Chapter 9

The Second Dimension

§9.1 Rows and Columns
2-dimensional arrays—matrix, functions that involve matrices, colon notation for submatrices, built-in function `size`

§9.2 Operations
Searching a 2-dimensional array and updating its values, built-in functions `rand` and `sprintf`, subfunctions

§9.3 Tables in Two Dimensions
Using 2-dimensional arrays to represent a function of two variables.

As we have said before, the ability to think at the array level is very important in computational science. This is challenging enough when the arrays involved are linear, i.e., one-dimensional. Now we consider the two-dimensional array using this chapter to set the stage for more involved applications that use this structure. Two-dimensional array thinking is essential in application areas that involve image processing. (A digitized picture is a 2-dimensional array.) Moreover, many 3-dimensional problems are solved by solving a sequence of 2-dimensional, "cross-section" problems.

We start by considering some array set-up computations in §9.1. The idea is to develop an intuition about the parts of a 2-dimensional array: its rows, its columns, and its subarrays.

Once an array is set up, it can be searched and its entries manipulated. Things are not too different from the 1-dimensional array setting, but we get additional row/column practice in §9.2 by considering a look-for-the-max problem and also a mean/standard deviation calculation typical in data analysis. Computations that involve both 1- and 2-dimensional arrays at the same time are explored through a cost/purchase order/inventory application. Using a 2-dimensional array to store a finite snapshot of a 2-dimensional continuous function $f(x, y)$ is examined in §9.3.

9.1 Rows and Columns

If $f(x)$ is a function of a single variable, then a vector can be used to represent a table of its values. Until now, we have only dealt with *1-dimensional* arrays. A single subscript is sufficient to specify the location of a value in a 1-dimensional array.

Many applications involve a function of *two* variables and a *2-dimensional array* is often handy for the representation of its values. In MATLAB, a 2-dimensional array is called a *matrix*. Our experience with 2-dimensional tables begins in grade school with the times table:

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

Times table construction is shown in `Example9_1` which illustrates the creation of a matrix and the printing of a *sub*matrix. The integers `r1` and `r2` define the range of the involved rows while `c1` and `c2` specify the column range. Extending our colon notation in the obvious way, we see that `Example9_1` displays the 4-by-6 subarray `T(6:9,5:10)`.

There are many things to discuss. We create a matrix `T` where each individual component has the value one with the statements

```
rowMax= 12;
colMax= 10;
T= ones(rowMax,colMax);
```

The result is a 12-row, 10-column array called `T` that looks like this:

T =

1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

A double subscript notation is used to indicate any location in the array. A fragment of the form

```
T(1,1)= 1*1;
T(1,2)= 1*2;
      ⟨:⟩
T(1,10)= 1*10;
```

sets up the first row of `T`:

```
% Example 9_1:  Times table

rowMax= 12;   % No.  of rows in table
colMax= 10;   % No.  of columns in table

T= ones(rowMax,colMax);   % Initialize times table to 1s

% Compute the times table
for r= 1:rowMax
   % Calculate and fill the r-th row
   for c= 1:colMax
      T(r,c)= r*c;
   end
end

% Print user-specified submatrix
anotherEg= 'y';
while (  anotherEg ~= 'n' )
   c1= input(sprintf('Enter c1, lower bound in x range.  1<=c1<=%d.  ',colMax));
   c2= input(sprintf('Enter c2, upper bound in x range.  1<=c2<=%d.  ',colMax));
   r1= input(sprintf('Enter r1, lower bound in y range.  1<=r1<=%d.  ',rowMax));
   r2= input(sprintf('Enter r2, upper bound in y range.  1<=r2<=%d.  ',rowMax));
   % Print T(r1:r2,c1:c2)
   for r= r1:r2
      % Print T(r,c1:c2)
      for c= c1:c2
         fprintf('%4d', T(r,c))
      end
      fprintf('\n')   % Start new line
   end
   anotherEg= input('Another example?  Enter y (yes) or n (no):  ', 's');
end
```

Sample output:

```
Enter c1, lower bound in x range.  1<=c1<=10.  5
Enter c2, upper bound in x range.  1<=c2<=10.  10
Enter r1, lower bound in y range.  1<=r1<=12.  6
Enter r2, upper bound in y range.  1<=r2<=12.  9
  30  36  42  48  54  60
  35  42  49  56  63  70
  40  48  56  64  72  80
  45  54  63  72  81  90
Another example?  Enter y (yes) or n (no).
n
```

T =

1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

In general, a reference to an entry in a 2-dimensional array has the form

⟨*Name*⟩(⟨*row index*⟩,⟨*column index*⟩)

Successful subscript-level "thinking" in 2-dimensions requires being able to keep track of both the row and column indices. *Remember that the row index comes first.* In `Example9_1`, the array `T` is filled row-by-row. The inner loop in the top portion of the program establishes the r-th row of the table. If `r=2`, then

```
for c= 1:colmax
   T(r,c)= r*c;
end
```

is equivalent to

```
T(2,1)= 2*1;
T(2,2)= 2*2;
      ⟨:⟩
T(2,10)= 2*10;
```

and the array `T` now looks like

T =

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

Double loops abound in matrix settings and the mastery of the subscript concept is essential. Here is what `T` looks like upon completion of the double loop:

T =

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100
11	22	33	44	55	66	77	88	99	110
12	24	36	48	60	72	84	96	108	120

If we reverse the order of the `r` and `c` loop, then `T` is filled column by column. Thus, after two passes through the outer loop

```
for c= 1:colmax
   {Calculate and fill the c-th column}
   for r= 1:rowmax
      T(r,c)= r*c;
   end
end
```

we have

T =

1	2	1	1	1	1	1	1	1	1
2	4	1	1	1	1	1	1	1	1
3	6	1	1	1	1	1	1	1	1
4	8	1	1	1	1	1	1	1	1
5	10	1	1	1	1	1	1	1	1
6	12	1	1	1	1	1	1	1	1
7	14	1	1	1	1	1	1	1	1
8	16	1	1	1	1	1	1	1	1
9	18	1	1	1	1	1	1	1	1
10	20	1	1	1	1	1	1	1	1
11	22	1	1	1	1	1	1	1	1
12	24	1	1	1	1	1	1	1	1

We also mention that it is legal to create the matrix "on-the-fly," or build it one row at a time, without first initializing the matrix to a certain size using function `ones`. Suppose we now wish to create a 3-by-4 times table. Consider the fragment

```
for r= 1:3
   % Calculate the r-th row
   for c= 1:4
      T(r,c)= r*c;
   end
end
```

Note that we have *not* used any function to first create a 3-by-4 matrix. In the first pass through the outer loop where r=1, we create a 1-by-4 matrix

T =

1	2	3	4

In the second pass through the outer loop, r=2 so we create the submatrix `T(2,1:4)`, *without* affecting the submatrix `T(1,1:4)` already built during the first pass of the outer loop. The result is a 2-by-4 matrix

T =

1	2	3	4
2	4	6	8

In the final pass through the outer loop, r=3 so we create the submatrix `T(3,1:4)`, resulting in the final 3-by-4 matrix

T =

1	2	3	4
2	4	6	8
3	6	9	12

We have used the colon notation to denote *sub*matrices several times. This colon notation can be used in MATLAB as well to specify a submatrix. Suppose `T` is the 3-by-4 matrix we have created above. The statement

```
subT1= T(2:3,2:4)
```

gives

subT1 =

4	6	8
6	9	12

Furthermore, MATLAB has a shortcut expression for specifying "all rows" or "all columns." Say we want to retrieve the first two columns of `T` and call the new matrix `subT2`. Then we can write `subT2= T(1:3,1:2)` or use the shortcut

```
subT2= T(:,2:4)
```

Instead of specifying "rows 1 through 3," we simply say "all rows" by using the colon to *replace* the row range `1:3`.

Let us now define a function `showMatrix` to print out the values in a matrix:

```
function showMatrix(m)
% Post:  Print all values in matrix m.
%        Each value is printed to 1 decimal place.
% Pre:   m is numeric and each value < 100000.  (For larger values, the
%        printed columns won't line up, but the values will be correct.

[nr,nc]= size(m);
for r= 1:nr
   for c= 1:nc
      fprintf('%8.1f', m(r,c))
   end
fprintf('\n')
end
```

Note the use of the built-in function `size`. Given the a matrix as the input argument, function `size` returns two values: the number of rows and the the number columns, in that order, of the matrix. Although function `showMatrix` prints the values in the entire matrix specified by the input parameter, you can use `showMatrix` to print a submatrix simply by specifying a submatrix *in the function call*. `Example9_2` does the same thing as `Example9_1` except that it uses `showMatrix` to print the user-specified part of the times table.

In `Example9_1` and `Example9_2`, we could have created and printed only the specified range of the times table *without* first creating the 12-by-10 times table. In fact, we can create any contiguous portion of the times table "on-the-fly" because the value in each cell can be calculated independent of the values in any other cell. For example, the fragment

```
% print lines 6 to 9 and columns 5 to 10 of times table
for i= 6:9
   for j= 5:10
      subT(i-5,j-4)= i*j;
   end
end
showMatrix(subT);
```

will print only lines 6 to 9 and columns 5 to 10 of the times table, like in `Example9_1` and `Example9_2`. Analyze the subscripts carefully. For each entry of matrix `subT`, the two numbers multiplied *are not* the row and column numbers of `subT`. The first line of the times table of interest is 6, but it corresponds to row 1 of the *sub*matrix, therefore an "offset" of 5 needs to be subtracted from multiplier 6 in order to store the value in a cell in row 1. Similarly, the column subscript has to be adjusted by an offset of 4.

Sometimes the entries in a matrix are defined in terms of other entries. As an example of such a recursive specification, consider the following definition of the n-by-n *Pascal array*, whose

```
% Example 9_2:  Times table

rowMax= 12;   % No.  of rows in table
colMax= 10;   % No.  of columns in table

T= ones(rowMax,colMax);   % Initialize times table to 1s

% Compute the times table
for r= 1:rowMax
   % Calculate and fill the r-th row
   for c= 1:colMax
      T(r,c)= r*c;
   end
end

% Print user-specified submatrix
anotherEg= 'y';
while (  anotherEg ~= 'n' )
   c1= input(sprintf('Enter c1, lower bound in x range.  1<=c1<=%d.  ',colMax));
   c2= input(sprintf('Enter c2, upper bound in x range.  1<=c2<=%d.  ',colMax));
   r1= input(sprintf('Enter r1, lower bound in y range.  1<=r1<=%d.  ',rowMax));
   r2= input(sprintf('Enter r2, upper bound in y range.  1<=r2<=%d.  ',rowMax));
   % Print T(r1:r2,c1:c2)
   showMatrix(T(r1:r2,c1:c2));
   anotherEg= input('Another example?  Enter y (yes) or n (no):  ', 's');
end
```

Sample output:

```
Enter c1, lower bound in x range.  1<=c1<=10.  5
Enter c2, upper bound in x range.  1<=c2<=10.  10
Enter r1, lower bound in y range.  1<=r1<=12.  6
Enter r2, upper bound in y range.  1<=r2<=12.  9
 30.0   36.0   42.0   48.0   54.0   60.0
 35.0   42.0   49.0   56.0   63.0   70.0
 40.0   48.0   56.0   64.0   72.0   80.0
 45.0   54.0   63.0   72.0   81.0   90.0
Another example?  Enter y (yes) or n (no):  n
```

(r,c) entry is defined by

$$p_{rc} = \begin{cases} 1 & \text{if } r=1 \text{ or } c=1 \\ \\ p_{r,c-1}+p_{r-1,c} & \text{otherwise} \end{cases}$$

Thus, the first row and column of the array are made up of ones. Elsewhere, the (r,c) entry is the sum of the "previous" entry in its row, $p_{r,c-1}$, and the previous entry in its column, $p_{r-1,c}$. Encapsulating this we obtain

```
function p = pascalMatrix(n)
% Post:  p is the pascal matrix with n rows and columns
% Pre:  n>1

for r= 1:n
   % Set up the r-th row
   for c= 1:n
      if  ( r==1 || c==1 )
         p(r,c)= 1;
      else
         p(r,c)= p(r,c-1) + p(r-1,c);
      end
   end
end
```

We also mention that instead of having a pair of "1-to-n" loops with an *or* operator, we could set up matrix `p` as follows:

```
for k= 1:n
   p(k,1)= 1;
   p(1,k)= 1;
end
for r= 2:n
   for c= 2:n
      p(r,c)= p(r,c-1) + p(r-1,c);
   end
end
```

Here, row 1 and column 1 are established first, and then `p(2:n,2:n)`. `Example9_3` illustrates the use of `pascalMatrix`.

Problem 9.1. Modify `Example9_1` so that it prints row and column "labels" for the user-specified portion of the times table. For example, if the user-specified range is lines 6 to 8 and columns 5 to 10, then the printed table should look like this:

	5	6	7	8	9	10
6	30	36	42	48	54	60
7	35	42	49	56	63	70
8	40	48	56	64	72	80

```
% Example9_3:  Pascal array

n= 6;   % Size of problem
M= pascalMatrix(n);
fprintf('The Pascal array with n = %d\n', n)
showMatrix(M);
```

Output:

```
The Pascal array with n = 6:
    1.0  1.0   1.0   1.0    1.0    1.0
    1.0  2.0   3.0   4.0    5.0    6.0
    1.0  3.0   6.0  10.0   15.0   21.0
    1.0  4.0  10.0  20.0   35.0   56.0
    1.0  5.0  15.0  35.0   70.0  126.0
    1.0  6.0  21.0  56.0  126.0  252.0
```

PROBLEM 9.2. Complete the following procedure

```
function [S,C] = sinCosMatrix(n, theta)
% Post:  S(i,j) = sin(i*j*theta) and C(i,j) = cos(i*j*theta), 1<=i<=n, 1<=j<=n
% Pre:  integer n>=1
```

Implement the simplest approach which is to call the functions `sin` and `cos` for each entry. The number of function evaluations can be reduced by exploiting the identities

$$\begin{aligned}\sin(ij\theta) &= \sin((i-1)j\theta)\cos(j\theta)+\cos((i-1)j\theta)\sin(j\theta)\\ \cos(ij\theta) &= \cos((i-1)j\theta)\cos(j\theta)-\sin((i-1)j\theta)\sin(j\theta)\end{aligned}$$

In particular, the identities can be used to generate the first row and column of `S` and `C` from `S(1,1)` and `C(1,1)`. Once the first rows and columns are available, the recursions can be used to generate `S(2:n,2:n)` and `C(2:n,2:n)`.

PROBLEM 9.3. The (r, c) entry of the n-by-n Pascal array P is also specified by

$$p_{rc} = \sum_{k=1}^{\min\{r,c\}} \binom{r-1}{k-1}\binom{c-1}{k-1}$$

Write an alternative `pascalMatrix1` to the function `pascalMatrix` that returns the n-by-n Pascal array using this recipe. Make use of the fact that $p_{rc} = p_{cr}$. Do not use MATLAB's built-in function `pascal`!

PROBLEM 9.4. A *magic square* of size n is an n-by-n array with the following properties:

- It is "made up" of the integers $1, 2, \ldots, n^2$.
- The numbers in every row, column, and diagonal sum to $n(n^2+1)/2$. (There are two diagonals, one from the upper left corner to the lower right corner, and one from the lower left corner to the upper right corner.)

Here are the magic squares of size 3, 5, and 7:

2	7	6
9	5	1
4	3	8

9	3	22	16	15
2	21	20	14	8
25	19	13	7	1
18	12	6	5	24
11	10	4	23	17

20	12	4	45	37	29	28
11	3	44	36	35	27	19
2	43	42	34	26	18	10
49	41	33	25	17	9	1
40	32	24	16	8	7	48
31	23	15	14	6	47	39
22	21	13	5	46	38	30

Pick up the pattern and complete the following function:

```
function M= magicSqr(n)
% Post:  M is an n-by-n magic square}
% Pre:  n is odd}
```

Do not use MATLAB's built-in function `magic`! For checking purposes, implement the following boolean-valued function:

```
function result = isMagic(M)
% Post:  {M is a magic square}, true or false
% Pre:  n is odd
```

PROBLEM 9.5. Complete the following function:

```
function vm = vanderMonde(x)
% Post:  vm(i,j) = x(i)^(j-1), 1<=i<=length(x), 1<=j<=length(x)
% Pre:  x is real vector
```

Do not use MATLAB's built-in function `vander`.

PROBLEM 9.6. Complete the following function:

```
function T = toeplitz(d)
% Post:  T(i,j) = d(|i-j|), 1<=i<=length(x), 1<=j<=length(x)
% Pre:  d is real vector
```

Do not use MATLAB's built-in function `toeplitz`.

9.2 Operations

Searching for a maximum or minimum value in a matrix is very similar to searching a vector. The only difference is that a nested loop is required, one for stepping through the columns and one for the rows. Here's a function that computes the row and column index of the maximum entry:

```
function [rmax, cmax]= indexOfMax(M)
% Post:  M(rmax,cmax) is the largest entry in M. If maximum value
% occurs in multiple places, identify the first one.
% Pre:  M is at least 1-by-1
```

```
rmax= 1;
cmax= 1;
maxSoFar= M(1,1);
for r= 1:size(M,1)
   % Scan the r-th row
   for c= 1:size(M,2)
      if  M(r,c)>maxSoFar
         % A new max has been found
         maxSoFar= M(r,c);
         rmax= r;
         cmax= c;
      end
   end
end
```

The search through the array is row-by-row. Within each row, the array entries are checked left to right. Any time a new largest value is encountered, its row and column indices are stored in `rmax` and `cmax` respectively. `Example9_4` illustrates the use of the procedure. Notice the use of

```
% Example 9_4:  Maximum entry in matrix

m= 4; n= 6;   % m-by-n problem
lBound= -5; uBound= 5;   % lower and upper bounds for array entries

anotherEg= 'y';
while (  anotherEg ='n' )
   M= rand(m,n)*(uBound-lBound)+lBound;
   showMatrix(M);
   [r,c]= indexOfMax(M);
   fprintf('Maximum value %.1f at (%d,%d)\n', M(r,c), r, c)
   anotherEg= input('Another example?  Enter y (yes) or n (no):  ', 's');
end
```

Output:

```
  -3.6  -2.3  -0.5   3.5   3.4   3.3
  -3.0  -3.0   4.3   0.3  -4.8   0.0
  -3.0  -4.8  -0.3  -3.0   1.8   2.1
   1.0   2.5  -0.8   1.7  -1.2  -0.7
Maximum value 4.3 at (2,3)
Another example?  Enter y (yes) or n (no):  n
```

built-in function `rand` to create a matrix of random real values, each within the interval of (0,1). The two arguments in the function call to `rand` specify the number of rows and columns, in that order, of random numbers to be generated.

Problem 9.7. Complete the following function:

```
function rmax = maxByCol(M)
% Post:  rmax(c) is the row index of the largest entry in column c
% Pre:  M is at least 1-by-1
```

Problem 9.8. We say that `M(i,j)` is a *saddle point* of the matrix `M` if it is at least as large as any other entry in its row and no bigger than any other entry in its column. Complete the functions

```
function rowNum = minInCol(M,c)
% Post:  rowNum is the index (row number) of the smallest entry in M(:,c)
% Pre:  M(:,c) has length>1

function colNum = maxInRow(M,r)
% Post:  colNum is the index (column number) of the largest entry in M(r,:)
% Pre:  M(r,:)  has length>1
```

and use them to implement the following function:

```
function [isSaddle,row,col] = isSaddlePt(M)
% Post:    If there is a saddle point, then isSaddle=1 and M(row,col) is a saddle point.
%          Otherwise isSaddle=0 and row=col=[].
% Pre:  M is at least 2-by-2
```

Problem 9.9. Suppose we want to find the thickest "border" around a cell `M(r,c)` such that the entries in the border are no larger than the value in `M(r,c)`. The thickness of the border is the number of cells to the left, right, top, and bottom of `M(r,c)`. For example, the border with a thickness of 3 around the cell `M(7,25)` refers to the submatrix `M(4:10,22:28)` (but excludes the center cell `M(7,25)`). Function `ringOK` checks that the ring of cells `d` cells from `M(r,c)` have entries that are no larger than `M(r,c)`. Function `findBorder` finds the thickness of the thickest border around `M(r,c)` such that the entries in the border are no larger than `M(r,c)`. Complete the two functions below. Use function `ringOK` in implementing function `findBorder`. Note that the maximum thickness may be zero.

```
function ok = ringOK(M,r,c,d)
% Post:    ok=1 if M(r,c) is at least as large as the cells in
%          M(r-d,c-d:c+d), M(r+d,c-d:c+d), M(r-d:r+d,c-d), M(r-d:r+d,c+d).
% Pre:     For matrix M of dimension nr-by-nc, 2<r<=nr and 2<c<=nc,
%          and 0<=d<=min(r-1,nr-r,c-1,nc-c).

function width = findBorder(M,r,c)
% Post:    width = maximum thickness of the border around M(r,c) such
%          that the entries in the border are no larger than M(r,c)
% Pre:     For matrix M of dimension nr-by-nc, 2<r<=nr and 2<c<=nc
```

A broad family of 2-dimensional array operations amount to a sequence of 1-dimensional array operations in which the same calculations are performed on each column (or row). For example, a common enterprise in data analysis is to "normalize" the values in a 1-dimensional array by their mean μ and standard deviation σ. Thus, if

x = | 10 | 40 | 20 | 30 |,

then $\mu = 25$, $\sigma = \sqrt{((10-25)^2 + (40-25)^2 + (20-25)^2 + (30-25)^2)/4} = 5\sqrt{5}$. Let `z` be the normalization of `x`, then

$$\texttt{z(i)} = \frac{\texttt{x(i)} - \mu}{\sigma}$$

where `i` is the subscript (or index) of `x`. Therefore,

$$\text{z} = \boxed{-3/\sqrt{5}}\boxed{3/\sqrt{5}}\boxed{-1/\sqrt{5}}\boxed{1/\sqrt{5}}.$$

The following fragment performs this task for an array `x` with length `n`:

```
% Compute the mean
mu= 0;
for i= 1:n
   mu= mu + x(i);
end
mu= mu/n;
% Compute the standard deviation
sigma= 0;
for i= 1:n
   sigma= sigma + (x(i)-mu)^2;
end
sigma= sqrt(sigma/n);
% Normalize
for i= 1:n
   z(i)= (x(i)-mu)/sigma;
end
```

It is easy to check that `z` has zero mean and unit standard deviation, hence the term "normalize." Here is a function that normalizes each of the columns in an array `M`:

```
function N= normalizeCols(M)
% Post:  Each column of N is the normalization of the corresponding column in M
% Pre:  M is at least 1-by-1

[nr,nc]= size(M);
for c= 1:nc
   % Normalize the c-th column
   % Compute the mean
   mu= 0;
   for r= 1:nr
      mu= mu + M(r,c);
   end
   mu= mu/nr;
   % Compute the standard deviation
   sigma= 0;
   for r= 1:nr
      sigma= sigma + (M(r,c)-mu)^2;
   end
   sigma= sqrt(sigma/nr);
```

```
    % Normalize
    for r= 1:nr
       N(r,c)= (M(r,c)-mu)/sigma;
    end
end
```

See Example9_5.

```
% Example 9_5:  Normalizing data

% Define constants
lBound= 0;
uBound= 10;
nr= 6; nc = 4;
anotherEg= 'y';

while (  anotherEg ~= 'n' )
   % nr-by-nc matrix where each entry is random in (lBound..uBound)
      M = rand(nr,nc)*(uBound-lBound) + lBound;
   fprintf('Original matrix:  \n');
   showMatrix(M);
   fprintf('Normalized matrix:  \n');
   showMatrix(normalizeCols(M));
   anotherEg= input('Another example?  Enter y (yes) or n (no):  ', 's');
end
```

Sample output:

```
Original matrix:
 9.5   4.6   9.2   4.1
 2.3   0.2   7.4   8.9
 6.1   8.2   1.8   0.6
 4.9   4.4   4.1   3.5
 8.9   6.2   9.4   8.1
 7.6   7.9   9.2   0.1
Normalized matrix:
  1.2   -0.3    0.8   -0.0
 -1.7   -1.9    0.2    1.4
 -0.2    1.1   -1.7   -1.1
 -0.7   -0.3   -0.9   -0.2
  1.0    0.3    0.9    1.2
  0.4    1.0    0.8   -1.2
```

PROBLEM 9.10. Complete the following function for smoothing data in a matrix:

```
function S = smooth(M)
% Post:    S is the smoothed data from matrix M.
%          For nr-by-nc M and S, S(i,j) = M(i,j) if i=1 or i=nr or j=1 or j=nc.
%          Otherwise, S(i,j) is the average of M(i-1,j),M(i+1,j),M(i,j-1),M(i,j+1).
% Pre:  M is at least 1-by-1
```

Computations that involve 1- and 2-dimensional arrays "at the same time" arise in numerous applications. Consider a situation where a company has m factories, each of which can produce any of n products. An $m \times n$ matrix `cost` can be used to store cost-of-production information, the value of `cost(f,p)` being the cost to factory f for producing one unit of product p. Here is a sample `cost` matrix:

`cost` =

10	32	21	73	5
12	27	25	67	6
9	30	26	73	4

Thus, Factory 2 can produce product 4 at a cost of $67 per unit.

A customer wishes to purchase a certain number of each product. A vector `d` can be used to represent this demand, e.g.,

`d` =

100	50	80	10	30

We wish to determine the factory that can fill the purchase order most cheaply. Note that the cost to factory f is a summation that involves numbers from the f-th row of `cost` multiplied by the corresponding numbers from `dq`:

$$
\begin{array}{lcrcrcrcrcrcr}
\text{Cost to Factory 1} & = & 10 \cdot 100 & + & 32 \cdot 50 & + & 21 \cdot 80 & + & 73 \cdot 10 & + & 5 \cdot 30 & = & 5160 \\
\text{Cost to Factory 2} & = & 12 \cdot 100 & + & 27 \cdot 50 & + & 25 \cdot 80 & + & 67 \cdot 10 & + & 6 \cdot 30 & = & 5400 \\
\text{Cost to Factory 3} & = & 9 \cdot 100 & + & 30 \cdot 50 & + & 26 \cdot 80 & + & 73 \cdot 10 & + & 4 \cdot 30 & = & 5330
\end{array}
$$

In terms of the arrays, the three production costs are given by

```
C1= cost(1,1)*d(1) + cost(1,2)*d(2) + cost(1,3)*d(3) + cost(1,4)*d(4) + cost(1,5)*d(5)
C2= cost(2,1)*d(1) + cost(2,2)*d(2) + cost(2,3)*d(3) + cost(2,4)*d(4) + cost(2,5)*d(5)
C3= cost(3,1)*d(1) + cost(3,2)*d(2) + cost(3,3)*d(3) + cost(3,4)*d(4) + cost(3,5)*d(5)
```

Picking up the pattern and letting `np` be the number of products, we see that the fragment

```
total= 0;
for p= 1:np
   total= total + cost(f,p)*d(p);
end
```

assigns to `total` the f-th factory's cost of production. Our goal is to determine which factory (the value of `f`) has the smallest possible cost. We can write the following function `cheapest`:

```
function [fMinCost, minCost] = cheapest(cost, d)
% Post:  minCost = the minimum cost of all factories to satisfay demand d.
%        fMinCost = the factory number that has minCost.
% Pre:  For a cost matrix that is nf-by-np, demand d is a vector of length np.

[nf,np]= size(cost);
minCost= realmax; % min cost initialized to a large number

for f= 1:nf
   % Calculate cost of factory f
   total= 0; % total cost factory f, initialized to 0
   for p= 1:np
      total= total + cost(f,p)*d(p);
   end

   if  (total<minCost)
      minCost= total;
      fMinCost= f;
   end
end
```

Some interesting boolean computations arise with the introduction of an "inventory array." Assume that `inventory` is initialized with the understanding that `inventory(f,p)` contains the number of units of product p on hand at factory f. Factory f can therefore process the purchase order if `inventory(f,p)>=d(p)` for $p = 1$ to np. Thus if

`inventory` =

150	200	100	120	110
200	40	130	20	40
300	50	100	12	80

and

`d` =

100	50	80	10	30

then factories 1 and 3 have sufficient inventory but factory 2 does not.

`Example9_6` contains a set of functions for experimenting with different cost-demand-inventory scenarios. The program prompts the user to enter a vector representing the purchase order, computes the costs, and determines whether the factories have sufficient inventory to fill the order. The function `eg9_6` is the "driver" and invokes functions `allcosts`, `fCanDo`, and `printIntArray`. Function `allcosts` calculates the costs of all the factories to fill the user specified purchase order. Function `fCanDo` evaluates whether factory `f` has sufficient inventory to fill the purchase order. Function `printIntArray` prints a matrix that contains integer values.

In the function `eg9_6`, the variable `prompt` stores a string that is created using concatenation and the built-in function `sprintf`. Function `sprintf` is used in the same way as `fprintf` to create a string that may include substituted variable values and special characters (e.g., `\n` for a line break). However, `sprintf` returns a string that *can be stored* while `fprintf`, as we have seen before, only prints the string to the screen.

In previous chapters, we have created scripts and functions that are stored separately—each under its own file name. A function that is stored under its own name can be used by any script

```
% Example 9_6:  Cost, demand, and inventory
function eg9_6()
% Program to display production cost, inventory level

% Define constants
nFact = 4; % no.  of factories
nProd = 8; % no.  of products
anotherEg = 'y';
prompt= sprintf('What is the purchase order for the %d products?', nProd);
prompt= [prompt sprintf('\nEnter the %d numbers as a vector:  ',nProd)];

while ( anotherEg ~= 'n')
   cost= floor(rand(nFact,nProd)*10+1); % random costs in (1..10)
   inv= floor(rand(nFact,nProd)*900+100); % random inventory in (100..999)
   fprintf('The Cost Array:')
   printIntArray(cost);
   purchase= input(prompt);
   fprintf('The Inventory Array:')
   printIntArray(inv);

   % Costs for filling the order (vector of length nFact)
   purchaseCosts= allCosts(cost, purchase);

   % Display results
   fprintf('Factory \t Cost \t Sufficient inventory?\n');
   for f= 1:nFact
      if  (fCanDo(inv(f,:), purchase))
         yesNo= 'yes';
      else
         yesNo= 'no';
      end
      fprintf('%4d \t %12d \t\t %s \n', f, purchaseCosts(f), yesNo);
   end

   anotherEg= input('\nAnother cost-inventory scenario (y/n)?  ','s');
end


%%%%%%%% Factory costs
function total = allCosts(cost, d)
% Post:  total(f) is the cost for factory f to fill demand d
% Pre:  For a cost matrix that is nf-by-np, demand vector d has length np

[nf,np]= size(cost);
total= zeros(nf,1);
for f= 1:nf % for factory f
   for p= 1:np % for product p
      total(f)= total(f) + cost(f,p)*d(p);
   end
end
```

Example 9_6 continues on next page

Example 9_6 continued

```
%%%%%%%% Check inventory of factory f
function canDo = fCanDo(v, d)
% Post:   canDo=1 if v(p)>=d(p) for all p, p=1:length(v)=length(d).
%         Otherwise canDo=0.
% Pre:  Inventory v and demand d are vectors of the same length

canDo= 1; % sufficient inventory so far, initialized to true.
p= 1; % product number
while (  p<=length(v) && canDo )
   canDo= v(p)>=d(p);
   p= p+1;
end


%%%%%%%% Print integer array
function printIntArray(A)
% Post:  Print the integer values in array A
% Pre:  A contains integer values and is at least 1-by-1

fprintf('\n');
[nr,nc]= size(A);
for r= 1:nr
   for c= 1:nc
      fprintf('%6d ', A(r,c));
   end
   fprintf('\n');
end
```

Sample output:

```
The Cost Array:
  1    4    8    2    9    1    1    4
 10    2    5    6    9    9    4    9
  2    9    4    7    7    2    9    5
  4    4    6    1    5    5    9    6
What is the purchase order for the 8 products?
Enter the 8 numbers as a vector:  [10 30 200 20 10 400 10 40]
The Inventory Array:
 654  559  970  217  112  353  930  195
 695  742  839  328  605  158  605  100
 654  563  385  822  509  528  687  587
 716  645  628  701  914  985  795  106
 Factory  Cost  Sufficient inventory?
    1     2430          no
    2     5370          no
    3     2390          yes
    4     3760          yes

Another cost-inventory scenario (y/n)?  n
```

or function in the current directory or on the search path. In `Example 9_6`, we write the four functions *in the same file* because they are specific to one particular problem and, except for `printIntArray`, are unlikely to be used by other programs. When there are multiple functions in a file, each has a function header as usual but no other "separators" are necessary. The first function is called the *main* or *top* function and any functions below the main function are called *subfunctions*. The subfunctions in a file can be accessed by the main function only but the subfunctions may call one another. The file name is the name of the main function with the extension `.m`.

PROBLEM 9.11. Complete the following:

```
function value = totalValue(cost, inv)
% Post:  value(f) = is the total value of factory f's entire inventory
% Pre:  cost, inv are matrices of same size, nf-by-np, so value is a length nf vector

function [fMinCost, minCost] = cheapestPossible(cost, inv, d)
% Post:    fminCost = 0 and minCost is negative if none of the factories
%          has enough inventory to fill demand d.  Otherwise minCost is
%          the lowest cost and fMinCost is the factory number with minCost.
% Pre:  cost, inv are matrices of same size, nf-by-np, and demand d is a length np vector
```

9.3 Tables in Two Dimensions

It is sometimes efficient to pre-compute function values and store them in a table for future use. Such a table would be a matrix if the function depends on two variables. To illustrate, consider the problem of finding the triangle with the longest perimeter assuming that the three vertices are chosen from a given finite point set. See FIGURE 9.1. Assume that `x(1:n)` and `y(1:n)` contain the coordinates of the points. The most obvious solution is simply to check every possible triangle:

```
pmax= 0;
for i= 1:n
   for j= 1:n
      for k= 1:n
         pij= distance(x(i),y(i),x(j),y(j));
         pjk= distance(x(j),y(j),x(k),y(k));
         pki= distance(x(k),y(k),x(i),y(i));
         pijk= pij+pjk+pki;
         if  (pijk > pmax)
            pmax=pijk; imax=i; jmax=j; kmax=k;
         end
      end
   end
end
```

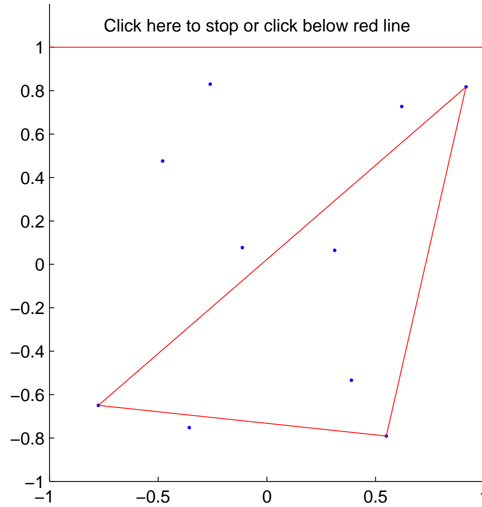

Figure 9.1 *Maximum Triangle*

Here, `distance(a,b,c,d)` returns $\sqrt{(a-c)^2+(b-d)^2}$, the distance between (a,b) and (c,d). The triply nested loop steps through all the possibilities, *but it is redundant.* Let the notation (i,j,k) stand for the triangle obtained by connecting points i, j, and k. Note that the triangles (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), and (3,2,1) are all the same. This 6-fold repetition can be avoided by abbreviating the loop ranges as follows:

```
for i = 1:  n
   for j = i+1:n
      for k= j+1:n
```

Think of the i-th point as the one selected first, the j-th point as the one selected second, and the k-th point as the one selected third. A little geometric thinking indicates that the optimum triangle will involve three *distinct* vertices, i.e., j is not the same as i and k should not be j or i. Therefore, the lower bound in an inner loop should be 1 larger than the index value of the immediate outer loop. Here is an enumeration of all the triangles that are checked in the $n=6$ case:

$(1,2,3)$	$(1,2,4)$	$(1,2,5)$	$(1,2,6)$
$(1,3,4)$	$(1,3,5)$	$(1,3,6)$	
$(1,4,5)$	$(1,4,6)$		
$(1,5,6)$			
$(2,3,4)$	$(2,3,5)$	$(2,3,6)$	
$(2,4,5)$	$(2,4,6)$		
$(2,5,6)$			
$(3,4,5)$	$(3,4,6)$		
$(3,5,6)$			
$(4,5,6)$			

There are four groups corresponding to i=1,2,3, and 4. Across each line we list all the triangles that involve a particular pair of points, avoiding the mention of triangles that have already been listed or that do not involve three distinct points.

Despite these modifications, there remains an even bigger redundancy. Each pairwise distance is computed about n times because each line segment that connects two points participates in about n triangles. This suggests that we pre-compute all the pairwise distances and store them in a matrix `D`. Incorporating this idea and casting the final solution in the form of a function we obtain:

```
function [imax, jmax, kmax] = MaxTriangle(x, y)
% Post:  imax, jmax, kmax are the 3 vertices of the triangle with the longest
%        perimeter:  (x(imax),y(imax)), (x(jmax),y(jmax)), (x(kmax),y(kmax)).
% Pre:  x, y are vectors of the same length and represent xy coordinates

n= length(x);

% Compute distance matrix between all pairs of points
D = zeros(n, n);
for i = 1:n
   for j=i+1:n
      D(i,j) = distance(x(i), y(i), x(j), y(j));
   end
end

% Find indices of triangle with longest perimeter
pmax = 0; % max perimeter found so far
for i= 1:n-2
   for j= i+1:n-1
      for k= j+1:n
         pijk= D(i,j) + D(j,k) + D(i,k);
         if  (pijk > pmax)
            pmax= pijk; imax= i; jmax= j; kmax= k;
         end
      end
   end
end
```

`Example 9_7` solicits a finite point set and then uses `maxTriangle` to compute the triangle with maximum perimeter.

Notice that only the upper triangular portion of the array is used. By setting up and using the `D` array, the number of calls to `Distance` is reduced by a factor of about n. Of course, `Distance` is not a particularly expensive function to execute, so this example of “trading space for time” is not very dramatic. But in other settings, the use of a 2-dimensional array to store function values can be crucial.

```
% Example9_7:  Maximum triangle

[x, y] = getPoints();
[imax, jmax, kmax] = maxTriangle(x,y);

% Draw triangle on figure opened via getPoints
xTri= [x(imax) x(jmax) x(kmax) x(imax)];
yTri= [y(imax) y(jmax) y(kmax) y(imax)];
plot(xTri,yTri,'-r')
```

The function `getPoints` was defined in Chapter 5. For sample output, see FIGURE 9.1.

PROBLEM 9.12. Complete the following function exploiting symmetry as much as possible:

```
function D = gridDist1(n)
% Post:  D(i,j) is the distance from (0,0) to (i,j) where i,j=1:n
% Pre:  n>=0
```

PROBLEM 9.13. Complete the following function exploiting symmetry as much as possible:

```
function D = gridDist2(n)
% Post:  D(i,j) is the distance from (0,n) to (i,j) where i,j=1:n
% Pre:  n>=0
```

PROBLEM 9.14. Complete the following function exploiting symmetry as much as possible:

```
function D = gridDist3(n)
% Post:  D(i,j) is the distance from (n/2,n/2) to (i,j) where i,j=1:n
% Pre:  n>=0 and is even
```