CHAPTER 10

Vectors as Arrays and Other Data Structures

THE OBJECTIVES OF THIS CHAPTER ARE:

- To enable you to solve problems which involve working carefully with array subscripts
- To introduce you to:
- Structures
- Cells and cell arrays

CONTENTS

In MATLAB an array is just another name for a vector. So why have a large part of a chapter on arrays, when we have already been using vectors for most of the book? It is helpful to talk about arrays (as opposed to vectors) when we are concerned with handling individual elements by means of their subscripts, rather than with the vector as a whole. Thus, in the first three sections of this chapter we look at a number of problems which are best solved by treating vectors as arrays, usually with the help of `for` loops.

In the last three sections of this chapter we deal with other, more advanced, data structures.

10.1 UPDATE PROCESSES

In Chapter 8 (**Loops**) we considered the problem of calculating the temperature of orange juice (OJ) in a can as it cools in a fridge. This is an example of an *update process*, where the main variable is repeatedly updated over a period of time. We now examine how to solve this problem more generally.

The OJ is initially at temperature 25 °C when it is placed in the fridge, where the ambient temperature F is 10°. The standard way to solve such an update process is to break the time period up into a number of small steps, each of

http://dx.doi.org/10.1016/B978-0-12-394398-9.00010-1

length dt. If T_i is the temperature at the beginning of step i, we can get T_{i+1} from T_i as follows:

$$T_{i+1} = T_i - K\,dt(T_i - F), \tag{10.1}$$

where K is a physical constant, and units are chosen so that time is in minutes.

10.1.1 Unit time steps

We first solve the problem using a unit time step, i.e., $dt = 1$. The simplest way is to use scalars for the time and the temperature, as we saw in Chapter 8 (although we didn't use a unit time step there):

```
K = 0.05;
F = 10;
T = 25;                        % initial temperature of OJ

for time = 1:100               % time in minutes
  T = T - K * (T - F);         % dt = 1
  if rem(time, 5) == 0
    disp( [time T] )
  end
end;
```

Note the use of `rem` to display the results every 5 min: When `time` is an integer multiple of 5 its remainder when divided by 5 will be zero.

While this is undoubtedly the easiest way of writing the script, we cannot easily plot the graph of temperature against time. In order to do that, `time` and `T` must be vectors. The index of the `for` loop must be used as the subscript of each element of `T`. Here's the script (`update1.m`):

```
K = 0.05;
F = 10;
time = 0:100;                 % initialize vector time
T = zeros(1,101);             % pre-allocate vector T
T(1) = 25;                    % initial temperature of OJ

for i = 1:100                             % time in minutes
  T(i+1) = T(i) - K * (T(i) - F);         % construct T
end;

disp([ time(1:5:101)' T(1:5:101)' ]);   % display results
plot(time, T), grid                     % every 5 mins
```

See Figure 10.1 for typical graphs.

Note:

- The statement `time=0:100` sets up a (row) vector for time where `time(1)` has the value 0 min, and `time(101)` has the value 100 min. This is necessary because the first subscript of a MATLAB vector must be 1.
- The statement `T=zeros(1,101)` sets up a corresponding (row) vector for the temperature, with every element initialized to zero (again there must be 101 elements, because the first element is for the temperature at time zero). This process is called *pre-allocation*. It serves two important purposes.

  a. Firstly, it clears a vector of the same name left over from a previous run. This could cause a conflict when attempting to display or plot `T` against `time`, if the vectors have different sizes. To see this, run `update1` as it stands. It should work perfectly. Now suppose you decide to do the calculations over a shorter time period, say 50 min. Remove the `zeros` statement, and make the following additional two changes and re-run the script (but *don't* clear the workspace):

```
time = 0:50;              % initialize vector time
...

for i = 1:50              % time in minutes
```

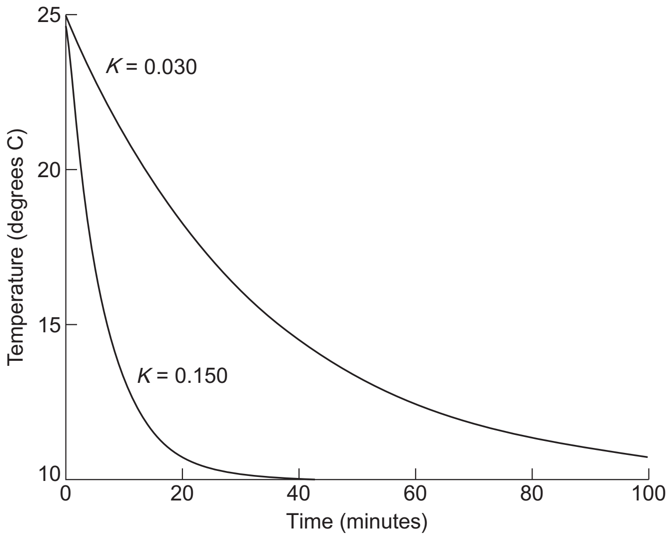

FIGURE 10.1 Cooling curves.

This time you get an error message:

```
??? Error using == > plot
Vectors must be the same lengths.
```

`whos` will reveal that `time` is correctly 51-by-1, but `T` is still 101-by-1. The `plot` command naturally requires these vectors to have the same lengths.
The problem is that, while the operation `0:50` redefines `time` correctly, the `for` loop does not have the same effect on `T`. Since `T` is being updated element by element, the unused elements from 51 to 101 from the previous run are left untouched in the workspace. Pre-allocation of the correct number of elements to `T` with `zeros` avoids this problem.

b. Secondly, although the script will work without the `zeros` statement, as we have seen, it will be much *slower*, since then `T` has to be *redimensioned* during each repeat of the `for` loop, in order to make space for a new element each time.
It's extremely instructive to do an experiment using a vector size of 10,000 elements, say, i.e.,

```
time = 0:9999;            % initialize vector time
T = zeros(1,10000);       % pre-allocate vector T
...
for i = 1:10000
...
```

(and comment out the `disp` and `plot` statements, since these will obscure the issue). My Pentium II takes 0.99 s to run the script with pre-allocation of `T`, but 13.19 s to run without pre-allocation—more than 10 times longer. This could be a critical consideration in a script which does a lot of such element-by-element processing.

- The first element of `T` is set to the initial temperature of the OJ. This is the temperature at time zero.
- The `for` loop computes the values for `T(2)`, ..., `T(101)`.
  This arrangement ensures that temperature `T(i)` corresponds to `time(i)`.
- The colon operator is used to display the results at 5-min intervals.

10.1.2 Non-unit time steps

It's not always appropriate and/or accurate enough to take $dt = 1$ in Equation (10.1). There is a standard way of generating the solution vector in MATLAB, given (almost) any value of dt. We introduce a more general notation to do this.

Call the initial time a, and the final time b. If we want time steps of length dt, the number m of such steps will be:

$$m = (b - a)/dt.$$

The time at the *end* of step i will therefore be $a + i\ dt$.

The following script, update2.m, implements this scheme. It has some additional features. It prompts you for a value of dt, and checks that this value gives an integer number of steps m. It also asks you for the output interval opint (the intervals in minutes at which the results are displayed in table form) and checks that this interval is an integer multiple of dt. Try the script with same sample values, e.g., $dt = 0.4$ and opint $= 4$.

```
K = 0.05;
F = 10;
a = 0;                     % initial time
b = 100;                   % final time
load train
dt = input( 'dt: ' );
opint = input( 'output interval (minutes): ' );
if opint/dt ~= fix(opint/dt)
  sound(y, Fs)
  disp( 'output interval is not a multiple of dt!' )
  break
end;

m = (b - a) / dt;          % m steps of length dt
if fix(m) ~= m             % make sure m is integer
  sound(y, Fs)
  disp( 'm is not an integer - try again!' );
  break

end;

T = zeros(1,m+1);          % pre-allocate (m+1) elements
time = a:dt:b;
T(1) = 25;                 % initial temperature

for i = 1:m
  T(i+1) = T(i) - K * dt * (T(i) - F);
end;

disp( [time(1:opint/dt:m+1)' T(1:opint/dt:m+1)'] )
plot(time, T),grid
```

Note:

- The vectors T and time must each have $m+1$ elements, because there are m time steps, and we need an extra element for the initial value of each vector.
- The expression opint/dt gives the index increment for displaying the results, e.g., $dt=0.1$ and opint $= 0.5$ displays every $(0.5/0.1)$th element, i.e., every 5th element.

10.1.3 Using a function

The really cool way of solving this problem is to write a function to do it. This makes it a lot easier to generate a table of results for different values of *dt*, say, using the function from the command line. Here is update2.m rewritten as a function cooler.m:

```
function [time, T, m] = cooler( a, b, K, F, dt, T0 )

m = (b - a) / dt;      % m steps of length dt
if fix(m) ~= m         % make sure m is integer
  disp( 'm is not an integer - try again!' );
  break
end;

T = zeros(1,m+1);      % pre-allocate
time = a:dt:b;
T(1) = T0;             % initial temperature

for i = 1:m
  T(i+1) = T(i) - K * dt * (T(i) - F);
end;
```

Suppose you want to display a table of temperatures against time at 5 min intervals, using $dt=1$ and $dt=0.1$. Here is how to do it (in the Command Window):

```
   dt = 1;
[t T m] = cooler(0, 100, 0.05, 10, dt, 25);
```

```
    table(:,1) = t(1:5/dt:m+1)';
    table(:,2) = T(1:5/dt:m+1)';
    dt = 0.1;
    [t T m] = cooler(0, 100, 0.05, 10, dt, 25);
    table(:,3) = T(1:5/dt:m+1)';
    format bank
    disp(table)
```

Output:

```
              0          25.00          25.00
           5.00          21.61          21.67
          10.00          18.98          19.09
            ...
         100.00          10.09          10.10
```

Note:

- The advantage of using a function which generates a vector output variable is that even if you forget to pre-allocate the vector inside the function (with `zeros`) MATLAB automatically clears any previous versions of the output vector before returning from the function.
- The variable `table` is a two-dimensional array (or matrix). Recall that the colon operator may be used to indicate all elements of a matrix in a row or column. So `table(:,1)` means the elements in every row and column 1, i.e., the entire first column. The vectors `t` and `T` are row vectors, so they must be transposed before being inserted into columns 1 and 2 of `table`. The third column of `table` is inserted in a similar way.
- The results in the third column (for $dt = 0.1$) will be more accurate.

10.1.4 Exact solution

This cooling problem has an exact mathematical solution. The temperature $T(t)$ at time t is given by the formula:

$$T(t) = F + (T_0 - F)e^{-Kt}, \tag{10.2}$$

where T_0 is the initial temperature. You can insert values for this exact solution into a fourth column of `table`, by vectorizing the formula, as follows:

```
    tab(:,4) = 10 + (T(1)-10)*exp(-0.05 * t(1:5/dt:m+1)');
```

The enlarged table should look something like this:

```
          0          25.00          25.00          25.00
       5.00          21.61          21.67          21.68
      10.00          18.98          19.09          19.10
        ...
```

Note that the numerical solution generated by Equation (10.1) gets more accurate as dt gets smaller. That is because Equation (10.2) (the exact solution) is derived from Equation (10.1) *in the limit* as $dt \to 0$.

Exercise

To complete the analysis of the cooling problem, you may like to superimpose graphs for different values of K and draw them in different colours. It would also be useful to label each graph at some convenient place, e.g., with the label `'K=0.08'`. The command:

```
gtext( 'text' )
```

causes an arrow to appear on the graphics screen. You can position the arrow with a mouse or the arrow keys. When you click (or press **Enter**), the specified text appears at that point.

The function `gtext` takes a string argument. If you want to label it with a variable numeric value, use `sprintf` to convert a number into a string. It works just like `fprintf`, except that the output goes into a string variable, e.g.,

```
gtext( sprintf('K = %5.3f', K) )
```

- Draw some graphs, on the same set of axes, for different values of K, in different colours, and label them. Figure 10.1 shows what to aim for. Also superimpose the exact solution, given by Equation (10.2).

Plots can also be labelled interactively in plot editing mode or with the Properties Editor.

10.2 FREQUENCIES, BAR CHARTS, AND HISTOGRAMS

10.2.1 A random walk

Imagine an ant walking along a straight line, e.g., the x-axis. She starts at $x=40$. She moves in steps of one unit along the line. Each step is to the left or the right with equal probability. We would like a visual representation of how much time she spends at each position.

Start by running the following script, `ant.m`:

```
f = zeros(1,100);
x = 40;

for i = 1:1000
  r = rand;
  if r >= 0.5
    x = x + 1;
  else
    x = x - 1;
  end
  if x ~= 0 | x ~= 100
```

Now enter the statement `bar(f)` in the Command Window. You should get a graph which is similar to the one in Figure 10.2.

Note:

- The function `rand` returns a random number in the range 0–1. If it's greater than 0.5, the ant moves right (`x=x+1`), otherwise she moves left (`x=x-1`).
- The vector `f` has 100 elements, initially all zero. We define `f(x)` as the number of times the ant lands at position `x`. Suppose her first step is to the right, so `x` has the value 41. The statement:

  `f(x) = f(x) + 1`

  then increases the value of `f(41)` to 1, meaning that she has been there once. When she next wanders past this value of `x`, `f(41)` will be increased to 2, meaning she's been there twice.
  When I ran this script, the final value of `f(41)` was 33—the number of times the ant was there.
- `f(x)` is called a *frequency distribution*, and the graph obtained from `bar(f)` is called a *bar graph*. Each element of `f` is represented by a vertical bar of proportional height. See also the `help bar`.

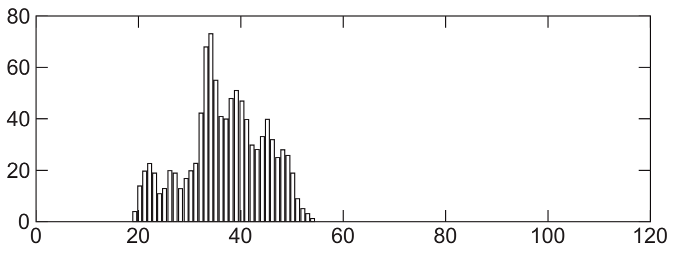

FIGURE 10.2 Random walk frequencies.

- The script `ant.m` *simulates* the random movement of the ant. If you re-run it, you will get a different bar graph, because `rand` will generate a different sequence of random numbers. Simulation is discussed more fully in Chapter 13.

10.2.2 Histograms

Another helpful way of representing data is with a *histogram*.

As an example, suppose 12 students write a test and obtain the following marks (expressed as percentages), which are assigned to the vector `m`:

```
0   25   29   35   50   55   55   59   72   75   95   100
```

The statement `hist(m)` will draw a histogram. Try it. The histogram shows the distribution of marks in 10 "bins" (categories) which are equally spaced between the minimum (0) and maximum (100) marks in the set of data. The number of bins (10 by default) can be specified by a second argument, e.g., `hist(m, 25)`.

To generate the frequences plotted by `hist` use the following form (which does not actually draw the histogram):

```
[n x] = hist(m)
```

`n` is a vector containing the frequences, viz.,

```
1    0    2    1    0    4    0    2    0    1    1
```

For example, there is one mark in the first bin (0–9), none in the second (10–19), two in the third, and so on.

The second output vector `x` contains the midpoints of the bins, such that `bar(x, n)` plots the histogram. See Help for more details.

Note the subtle difference between a histogram and a bar graph. The values plotted by `hist` are computed from the distribution of values in a vector, whereas `bar` generates a bar graph directly from the values themselves.

10.3 SORTING

One of the standard applications of arrays is sorting a list of numbers into, say, ascending order. Although MATLAB has its own sorting function (`sort`), you may be interested in how sorting algorithms actually work.

10.3.1 Bubble sort

The basic idea is that the unsorted list is assigned to a vector. The numbers are then ordered by a process which essentially passes through the vector many times, swopping consecutive elements that are in the wrong order, until all the elements are in the right order. Such a process is called a Bubble Sort, because the smaller numbers rise to the top of the list, like bubbles of air in water. (In fact, in the version shown below, the largest number will "sink" to the bottom of the list after the first pass, which really makes it a "Lead Ball" sort.) There are many other methods of sorting, such as the Quick Sort, which may be found in most textbooks on computer science. These are generally more efficient than the Bubble Sort, but its advantage is that it is by far the easiest method to program. A structure plan for the Bubble Sort is as follows:

```
1. Input the list $X$
2. Set $N$ to the length of $X$
3. Repeat $N - 1$ times with counter $K$:
        Repeat $N - K$ times with counter $J$:
            If $X_j > X_{j+1}$ then
                Swop the contents of $X_j$ and $X_{j+1}$
4. Stop since the list $X$ is now sorted.
```

As an example, consider a list of five numbers: 27, 13, 9, 5, and 3. They are initially input into the vector X. Part of MATLAB's memory for this problem is sketched in Table 10.1. Each column shows the list during each *pass*. A stroke in a row indicates a change in that variable during the pass as the script works down the list. The number of tests ($X_j > X_{j+1}$?) made on each pass is also shown in the table. Work through the table by hand with the structure plan until you understand how the algorithm works.

Sorting algorithms are compared by calculating the number of tests (comparisons) they carry out, since this takes up most of the execution time during the sort. On the Kth pass of the Bubble Sort there are exactly $N - K$ tests, so the total number of tests is:

$$1 + 2 + 3 + \cdots + (N - 1) = N(N - 1)/2$$

(approximately $N^2/2$ for large N). For a list of five numbers there are therefore 10 tests, but for 10 numbers there are 45 tests. The computer time needed goes up as the square of the length of the list.

Table 10.1 Memory During a Bubble Sort

	1st pass	2nd pass	3rd pass	4th pass
X_1:	27/13	13/9	9/5	5/3
X_2:	13/27/9	9/13/5	5/9/3	3/5
X_3:	9/27/5	5/13/3	3/9	9
X_4:	5/27/3	3/13	13	13
X_5:	3/27	27	27	27
	4 tests	3 tests	2 tests	1 test

The function M-file `bubble.m` below departs slightly from the structure plan above, which will make $N - 1$ passes, *even if the list is sorted before the last pass*. Since most real lists are partially sorted, it makes sense to check after each pass if any swops were made. If none were, the list must be sorted, so unnecessary (and therefore time-wasting) tests can be eliminated. In the function, the variable `sorted` is used to detect when the list is sorted, and the outer loop is coded instead as a non-deterministic `while` loop. Here it is:

```
function y = bubble( x )
n = length(x);
sorted = 0;                % flag to detect when sorted
k = 0;                     % count the passes

while ~sorted
  sorted = 1;              % they could be sorted
  k = k + 1;               % another pass
  for j = 1:n-k            % fewer tests on each pass
    if x(j) > x(j+1)       % are they in order?
      temp = x(j);         % no ...
      x(j) = x(j+1);
      x(j+1) = temp;
      sorted = 0;          % a swop was made
    end
    end
  end;

  y = x;
```

You can test it on the command line to sort say 20 random numbers as follows:

```
  r = rand(1,20);
r = bubble( r );
```

Note how `bubble` changes its input vector.

On my PC `bubble` takes 1.81 s to sort 200 random numbers, but 7.31 s to sort 400 numbers. This is consistent with the theoretical result obtained above.

10.3.2 MATLAB's `sort`

The built-in MATLAB function `sort` returns two output arguments: The sorted list (in ascending order) and a vector containing the *indexes* used in the sort, i.e., the positions of the sorted numbers in the original list. If the random numbers:

```
r = 0.4175   0.6868   0.5890   0.9304   0.8462
```

are sorted with the command:

```
[y,i] = sort(r)
```

the output variables are:

```
y =  0.4175    0.5890    0.6868    0.8462    0.9304
i =  1      3      2      5      4
```

For example, the index of the second largest number (0.5890) is 3, which is its subscript in the original unsorted list `r`.

As a matter of fact, the built-in functions `max` and `min` also return second output variables giving indexes.

MATLAB's `sort` is *very* fast. My PC takes 2.36 s to sort a list of one million random numbers! This is because (a) a Quick Sort is used, and (b) the script has been compiled as a built-in function; this makes for faster code.

10.4 STRUCTURES

Up to now we have seen arrays with only one type of element—all numeric, or all character. A MATLAB *structure* allows you to put different kinds of data in its various *fields*. For example, we can create a structure called `student` with one field for a student's name,

```
student.name = 'Thandi Mangwane';
```

a second field for her student ID number,

```
student.id = 'MNGTHA003';
```

and a third field for all her marks to date,

```
student.marks = [36 49 74];
```

To see whole structure, enter its name:

```
student

student =
     name: 'Thandi Mangwane'
       id: 'MNGTHA003'
    marks: [36 49 74]
```

Here's how to access her second mark:

```
student.marks(2)
ans =
    49
```

Note the use of the dot to separate the structure name from its fields when creating it and when accessing its fields.

To add further elements to the structure, use subscripts after the structure name:

```
student(2).name = 'Charles Wilson'
student(2).id = 'WLSCHA007'
student(2).marks = [49 98]
```

(the original student, Thandi Mangwane, is now accessed as `student(1)`). Note that field sizes do not have to conform across elements of a structure array: `student(1).marks` has three elements, while `student(2).marks` has only two elements.

The `student` structure now has size 1-by-2: It has two elements, each element being a student with three fields. Once a structure has more than one element MATLAB does not display the contents of each field when you type the structure name at the command line. Instead, it gives the following summary:

```
student

student =
1x2 struct array with fields:
    name
    id
    marks
```

You can also use `fieldnames(student)` to get this information.

A structure array can be preallocated with the `struct` function. See Help.

A structure field can contain any kind of data, even another structure—why ever not? Thus we can create a structure `course` of courses taken by students, where one field is the name of the course, and another field is a `student` structure with information on all the students taking that particular course:

```
course.name = 'MTH101';
course.class = student;
course

course =
     name: 'MTH101'
    class: [1x2 struct]
```

We can set up a second element of `course` for a different class of students:

```
course(2).name = 'PHY102';
course(2).class = ...
```

To see all the courses:

```
course(1:2).name
ans =
MTH101
ans =
PHY102
```

To see all the students in a particular course:

```
course(1).class(1:2).name
ans =
Thandi Mangwane
ans =
Charles Wilson
```

There is a curious function called `deal` which "deals inputs to outputs." You can use it to generate "comma-separated variable lists" from structure fields:

```
[name1, name2] = deal(course(1).class(1:2).name);
```

(but you don't actually need the commas here …).

You can use the `rmfield` function to remove fields from a structure.

10.5 CELL ARRAYS

A *cell* is the most general data object in MATLAB. A cell may be thought of as a "data container," which can contain any type of data: numeric arrays, strings, structures, or cells. An array of cells (and they almost always occur in arrays) is called a *cell array*. While you might think a cell sounds the same as a structure, a cell is more general, and there are also notational differences (which are confusing).

10.5.1 Assigning data to cell arrays

There are a number of ways of assigning data to cell arrays.

- Cell indexing:

```
c(1,1) = {rand(3)};
c(1,2) = {char('Bongani', 'Thandeka')};
c(2,1) = {13};
c(2,2) = {student};

c =
    [3x3 double]    [2x8 char  ]
    [         13]    [1x2 struct]
```

(assuming that the structure `student` created above still exists). Here the round brackets on the lefthand side of the assignments refer in the normal way to the elements of the cell array. What is different are the curly braces on the right. Curly braces indicate the *contents* of a cell; on the righthand side of an assignment they are technically cell array *constructors* (remember that each element of this array is a cell).

So read the first statement as:

"Construct a cell containing `rand(3)` and assign the cell to element `1,1` of the cell array `c`."

- Content indexing:

```
c{1,1} = rand(3);
c{1,2} = char('Bongani', 'Thandeka');
c{2,1} = 13;
c{2,2} = student;
```

  Here the curly braces on the left indicate the *contents* of the cell element at that particular location. So read the first statement as:
  "The contents of the cell at location `1,1` becomes `rand(3)`."

- You can use the curly braces to construct an entire cell array in one statement:

```
b = {[1:5], rand(2); student, char('Jason', 'Amy')}
b =
     [1x5 double]     [2x2 double]
     [1x2 struct]     [2x5 char  ]
```

  A cell may contain another cell array; nested curly braces may be used to create nested cell arrays.

- The `cell` function enables you to preallocate empty cell arrays, e.g.,

```
a = cell(3,2)     % empty 3-by-2 cell array
a =
      []      []
      []      []
      []      []
```

You can then use assignment statements to fill the cells, e.g.,

`a(2,2) = {magic(3)}`

Note: if you already have a numeric array with a certain name, don't try to create a cell array of the same name by assignment without first clearing the numeric array. If you don't clear the numeric array, MATLAB will generate an error (it will think you are trying to mix cell and numeric syntaxes).

10.5.2 Accessing data in cell arrays

You can access cell contents using content indexing (curly braces):

```
r = c{1,1}
r =
    0.4447    0.9218    0.4057
    0.6154    0.7382    0.9355
    0.7919    0.1763    0.9169
```

To access a subset of a cell's contents you can concatenate curly braces and round brackets if necessary:

```
rnum = c{1,1}(2,3)
rnum =
    0.9355
```

Here the curly braces (content indexing) indicate the contents of the cell array element `c(1,1)`, which is a 3-by-3 numeric matrix. The subscripts `(2,3)` then indicate the appropriate element within the matrix.

Curly braces may be concatenated to access nested cell arrays.

10.5.3 Using cell arrays

Cell arrays come into their own when you need to access (different types of) data as "comma-separated variable lists," as the next example demonstrates.

The functions `varargin` and `varargout`, which allow a function to have any number of input or output arguments, are none other than cell arrays. The function `testvar` has a variable number of input arguments, which are doubled into the variable number of output arguments (assuming that the number of output arguments does not exceed the number of input arguments):

```
function [varargout] = testvar(varargin)

for i = 1:length(varargin)
    x(i) = varargin{i};     % unpack the input args
end

for i = 1:nargout           % how many output arguments?
    varargout{i} = 2*x(i);  % pack up the output args
end
```

Command line:

```
[a b c] = testvar(1, 2, 3, 4)
a =
     2
b =
     4
c =
     6
```

When a function is called with the input argument `varargin` MATLAB automatically packs the list of corresponding input arguments into a cell array. You then simply unpack the cell array inside the function using the curly brace content indexing. You similarly pack up the output arguments into the cell array `varargout`.

Note that, should your function have some compulsory input and output arguments, `varargin` and `varargout` must appear at the end of their respective argument lists.

MATLAB has a discussion on when to use cell arrays in the section **Organizing Data in Cell Arrays**, to be found in **MATLAB Help: Programming and Data Types: Structures and Cell Arrays**.

10.5.4 Displaying and visualizing cell arrays

The function `celldisp` recursively displays the contents of a cell array.

The function `cellplot` draws a visualization of a cell array. Figure 10.3 depicts the contents of the cell array `c`, created above. Non-empty array elements are shaded.

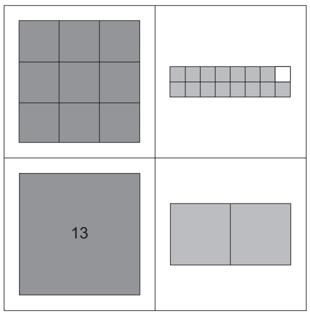

FIGURE 10.3 Result of `cellplot(c)`.

10.6 CLASSES AND OBJECTS

You'd have to be living face down in a moon crater not to have heard about object-oriented programming. *Tom Swan.*
Object-oriented programming is an exceptionally bad idea which could only have originated in California. *Edsger Dijkstra.*
MATLAB, along with most other modern programming languages, espouses the cause of object-oriented programming, and has all the usual paraphernalia associated with this paradigm: classes, objects, encapsulation, inheritance, operator overloading, and so on.

Object-oriented programming is a subject which requires an entire book to do justice to it. If you want to learn about the concepts, there are many excellent books on the subject. If you want to see how MATLAB implements object-oriented programming, consult the online Help: **MATLAB Help: Programming and Data Types: MATLAB Classes and Objects**.

SUMMARY

- A MATLAB structure allows you to store different types of data in its various fields.
- Arrays of structures may be created.
- A cell is the most general data object in MATLAB, and can store any type of data.
- An array of cells is called a cell array—each cell in the array can store different types of data, including other cell arrays.
- A cell array is constructed with curly braces {}.
- The contents of cell in a cell array is accessed with content indexing (curly braces).
- Cell elements in a cell array are accessed in the usual way with round brackets (cell indexing).
- Arguments of functions using a variable number of arguments are packed into cell arrays.
- The function `cellplot` gives a visualization of a cell array.
- MATLAB implements object-oriented programming.