15 Simulation

The objective of this chapter is to introduce you to:

- **Simulation of "real-life" events.**

Simulation is an area of application where computers have come into their own. A simulation is a *computer experiment* which mirrors some aspect of the real world that appears to be based on random processes, or is too complicated to understand properly. (Whether events can be really random is actually a philosophical or theological question.) Some examples are: radioactive decay, rolling dice, bacteria division and traffic flow. The essence of a simulation program is that the programmer is unable to predict beforehand exactly what the outcome of the program will be, which is true to the event being simulated. For example, when you spin a coin, you do not know for sure what the result will be.

15.1 Random number generation

Random events are easily simulated in MATLAB with the function `rand`, which we have briefly encountered already. By default, `rand` returns a *uniformly distributed pseudo-random* number in the range $0 \leq$ `rand` < 1. (A computer cannot generate truly random numbers, but they can be practically unpredictable.) `rand` can also generate row or column vectors, e.g. `rand(1,5)` returns a row vector of five random numbers (1 row, 5 column) as shown:

```
0.9501    0.2311    0.6068    0.4860    0.8913
```

If you generate more random numbers during the same MATLAB session, you will get a different sequence each time, as you would expect. However, each time you start a MATLAB session, the random number sequence begins at the same place (0.9501), and continues in the same way. This is not true to life, as every gambler knows. To produce a different sequence each time you start a session, `rand` can be initially *seeded* in a different way each time you start.

15.1.1 Seeding `rand`

The random number generator `rand` can be seeded with the statement

```
rand('state', n)
```

where `n` is any integer. (By default, `n` is set to 0 when a MATLAB session starts.) This is useful if you want to generate the *same* random sequence every time a script runs, e.g. in order to debug it properly. Note that this statement does not generate any random numbers; it only initializes the generator.

However, you can also arrange for `n` to be different each time you start MATLAB by using the system time. The function `clock` returns the date and time in a six-element vector with seconds to two decimal places, so the expression `sum(100*clock)` never has the same value (well, hardly ever). You can use it to seed `rand` as follows:

```
≫ rand('state', sum(100*clock))
≫ rand(1,7)
ans =
  0.3637   0.2736   0.9910   0.3550   0.8501   0.0911   0.4493
≫ rand('state', sum(100*clock))
≫ rand(1,7)
ans =
  0.9309   0.2064   0.7707   0.7644   0.2286   0.7722   0.5315
```

Theoretically `rand` can generate over 2^{1492} numbers before repeating itself.

15.2 Spinning coins

When a fair (unbiased) coin is spun, the probability of getting heads or tails is 0.5 (50%). Since a value returned by `rand` is equally likely to anywhere in the interval [0, 1) we can represent heads, say, with a value less than 0.5, and tails otherwise.

Suppose an experiment calls for a coin to be spun 50 times, and the results recorded. In real life you may need to repeat such an experiment a number of times; this is where computer simulation is handy. The following script simulates spinning a coin 50 times:

```
for i = 1:50
  r = rand;
  if r < 0.5
    fprintf( 'H' )
  else
    fprintf( 'T' )
  end
end
fprintf( '\n' )         % newline
```

Here is the output from two sample runs:

```
THHTTHHHHTTTTTHTHTTTHHTHTTTHHTTTTHTTHHTHHHHHHTTHTT
THTHHHTHTHHTTHTHTTTHHTTTTTTTHHHTTTHTHHTHHHHTTHTHTT
```

Note that it should be impossible in principle to tell from the output alone whether the experiment was simulated or real (if the random number generator is sufficiently random).

Can you see why it would be wrong to code the `if` part of the coin simulation like this:

```
if rand < 0.5 fprintf( 'H' ), end
if rand >= 0.5 fprintf( 'T' ), end
```

The basic principle is that `rand` should be called only *once* for each 'event' being simulated. Here the single event is spinning a coin, but `rand` is called twice. Also, since two different random numbers are generated, it is quite possible that *both* logical expressions will be true, in which case '`H`' and '`T`' will both be displayed for the same coin!

15.3 Rolling dice

When a fair dice is rolled, the number uppermost is equally likely to be any integer from 1 to 6. We saw in **Counting random numbers** (Chapter 5) how to use `rand` to simulate this. The following statement generates a vector with 10 random integers in the range 1–6:

```
d = floor( 6 * rand(1,10) + 1 )
```

Here are the results of two such simulations:

```
2      1      5      5      6      3      4      5      1      1
4      5      1      3      1      3      5      4      6      6
```

We can do statistics on our simulated experiment, just as if it were a real one. For example, we could estimate the mean of the number obtained when the dice is rolled 100 times, and the probability of getting a six, say.

15.4 Bacteria division

If a fair coin is spun, or a fair dice is rolled, the different events (e.g. getting 'heads', or a 6) happen with equal likelihood. Suppose, however, that a certain type of bacteria divides (into two) in a given time interval with a probability of 0.75 (75%), and that if it does not divide, it dies. Since a value generated by `rand` is equally likely to be anywhere between 0 and 1, the chances of it being *less than* 0.75 are precisely 75%. We can therefore simulate this situation as follows:

```
r = rand;
if r < 0.75
  disp( 'I am now we' )
else
  disp( 'I am no more' )
end
```

Again, the basic principle is that one random number should be generated for each event being simulated. The single event here is the bacterium's life history over the time interval.

15.5 A random walk

A seriously short-sighted sailor has lost his contact lenses returning from a symphony concert, and has to negotiate a jetty to get to his ship. The jetty is 50 paces long and 20 wide. He is in the middle of the jetty at the quay-end, pointing toward the ship. Suppose at every step he has a 60 percent chance of stumbling blindly toward the ship, but a 20 percent chance of lurching to the left or right (he manages to be always facing the ship). If he reaches the ship-end of the jetty, he is hauled aboard by waiting mates.

The problem is to simulate his progress along the jetty, and to estimate his chances of getting to the ship without falling into the sea. To do this correctly, we must simulate one *random walk* along the jetty, find out whether or not he reaches the ship, and then repeat this simulation 1000 times, say (if we have a fast enough computer!). The proportion of simulations that end with the sailor safely in the ship will be an estimate of his chances of making it to the ship. For a given walk we assume that if he has not either reached the ship or fallen into the sea after, say, 10 000 steps, he dies of thirst on the jetty.

To represent the jetty, we set up coordinates so that the *x*-axis runs along the middle of the jetty with the origin at the quay-end. *x* and *y* are measured in steps. The sailor starts his walk at the origin each time. The structure plan and script are as follows:

```
1. Initialize variables, including number of walks n
2. Repeat n simulated walks down the jetty:
       Start at the quay-end of the jetty
       While still on the jetty and still alive repeat:
           Get a random number R for the next step
           If $R < 0.6$ then
               Move forward (to the ship)
           Else if $R < 0.8$ then
               Move port (left)
           Else
               Move starboard
       If he got to the ship then
           Count that walk as a success
3. Compute and print estimated probability of reaching the ship
4. Stop.
```

```
% random walk
n = input( 'Number of walks: ' );
nsafe = 0;                    % number of times he makes it

for i = 1:n
  steps = 0;                  % each new walk ...
  x = 0;                      % ... starts at the origin
  y = 0;
  while x <= 50 & abs(y) <= 10 & steps < 1000
    steps = steps + 1;          % that's another step
    r = rand;                   % random number for that step
    if r < 0.6                  % which way did he go?
      x = x + 1;                % maybe forward ...
    elseif r < 0.8
      y = y + 1;                % ... or to port ...
    else
      y = y - 1;                % ... or to starboard
    end;
  end;

  if x > 50
    nsafe = nsafe + 1;          % he actually made it this
                                  time!
  end;

end;

prob = 100 * nsafe / n;
disp( prob );
```

A sample run of 100 walks took about 1.7 seconds on Hahn's Celeron, and gave an 93% probability of reaching the ship.

You can speed up the script by about 20% if you generate a vector of 1000 random numbers, say, at the start of each walk (with `r = rand(1,1000);`) and then reference elements of the vector in the `while` loop, e.g.

```
if r(steps) < 0.6 ...
```

15.6 Traffic flow

A major application of simulation is in modeling the traffic flow in large cities, in order to test different traffic light patterns before inflicting them on the real traffic. In this example we look at a very small part of the problem: how to simulate the flow of a single line of traffic through one set of traffic lights. We make the following assumptions (you can make additional or different ones if you like):

1. Traffic travels straight, without turning.
2. The probability of a car arriving at the lights in a particular second is independent of what happened during the previous second. This is called a *Poisson process*. This probability (call it *p*) may be estimated by watching cars at

the intersection and monitoring their arrival pattern. In this simulation we take $p = 0.3$.

3. When the lights are green, assume the cars move through at a steady rate of, say, eight every ten seconds.
4. In the simulation, we will take the basic time period to be ten seconds, so we want a display showing the length of the queue of traffic (if any) at the lights every ten seconds.
5. We will set the lights red or green for variable multiples of ten seconds.

The situation is modeled with a script file `traffic.m` which calls three function files: `go.m`, `stop.m`, and `prq.m`. Because the function files need access to a number of base workspace variables created by `traffic.m`, these variables are declared `global` in `traffic.m`, and in all three function files.

In this example the lights are red for 40 seconds (`red = 4`) and green for 20 seconds (`green = 2`). The simulation runs for 240 seconds (`n = 24`).

The script, `traffic.m`, is as follows:

```
clc
clear                     % clear out any previous garbage!
global CARS GTIMER GREEN LIGHTS RED RTIMER T

CARS = 0;                 % number of cars in queue
GTIMER = 0;               % timer for green lights
GREEN = 2;                % period lights are green
LIGHTS = 'R';             % color of lights
n = 48;                   % number of 10-sec periods
p = 0.3;                  % probability of a car arriving
RED = 4;                  % period lights are red
RTIMER = 0;               % timer for red lights

for T = 1:n               % for each 10-sec period

  r = rand(1,10);         % 10 seconds means 10 random
                            numbers
  CARS = CARS + sum(r < p);  % cars arriving in 10
                               seconds

  if LIGHTS == 'G'
    go                    % handles green lights
  else
    stop                  % handles red lights
  end;

end;
```

The function files `go.m`, `stop.m` and `prq.m` (all separate M-files) are as follows:

```
% ----------------------------------------------------------
function go
global CARS GTIMER GREEN LIGHTS
GTIMER = GTIMER + 1;          % advance green timer
CARS = CARS - 8;              % let 8 cars through

if CARS < 0                   % ... there may have been < 8
  CARS = 0;
end;

prq;                          % display queue of cars

if GTIMER == GREEN            % check if lights need to
                                change
  LIGHTS = 'R';
  GTIMER = 0;
end;

% ----------------------------------------------------------
function stop
global LIGHTS RED RTIMER
RTIMER = RTIMER + 1;         % advance red timer
prq;                         % display queue of cars

if RTIMER == RED             % check if lights must be
                               changed
  LIGHTS = 'G';
  RTIMER = 0;
end;

% ----------------------------------------------------------
function prq
global CARS LIGHTS T
fprintf( '%3.0f ', T );       % display period number

if LIGHTS == 'R'              % display color of lights
  fprintf( 'R    ' );
else
  fprintf( 'G    ' );
end;
for i = 1:CARS                % display * for each car
   fprintf( '*' );
```

```
end;

fprintf( '\n' )             % new line
```

Typical output looks like this:

```
 1  R     ****
 2  R     ********
 3  R     ***********
 4  R     **************
 5  G     **********
 6  G     *****
 7  R     ********
 8  R     *************
 9  R     ****************
10  R     *******************
11  G     **************
12  G     **********
13  R     **************
14  R     *****************
15  R     ********************
16  R     ************************
17  G     **********************
18  G     ****************
19  R     ******************
20  R     **********************
21  R     ************************
22  R     ******************************
23  G     **************************
24  G     ***********************
```

From this particular run it seems that a traffic jam is building up, although more and longer runs are needed to see if this is really so. In that case, one can experiment with different periods for red and green lights in order to get an acceptable traffic pattern before setting the real lights to that cycle. Of course, we can get closer to reality by considering two-way traffic, and allowing cars to turn in both directions, and occasionally to break down, but this program gives the basic ideas.

15.7 Normal (Gaussian) random numbers

The function `randn` generates *Gaussian* or *normal* random numbers (as opposed to *uniform*) with a mean (μ) of 0 and a variance (σ^2) of 1.

- Generate 100 normal random numbers `r` with `randn(1,100)` and draw their histogram. Use the functions `mean(r)` and `std(r)` to find their mean and standard deviation (σ).
- Repeat with 1000 random numbers. The mean and standard deviation should be closer to 0 and 1 this time.

The functions `rand` and `randn` have separate generators, each with its own seed.

Summary

- A simulation is a computer program written to mimic a "real-life" situation which is apparently based on chance.
- The pseudo-random number generator `rand` returns uniformly distributed random numbers in the range [0, 1), and is the basis of the simulations discussed in this chapter.
- `randn` generates normally distributed (Gaussian) random numbers.
- `rand('state', n)` enables the user to seed `rand` with any integer `n`. A seed may be obtained from `clock`, which returns the system clock time.
  `randn` may be seeded (independently) in a similar way.
- Each independent event being simulated requires one and only one random number.

EXERCISES

15.1 Write some statements to simulate spinning a coin 50 times using 0-1 vectors instead of a `for` loop. **Hints:** Generate a vector of 50 random numbers, set up 0-1 vectors to represent the heads and tails, and use `double` and `char` to display them as a string of `H`s and `T`s.

15.2 In a game of Bingo the numbers 1 to 99 are drawn at random from a bag. Write a script to simulate the draw of the numbers (each number can be drawn only once), printing them ten to a line.

15.3 Generate some strings of 80 random alphabetic letters (lowercase only). For fun, see how many real words, if any, you can find in the strings.

15.4 A random number generator can be used to estimate π as follows (such a method is called a *Monte Carlo* method). Write a script which generates random points in a square with sides of length 2, say, and which counts what proportion of these points falls inside the circle of unit radius that fits exactly into the square. This proportion will be the ratio of the area of the circle to that of the square. Hence estimate π. (This is not a very efficient method, as you will see from the number of points required to get even a rough approximation.)

15.5 Write a script to simulate the progress of the short-sighted student in Chapter 16 (**Markov Processes**). Start him at a given intersection, and generate a random number to decide whether he moves toward the internet cafe or home, according to the probabilities in the transition matrix. For each simulated walk, record whether he ends up at home or in the cafe. Repeat a large number of times. The proportion of walks that end up in either place should approach the limiting probabilities computed using the Markov model described in Chapter 16. **Hint:** If the random number is less than 2/3 he moves toward the cafe (unless he is already at home or in the cafe, in which case that random walk ends), otherwise he moves toward home.

15.6 The aim of this exercise is to simulate bacteria growth.

Suppose that a certain type of bacteria divides or dies according to the following assumptions:

(a) during a fixed time interval, called a *generation*, a single bacterium divides into two identical replicas with probability p;

(b) if it does not divide during that interval, it dies;

(c) the offspring (called daughters) will divide or die during the next generation, independently of the past history (there may well be no offspring, in which case the colony becomes extinct).

Start with a single individual and write a script which simulates a number of generations. Take $p = 0.75$. The number of generations which you can simulate will depend on your computer system. Carry out a large number (e.g. 100) of such simulations. The probability of ultimate extinction, $p(E)$, may be estimated as the proportion of simulations that end in extinction. You can also estimate the mean size of the nth generation from a large number of simulations. Compare your estimate with the theoretical mean of $(2p)^n$.

Statistical theory shows that the expected value of the extinction probability $p(E)$ is the smaller of 1, and $(1-p)/p$. So for $p = 0.75$, $p(E)$ is expected to be 1/3. But for $p \le 0.5$, $p(E)$ is expected to be 1, which means that extinction is certain (a rather unexpected result). You can use your script to test this theory by running it for different values of p, and estimating $p(E)$ in each case.

15.7 Brian Hahn (the original author of this book) is indebted to a colleague, Gordon Kass, for suggesting this problem.

Dribblefire Jets Inc. make two types of airplane, the two-engined DFII, and the four-engined DFIV. The engines are terrible and fail with probability 0.5 on a standard flight (the engines fail independently of each other). The manufacturers claim that the planes can fly if at least half of their engines are working, i.e. the DFII will crash only if both its engines fail, while the DFIV will crash if all four, or if any three engines fail.

You have been commissioned by the Civil Aviation Board to ascertain which of the two models is less likely to crash. Since parachutes are expensive, the cheapest (and safest!) way to do this is to simulate a large number of flights of each model. For example, two calls of `Math.random` could represent one standard DFII flight: if both random numbers are less than 0.5, that flight crashes, otherwise it doesn't. Write a script which simulates a large number of flights of both models, and estimates the probability of a crash in each case. If you can run enough simulations, you may get a surprising result. (Incidentally, the probability of n engines failing on a given flight is given by the binomial distribution, but you do not need to use this fact in the simulation.)

15.8 Two players, A and B, play a game called *Eights*. They take it in turns to choose a number 1, 2 or 3, which may not be the same as the last number chosen (so if A starts with 2, B may only choose 1 or 3 at the next move). A starts, and may choose any of the three numbers for the first move. After each move, the number chosen is added to a common running total. If the total reaches 8 exactly, the player whose turn it was wins the game. If a player causes the total to go over 8, the other player wins. For example, suppose A starts with 1 (total 1), B chooses 2 (total 3), A chooses 1 (total 4) and B chooses 2 (total 6). A would like to play 2 now, to win, but he can't because B cunningly played it on the last move, so A chooses 1 (total 7). This is even smarter, because B is forced to play 2 or 3, making the total go over 8 and thereby losing.

Write a script to simulate each player's chances of winning, if they always play at random.

15.9 If r is a normal random number with mean 0 and variance 1 (as generated by `randn`), it can be transformed into a random number X with mean μ and standard deviation σ by the relation

$$X = \sigma r + \mu.$$

In an experiment a Geiger counter is used to count the radioactive emissions of cobalt 60 over a 10-second period. After a large number of such readings are taken, the count rate is estimated to be normally distributed with a mean of 460 and a standard deviation of 20.

1. Simulate such an experiment 200 times by generating 200 random numbers with this mean and standard deviation. Plot the histogram (use 10 bins).
2. Repeat a few times to note how the histogram changes each time.

15.10 Radioactive carbon 11 has a decay-rate k of 0.0338 per minute, i.e. a particular C^{11} atom has a 3.38% chance of decaying in any one minute.

Suppose we start with 100 such atoms. We would like to simulate their fate over a period of 100 minutes, say. We want to end up with a bar graph showing how many atoms remain undecayed after 1, 2, ..., 100 minutes.

We need to simulate when each of the 100 atoms decays. This can be done, for each atom, by generating a random number r for each of the 100 minutes, until either $r > k$ (that atom decays), or the 100 minutes is up. If the atom decayed at time $t < 100$, increment the frequency distribution $f(t)$ by 1. $f(t)$ will be the number of atoms decaying at time t minutes.

Now convert the number $f(t)$ decaying each minute to the number $R(t)$ *remaining* each minute. If there are n atoms to start with, after one minute, the number $R(1)$ remaining will be $n - f(1)$, since $f(1)$ is the number decaying during the first minute. The number $R(2)$ remaining after two minutes will be $n - f(1) - f(2)$. In general, the number remaining after t minutes will be (in MATLAB notation)

```
R(t) = n - sum( f(1:t) )
```

Write a script to compute $R(t)$ and plot its bar graph. Superimpose on the bar graph the theoretical result, which is

$$R(t) = 100 \exp^{-kt} .$$

Typical results are shown in Figure 15.1.

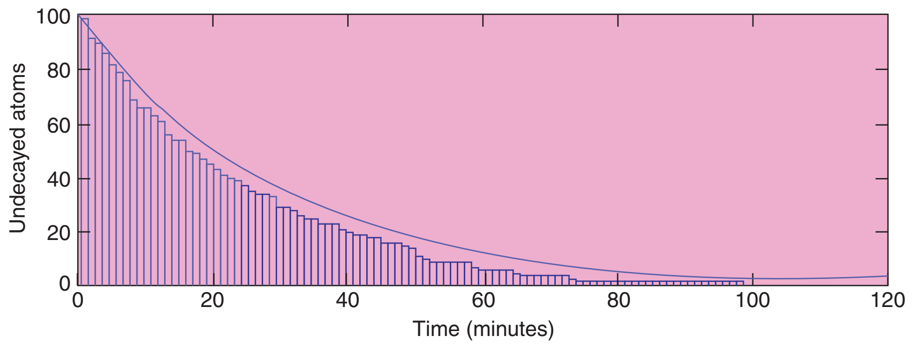

Figure 15.1 Radioactive decay of carbon 11: simulated and theoretical

16

*More matrices

The objectives of this chapter are to demonstrate the use of matrices in a number of application areas, including:

- Population dynamics.
- Markov processes.
- Linear equations.

And to introduce:

- MATLAB's sparse matrix facilities.

16.1 Leslie matrices: population growth

Our first application of matrices is in population dynamics.

Suppose we want to model the growth of a population of rabbits, in the sense that given their number at some moment, we would like to estimate the size of the population in a few years' time. One approach is to divide the rabbit population up into a number of age classes, where the members of each age class are one time unit older than the members of the previous class, the time unit being whatever is convenient for the population being studied (days, months, etc.).

If X_i is the size of the ith age class, we define a *survival factor* P_i as the proportion of the ith class that survive to the $(i+1)$th age class, i.e. the proportion that 'graduate'. F_i is defined as the *mean fertility* of the ith class. This is the mean number of newborn individuals expected to be produced during one time interval by each member of the ith class at the beginning of the interval (only females count in biological modeling, since there are always enough males to go round!).

Suppose for our modified rabbit model we have three age classes, with X_1, X_2 and X_3 members, respectively. We will call them young, middle-aged and old-aged for convenience. We will take our time unit as one month, so X_1 is the number that were born during the current month, and which will be considered as youngsters at the end of the month. X_2 is the number of middle-aged rabbits at the end of the month, and X_3 the number of oldsters. Suppose the youngsters cannot reproduce, so that $F_1 = 0$. Suppose the fertility rate for middle-aged rabbits is 9, so $F_2 = 9$, while for oldsters $F_3 = 12$. The probability of survival from youth to middle-age is one third, so $P_1 = 1/3$, while no less than half the middle-aged rabbits live to become oldsters, so $P_2 = 0.5$ (we are assuming for the sake of illustration that all old-aged rabbits die at the end of the month—this can be corrected easily). With this information we can quite easily compute the changing population structure month by month, as long as we have the population breakdown to start with.

If we now denote the current month by t, and next month by $(t + 1)$, we can refer to this month's youngsters as $X_1(t)$, and to next month's as $X_1(t + 1)$, with similar notation for the other two age classes. We can then write a scheme for updating the population from month t to month $(t + 1)$ as follows:

$$\begin{aligned}
X_1(t+1) &= F_2X_2(t) + F_3X_3(t),\\
X_2(t+1) &= P_1X_1(t),\\
X_3(t+1) &= P_2X_2(t).
\end{aligned}$$

We now define a population vector $\mathbf{X}(t)$, with three components, $X_1(t)$, $X_2(t)$, and $X_3(t)$, representing the three age classes of the rabbit population in month t. The above three equations can then be rewritten as

$$\begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix}_{(t+1)} = \begin{bmatrix} 0 & F_2 & F_3 \\ P_1 & 0 & 0 \\ 0 & P_2 & 0 \end{bmatrix} \times \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix}_t$$

where the subscript at the bottom of the vectors indicates the month. We can write this even more concisely as the matrix equation

$$\mathbf{X}(t+1) = \mathbf{L}\,\mathbf{X}(t), \tag{16.1}$$

where $\mathbf{L}$ is the matrix

$$\begin{bmatrix} 0 & 9 & 12 \\ 1/3 & 0 & 0 \\ 0 & 1/2 & 0 \end{bmatrix}$$

in this particular case. **L** is called a *Leslie matrix*. A population model can always be written in the form of Equation (16.1) if the concepts of age classes, fertility, and survival factors, as outlined above, are used.

Now that we have established a matrix representation for our model, we can easily write a script using matrix multiplication and repeated application of Equation (16.1):

$$\mathbf{X}(t+2) = \mathbf{L}\,\mathbf{X}(t+1),$$
$$\mathbf{X}(t+3) = \mathbf{L}\,\mathbf{X}(t+2), \text{ etc.}$$

We will assume to start with that we have one old (female) rabbit, and no others, so $X_1 = X_2 = 0$, and $X_3 = 1$. Here is the script:

```
% Leslie matrix population model
n = 3;
L = zeros(n);    % all elements set to zero
L(1,2) = 9;
L(1,3) = 12;
L(2,1) = 1/3;
L(3,2) = 0.5;
x = [0 0 1]';    % remember x must be a column vector!

for t = 1:24
  x = L * x;
  disp( [t x' sum(x)] )    % x' is a row
end
```

The output, over a period of 24 months (after some editing), is:

```
Month      Young    Middle       Old      Total

    1         12         0         0         12
    2          0         4         0          4
    3         36         0         2         38
    4         24        12         0         36
    5        108         8         6        122
  ...
   22   11184720   1864164    466020   13514904
   23   22369716   3728240    932082   27030038
   24   44739144   7456572   1864120   54059836
```

It so happens that there are no ‘fractional’ rabbits in this example. If there are any, they should be kept, and not rounded (and certainly not truncated). They occur because the fertility rates and survival probabilities are averages.

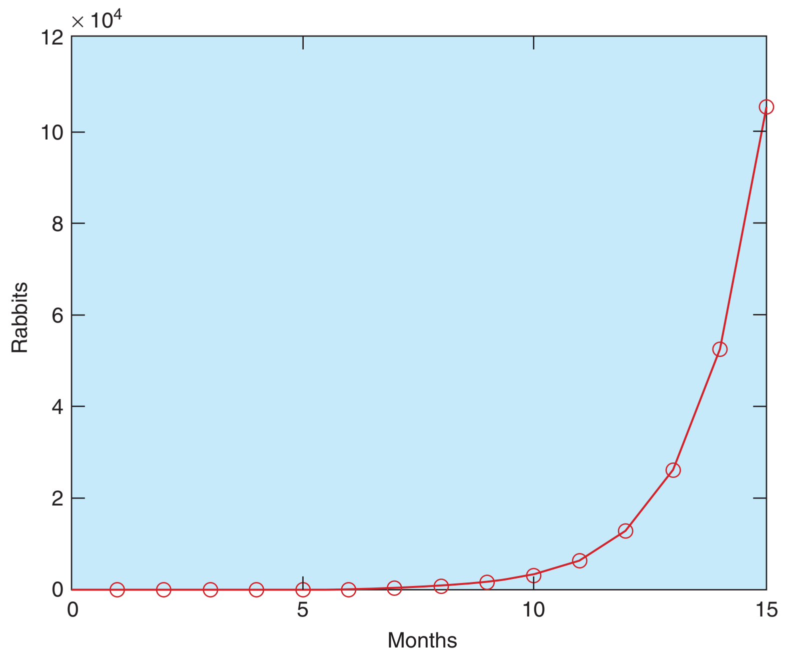

Figure 16.1 Total rabbit population over 15 months

If you look carefully at the output you may spot that after some months the total population doubles every month. This factor is called the *growth factor*, and is a property of the particular Leslie matrix being used (for those who know about such things, it's the *dominant eigenvalue* of the matrix). The growth factor is 2 in this example, but if the values in the Leslie matrix are changed, the long-term growth factor changes too (try it and see).

Figure 16.1 shows how the total rabbit population grows over the first 15 months. To draw this graph yourself, insert the line

```
p(t) = sum(x);
```

in the `for` loop after the statement `x = L * x;`, and run the program again. The vector `p` will contain the total population at the end of each month. Then enter the commands

```
plot(1:15, p(1:15)), xlabel('months'), ylabel('rabbits')
hold, plot(1:15, p(1:15),'o')
```

The graph demonstrates *exponential* growth. If you plot the population over the full 24-month period, you will see that the graph gets much steeper. This is a feature of exponential growth.

You probably didn't spot that the numbers in the three age classes tend to a limiting ratio of 24:4:1. This can be demonstrated very clearly if you run the model with an initial population structure having this limiting ratio. The limiting ratio is called the *stable age distribution* of the population, and again it is a property of the Leslie matrix (in fact, it is the *eigenvector* belonging to the dominant eigenvalue of the matrix). Different population matrices lead to different stable age distributions.

The interesting point about this is that a given Leslie matrix always eventually gets a population into the *same* stable age distribution, which increases eventually by the *same* growth factor each month, *no matter what the initial population breakdown is*. For example, if you run the above model with any other initial population, it will always eventually get into a stable age distribution of 24:4:1 with a growth factor of 2 (try it and see).

See `help eig` if you're interested in using MATLAB to compute eigenvalues and eigenvectors.

16.2 Markov processes

Often a process that we wish to model may be represented by a number of possible *discrete* (i.e. discontinuous) states that describe the outcome of the process. For example, if we are spinning a coin, then the outcome is adequately represented by the two states 'heads' and 'tails' (and nothing in between). If the process is random, as it is with spinning coins, there is a certain probability of being in any of the states at a given moment, and also a probability of changing from one state to another. If the probability of moving from one state to another depends on the present state only, and not on any previous state, the process is called a *Markov chain*. The progress of the myopic sailor in Chapter 15 is an example of such a process. Markov chains are used widely in such diverse fields as biology and business decision-making, to name just two areas.

16.2.1 A random walk

This example is a variation on the random walk simulation of Chapter 15. A street has six intersections. A short-sighted student wanders down the street. His home is at intersection 1, and his favorite internet cafe at intersection 6. At each intersection other than his home or the cafe he moves in the direction of the cafe with probability 2/3, and in the direction of his home with probability 1/3. In other words, he is twice as likely to move towards the cafe as towards his home. He never wanders down a side street. If he reaches his home or the cafe, he disappears into them, never to reappear (when he disappears we say in Markov jargon that he has been *absorbed*).

We would like to know: what are the chances of him ending up at home or in the cafe, if he starts at a given corner (other than home or the cafe, obviously)? He can clearly be in one of six states, with respect to his random walk, which can be labeled by the intersection number, where state 1 means *Home* and state 6 means *Cafe*. We can represent the probabilities of being in these states by a six-component *state vector* $\mathbf{X}(t)$, where $X_i(t)$ is the probability of him being at intersection i at moment t. The components of $\mathbf{X}(t)$ must sum to 1, since he has to be in one of these states.

We can express this Markov process with the following *transition probability matrix*, **P**, where the rows represent the next state (i.e. corner), and the columns represent the present state:

	Home	2	3	4	5	*Cafe*
Home	1	1/3	0	0	0	0
2	0	0	1/3	0	0	0
3	0	2/3	0	1/3	0	0
4	0	0	2/3	0	1/3	0
5	0	0	0	2/3	0	0
Cafe	0	0	0	0	2/3	1

The entries for *Home-Home* and *Cafe-Cafe* are both 1 because he stays there with certainty.

Using the probability matrix **P** we can work out his chances of being, say, at intersection 3 at moment $(t+1)$ as

$$X_3(t+1) = 2/3X_2(t) + 1/3X_4(t).$$

To get to 3, he must have been at either 2 or 4, and his chances of moving from there are 2/3 and 1/3, respectively.

Mathematically, this is identical to the Leslie matrix problem. We can therefore form the new state vector from the old one each time with a matrix equation:

$$\mathbf{X}(t+1) = \mathbf{P}\,\mathbf{X}(t).$$

If we suppose the student starts at intersection 2, the initial probabilities will be (0; 1; 0; 0; 0; 0). The Leslie matrix script may be adapted with very few changes to generate future states:

```
n = 6;
P = zeros(n);    % all elements set to zero

for i = 3:6
  P(i,i-1) = 2/3;
  P(i-2,i-1) = 1/3;
end

P(1,1) = 1;
P(6,6) = 1;
x = [0 1 0 0 0 0]'; % remember x must be a column vector!

for t = 1:50
  x = P * x;
  disp( [t x'] )
end
```

Edited output:

```
Time     Home         2         3         4         5     Cafe

   1   0.3333         0    0.6667         0         0         0
   2   0.3333    0.2222         0    0.4444         0         0
   3   0.4074         0    0.2963         0    0.2963         0
   4   0.4074    0.0988         0    0.2963         0    0.1975
   5   0.4403         0    0.1646         0    0.1975    0.1975
   6   0.4403    0.0549         0    0.1756         0    0.3292
   7   0.4586         0    0.0951         0    0.1171    0.3292
   8   0.4586    0.0317         0    0.1024         0    0.4073
 ...
  20   0.4829    0.0012         0    0.0040         0    0.5119
 ...
  40   0.4839    0.0000         0    0.0000         0    0.5161
 ...
  50   0.4839    0.0000         0    0.0000         0    0.5161
```

By running the program for long enough, we soon find the limiting probabilities: he ends up at home about 48 percent of the time, and at the cafe about 52 percent of the time. Perhaps this is a little surprising; from the transition probabilities, we might have expected him to get to the cafe rather more easily.

It just goes to show that you should never trust your intuition when it comes to statistics!

Note that the Markov chain approach is *not* a simulation: one gets the *theoretical* probabilities each time (this can all be done mathematically, without a computer). But it is interesting to confirm the limiting probabilities by *simulating* the student's progress, using a random number generator (see Exercise 15.5 at the end of Chapter 15).

16.3 Linear equations

A problem that often arises in scientific applications is the solution of a system of linear equations, e.g.

$$3x + 2y - z = 10 \tag{16.2}$$

$$-x + 3y + 2z = 5 \tag{16.3}$$

$$x - y - z = -1. \tag{16.4}$$

MATLAB was designed to solve a system like this directly and very easily, as we shall now see.

If we define the matrix **A** as

$$\mathbf{A} = \begin{bmatrix} 3 & 2 & -1 \\ -1 & 3 & 2 \\ 1 & -1 & -1 \end{bmatrix},$$

and the vectors **x** and **b** as

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 10 \\ 5 \\ -1 \end{bmatrix},$$

we can write the above system of three equations in matrix form as

$$\begin{bmatrix} 3 & 2 & -1 \\ -1 & 3 & 2 \\ 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 10 \\ 5 \\ -1 \end{bmatrix},$$

or even more concisely as the single matrix equation

$$\mathbf{Ax} = \mathbf{b}. \tag{16.5}$$

The solution may then be written as

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}, \tag{16.6}$$

where $\mathbf{A}^{-1}$ is the matrix inverse of $\mathbf{A}$ (i.e. the matrix which when multiplied by $\mathbf{A}$ gives the identity matrix $\mathbf{I}$).

16.3.1 MATLAB's solution

To see how MATLAB solves this system, first recall that the *left division* operator \ may be used on scalars, i.e. `a \ b` is the same as `b / a` if `a` and `b` are scalars. However, it can also be used on vectors and matrices, in order to solve linear equations. Enter the following statements on the command line to solve Equations (16.2)–(16.4):

```
A = [3 2 -1; -1 3 2; 1 -1 -1];
b = [10 5 -1]';
x = A \ b
```

This should result in

```
x =
   -2.0000
    5.0000
   -6.0000
```

In terms of our notation, this means that the solution is $x = -2, y = 5, z = -6$.

You can think of the matrix operation `A \ b` as '`b` divided by `A`', or as 'the inverse of `A` multiplied by `b`', which is essentially what Equation (16.6) means. A colleague of mine reads this operation as '`A` under `b`', which you may also find helpful.

You may be tempted to implement Equation (16.6) in MATLAB as follows:

```
x = inv(A) * b
```

since the function `inv` finds a matrix inverse directly. However, `A \ b` is actually more accurate and efficient, since it uses Gauss elimination, and therefore requires fewer *flops* (floating point operations), e.g. with a random 30×30 system `a \ b` needs 24051 flops, whereas `inv(a) * b` needs 59850—more than twice as many. (In case you're wondering where I got the number of flops from with such authority, earlier versions of MATLAB had a function `flops` which counted the number of flops. In Version 7 this is no longer practical since the algorithms have been changed.)

16.3.2 The residual

Whenever we solve a system of linear equations numerically we need to have some idea of how accurate the solution is. The first thing to check is the *residual*, defined as

```
r = A*x - b
```

where `x` is the result of the operation `x = A \ b`. Theoretically the residual `r` should be zero, since the expression `A * x` is supposed to be equal to `b`, according to Equation (16.5), which is the equation we are trying to solve. In our example here the residual is (check it)

```
r =
   1.0e-015 *
          0
     0.8882
     0.6661
```

This seems pretty conclusive: all the elements of the residual are less than 10^{-15} in absolute value. Unfortunately, there may still be problems lurking beneath the surface, as we shall see shortly.

We will, however, first look at a situation where the residual turns out to be far from zero.

16.3.3 Overdetermined systems

When we have more equations than unknowns, the system is called *over-determined*, e.g.

$$\begin{aligned} x - y &= 0 \\ y &= 2 \\ x &= 1. \end{aligned}$$

Surprisingly, perhaps, MATLAB gives a solution to such a system. If

```
A = [1 -1; 0 1; 1 0];
b = [0 2 1]';
```

the statement

```
x = A \ b
```

results in

```
x =
   1.3333
   1.6667
```

The residual `r = A*x - b` is now

```
r =
   -0.3333
   -0.3333
    0.3333
```

What happens in this case is that MATLAB produces the *least squares best fit*. This is the value of `x` which makes the *magnitude* of `r`, i.e.

$$\sqrt{\mathtt{r}(1)^2 + \mathtt{r}(2)^2 + \mathtt{r}(3)^3},$$

as small as possible. You can compute this quantity (0.5774) with `sqrt(r' * r)` or `sqrt(sum(r .* r))`. There is a nice example of fitting a decaying exponential function to data with a least squares fit in **Using MATLAB: Mathematics: Matrices and Linear Algebra: Solving Linear Equations: Overdetermined Systems**.

16.3.4 Underdetermined systems

If there are fewer equations than unknowns, the system is called *underdetermined*. In this case there are an infinite number of solutions; MATLAB will find one which has zeros for some of the unknowns.

The equations in such a system are the constraints in a linear programming problem.

16.3.5 Ill conditioning

Sometimes the coefficients of a system of equations are the results of an experiment, and may be subject to error. We need in that case to know how sensitive the solution is to the experimental errors. As an example, consider the system

$$\begin{aligned}10x + 7y + 8z + 7w &= 32\\ 7x + 5y + 6z + 5w &= 23\\ 8x + 6y + 10z + 9w &= 33\\ 7x + 5y + 9z + 10w &= 31\end{aligned}$$

Use matrix left division to show that the solution is $x = y = z = w = 1$. The residual is exactly zero (check it), and all seems well.

However, if we change the right-hand side constants to 32.1, 22.9, 32.9 and 31.1, the 'solution' is now given by $x = 6, y = -7.2, z = 2.9, w = -0.1$. The residual is very small.

A system like this is called *ill-conditioned*, meaning that a small change in the coefficients leads to a large change in the solution. The MATLAB function `rcond` returns the *condition estimator*, which tests for ill conditioning. If `A` is the coefficient matrix, `rcond(A)` will be close to zero if `A` is ill-conditioned, but close to 1 if it is well-conditioned. In this example, the condition estimator is about 2×10^{-4}, which is pretty close to zero.

Some authors suggest the rule of thumb that a matrix is ill-conditioned if its determinant is small compared to the entries in the matrix. In this case the determinant of `A` is 1 (check with the function `det`) which is about an order of magnitude smaller than most of its entries.

16.3.6 Matrix division

Matrix left division, `A\B`, is defined whenever `B` has as many rows as `A`. This corresponds formally to `inv(A)*B` although the result is obtained without computing the inverse explicitly. In general,

```
x = A \ B
```

is a solution to the system of equations defined by $\mathbf{A}x = \mathbf{B}$.

If `A` is square, matrix left division is done using Gauss elimination.

If `A` is not square the over- or underdetermined equations are solved in the least squares sense. The result is a $m \times n$ matrix `X`, where m is the number of columns of `A` and n is the number of columns of `B`.

Matrix right division, `B/A`, is defined in terms of matrix left division such that `B/A` is the same as `(A'\B')'`. So with `a` and `b` defined as for Equations (16.2)–(16.4), this means that

```
x = (b'/a')'
```

gives the same solution, doesn't it? Try it, and make sure you can see why.

Sometimes the least squares solutions computed by \ or / for over- or under-determined systems can cause surprises, since you can legally divide one *vector* by another. For example, if

```
a = [1 2];
b = [3 4];
```

the statement

```
a / b
```

results in

```
ans =
     0.4400
```

This is because `a/b` is the same as `(b'\a')'`, which is formally the solution of $\mathbf{b}' x' = \mathbf{a}'$. The result is a scalar, since `a'` and `b'` each have one column, and is the least squares solution of

$$\begin{pmatrix} 3 \\ 4 \end{pmatrix} x = \begin{pmatrix} 1 \\ 2 \end{pmatrix}.$$

With this under your belt, can you explain why

```
a \ b
```

gives

```
ans =
          0         0
     1.5000    2.0000
```

(try writing the equations out in full)?

A complete discussion of the algorithms used in solving simultaneous linear equations may be found in the online documentation under **MATLAB: Reference: MATLAB Function Reference: Alphabetical List of Functions: Arithmetic Operators** `+ - * / \ ^ '`.

16.4 Sparse matrices

Matrices can sometimes get very large, with thousands of entries. Very large matrices can occupy huge amounts of memory, and processing them can take up a lot of computer time. For example, a system of n simultaneous linear equations requires n^2 matrix entries, and the computing time to solve them is proportional to n^3.

However, some matrices have relatively few *non-zero* entries. Such matrices are called *sparse* as opposed to *full*. The father of modern numerical linear algebra, J.H. Wilkinson, has remarked that a matrix is sparse if 'it has enough zeros that it pays to take advantage of them'. MATLAB has facilities for exploiting sparsity, which have the potential for saving huge amounts of memory and processing time. For example, the matrix representation of a certain type of partial differential equation (a 5-point Laplacian) on a square 64×64 grid is a 4096×4096 element matrix with 20 224 non-zero elements. The sparse form of this matrix in MATLAB occupies only 250 kB of memory, whereas the full version of the same matrix would occupy 128 MB, which is way beyond the limits of most desktop computers. The solution of the system $\mathbf{Ax} = \mathbf{b}$ using sparse techniques is about 4000 times faster than solving the full case, i.e. 10 seconds instead of 12 hours!

In this section (which you can safely skip, if you want to) we will look briefly at how to create sparse matrices in MATLAB. For a full description of sparse matrix techniques consult **Using MATLAB: Mathematics: Sparse Matrices**.

First an example, then an explanation. The transition probability matrix for the random walk problem in Section 16.2 is a good candidate for sparse representation. Only ten of its 36 elements are non-zero. Since the non-zeros appear only on the diagonal and the sub- and superdiagonals, a matrix representing more intersections would be even sparser. For example, a 100×100 representation of the same problem would have only 198 non-zero entries, i.e. 1.98%.

To represent a sparse matrix all that MATLAB needs to record are the non-zero entries with their row and column indices. This is done with the `sparse` function. The transition matrix of Section 16.2 can be set up as a sparse matrix with the statements

```
n = 6;
P = sparse(1, 1, 1, n, n);
P = P + sparse(n, n, 1, n, n);
P = P + sparse(1:n-2, 2:n-1, 1/3, n, n);
P = P + sparse(3:n, 2:n-1, 2/3, n, n)
```

which result in (with `format rat`)

```
P =

   (1,1)       1
   (1,2)      1/3
   (3,2)      2/3
   (2,3)      1/3
   (4,3)      2/3
   (3,4)      1/3
   (5,4)      2/3
   (4,5)      1/3
   (6,5)      2/3
   (6,6)       1
```

Each line of the display of a sparse matrix gives a non-zero entry with its row and column, e.g. 2/3 in row 3 and column 2. To display a sparse matrix in full form, use the function

```
full(P)
```

which results in

```
ans =
     1    1/3     0      0      0      0
     0     0     1/3     0      0      0
     0    2/3     0     1/3     0      0
     0     0     2/3     0     1/3     0
     0     0      0     2/3     0      0
     0     0      0      0     2/3     1
```

(also with `format rat`).

The form of the `sparse` function used here is

```
sparse(rows, cols, entries, m, n)
```

This generates an `m` × `n` sparse matrix with non-zero `entries` having subscripts (`rows`, `cols`) (which may be vectors), e.g. the statement

```
sparse(1:n-2, 2:n-1, 1/3, n, n);
```

(with `n = 6`) creates a 6 × 6 sparse matrix with 4 non-zero elements, being 1/3's in rows 1–4 and columns 2–5 (most of the superdiagonal). Note that repeated use of `sparse` produces a number of 6 × 6 matrices, which must be added together to give the final form. Sparsity is therefore preserved by operations on sparse matrices. See `help` for more details of `sparse`.

It's quite easy to test the efficiency of sparse matrices. Construct a (full) identity matrix

```
a = eye(1000);
```

It takes about 27.5 seconds on Hahn's Celeron to compute `a^2`. However, `a` is an ideal candidate for being represented as a sparse matrix, since only 1000 of its one million elements are non-zero. It is represented in sparse form as

```
s = sparse(1:1000, 1:1000, 1, 1000, 1000);
```

It only takes about 0.05 seconds to compute `s^2`—more than 500 times faster!

The function `full(a)` returns the full form of the sparse matrix `a` (without changing the sparse representation of `a` itself). Conversely, `sparse(a)` returns the sparse form of the full matrix `a`.

The function `spy` provides a neat visualization of sparse matrices. Try it on `P`. Then enlarge `P` to about 50 × 50, and `spy` it.

Summary

- The matrix left division operator \ is used for solving systems of linear equations directly. Because the matrix division operators \ and \ can produce surprising results with the least squares solution method, you should always compute the residual when solving a system of equations.

➤ If you work with large matrices with relatively few non-zero entries you should consider using MATLAB's sparse matrix facilities.

EXERCISES

16.1 Compute the limiting probabilities for the student in Section 16.2 when he starts at each of the remaining intersections in turn, and confirm that the closer he starts to the cafe, the more likely he is to end up there.

Compute `P^50` directly. Can you see the limiting probabilities in the first row?

16.2 Solve the equations

$$\begin{aligned} 2x - y + z &= 4 \\ x + y + z &= 3 \\ 3x - y - z &= 1 \end{aligned}$$

using the left division operator. Check your solution by computing the residual. Also compute the determinant (`det`) and the condition estimator (`rcond`). What do you conclude?

16.3 This problem, suggested by R.V. Andree, demonstrates ill conditioning (where small changes in the coefficients cause large changes in the solution). Use the left division operator to show that the solution of the system

$$\begin{aligned} x + 5.000y &= 17.0 \\ 1.5x + 7.501y &= 25.503 \end{aligned}$$

is $x = 2$, $y = 3$. Compute the residual.

Now change the term on the right-hand side of the second equation to 25.501, a change of about one part in 12 000, and find the new solution and the residual. The solution is completely different. Also try changing this term to 25.502, 25.504, etc. If the coefficients are subject to experimental errors, the solution is clearly meaningless. Use `rcond` to find the condition estimator and `det` to compute the determinant. Do these values confirm ill conditioning?

Another way to anticipate ill conditioning is to perform a *sensitivity analysis* on the coefficients: change them all in turn by the same small percentage, and observe what effect this has on the solution.

16.4 Use `sparse` to represent the Leslie matrix in Section 16.1. Test your representation by projecting the rabbit population over 24 months.

16.5 If you are familiar with *Gauss reduction* it is an excellent programming exercise to code a Gauss reduction directly with operations on the rows of the augmented coefficient matrix. See if you can write a function

x = mygauss(a, b)

to solve the general system $\mathbf{Ax}=\mathbf{b}$. Skillful use of the colon operator in the row operations can reduce the code to a few lines!

Test it on **A** and **b** with random entries, and on the systems in Section 16.3 and Exercise 16.4. Check your solutions with left division.

17

*Introduction to numerical methods

The objectives of this chapter are to introduce numerical methods for:

- **Solving equations.**
- **Evaluating definite integrals.**
- **Solving systems of ordinary differential equations.**
- **Solving a parabolic partial differential equation.**

A major scientific use of computers is in finding numerical solutions to mathematical problems which have no analytical solutions (i.e. solutions which may be written down in terms of polynomials and standard mathematical functions). In this chapter we look briefly at some areas where *numerical methods* have been highly developed, e.g. solving nonlinear equations, evaluating integrals, and solving differential equations.

17.1 Equations

In this section we consider how to solve equations in one unknown numerically. The usual way of expressing the problem is to say that we want to solve the equation $f(x) = 0$, i.e. we want to find its *root* (or roots). This process is also described as finding the *zeros* of $f(x)$. There is no general method for finding roots analytically for an arbitrary $f(x)$.

17.1.1 Newton's method

Newton's method is perhaps the easiest numerical method to implement for solving equations, and was introduced briefly in earlier chapters. It is an *iterative*

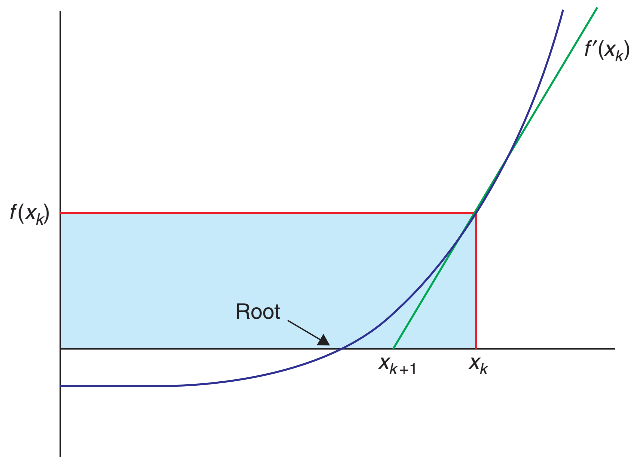

Figure 17.1 Newton's method

method, meaning that it repeatedly attempts to improve an estimate of the root. If x_k is an approximation to the root, we can relate it to the next approximation x_{k+1} using the right-angle triangle in Figure 17.1:

$$f'(x_k) = \frac{f(x_k) - 0}{x_k - x_{k+1}},$$

where $f'(x)$ is df/dx. Solving for x_{k+1} gives

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

A structure plan to implement Newton's method is:

1. Input starting value x_0 and required relative error e
2. While relative error $|(x_k - x_{k-1})/x_k| \geq e$ repeat up to $k = 20$, say:
   $x_{k+1} = x_k - f(x_k)/f'(x_k)$
   Print x_{k+1} and $f(x_{k+1})$
3. Stop.

It is necessary to limit step 2 since the process may not converge.

A script using Newton's method (without the subscript notation) to solve the equation $x^3 + x - 3 = 0$ is given in Chapter 10. If you run it you will see that the values of x converge rapidly to the root.

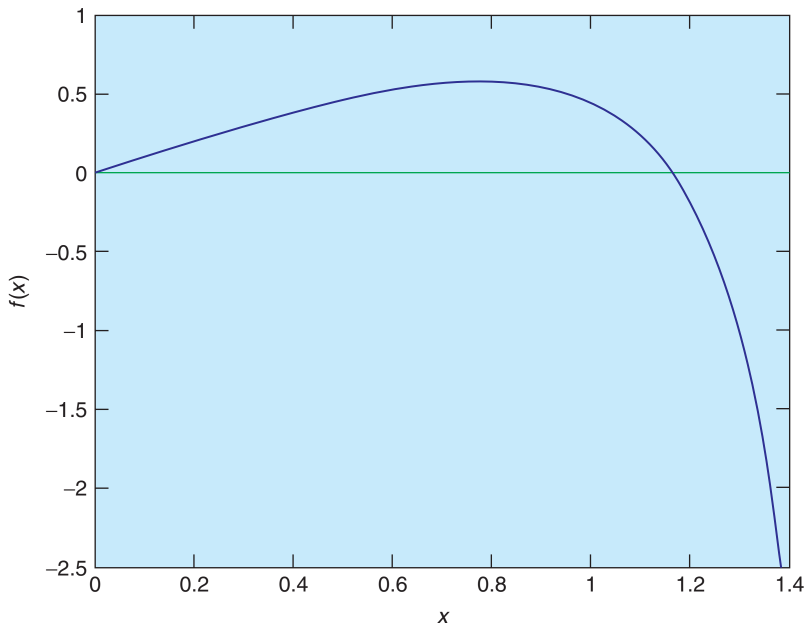

Figure 17.2 $f(x) = 2x - \tan(x)$

As an exercise, try running the script with different starting values of x_0 to see whether the algorithm always converges.

If you have a sense of history, use Newton's method to find a root of $x^3 - 2x - 5 = 0$. This is the example used when the algorithm was first presented to the French Academy.

Also try finding a *non-zero* root of $2x = tan(x)$, using Newton's method. You might have some trouble with this one. If you do, you will have discovered the one serious problem with Newton's method: it converges to a root only if the starting guess is 'close enough'. Since 'close enough' depends on the nature of $f(x)$ and on the root, one can obviously get into difficulties here. The only remedy is some intelligent trial-and-error work on the initial guess—this is made considerably easier by sketching $f(x)$ or plotting it with MATLAB (see Figure 17.2).

If Newton's method fails to find a root, the Bisection method, discussed below, can be used.

Complex roots

Newton's method can also find complex roots, but only if the starting guess is complex. Use the script in Chapter 10 to find a complex root of $x^2 + x + 1 = 0$.

Start with a complex value of `1 + i` say, for x. Using this starting value for x gives the following output (if you replace `disp( [x f(x)] )` in the script with `disp( x )`):

```
 0.0769 + 0.6154i
-0.5156 + 0.6320i
-0.4932 + 0.9090i
-0.4997 + 0.8670i
-0.5000 + 0.8660i
-0.5000 + 0.8660i
Zero found
```

Since complex roots occur in complex conjugate pairs, the other root is $-0.5 - 0.866i$.

17.1.2 The Bisection method

Consider again the problem of solving the equation $f(x) = 0$, where

$$f(x) = x^3 + x - 3.$$

We attempt to find by inspection, or trial and error, two values of x, call them x_L and x_R, such that $f(x_L)$ and $f(x_R)$ have different signs, i.e. $f(x_L)f(x_R) < 0$. If we can find two such values, the root must lie somewhere in the interval between them, since $f(x)$ changes sign on this interval (see Figure 17.3). In this example, $x_L = 1$ and $x_R = 2$ will do, since $f(1) = -1$ and $f(2) = 7$. In the Bisection method, we estimate the root by x_M, where x_M is the mid-point of the interval $[x_L, x_R]$, i.e.

$$x_M = (x_L + x_R)/2. \tag{17.1}$$

Then if $f(x_M)$ has the same sign as $f(x_L)$, as drawn in the figure, the root clearly lies between x_M and x_R. We must then redefine the left-hand end of the interval as having the value of x_M, i.e. we let the new value of x_L be x_M. Otherwise, if $f(x_M)$ and $f(x_L)$ have *different* signs, we let the new value of x_R be x_M, since the root must lie between x_L and x_M in that case. Having redefined x_L or x_R, as the case may be, we bisect the new interval again according to Equation (17.1) and repeat the process until the distance between x_L and x_R is as small as we please.

The neat thing about this method is that, *before* starting, we can calculate how many bisections are needed to obtain a certain accuracy, given initial values of x_L and x_R. Suppose we start with $x_L = a$, and $x_R = b$. After the first bisection the worst possible error (E_1) in x_M is $E_1 = |a - b|/2$, since we are estimating the

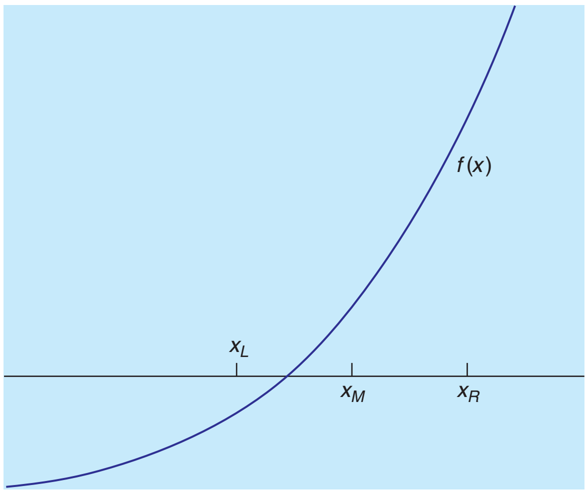

Figure 17.3 The Bisection method

root as being at the mid-point of the interval $[a, b]$. The worst that can happen is that the root is actually at x_L or x_R, in which case the error is E_1. Carrying on like this, after n bisections the worst possible error E_n is given by $E_n = |a - b|/2^n$. If we want to be sure that this is less than some specified error E, we must see to it that n satisfies the inequality $|a - b|/2^n < E$, i.e.

$$n > \frac{\log(|a - b|/E)}{\log(2)} \tag{17.2}$$

Since n is the number of bisections, it must be an integer. The smallest integer n that *exceeds* the right-hand side of Inequality (17.2) will do as the maximum number of bisections required to guarantee the given accuracy E.

The following scheme may be used to program the Bisection method. It will work for any function $f(x)$ that changes sign (in either direction) between the two values a and b, which must be found beforehand by the user.

```
1. Input $a, b$ and $E$
2. Initialize $x_L$ and $x_R$
3. Compute maximum bisections $n$ from Inequality (17.2)
4. Repeat $n$ times:
         Compute $x_M$ according to Equation (17.1)
         If $f(x_L)f(x_M) > 0$ then
            Let $x_L = x_M$
```

```
          otherwise
             Let $x_R = x_M$
5.   Display root $x_M$
6.   Stop.
```

We have assumed that the procedure will not find the root exactly; the chances of this happening with real variables are infinitesimal.

The main advantage of the Bisection method is that it is guaranteed to find a root if you can find two starting values for x_L and x_R between which the function changes sign. You can also compute in advance the number of bisections needed to attain a given accuracy. Compared to Newton's method it is inefficient. Successive bisections do not necessarily move closer to the root, as usually happens with Newton's method. In fact, it is interesting to compare the two methods on the same function to see how many more steps the Bisection method requires than Newton's method. For example, to solve the equation $x^3 + x - 3 = 0$, the Bisection method takes 21 steps to reach the same accuracy as Newton's in five steps.

17.1.3 `fzero`

The MATLAB function `fzero(@f, a)` finds the zero nearest to the value `a` of the function `f` represented by the function `f.m`.

Use it to find a zero of $x^3 + x - 3$.

`fzero` doesn't appear to be able to find complex roots.

17.1.4 `roots`

The MATLAB function M-file `roots(c)` finds all the roots (zeros) of the polynomial with coefficients in the vector `c`. See `help` for details.

Use it to find a zero of $x^3 + x - 3$.

17.2 Integration

Although most 'respectable' mathematical functions can be differentiated analytically, the same cannot be said for integration. There are no general rules for integrating, as there are for differentiating. For example, the indefinite integral of a function as simple as e^{-x^2} cannot be found analytically. We therefore need numerical methods for evaluating integrals.

This is actually quite easy, and depends on the fact that the definite integral of a function $f(x)$ between the limits $x = a$ and $x = b$ is equal to the area under $f(x)$ bounded by the x-axis and the two vertical lines $x = a$ and $x = b$. So all numerical methods for integrating simply involve more or less ingenious ways of estimating the area under $f(x)$.

17.2.1 The Trapezoidal rule

The Trapezoidal (or Trapezium) rule is fairly simple to program. The area under $f(x)$ is divided into vertical panels each of width h, called the *step-length*. If there are n such panels, then $nh = b - a$, i.e. $n = (b - a)/h$. If we join the points where successive panels cut $f(x)$, we can estimate the area under $f(x)$ as the sum of the area of the resulting trapezia (see Figure 17.4). If we call this approximation to the integral S, then

$$S = \frac{h}{2}\left[f(a) + f(b) + 2\sum_{i=1}^{n-1} f(x_i)\right], \tag{17.3}$$

where $x_i = a + ih$. Equation (17.3) is the Trapezoidal rule, and provides an estimate for the integral

$$\int_a^b f(x)\,dx.$$

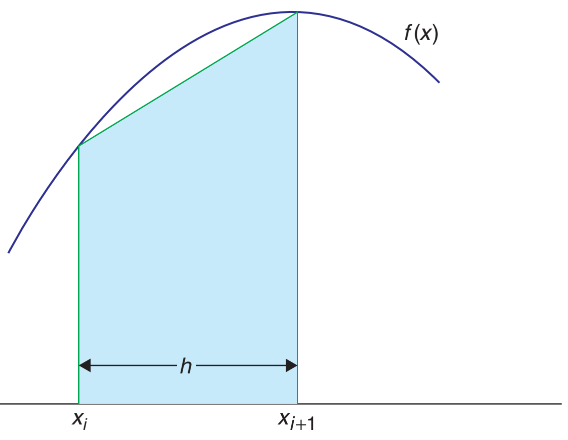

Figure 17.4 The Trapezoidal rule

Here is a function to implement the Trapezoidal rule:

```
function y = trap( fn, a, b, h )
n = (b-a)/h;
x = a + [1:n-1]*h;
y = sum(feval(fn, x));
y = h/2*(feval(fn, a) + feval(fn, b) + 2*y);
```

Note:

1. Since the summation in the rule is implemented with a vectorized formula rather than a `for` loop (to save time), the function to be integrated must use array operators where appropriate in its M-file implementation.
2. The user must choose *h* in such a way that the number of steps *n* will be an integer—a check for this could be built in.

As an exercise, integrate $f(x) = x^3$ between the limits 0 and 4 (remember to write `x.^3` in the function M-file). Call `trap` as follows:

```
s = trap(@f, 0, 4, h);
```

With $h = 0.1$, the estimate is 64.04, and with $h = 0.01$ it is 64.0004 (the exact integral is 64). You will find that as *h* gets smaller, the estimate gets more accurate.

This example assumes that $f(x)$ is a continuous function which may be evaluated at any *x*. In practice, the function could be defined at discrete points supplied as results of an experiment. For example, the speed of an object $v(t)$ might be measured every so many seconds, and one might want to estimate the distance traveled as the area under the speed-time graph. In this case, `trap` would have to be changed by replacing `fn` with a vector of function values. This is left as an exercise for the curious. Alternatively, you can use the MATLAB function `interp1` to interpolate the data. See `help`.

17.2.2 Simpson's rule

Simpson's rule is a method of numerical integration which is a good deal more accurate than the Trapezoidal rule, and should always be used before you try anything fancier. It also divides the area under the function to be integrated, $f(x)$, into vertical strips, but instead of joining the points $f(x_i)$ with straight lines, every set of three such successive points is fitted with a parabola. To ensure that there are always an even number of panels, the step-length h is usually chosen so that there are $2n$ panels, i.e. $n=(b-a)/(2h)$.

Using the same notation as above, Simpson's rule estimates the integral as

$$S=\frac{h}{3}\left[f(a)+f(b)+2\sum_{i=1}^{n-1}f(x_{2i})+4\sum_{i=1}^{n}f(x_{2i-1})\right]. \tag{17.4}$$

Coding this formula into a function M-file is left as an exercise.

If you try Simpson's rule on $f(x)=x^3$ between any limits, you will find rather surprisingly, that it gives the same result as the exact mathematical solution. This is a nice extra benefit of the rule: it integrates cubic polynomials exactly (which can be proved).

17.2.3 `quad`

Not surprisingly, MATLAB has a function `quad` to carry out numerical integration, or *quadrature* as it is also called. See `help`.

You may think there is no point in developing our own function files to handle these numerical procedures when MATLAB has its own. If you have got this far, you should be curious enough to want to know how they work, rather than treating them simply as 'black boxes'.

17.3 Numerical differentiation

The *Newton quotient* for a function $f(x)$ is given by

$$\frac{f(x+h)-f(x)}{h}, \tag{17.5}$$

where h is 'small'. As h tends to zero, this quotient approaches the first derivative, df/dx. The Newton quotient may therefore be used to estimate a derivative numerically. It is a useful exercise to do this with a few functions for which you know the derivatives. This way you can see how small you can make h before rounding errors cause problems. Such errors arise because expression (17.5) involves subtracting two terms that eventually become equal when the limit of the computer's accuracy is reached.

As an example, the following script uses the Newton quotient to estimate $f'(x)$ for $f(x) = x^2$ (which must be supplied as a function file `f.m`) at $x = 2$, for smaller and smaller values of h (the exact answer is 4).

```
h = 1;
x = 2;
format short e
for i = 1:20
  nq = (f(x+h) - f(x))/h;
  disp( [h nq] )
  h = h / 10;
end
```

Output:

```
1             5
1.0000e-001  4.1000e+000
1.0000e-002  4.0100e+000
1.0000e-003  4.0010e+000
1.0000e-004  4.0001e+000
1.0000e-005  4.0000e+000
1.0000e-006  4.0000e+000
1.0000e-007  4.0000e+000
1.0000e-008  4.0000e+000
1.0000e-009  4.0000e+000
1.0000e-010  4.0000e+000
1.0000e-011  4.0000e+000
1.0000e-012  4.0004e+000
1.0000e-013  3.9968e+000
1.0000e-014  4.0856e+000
1.0000e-015  3.5527e+000
1.0000e-016            0
...
```

The results show that the best h for this particular problem is about 10^{-8}. But for h much smaller than this the estimate becomes totally unreliable.

Generally, the best h for a given problem can only be found by trial and error. Finding it can be a non-trivial exercise. This problem does not arise with numerical integration, because numbers are *added* to find the area, not subtracted.

17.3.1 `diff`

If `x` is a row or column vector

```
[x(1)  x(2)  ...  x(n)]
```

then the MATLAB function `diff(x)` returns a vector of differences between adjacent elements:

```
[x(2)-x(1)  x(3)-x(2)  ...  x(n)-x(n-1)]
```

The output vector is one element shorter than the input vector.

In certain problems, `diff` is helpful in finding approximate derivatives, e.g. if `x` contains displacements of an object every `h` seconds, `diff(x)/h` will be its speed.

17.4 First-order differential equations

The most interesting situations in real life that we may want to model, or represent quantitatively, are usually those in which the variables change in time (e.g. biological, electrical or mechanical systems). If the changes are continuous, the system can often be represented with equations involving the derivatives of the dependent variables. Such equations are called *differential* equations. The main aim of a lot of modeling is to be able to write down a set of differential equations (DEs) that describe the system being studied as accurately as possible. Very few DEs can be solved analytically, so once again, numerical methods are required. We will consider the simplest method of numerical solution in this section: Euler's method (Euler rhymes with 'boiler'). We also consider briefly how to improve it.

17.4.1 Euler's method

In general we want to solve a first-order DE (strictly an ordinary—ODE) of the form

$$dy/dx = f(x,y), \quad y(0) \text{ given.}$$

Euler's method for solving this DE numerically consists of replacing dy/dx with its Newton quotient, so that the DE becomes

$$\frac{y(x+h) - y(x)}{h} = f(x,y).$$

After a slight rearrangement of terms, we get

$$y(x+h) = y(x) + hf(x,y). \tag{17.6}$$

Solving a DE numerically is such an important and common problem in science and engineering that it is worth introducing some general notation at this point. Suppose we want to integrate the DE over the interval $x = a$ ($a = 0$ usually) to $x = b$. We break this interval up into m steps of length h, so

$$m = (b - a)/h$$

(this is the same as the notation used in the update process of Chapter 11, except that dt used there has been replaced by the more general h here).

For consistency with MATLAB's subscript notation, if we define y_i as $y(x_i)$ (the Euler estimate at the *beginning* of step i), where $x_i = (i - 1)h$, then $y_{i+1} = y(x + h)$, at the end of step i. We can then replace Equation (17.6) by the iterative scheme

$$y_{i+1} = y_i + hf(x_i, y_i), \tag{17.7}$$

where $y_1 = y(0)$. Recall from Chapter 11 that this notation enables us to generate a vector y which we can then plot. Note also the striking similarity between Equation (17.7) and the equation in Chapter 11 representing an update process. This similarity is no coincidence. Update processes are typically modeled by DEs, and Euler's method provides an approximate solution for such DEs.

17.4.2 Example: bacteria growth

Suppose a colony of 1000 bacteria is multiplying at the rate of $r = 0.8$ per hour per individual (i.e. an individual produces an average of 0.8 offspring every hour). How many bacteria are there after 10 hours? Assuming that the colony grows continuously and without restriction, we can model this growth with the DE

$$\mathrm{d}N/\mathrm{d}t = rN, \quad N(0) = 1000, \tag{17.8}$$

where $N(t)$ is the population size at time t. This process is called *exponential growth*. Equation (17.8) may be solved analytically to give the well-known formula for exponential growth:

$$N(t) = N(0)e^{rt}.$$

To solve Equation (17.8) numerically, we apply Euler's algorithm to it to get

$$N_{i+1} = N_i + rhN_i, \tag{17.9}$$

where the initial value $N_1 = 1000$.

It is very easy to program Euler's method. The following script implements Equation (17.9), taking $h = 0.5$. It also computes the exact solution for comparison.

```
h = 0.5;
r = 0.8;
a = 0;
b = 10;
m = (b - a) / h;
N = zeros(1, m+1);
N(1) = 1000;
t = a:h:b;

for i = 1:m
   N(i+1) = N(i) + r * h * N(i);
end

Nex = N(1) * exp(r * t);
format bank
disp( [t' N' Nex'] )

plot(t, N ), xlabel( 'Hours' ), ylabel( 'Bacteria' )
hold on
plot(t, Nex ), hold off
```

Results are shown in Table 17.1, and also in Figure 17.5. The Euler solution is not too good. In fact, the error gets worse at each step, and after 10 hours of bacteria time it is about 72 percent. The numerical solution will improve if we make h smaller, but there will always be some value of t where the error exceeds some acceptable limit.

In some cases, Euler's method performs better than it does here, but there are other numerical methods which always do better than Euler. Two of them are discussed below. More sophisticated methods may be found in most textbooks on numerical analysis. However, Euler's method may always be used as a first approximation as long as you realize that errors may arise.

17.4.3 Alternative subscript notation

Equation 17.9 is an example of a *finite difference scheme*. The conventional finite difference notation is for the initial value to be represented by N_0, i.e. with subscript $i = 0$. N_i is then the estimate at the *end* of step i. If you want the MATLAB subscripts in the Euler solution to be the same as the finite difference subscripts, the initial value N_0 must be represented by the MATLAB scalar `N0`,

Table 17.1 Bacteria growth

Time (hours)	Euler	Predictor-Corrector	Exact
0.0	000	1000	1000
0.5	1400	1480	1492
1.0	1960	2190	2226
1.5	2744	3242	3320
2.0	3842	4798	4953
...			
5.0	28925	50422	54598
...			
8.0	217795	529892	601845
...			
10.0	836683	2542344	2980958

and you have to compute `N(1)` separately, before the `for` loop starts. You also have to display or plot the initial values separately since they will no longer be included in the MATLAB vectors `t`, `N` and `Nex` (which now have m instead of $m+1$ elements). Here is a complete script to generate the Euler solution using finite difference subscripts:

```
h = 0.5;
r = 0.8;
a = 0;
b = 10;
m = (b - a) / h;
N = zeros(1, m);      % one less element now
N0 = 1000;
N(1) = N0 + r*h*N0;   % no longer 'self-starting'

for i = 2:m
  N(i) = N(i-1) + r * h * N(i-1); %finite difference
          notation
end
```

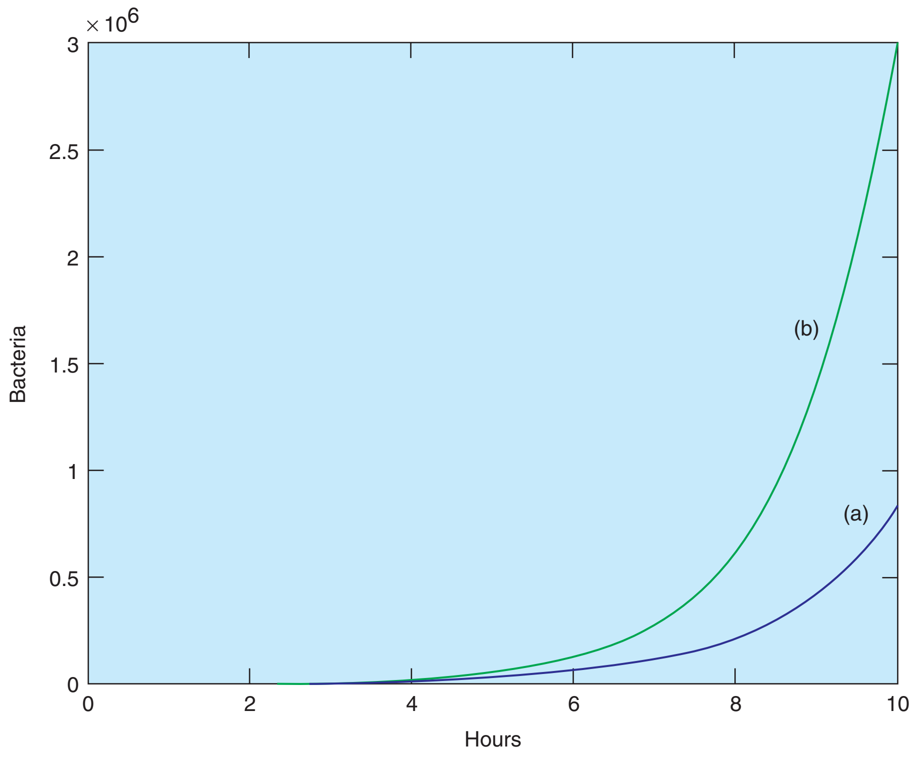

Figure 17.5 Bacteria growth: (a) Euler's method; (b) the exact solution

```
t = a+h:h:b;           % exclude initial time = a
Nex = N0 * exp(r * t);
disp( [a N0 N0] )      % display initial values separately
disp( [t' N' Nex'] )

plot(a, N0)            % plot initial values separately
hold on
plot(t, N ), xlabel( 'Hours' ), ylabel( 'Bacteria' )
plot(t, Nex ), hold off
```

17.4.4 A predictor-corrector method

One improvement on the numerical solution of the first-order DE

$$dy/dx = f(x, y), \quad y(0) \text{ given},$$

is as follows. The Euler approximation, which we are going to denote by an asterisk, is given by

$$y^*_{i+1} = y_i + hf(x_i, y_i) \tag{17.10}$$

But this formula favors the old value of y in computing $f(x_i, y_i)$ on the right-hand side. Surely it would be better to say

$$y^*_{i+1} = y_i + h[f(x_{i+1}, y^*_{i+1}) + f(x_i, y_i)]/2, \tag{17.11}$$

where $x_{i+1} = x_i + h$, since this also involves the new value y^*_{i+1} in computing f on the right-hand side? The problem of course is that y^*_{i+1} is as yet unknown, so we can't use it on the right-hand side of Equation (17.11). But we could use Euler to estimate (predict) y^*_{i+1} from Equation (17.10) and then use Equation (17.11) to correct the prediction by computing a *better* version of y^*_{i+1}, which we will call y_{i+1}. So the full procedure is:

Repeat as many times as required:

Use Euler to predict: $y^*_{i+1} = y_i + hf(x_i, y_i)$
Then correct y^*_{i+1} to: $y_{i+1} = y_i + h[f(x_{i+1}, y^*_{i+1}) + f(x_i, y_i)]/2$.

This is called a *predictor-corrector* method. The script above can easily be adapted to this problem. The relevant lines of code are:

```
for i = 1:m     % m steps of length dt
  ne(i+1) = ne(i) + r * h * ne(i);
  np = nc(i) + r * h * nc(i);
  nc(i+1) = nc(i) + r * h * (np + nc(i))/2;
  disp( [t(i+1) ne(i+1) nc(i+1) nex(i+1)] )
end;
```

`ne` stands for the 'straight' (uncorrected) Euler solution, `np` is the Euler predictor (since this is an intermediate result a vector is not needed for `np`), and `nc` is the corrector. The worst error is now only 15 percent. This is much better than the uncorrected Euler solution, although there is still room for improvement.

17.5 Linear ordinary differential equations (LODEs)

Linear ODEs with constant coefficients may be solved analytically in terms of *matrix exponentials*, which are represented in MATLAB by the function `expm`. For an example see **Using MATLAB: Mathematics: Matrices and Linear Algebra: Matrix Powers and Exponentials**.

17.6 Runge-Kutta methods

There are a variety of algorithms, under the general name of Runge-Kutta, which can be used to integrate systems of ODEs. The formulae involved are rather complicated; they can be found in most books on numerical analysis.

However, as you may have guessed, MATLAB has plenty of ODE solvers, which are discussed in **Using MATLAB: Mathematics: Differential Equations**. Among them are `ode23` (second/third order) and `ode45` (fourth/fifth order), which implement Runge-Kutta methods. (The *order* of a numerical method is the power of *h* (i.e. *dt*) in the leading error term. Since *h* is generally very small, the higher the power, the smaller the error.) We will demonstrate the use of `ode23` and `ode45` here, first with a single first-order DE, and then with systems of such equations.

17.6.1 A single differential equation

Here's how to use `ode23` to solve the bacteria growth problem, Equation (17.8):

$$dN/dt = rN, \quad N(0) = 1000.$$

1. Start by writing a function file for the *right-hand side* of the DE to be solved. The function must have input variables *t* and *N* in this case (i.e. independent and dependent variables of the DE), *in that order*, e.g. create the function file **f.m** as follows:

```
function y = f(t, Nr)
y = 0.8 * Nr;
```

2. Now enter the following statements in the Command Window:

```
a = 0;
b = 10;
n0 = 1000;
[t, Nr] = ode23(@f, [a:0.5:b], n0);
```

3. Note the input arguments of `ode23`:

   `@f`: a handle for the function `f`, which contains the right-hand side of the DE;

   `[a:0.5:b]`: a vector (`tspan`) specifying the range of integration. If `tspan` has two elements (`[a b]`) the solver returns the solution evaluated at every integration step (the solver chooses the integration steps and may vary them). This form would be suitable for plotting. However, if you want to display the solution at regular time intervals, as we want to here, use the form of `tspan` with three elements as above. The solution is then returned evaluated at each time in `tspan`. The accuracy of the solution is not affected by the form of `tspan` used.

   `n0`: the initial value of the solution *N*.

4. The output arguments are two vectors: the solutions `Nr` at times `t`. After 10 hours `ode23` gives a value of 2961338 bacteria. From the exact solution in Table 17.1 we see that the error here is only 0.7 percent.

If the solutions you get from `ode23` are not accurate enough, you can request greater accuracy with an additional optional argument. See `help`.

If you need still more accurate numerical solutions, you can use `ode45` instead. It gives a final value for the bacteria of 2981290—an error of about 0.01%.

17.6.2 Systems of differential equations: chaos

The reason that weather prediction is so difficult and forecasts are so erratic is no longer thought to be the complexity of the system but the nature of the DEs modeling it. These DEs belong to a class referred to as *chaotic*. Such equations will produce wildly different results when their initial conditions are changed infinitesimally. In other words, accurate weather prediction depends crucially on the accuracy of the measurements of the initial conditions.

Edward Lorenz, a research meteorologist, discovered this phenomenon in 1961. Although his original equations are far too complex to consider here, the following much simpler system has the same essential chaotic features:

$$\mathrm{d}x/\mathrm{d}t = 10(y - x), \tag{17.12}$$

$$\mathrm{d}y/\mathrm{d}t = -xz + 28x - y, \tag{17.13}$$

$$\mathrm{d}z/\mathrm{d}t = xy - 8z/3. \tag{17.14}$$

This system of DEs may be solved very easily with the MATLAB ODE solvers. The idea is to solve the DEs with certain initial conditions, plot the solution, then change the initial conditions very slightly, and superimpose the new solution over the old one to see how much it has changed.

We begin by solving the system with the initial conditions $x(0) = -2$, $y(0) = -3.5$ and $z(0) = 21$.

1. Write a function file `lorenz.m` to represent the right-hand sides of the system as follows:

   ```
   function f = lorenz(t, x)
   f = zeros(3,1);
   f(1) = 10 * (x(2) - x(1));
   f(2) = -x(1) * x(3) + 28 * x(1) - x(2);
   f(3) = x(1) * x(2) - 8 * x(3) / 3;
   ```

   The three elements of the MATLAB vector `x`, i.e. `x(1)`, `x(2)` and `x(3)`, represent the three dependent scalar variables x, y and z respectively. The elements of the vector `f` represent the right-hand sides of the three DEs. When a vector is returned by such a DE function it must be a column vector, hence the statement

   ```
   f = zeros(3,1);
   ```

2. Now use the following commands to solve the system from $t = 0$ to $t = 10$, say:

   ```
   x0 = [-2 -3.5 21];    % initial values in a vector
   [t, x] = ode45(@lorenz, [0 10], x0);
   plot(t,x)
   ```

   Note that we are use `ode45` now, since it is more accurate. (If you aren't using a Pentium it will take quite a few seconds for the integration to be completed.)

   You will see three graphs, for x, y and z (in different colors).

3. It's easier to see the effect of changing the initial values if there is only one graph in the figure to start with. It is in fact best to plot the solution $y(t)$ on its own.

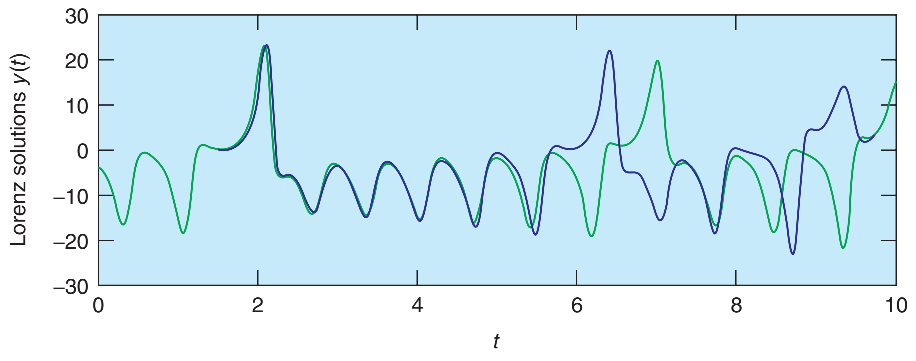

Figure 17.6 Chaos?

The MATLAB solution `x` is actually a matrix with three columns (as you can see from `whos`). The solution $y(t)$ that we want will be the second column, so to plot it by itself use the command

```
plot(t,x(:,2),'g')
```

Then keep the graph on the axes with the command `hold`.

Now we can see the effect of changing the initial values. Let's just change the initial value of $x(0)$, from -2 to -2.04—that's a change of only 2 percent, and in only one of the three initial values. The following commands will do this, solve the DEs, and plot the new graph of $y(t)$ (in a different color):

```
x0 = [-2.04 -3.5 21];
[t, x] = ode45(@lorenz, [0 10], x0);
plot(t,x(:,2),'r')
```

You should see (Figure 17.6) that the two graphs are practically indistinguishable until t is about 1.5. The discrepancy grows quite gradually, until t reaches about 6, when the solutions suddenly and shockingly flip over in opposite directions. As t increases further, the new solution bears no resemblance to the old one.

Now solve the system (17.12)–(17.14) with the original initial values using `ode23` this time:

```
x0 = [-2 -3.5 21];
[t,x] = ode23(@lorenz, [0 10], x0);
```

Plot the graph of $y(t)$ only—`x(:,2)`—and then superimpose the `ode45` solution *with the same initial values* (in a different color).

A strange thing happens—the solutions begin to deviate wildly for $t > 1.5$! The initial conditions are the same—the only difference is the order of the Runge-Kutta method.

Finally solve the system with `ode23s` and superimpose the solution. (The `s` stands for 'stiff'. For a stiff DE, solutions can change on a time scale that is very short compared to the interval of integration.) The `ode45` and `ode23s` solutions only start to diverge at $t > 5$.

The explanation is that `ode23`, `ode23s` and `ode45` all have numerical inaccuracies (if one could compare them with the exact solution—which incidentally can't be found). However, the numerical inaccuracies are *different* in the three cases. This difference has the same effect as starting the numerical solution with very slightly different initial values.

How do we ever know when we have the 'right' numerical solution? Well, we don't—the best we can do is increase the accuracy of the numerical method until no further wild changes occur over the interval of interest. So in our example we can only be pretty sure of the solution for $t < 5$ (using `ode23s` or `ode45`). If that's not good enough, you have to find a more accurate DE solver.

So beware: 'chaotic' DEs are very tricky to solve!

Incidentally, if you want to see the famous 'butterfly' picture of chaos, just plot x against z as time increases (the resulting graph is called a *phase plane* plot). The following command will do the trick:

```
plot(x(:,1), x(:,3))
```

What you will see is a static 2-D projection of the *trajectory*, i.e. the solution developing in time. **Demos** in the MATLAB Launch Pad include an example which enables you to see the trajectory evolving dynamically in 3-D (**Demos: Graphics: Lorenz attractor animation**).

17.6.3 Passing additional parameters to an ODE solver

In the above examples of the MATLAB ODE solvers the coefficients in the right-hand sides of the DEs (e.g. the value 28 in Equation (17.13)) have all been

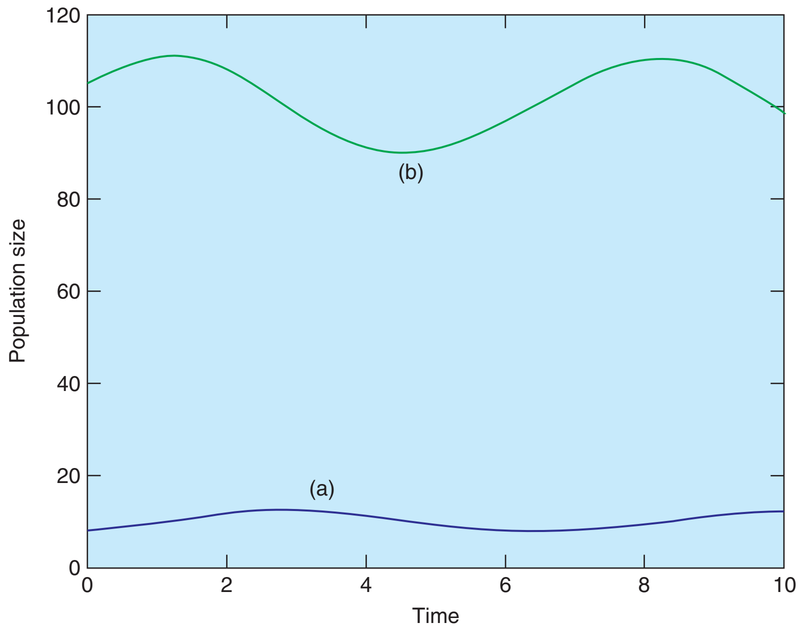

Figure 17.7 Lotka-Volterra model: (a) predator; (b) prey

constants. In a real modeling situation, you will most likely want to change such coefficients frequently. To avoid having to edit the function files each time you want to change a coefficient, you can pass the coefficients as additional parameters to the ODE solver, which in turn passes them to the DE function. To see how this may be done, consider the Lotka-Volterra *predator-prey* model:

$$\mathrm{d}x/\mathrm{d}t = px - qxy \tag{17.15}$$

$$\mathrm{d}y/\mathrm{d}t = rxy - sy, \tag{17.16}$$

where $x(t)$ and $y(t)$ are the prey and predator population sizes at time t, and p, q, r and s are biologically determined parameters. For this example, we take $p = 0.4$, $q = 0.04$, $r = 0.02$, $s = 2$, $x(0) = 105$ and $y(0) = 8$.

First, write a function M-file, `volterra.m` as follows:

```
function f = volterra(t, x, p, q, r, s)
f = zeros(2,1);
f(1) = p*x(1) - q*x(1)*x(2);
f(2) = r*x(1)*x(2) - s*x(2);
```

Then enter the following statements in the Command Window, which generate the characteristically oscillating graphs in Figure 17.7:

```
p = 0.4; q = 0.04; r = 0.02; s = 2;
[t,x] = ode23(@volterra,[0 10],[105; 8],[],p,q,r,s);
plot(t, x)
```

Note:

- The additional parameters (`p`, `q`, `r` and `s`) have to follow the fourth input argument (`options`—see `help`) of the ODE solver. If no options have been set (as in our case), use `[]` as a placeholder for the `options` parameter.

You can now change the coefficients from the Command Window and get a new solution, without editing the function file.

17.7 A partial differential equation

The numerical solution of partial differential equations (PDEs) is a vast subject, which is beyond the scope of this book. However, a class of PDEs called *parabolic* often lead to solutions in terms of sparse matrices, which were mentioned briefly in Chapter 16. One such example is considered in this section.

17.7.1 Heat conduction

The conduction of heat along a thin uniform rod may be modeled by the partial differential equation

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \tag{17.17}$$

where $u(x, t)$ is the temperature distribution a distance x from one end of the rod at time t, and assuming that no heat is lost from the rod along its length.

Half the battle in solving PDEs is mastering the notation. We set up a rectangular grid, with step-lengths of h and k in the x and t directions respectively. A general point on the grid has coordinates $x_i = ih$, $y_j = jk$. A concise notation for $u(x, t)$ at x_i, y_j is then simply $u_{i,j}$.

Truncated Taylor series may then be used to approximate the PDE by a *finite difference scheme*. The left-hand side of Equation (17.17) is usually approximated by a *forward difference*:

$$\frac{\partial u}{\partial t} = \frac{u_{i,j+1} - u_{i,j}}{k}$$

One way of approximating the right-hand side of Equation (17.17) is by the scheme

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}. \tag{17.18}$$

This leads to a scheme, which although easy to compute, is only conditionally stable.

If however we replace the right-hand side of the scheme in Equation (17.18) by the mean of the finite difference approximation on the jth and $(j+1)$th time rows, we get (after a certain amount of algebra!) the following scheme for Equation (17.17):

$$-ru_{i-1,j+1} + (2+2r)u_{i,j+1} - ru_{i+1,j+1} = ru_{i-1,j} + (2-2r)u_{i,j} + ru_{i+1,j}, \tag{17.19}$$

where $r = k/h^2$. This is known as the Crank-Nicolson *implicit* method, since it involves the solution of a system of simultaneous equations, as we shall see.

To illustrate the method numerically, let's suppose that the rod has a length of 1 unit, and that its ends are in contact with blocks of ice, i.e. the *boundary conditions* are

$$u(0,t) = u(1,t) = 0. \tag{17.20}$$

Suppose also that the initial temperature (*initial condition*) is

$$u(x,0) = \begin{cases} 2x, & 0 \le x \le 1/2, \\ 2(1-x), & 1/2 \le x \le 1. \end{cases} \tag{17.21}$$

(This situation could come about by heating the center of the rod for a long time, with the ends kept in contact with the ice, removing the heat source at time $t = 0$.) This particular problem has symmetry about the line $x = 1/2$; we exploit this now in finding the solution.

If we take $h = 0.1$ and $k = 0.01$, we will have $r = 1$, and Equation (17.19) becomes

$$-u_{i-1,j+1} + 4u_{i,j+1} - u_{i+1,j+1} = u_{i-1,j} + u_{i+1,j}. \tag{17.22}$$

Putting $j = 0$ in Equation (17.22) generates the following set of equations for the unknowns $u_{i,1}$ (i.e. after one time step k) up to the mid-point of the rod, which is represented by $i = 5$, i.e. $x = ih = 0.5$. The subscript $j = 1$ has been dropped for clarity:

$$
\begin{aligned}
0 + 4u_1 - u_2 &= 0 + 0.4 \\
-u_1 + 4u_2 - u_3 &= 0.2 + 0.6 \\
-u_2 + 4u_3 - u_4 &= 0.4 + 0.8 \\
-u_3 + 4u_4 - u_5 &= 0.6 + 1.0 \\
-u_4 + 4u_5 - u_6 &= 0.8 + 0.8.
\end{aligned}
$$

Symmetry then allows us to replace u_6 in the last equation by u_4. These equations can be written in matrix form as

$$
\begin{bmatrix} 4 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & -2 & 4 \end{bmatrix}
\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix}
=
\begin{bmatrix} 0.4 \\ 0.8 \\ 1.2 \\ 1.6 \\ 1.6 \end{bmatrix}. \tag{17.23}
$$

The matrix (**A**) on the left of Equations (17.23) is known as a *tridiagonal* matrix. Having solved for the $u_{i,1}$ we can then put $j = 1$ in Equation (17.22) and proceed to solve for the $u_{i,2}$, and so on. The system (17.23) can of course be solved directly in MATLAB with the left division operator. In the script below, the general form of Equations (17.23) is taken as

$$
\mathbf{Av} = \mathbf{g}. \tag{17.24}
$$

Care needs to be taken when constructing the matrix **A**. The following notation is often used:

$$
\mathbf{A} = \begin{bmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & a_3 & b_3 & c_3 & & \\ & & & \dots & & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{bmatrix}.
$$

A is an example of a sparse matrix (see Chapter 16).

The script below implements the general Crank-Nicolson scheme of Equation (17.19) to solve this particular problem over 10 time steps of $k = 0.01$. The step-length is specified by $h = 1/(2n)$ because of symmetry. r is therefore not restricted to the value 1, although it takes this value here. The script exploits the sparsity of **A** by using the `sparse` function.

```
format compact
n = 5;
k = 0.01;
h =  1 / (2 * n);                       % symmetry assumed
r = k / h ^ 2;

% set up the (sparse) matrix A
b = sparse(1:n, 1:n, 2+2*r, n, n);      % b(1) .. b(n)
c = sparse(1:n-1, 2:n, -r, n, n);       % c(1) .. c(n-1)
a = sparse(2:n, 1:n-1, -r, n, n);       % a(2) ..
A = a + b + c;
A(n, n-1) = -2 * r;                     % symmetry: a(n)
full(A)                                 %
disp(' ')

u0 = 0;                     % boundary condition (Eq 19.20)
u = 2*h*[1:n]               % initial conditions (Eq 19.21)
u(n+1) = u(n-1);            % symmetry
disp([0 u(1:n)])

for t = k*[1:10]
  g = r * ([u0 u(1:n-1)] + u(2:n+1)) ...
                         + (2 - 2 * r) * u(1:n);
                            % Eq 19.19
  v = A\g';                 % Eq 19.24
  disp([t v'])
  u(1:n) = v;
  u(n+1) = u(n-1);          % symmetry
end
```

Note:

➤ to preserve consistency between the formal subscripts of Equation (17.19) etc. and MATLAB subscripts, u_0 (the boundary value) is represented by the scalar `u0`.

In the following output the first column is time, and subsequent columns are the solutions at intervals of *h* along the rod:

```
     0    0.2000    0.4000    0.6000    0.8000    1.0000
0.0100    0.1989    0.3956    0.5834    0.7381    0.7691
0.0200    0.1936    0.3789    0.5397    0.6461    0.6921
   ...
0.1000    0.0948    0.1803    0.2482    0.2918    0.3069
```

MATLAB has some built-in PDE solvers. See **Using MATLAB: Mathematics: Differential Equations: Partial Differential Equations**.

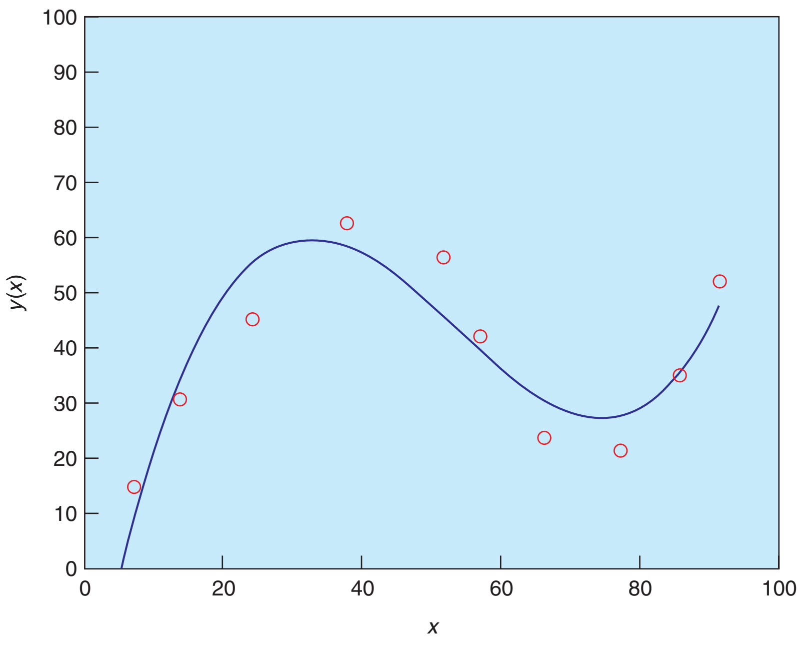

Figure 17.8 A cubic polynomial fit

17.8 Other numerical methods

The ODEs considered earlier in this chapter are all *initial value* problems. For *boundary value* problem solvers, see **Using MATLAB: Mathematics: Differential Equations: Boundary Value Problems for ODEs**.

MATLAB has a large number of functions for handling other numerical procedures, such as curve fitting, correlation, interpolation, minimization, filtering and convolution, and (fast) Fourier transforms. Consult **Using MATLAB: Mathematics: Polynomials and Interpolation** and **Data Analysis and Statistics**.

Here's an example of curve fitting. The following script enables you to plot data points interactively. When you have finished plotting points (signified when the *x* coordinates of your last two points differ by less than 2 in absolute value) a cubic polynomial is fitted and drawn (see Figure 17.8).

```
% Interactive script to fit a cubic to data points

clf
hold on
axis([0 100 0 100]);
```

```
diff = 10;
xold = 68;
i = 0;
xp = zeros(1);       % data points
yp = zeros(1);
while diff > 2
[a b] = ginput(1);
diff = abs(a - xold);
if diff > 2
    i = i + 1;
    xp(i) = a;
    yp(i) = b;
    xold = a;
    plot(a, b, 'ok')
  end
end

p = polyfit(xp, yp, 3 );
x = 0:0.1:xp(length(xp));
y= p(1)*x.^3 + p(2)*x.^2 + p(3)*x + p(4);
plot(x,y), title( 'cubic polynomial fit'), ...
   ylabel('y(x)'), xlabel('x')
hold off
```

Polynomial fitting may also be done interactively in a figure window, with **Tools** -> **Basic Fitting**.

Summary

- A numerical method is an approximate computer method for solving a mathematical problem which often has no analytical solution.
- A numerical method is subject to two distinct types of error: rounding error in the computer solution, and *truncation error*, where an infinite mathematical process, like taking a limit, is approximated by a finite process.
- MATLAB has a large number of useful functions for handling numerical methods.

EXERCISES

17.1 Use Newton's method in a script to solve the following (you may have to experiment a bit with the starting values). Check all your answers with `fzero`. Check the answers involving polynomial equations with `roots`.

Hint: Use `fplot` to get an idea of where the roots are, e.g.

```
fplot('x^3-8*x^2+17*x-10', [0 3])
```

The Zoom feature also helps. In the figure window select the Zoom In button (magnifying glass) and click on the part of the graph you want to magnify.

(a) $x^4 - x = 10$ (two real roots and two complex roots)
(b) $e^{-x} = \sin x$ (infinitely many roots)
(c) $x^3 - 8x^2 + 17x - 10 = 0$ (three real roots)
(d) $\log x = \cos x$
(e) $x^4 - 5x^3 - 12x^2 + 76x - 79 = 0$ (four real roots)

17.2 Use the Bisection method to find the square root of 2, taking 1 and 2 as initial values of x_L and x_R. Continue bisecting until the maximum error is less than 0.05 (use Inequality (17.2) of Section 17.1 to determine how many bisections are needed).

17.3 Use the Trapezoidal rule to evaluate $\int_0^4 x^2 dx$, using a step-length of $h = 1$.

17.4 A human population of 1000 at time $t = 0$ grows at a rate given by

$$dN/dt = aN,$$

where $a = 0.025$ per person per year. Use Euler's method to project the population over the next 30 years, working in steps of (a) $h = 2$ years, (b) $h = 1$ year and (c) $h = 0.5$ years. Compare your answers with the exact mathematical solution.

17.5 Write a function file `euler.m` which starts with the line

```
function [t, n] = euler(a, b, dt)
```

and which uses Euler's method to solve the bacteria growth DE (17.8). Use it in a script to compare the Euler solutions for $dt = 0.5$ and 0.05 with the exact solution. Try to get your output looking like this:

```
time  dt = 0.5  dt = 0.05  exact

   0  1000.00   1000.00    1000.00
0.50  1400.00   1480.24    1491.82
1.00  1960.00   2191.12    2225.54
...
5.00  28925.47  50504.95   54598.15
```

17.6 The basic equation for modeling radioactive decay is

$$dx/dt = -rx,$$

where x is the amount of the radioactive substance at time t, and r is the decay rate.

Some radioactive substances decay into other radioactive substances, which in turn also decay. For example, Strontium 92 ($r_1 = 0.256$ per hr) decays into Yttrium 92 ($r_2 = 0.127$ per hr), which in turn decays into Zirconium. Write down a pair of differential equations for Strontium and Yttrium to describe what is happening.

Starting at $t = 0$ with 5×10^{26} atoms of Strontium 92 and none of Yttrium, use the Runge-Kutta method (`ode23`) to solve the equations up to $t = 8$ hours in steps of 1/3 hr. Also use Euler's method for the same problem, and compare your results.

17.7 The springbok (a species of small buck, not rugby players!) population $x(t)$ in the Kruger National Park in South Africa may be modeled by the equation

$$\mathrm{d}x/\mathrm{d}t = (r - bx \sin at)x,$$

where r, b, and a are constants. Write a program which reads values for r, b, and a, and initial values for x and t, and which uses Euler's method to compute the impala population at monthly intervals over a period of two years.

17.8 The luminous efficiency (ratio of the energy in the visible spectrum to the total energy) of a black body radiator may be expressed as a percentage by the formula

$$E = 64.77T^{-4} \int_{4\times10^{-5}}^{7\times10^{-5}} x^{-5}(e^{1.432/Tx} - 1)^{-1}\mathrm{d}x,$$

where T is the absolute temperature in degrees Kelvin, x is the wavelength in cm, and the range of integration is over the visible spectrum.

Write a general function `simp(fn, a, b, h)` to implement Simpson's rule as given in Equation (17.4).

Taking $T = 3500°$K, use `simp` to compute E, firstly with 10 intervals ($n = 5$), and then with 20 intervals ($n = 10$), and compare your results.

(Answers: 14.512725% for $n = 5$; 14.512667% for $n = 10$)

17.9 Van der Pol's equation is a second-order nonlinear differential equation which may be expressed as two first-order equations as follows:

$$\mathrm{d}x_1/\mathrm{d}t = x_2$$
$$\mathrm{d}x_2/\mathrm{d}t = \epsilon(1 - x_1^2)x_2 - b^2x_1.$$

The solution of this system has a stable limit cycle, which means that if you plot the phase trajectory of the solution (the plot of x_1 against x_2) starting at any point in the positive x_1–x_2 plane, it always moves continuously into the same

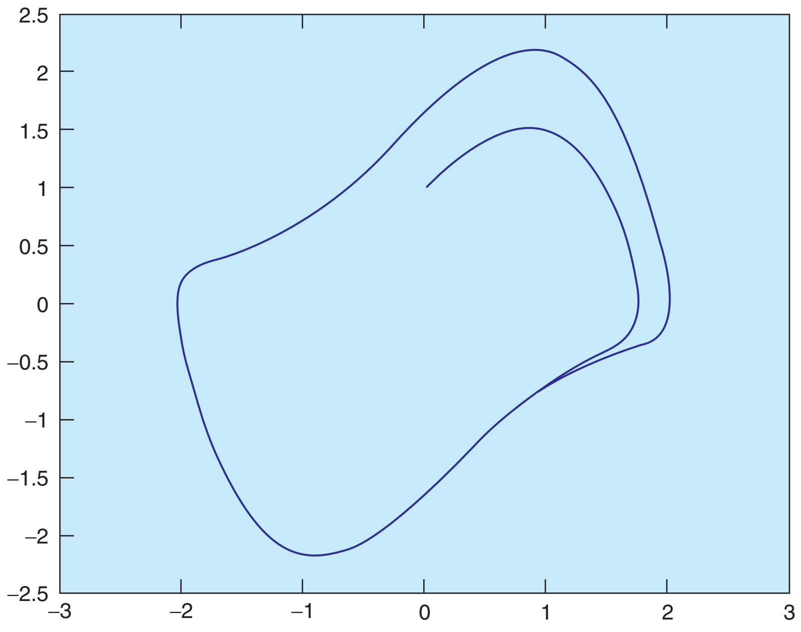

Figure 17.9 A trajectory of Van der Pol's equation

closed loop. Use `ode23` to solve this system numerically, for $x_1(0) = 0$, and $x_2(0) = 1$. Draw some phase trajectories for $b = 1$ and ϵ ranging between 0.01 and 1.0. Figure 17.9 shows you what to expect.