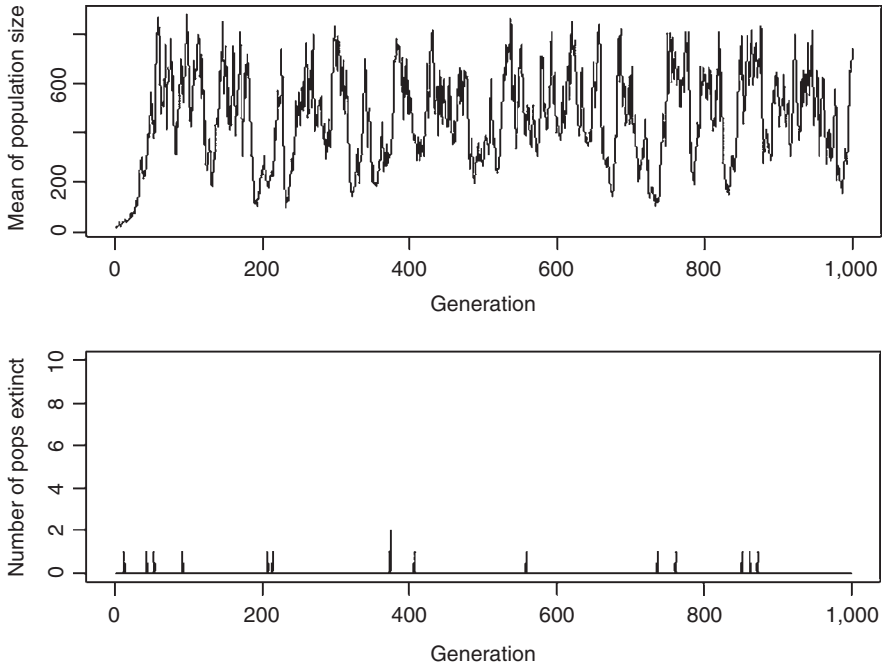

Figure 1.7 Population size and number of extinct populations for model 5 (model 4 plus a carrying capacity). For display purposes the graph page has been split horizontally using `par(mfcol=c(2,1))`. Note that population size is not log transformed.

Assumption 2 is coded as

```
Npop[Npop>1000]   <- 1000 # R code setting limit at 1000
Npop(Npop>1000)   = 1000; % MATLAB code setting limit at 1000
```

Incorporation of this restriction produces a dramatic drop in population size, an increase in the number of temporally extinct populations, but no indication of a significant decline in population persistence (Figure 1.7). The mean population size over the last 100 generations, obtained from the code `mean(Npop.Sizes[900:1000])` is only 473.2975, which is less than one half of the carrying capacity. Thus, the imposition of a carrying capacity in a heterogeneous environment can have profound effects on population size even though a naive census would suggest that the carrying capacity was not being exceeded (see Roff [1974a, 1974b]).

Step 19: More on graphics: functions `expand.grid`, `outer`, `contour`, `persp`

Many analyses will require investigation of effects due to variation in multiple parameters: one method of graphically viewing such variation is to use a contour plot and a 3-D perspective plot. There are several ways to generate such plot: here I shall present two approaches. Before applying either method to the present

model I shall consider a much simpler model to better illustrate the procedures. The object is to plot the equation of a circle:

$$z = \sqrt{x^2 + y^2}, -10 < x < 10, 0 < y < 10 \tag{1.35}$$

First we define the function for z.

R CODE:

```
FUNC.Z  <- function(x,y) {sqrt(x^2+y^2)}
```

MATLAB CODE:

```
function z = FUNC_Z(x,y)
z = sqrt(x^2+y^2)
```

The ranges of x and y are divided into 10 parts (this will generate a matrix of 100 values):

R CODE:

```
n1 <- 4; n2 <- 3
x  <- seq(from=-10, to= 10, length=n1)
y  <- seq(from=0, to = 10,  length=n2)
```

MATLAB CODE:

```
n1 = 4;
n2 = 3;
x  =  linspace(-10, 10, n1);
y  =  linspace(0, 10, n2);
```

Now we generate the matrix of z values for all combinations of x and y. In R this can be done by either of the following:

1. Using the R function `expand.grid` which takes the two vectors and generates a two-column matrix of all combinations, with the x variable changing most rapidly:

```
d  <- expand.grid(x,y)      # x values vary first
z  <- FUNC.Z(d[,1], d[,2])  # Create z
```

The two-column matrix is now converted into the appropriate matrix:

```
z.matrix  <- matrix(z,n1,n2)
```

2. An alternate method is to use the function `outer`

```
z.matrix  <- outer(x, y, func.z)
```

The equivalent in MATLAB is `meshgrid` and rather than the function `FUNC_Z` described above, I use here vectorization to give

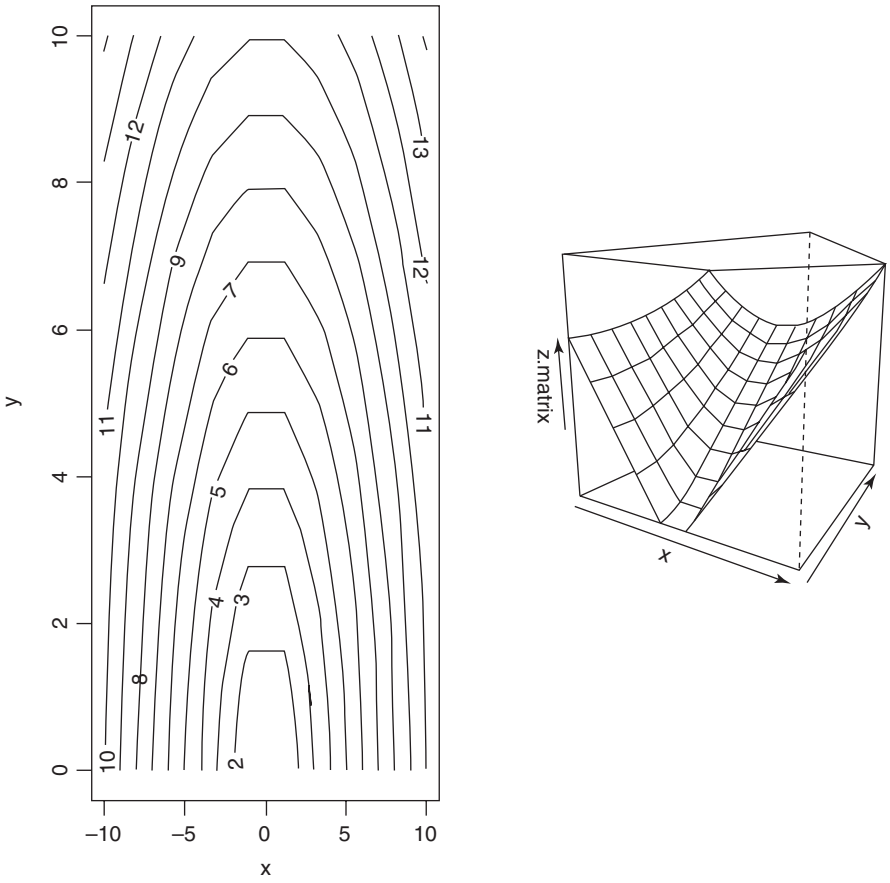

Figure 1.8 An example of contour and 3-D plots.

```
[xx,yy]  = meshgrid(x, y)
zz       = x.^2 + y.^2
```

The data are now in a format suitable for plotting the contours using `contour` or a 3-D plot using the R function `persp` (Figure 1.8) or `surfc` in MATLAB.

R CODE:

```
par(mfrow=c(1,2))                  # Divide graphics page into two
contour(x,y, z.matrix, xlab="x", ylab="y")   # Contour plot
persp(x, y, z.matrix, theta=30, phi=10)      # 3-D plot
```

The parameters `theta` and `phi` are angles defining the viewing direction: `theta` gives the azimuthal direction and `phi` the colatitude.

MATLAB CODE:

```
subplot(1,2,1); % Divide graphics page in two and plot contour in left
[C,h] = contour(x, y, zz)  % Create contour plot
```

```
% clabel(C,h) rotates the labels and inserts them in the contour
lines
clabel(C,h);
xlabel('x'); ylabel('y'); % Add text
subplot(1,2,2); % Divide graphics page in two and plot contour on
                  right
surfc(xx,yy,zz); % Plot a 3-D surface
xlabel('Foraging'); ylabel('Vigilance'); zlabel('Fitness') % Add text
```

Now consider the analysis of variation in `P.mig` and `P.surv` in model 5. In this case I want to plot variation in mean population size for the ranges

```
P.mig  <- seq(from=0.1, to=0.9, length=10) # Proportion migrating
P.surv <- seq(from=0.8, to=0.9, length=10) # Survival rate of mi-
                                             grants
```

To do this I make the previous main program into a function

```
MAIN.PROG  <- function(D)
{
P.mig      <- D[1]
P.surv     <- D[2]
```

Same lines as previously except for deletion of plotting codes

```
} # End of function MAIN.PROG
```

I have left the `set.seed` code in `MAIN.PROG` which means that the same sequence of random numbers is used for each combination: thus I replicate exactly the same environmental variation for each simulation. Now I can use `expand.grid` to generate the necessary variation and pass this to `MAIN.PROG`. However, if I do this exactly as before I will get an error message, because I need to pass only one combination at a time. To pass only one combination per run use the R function `apply` telling the function to use rows:

```
d          <- expand.grid(P.mig,P.surv)
z          <- apply(d, 1, MAIN.PROG)
z.matrix   <- matrix(z, length(P.mig), length(P.surv))
par(mfrow=c(1,2))
contour(P.mig, P.surv,z.matrix, xlab ="P.mig", ylab="P.surv")
persp(P.mig, P.surv, z.matrix, theta=20, phi=20)
```

OUTPUT: (Figure 1.9)

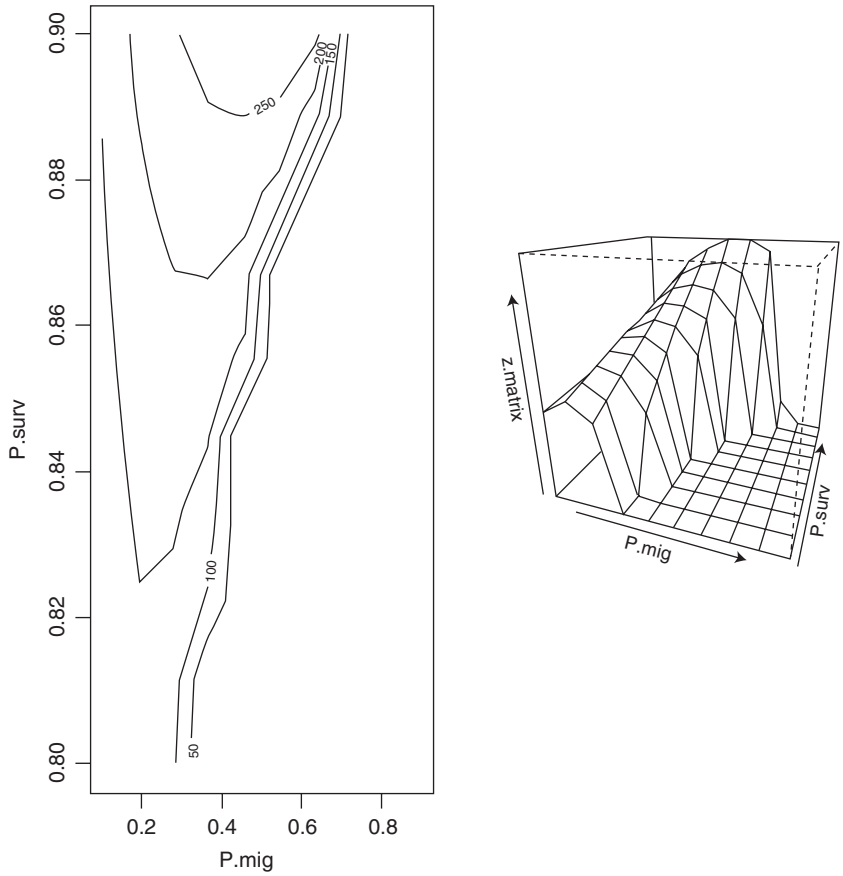

Figure 1.9 Contour and perspective plots for model 5.

The results plotted in Figure 1.9 show that population size increases with `P.surv` (not surprisingly) and that it is largest at an intermediate value of `P.mig`.

Step 20: Adding inheritance: functions `pnorm`, `dnorm` and numerical integration

The observation from the above analysis of the potential importance of migration in a heterogeneous environment raises the question of what is the optimal migration rate. This question is made difficult to answer analytically because at the metapopulation level fitness is frequency-dependent (Roff 1994a). Migrants can be reasonably assumed to suffer an increased mortality rate (in models 4 and 5) and a decrease in reproductive opportunities by a lack of time and by the allocation of energy to migration, both in terms of the energy used in migration and the energy sequested for the capability of migration, as found in wing-dimorphic insects (Roff 1996; Roff and Fairbairn 2007). For the present analysis I shall use a threshold model for the genetic basis of migrant and nonmigrant phenotypes.

1.4.7 Mathematical assumptions of model 6

1. Assumptions of model 5 hold for model 6 except that migration propensity is inherited.
2. Mating is at random and occurs prior to migration.
3. Individuals can be divided into two classes, nonmigrants and potential migrants. Individuals in the latter class migrate with a probability P_{mig} and survive the migratory episode with probability P_{surv}. Additionally, potential migrants suffer a reduction in reproductive fitness because of their allocation of resources to the capability of migration (e.g., presence or absence of a functional flight apparatus in wing-dimorphic insects).
4. Migratory propensity is inherited as a quantitative trait as specified by the threshold model. According to this model there is a continuously normally distributed underlying trait called the liability: individuals above a threshold develop into one type of morph, while individuals below the threshold develop into the alternate (Figure 1.10). Without loss of generality, we can set the threshold at zero and the variance of the distribution at 1. Assuming, again without loss of generality, that nonmigrants are those individuals lying above the threshold the proportion of nonmigrants in the population is given by the normal distribution function

$$P = \int_0^{\infty} \phi\left(x - \mu\right)dx, \text{where } \phi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} \tag{1.36}$$

where μ is the mean value of the liability. This function cannot be integrated symbolically but there is an R function `pnorm` and a MATLAB function `normcdf` that provide a solution. The integral from zero to infinity, given a mean of `Mu` (which can also be vector of values) is obtained as

```
P  <- pnorm( 0, mean= -Mu, sd=1) # R code
P  =  normcdf(0, −Mu, 1);% MATLAB code
```

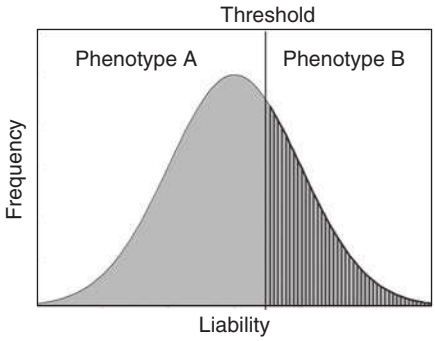

Figure 1.10 Graphical illustration of the threshold model. Individuals with liabilities below the threshold develop into phenotype A, whereas individuals with liabilities above the threshold develop into phenotype B.

The function $\phi(x)$ can be calculated using the R function `dnorm` or the MATLAB function `normcdf`. Numerical integration can be done in R using the function `integrate` and in MATLAB the functions `int` or `quad` (see, e.g., Scenario 4 of chapter 2). The mean values of nonmigrants, X_{NM}, and potential migrants, X_{PM}, can be calculated from the respective truncated normal distributions, giving

$$X_{\mathrm{NM}} = \mu + \frac{\phi(\mu)}{P}, \quad X_{\mathrm{PM}} = \mu - \frac{\phi(\mu)}{1-P} \tag{1.37}$$

Given random mating prior to migration, the phenotypic value of the liability of nonmigrants, Y_{NM}, and potential migrants, Y_{PM}, is (Roff 1994)

$$Y_{\mathrm{NM}} = \mu + \frac{\phi(\mu)h^2}{2P}, \quad Y_{\mathrm{PM}} = \mu - \frac{\phi(\mu)h^2}{2(1-P)} \tag{1.38}$$

which can be coded in R as (respectively)

```
Y.nonmigrants <- Mu +  dnorm(0, mean=Mu,  sd=1)*h2/(2*P)
Y.migrants    <- Mu -  dnorm(0, mean=Mu,  sd=1)*h2/(2*(1-P))
```

where h^2 is the heritability of liability (set at 0.5 in the present model). The mean phenotypic value of offspring in the ith patch at generation $t+1$, $\mu_{t+1,i}$ is given by (Roff 1994)

$$\mu_{t+1,i} = \frac{N_{t,i}[Y_{\mathrm{NM},t,i}P_{t,i} + Y_{\mathrm{PM},t,i}(1-P_{t,i})(1-P_{\mathrm{mig}})C] + \frac{Y_t^*}{n}}{N_{t,i}[(1-P_{t,i}) + (1-P_{\mathrm{mig}})C] + \frac{N_{t,\mathrm{T}}}{n}} \tag{1.39}$$

where n is the number of patches, C is the "cost" incurred by potential migrants for the ability to migrate (whether done so or not) and set to 0.6 in the present simulation (this cost is consistent with the loss in fecundity of the winged morph of the sand cricket [Roff 1984a]), $N_{t,i}$ is the number in the ith patch before migration, $N_{i,\mathrm{T}}$ is the total number of migrants, and Y_t^* is the weighted phenotypic value of the offspring from migrants. The latter two variables are defined as

$$\begin{aligned} N_{t,\mathrm{T}} &= CP_{\mathrm{surv}}P_{\mathrm{mig}}\sum_{i=1}^{n} N_{t,i}(1-P_{t,i}) \\ Y_t^* &= CP_{\mathrm{surv}}P_{\mathrm{mig}}\sum_{i=1}^{n} N_{t,i}Y_{\mathrm{PM},t,i}(1-P_{t,i}) \end{aligned} \tag{1.40}$$

Equations (1.39) and (1.40) can be coded in R as

```
Emigrants    <- P.mig*Npop*(1-P)          # Nos of emigrants
Nos.migrants <- P.surv*sum(Emigrants)     # N_t,T
Y.star       <- P.mig*sum(Npop*P.surv*Y.migrants*(1-P))
Mu           <- (Npop*(Y.nonmigrants*P+Y.migrants*(1-P)*(1-P.
                mig)*Cost) + (Y.star/N.patches))/(Npop*(P+ (1-
                P)*(1-P.mig)*Cost) + Nos.migrants/N.patches)
```

Finally, we compute the change in population size within each patch and return the new population size and new mean liability for each population:

```
#  New population size before reproduction
  Npop             <-  Npop - Emigrants + Nos.migrants/N.patches
  Npop             <-  Npop*LAMBDA      #  Population size before
                                           constraints
  Npop[Npop<1]     <-  0                #  Check for extinction
  Npop[Npop>1000] <-  1000              #  Carrying capacity
# Return the vector of new population sizes and means
  return (c(Npop,Mu))
```

The full coding is

R CODE:

```
    rm(list=ls())                                          #Clearmemory
# Population and inheritance function
    POP   <-function(MAX.Lambda,Npop,N.patches,Mu,P.surv,P.mig)
{
    h2    <-  0.5; Cost <- 0.6                             # parameters
    LAMBDA <-runif(N.patches,min=0,max=MAX.Lambda) #randomlamb-
                                                            das
    P   <-  pnorm( 0, mean= -Mu, sd=1)  #  Proportion of nonmigrants
    Y.nonmigrants   <- Mu + dnorm(0,mean=Mu, sd=1)*h2/(2*P)
    Y.migrants      <- Mu - dnorm(0, mean=Mu, sd=1)*h2/(2*(1-P))
    Emigrants       <- P.mig*Npop*(1-P)     # vector of surviving
                                              emigrants
    Nos.migrants    <-P.surv*sum(Emigrants)# =N(t,T)
    Y.star          <- P.mig*sum(Npop*P.surv*Y.migrants*(1-P))
    Mu     <-  (Npop*(Y.nonmigrants*P+Y.migrants*(1-P)*(1-P.mig)
          *Cost)  +  (Y.star/N.patches))/(Npop*(P+  (1-P)*(1-P.
          mig)*Cost) + Nos.migrants/N.patches)# Calculate Nos in
          new populations
    Npop             <- Npop - Emigrants + Nos.migrants/N.patches
    Npop <- Npop*LAMBDA       # Population size before constraints
    Npop[Npop<1] 0            # Check for extinction
    Npop[Npop>1000] <- 1000   # Carrying capacity
    return (c(Npop,Mu)) # Return the vector of new popn sizes and means
} # End of function

#################### MAIN PROGRAM ####################
    set.seed(100)             #  set seed
    MAXGEN        <- 2000     #  Set maximum number of generations
    N.init        <- 20       #  Initial population size
    MAX.LAMBDA    <- 2.2      #  Maximum value of lambda
    N.patches     <- 10       #  Number of patches
    P.surv        <- 0.95     #  Survival rate of migrants
    P.mig   <-.8   #  Proportion of potential migrants migrating
    Mu  <- matrix(0,N.patches,1)   #  Initial mean liability values
    Npop <- matrix(N.init, N.patches, 1) #  Initialise populations
```

```
   Npop.Sizes<- matrix(0,MAXGEN)  # Pre-assign storage for means
   Npop.Sizes[1] <- mean(Npop)           # Store 1st generation
                                           mean population size
   N.extinct     <- matrix(0,N.patches,1)# Assign storage for nos
                                          extinct
# Pre-assign space for mean propn non-migrants
   Mean.nonmig <- matrix(0,MAXGEN) # Storage for propn nonmigrants
# Mean propn nonmigrants
   Mean.nonmig[1] <- mean(pnorm( 0, mean= -Mu, sd=1))
   Mean.mig     <- matrix(0, MAXGEN) # Storage for propn migrants
   Mean.mig[1]    <- 1-Mean.nonmig[1]   # Proportion migrants
   Igen           <- 1              # Initial generation number
   while ( Igen<MAXGEN && Npop.Sizes[Igen]>0)  # Enter while loop
{
   Igen           <- Igen+1      # Increment generation counter
# Get new population sizes and mean liabilities
   OUT      <- POP(MAX.LAMBDA, Npop, N.patches, Mu, P.surv, P.mig)
   Npop     <- OUT[1:N.patches]  # Vector of Population sizes
   n1 <- N.patches+1; n2 <-2*N.patches  # Range for mean liabilities
   Mu             <- OUT[n1:n2]       # Mean liabilities
   P.nonmigrants  <- pnorm( 0, mean=-Mu, sd=1)  # Vector of prop
                                                  nonmigrants
# Mean proportion of nonmigrants in metapopulation
   Mean.nonmig[Igen] <- sum(Npop*P.nonmigrants)/sum(Npop)
                    # mean proportion of population migrating
   Mean.mig[Igen] <- sum(Npop*(1-P.nonmigrants)*P.mig)/sum(Npop)
   Npop.Sizes[Igen]   <- mean(Npop) # Store mean population size
   N.extinct[Igen]    <- length(Npop[Npop==0]) # Store nos of ex-
                                                 tinct popns
}
   par(mfcol=c(2,2)) # Divide graphics page into four quadrants
   Gen <- seq(1,Igen)  # vector of generation numbers
   plot(Gen, Npop.Sizes[1:Igen], xlab="Generation", ylab="Mean
population size",type="l")# Mean population size over generations
plot(Gen, N.extinct[1:Igen], xlab="Generation", ylab="Number of
pops extinct", type="l", ylim=c(0,N.patches))  # Nos extinct
over generations
   plot(Gen, Mean.nonmig[1:Igen], xlab="Generation", ylab="Mean
Proportion of nonmigrants",type='l') # Mean proportion of nonnon-
                                       migrants over generations
   plot(Gen, Mean.mig[1:Igen], xlab="Generation", ylab="Mean
Proportion of migrants",type='l')  # Mean proportion of migrants
over generations
```

OUTPUT: (Figure 1.11)

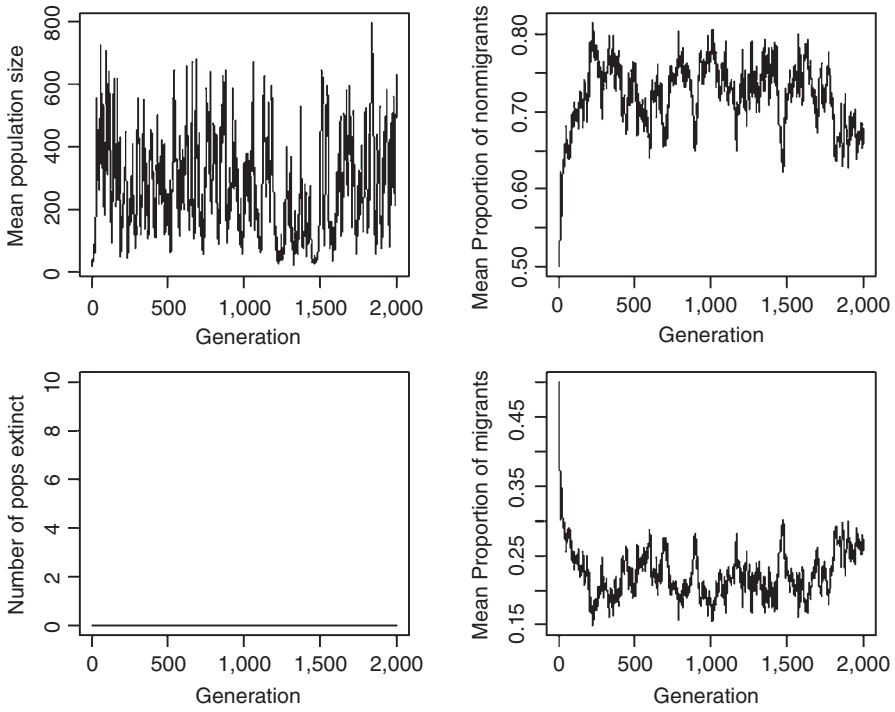

Figure 1.11 Mean population size, mean proportion of nonmigrants and actual migrants, and number of extinct populations for model 6. For display purposes the graph page has been split into four quadrats using `par(mfcol=c(2,2))`.

In addition to the population size data the program also plots the mean proportion of nonmigrants and the mean proportion of the population that actually migrate. As expected from the contour analysis, the system evolves to an intermediate level of migration. An interesting question, which I leave the reader to address is whether or not the evolutionarily stable proportion maximizes population size.

1.5 Summary of modeling approaches described in this book

1.5.1 Fisherian optimality analysis (Chapter 2)

Fisher's general analysis of evolution was based on the characteristic equation and the maximization of the Malthusian parameter, r (see Section 2.2). This approach has formed the backbone of much of the study of the evolution of trait variation. While I do not wish to imply that Fisher used only this approach, I think that he can be acknowledged as its originator. To distinguish this type of analysis from others in the book, which are also optimality models in one sense or another, I shall refer to it as "Fisherian" optimality analysis. The general assumptions of these models are