Matlab Primer #2 - *Programming*

Introduction:
Although it has much of the attractive high-level functionality of a software "package", Matlab is most definitely a *bona-fide* computer language with its own programming constructs and syntax. It is naturally outside the scope and purpose of this primer centred on Matlab image processing to offer a comprehensive treatment of the Matlab language[1]. Rather, our aim here is to offer a brief discussion (with illustrative examples) of the key programming constructs within (and features of) Matlab. Thus, our brief presentation is limited to *essentials* in the belief that mastery of the details and finer points, as with all computer languages, will come through a combination of practice and experience.

Matrices and Arrays
Most introductions to the Matlab language begin with a discussion of arrays and matrices. This discussion is no exception. The way in which Matlab defines and deals with arrays and array operations is a strong distinguishing feature which allows efficient and compact code to be written so that grasping the essentials of array handling is one of the most important things for the new user to grasp. In fact, it is useful in the beginning to consider that *every quantity (whether numeric or text) created by* Matlab *is simply an array*[2].

Matlab (an amalgam of **MAT**rix **LAB**oratory) was originally designed specifically to allow easy and comprehensive calculation/manipulation of matrices. Mathematically speaking, we need to be aware of the distinction between *matrices and arrays*. As computational objects, matrices are simply arrays – i.e. ordered rows and columns of numerical values. However, matrices are a type of array to which we attach a definite mathematical meaning and to which certain operations (e.g. matrix multiplication, transpose etc) can be meaningfully applied. Arrays on the other hand are often just convenient ways of storing and organizing data which can be numeric, textual or a combination of both and Matlab provides many useful ways of manipulating and dealing with these quite general arrays. The key point then is that within Matlab *there is no distinction between arrays and matrices*. Arrays and matrices are identical entities and it is only the user who can decide if the quantity he has defined is a "matrix" in the proper mathematical sense or simply an array. In the following discussion, no attempt is therefore made to distinguish between them and it will generally be clear from context whether a given array is a matrix.

[1] A comprehensive treatment is given in Mastering Matlab, Hanselman and Littlefield, Prentice Hall. The extensive Matlab documentation provided with the software is also very readable and informative .
[2] This is not strictly true but sufficiently close to the truth that it will serve its purpose.

Creating arrays in Matlab

The simplest way of creating small arrays in Matlab is to manually enter the elements in the command window, row by row. Try entering the commands given below:–

Matlab Commands	**What's Happening**
>>A=[1 2 3; 4 5 6]	%Create a 2 x 3 matrix A
>>B=[6 5 4; 3 2 1]	%Create a 2 x 3 matrix B

Comments

The **enclosing square brackets** indicate that the contents form an array. A **semi-colon** within the brackets indicates the start of a new row.

A vector is just a 1-D array :–

```
>>vec=[0 1 1]                    %create 3 element row vector, vec
```

We can turn a row vector into a column vector, by using the transpose operator

```
>> vec=vec'                      %make vec a transposed version of itself
```

or, of course, transpose a 2-D array :–

```
>>A'                              %Transpose A
```

We can also create 'string' arrays which contain text –

```
>> alph=['ab';'cd']              %Create  2x2 array of characters
>> alph(2,2)                     %Display element in 2nd row, 2nd column (d)
```

All 2-D arrays must be rectangular or square – in other words, to be a legal array each row in the array must have the same number of elements[3]. The same principle applies to arrays of higher dimension. Thus a legal 3-D array must comprise a "stack" of 2-D arrays

[3] So-called cell arrays and structures are advanced data structures which can be used to circumvent this restriction when needed. We must defer discussion of these data structures for later examples in the book but for the impatient reader, typing >> *doc struct* and >> *doc cell* at the Matlab prompt for information on the basic idea and use.

of *similar* dimensions, a legal 4-D array a group of 3-D arrays of similar dimensions and so on.

Accessing individual elements in an array

Individual array elements are accessed using subscripts as follows:-

Matlab Commands	**What's Happening**
>>vec(2)	% vec(2) is the second element of vec
>>A(1,2)	% Return element in the 1st row, 2nd column

We can change or correct the values of array elements by indexing them individually:–

>>A(1,2)=6	%Set element in 1st row, 2nd column to 6

Arrays can be 3-D (or have even higher dimensionality) :–

Matlab Commands	**What's Happening**
>>C=imread('football.jpg');	%Read in RGB (colour) image of football
>>C(128,39,1)	%Get R (red) value at given pixel (=32)

Comments

Note the use of the semi-colon after Matlab command to *suppress* the output of the values of the matrix variable C to the screen.

Accessing groups of elements in an array – the colon operator

Matlab has a very important and powerful *colon* operator (**:**). This can be used to create vectors and for subscripting arrays. Here's a couple of examples of its use in creating vectors -

Matlab Commands	**What's Happening**
>>x=1:10	% x=[1 2 3… 10] Increment is 1
>>y=0:5:100	%y=[0 5 10 ..95 100] Increment is 5

We can also access groups of elements in an array by using the colon operator to specify the vector indices**.** Try the following :-

Matlab Commands	**What's Happening**
>> vec(2:3)	%Extract elements 2 and 3 inclusive
>>B=A(1:2,2:3)	%Extract sub-array rows 1- 2, columns 2-3
>>C=A(1,1:2:3)	%Extract 1st and 3rd elements of 1st row

We can also use the colon operator to extract entire rows or columns of an array:-

Matlab Commands	**What's Happening**
>>A(:,1)	%Extract first column of A (ans=[1 ; 4])
>>A(2,:)	%Extract 2nd row of A (ans=[2 3 4])

When we use the colon operator along one of the array dimensions in this way, we are effectively allowing the index to *range over the entire length* of that dimension. Thus in the first example, A(:,1) means we fix the column number equal to 1 and extract all the elements in that column (i.e. the row index varies from 1 up to its maximum value). In the second example, A(2,:) we fix the row number equal to 2 and extract all the elements in that row (i.e. the column index varies from 1 up to its maximum value).

Try the exercise below.

Primer Exercise 2.1

- Read in the Matlab image 'liftingbody.png' (use function ***imread***).
- Extract the central 50% of the image pixels and display the image (use ***imshow***).
- Extract the first 64 columns of the image and display them as a new image.
- Extract the rows 33 to 64 inclusive of the image and display them as a new image.

Concatenation of arrays

We can easily concatenate (join together) arrays to make a larger array. In effect, we just consider the elements of an array *to be arrays themselves*.

Matlab Commands	**What's Happening**
>> clear;	%Clear workspace
>> A=[1 2; 3 4], B=[5 6; 7 8], C=[9 10 11 12]	%Create arrays
>> [A B]	%A B are the "columns" of new %array. Result has dimension 2x4.
>> [A; B]	%A B are the "rows" of new %array. Result has dimension 4x2.
>> [A B; C]	%Result has dimension 3x4.

The arrays which form the elements of the concatenated array *must be of conformable dimension* – i.e. the resulting array must be rectangular. For example, trying to form the arrays :–

```
>> [A; C]
>> [A B C]
```

does not work and Matlab responds with the appropriate error message –

```
??? Error using ==> horzcat
All matrices on a row in the bracketed expression must have the
same number of rows.
```

Try the exercises below:-

Primer Exercise 2.2

- Read in the Matlab images *cameraman.tif* and *rice.tif* (use the **imread** function).
- Concatenate them to make a single image and display the result (use **imagesc** or **imshow** function).

Primer Exercise 2.3

- Generate the same result as in exercise 2.2. but this time using the Matlab function ***repmat***. (Use the Matlab help facility to find out how this function is used).

Creating and Dealing With Larger Arrays

Manual entry as described above is suitable only for relatively small arrays. Larger arrays can be created in 2 basic ways.

- ***Matlab*** provides a number of built in functions for creating and manipulating simple arrays of arbitrary dimension. These are outlined in the table 2.1 below.

Function	Description	Simple example
zeros	Array of zeros	A=zeros(4);
ones	Array of ones	A=ones(4);
eye	Identity matrix	A=eye(4);
repmat	Make tiled copies of input array	A=eye(4); B=repmat(A,1,2);
rand	Random array – elements 0 – 1	A=rand(4);
randn	Normally distributed random array	A=randn(4);
linspace	Vector with linear increments	A=linspace(0,1,12);
logspace	Vector with logarithmic increments	A=logspace(0,1,12);
meshgrid	Make 2-D array from 2 input vectors	[X,Y]=meshgrid(1:16,1:16);

Table 2.1: Matlab functions for creating arrays

By suitable manipulation, we can create large arrays with meaningful contents.
Try the following examples :–

Matlab Commands	**What's Happening**
>>A=randn(4)	%Create normally distributed random 4 x 4 array.
>>x=linspace(-1,1,50); y=x	%Make vector x of 50 points linearly spaced from -1 to 1 %Make copy y
>>[X,Y]=meshgrid(x,y); >>mesh(X.^2+Y.^2)	%Produce 2-D grid array for all combinations of x and y. %Surface height display of function X^2+Y^2
>>clear; >>A=ones(128); surf(A);	%Generate array of ones, display as surface height.
>>A=imread('rice.tif'); >>B=repmat(A,2,2); imshow(B); >> thresh=100.*rand(size(A)); >> I=find(thresh>A); >> A(I)=0; imshow(A)	%Read in rice image %Make 2 x 2 tiled copy and display %Random array same size as A – range 0-100 %Find all pixels at which random array is > A %Set them to zero and display modified image

Primer Exercise 2.4

- Read in the images *rice.tif* and *cameraman.tif*.
- Find all pixels for which the *rice* image has a value greater than that of the *cameraman* image and copy the corresponding pixel values into the *cameraman* image. (Use the Matlab function ***find***)
- Display the result.

The last example uses the powerful in-built ***find*** function and so-called linear indexing. We'll discuss this function and the technique of *linear indexing* shortly, so don't worry if all the last example is not completely transparent.

Those with some previous programming experience will note from the examples above that there are no loop constructs (for, do, while etc) used to build and manipulate the arrays. A feature of Matlab which distinguishes it from many other languages is called *implicit vectorisation*. Namely, if the input to a Matlab expression is a vector or array, ***Matlab*** knows to evaluate the expression at every value of the input, producing an output that is also a vector or array.

A second way in which larger arrays can be constructed is through use of appropriate loop constructs. This is done in a way similar to most other languages. The following example assigns the value 1 to all the elements in column 1, 2 to all the elements in column 2 and so on :–

Matlab Commands	**What's Happening**
>>A=zeros(16);	%Make 16 x 16 zero matrix
>>for i=1:16;	%begin for loop
A(:,i) = i.*ones(16,1);	%Build the columns of A
End;	%End for loop

The next example creates an array that randomly allocates values of 0 or 1 to the rows of an array by "flipping a coin":–

Matlab Commands	**What's Happening**
>>A=zeros(16);	%Create array of zeros
>>for i=1:16	
if rand > 0.5	%Flip a coin with rand (>0.5 == heads)
A(i , :) =ones(1,16)	%If "heads" occurs, assign value 1 to
end	%elements of column i
end	
>>imshow(A);	%Display resulting array

As a rule, you should try to avoid loop constructs in Matlab where this is possible - especially for large arrays. Matlab is an *interpreted* language and this means that loops execute slowly compared to compiled code. Loops are sometimes unavoidable and have their place in the Matlab language and we shall say more about loop constructs later in this section. The example below *achieves the same purpose* as the one above but uses Matlab's implicit vectorization capabilities

Matlab Commands	What's Happening
A=zeros(16);	%Create 16x16 array of zeros
I=find(rand(16,1)>0.5); A(:,I)=1;	%Find indices at which 16 random %numbers exceed threshold and assign %corresponding columns of A equal to 1

Comments

Note how the Matlab code is very compact and that the output of one function can directly constitute the input to another (e.g. *find(rand.. etc)*)

Determining the size of arrays

The function ***whos*** only prints information to the screen about the size and type of variables currently existing within the workspace. To determine the size of an array and actually assign this information to a variable in the workspace, we must use the in-built ***size*** function. The basic syntax is *[m,n]=size(A)* where the number of rows and columns in A are assigned to *m* and *n* respectively.

For example :–

Matlab Commands	What's Happening
>>A=imread('circuit.tif'); dims=size(A);	%Read in image and ascertain size. [280 272]

Here, *dims* is a 2 element row vector containing the number of rows and number of columns. Try the following :–

Matlab Commands	What's Happening
>>A=imread('onion.png); dims=size(A)	%Read in colour image and ascertain size. %[135 198 3]

This time, dims is a 3 element row vector because A is an RGB colour image and RGB colour images are a composite of three separate 2-D images, one each for the red, green and blue components.

Array Utility functions

Matlab provides some further in-built functions for array manipulation. Some of these are summarised in the table below :–

Function	Description
reshape	Change the shape of an array
fliplr	Orders the columns in reverse order
flipud	Orders the rows in reverse order
tril	Extracts lower triangular part of array
triu	Extracts upper triangular part of array
rot90	Rotates array counter-clockwise by 90^o

Here are some fun examples to try out :–

Matlab Commands	What's Happening
>>A=imread('cameraman.tif'); imshow(A);	%Display original image matrix
>>B=fliplr(A); imshow(B)	%Flip left-right and display
>>C=flipud(B); imshow(C);	%Flip upside-down and display
>>subplot(1,3,1), imshow(A);	
>>subplot(1,3,2), imshow(B);	
>>subplot(1,3,3), imshow(C);	%Display together in same window

Primer Exercise 2.5
Read in the images *rice.png* and *cameraman.tif* . Reflect the rice image about the x axis, the cameraman image about the y axis and add them together using Matlab's ordinary addition operator. Display the result using the function ***imshow***

Relational and logical operators

Comparison of two quantities is fundamental to any language. Matlab includes all the common relational operators, summarized in the table below:-

Relational Operator	Description
<	Less than
<=	Less than or equal to
~=	Not equal to
>	Greater than
>=	Greater than or equal to
==	Equal to

Relational operators can be used to compare two arrays or an array to a scalar. The output of all relational expressions produces logical arrays with 1 where the expression is True and 0 where it is False. Thus, if A = [1 3 2 6] and B = [0 4 3 4], then A < B = [0 1 1 0].

Try the examples below :-

Matlab Commands	What's Happening
>> A=1; B=2; tf = A<B	%tf assigned value 1 because A is less than B
>>A=[1 2 3]; B=[0 2 4]; tf = A>=B	%tf=[1 1 0] - first two elements of A are >=those in B
>>A=3; B=[2 4 0 3 1]; tf = A==B	%tf=[0 0 0 1 0] because 4th element is the same as A

Note that comparison takes place on an *element by element* basis, returning 1 where the relation is satisfied and 0 where it is not. Note also the difference between the relational equality (==) and the assignment operator (=).

Warning !
== and = are *distinct operators.*
The relational operator == tests for the equality of quantities; the assignment operator = is used to assign the output of an operation to a variable. This is a common source of error, especially for beginners.

Logical operators
Logical operators allow us to combine or negate relational expressions. Applying these operators to relational expressions also results in logical (0-1) arrays having value 1 where the expression is TRUE and 0 where it is FALSE.

The basic logical operators in ***Matlab*** are given in the table below. Some examples of their use is illustrated below :-

Matlab Commands	What's Happening
>>A=[1 2 3]; B=[0 2 4]; C=[1 3 2];	%Assign matrices
>>B>A & B>C	%ans=[0 0 1] – only last element of B satisfies both %conditions
>>~(B>A & B>C)	%ans=[1 1 0] – Logical NOT of previous example
>>A<C | B~=C	%ans=[1 1 1] – All elements of B are different from C

Matlab Logical Operator	Description
&	Logical AND
|	Logical OR
~	Logical NOT
Matlab Logical function	**Description**
and	Logical AND
or	Logical OR
xor	Logical exclusive OR
any	Logical TRUE if any element is TRUE
all	Logical TRUE if all elements TRUE
isempty	Logical TRUE if matrix is empty
isequal	Logical TRUE if matrices are identical
ismember	

Try some of these basic examples –

Matlab Commands	**What's Happening**
>> A=[1 2 3]; B=[0 2 4]; C=[1 3 2];	%Assign matrices
>> and(A<B,C<B)	%Logical AND between logical arrays A<B and C<B: ans=[0 0 1]
>> xor(A<B , B<C)	%Exclusive OR between arrays A<B and C<B: %ans=[1 1 1]
>> isempty(find(A>4))	%A has no elements > 4. Find function returns the empty %array. isempty tests for this. ans=1
>> any(B>A)	%if any elements of B>A returns logical true. ans=1

Note from the last example how the output from the function ***find*** can directly form the input to the function ***isempty***. This is possible in this specific case because the output from ***find*** is a single array/matrix and the required input to ***isempty*** is also a single array/matrix. The reader should consult the online help facility for further details of the logical functions described above .

Matlab Flow Control

Matlab provides flow control statements in a way very similar to most other programming languages. Conditional statements where certain operations are carried out only if certain conditions are satisfied can be constructed in two basic ways - ***if*** and ***switch*** statements.

If statements

Matlab supports these variants of the ***if*** construct -

if …(statement)... end
if …(statement)... else …(statement)... end
if …(statement)... elseif …(statement)... else …(statement)... end

A single example of each variant is sufficient to illustrate its use :-

Matlab Commands	**What's Happening**
>> d = b^2 - 4*a*c;	%Define d
>> if d<0	%Begin conditional ***if***
>> disp('warning: discriminant negative, roots imag');	%Statement
>> end	%End ***if***

Matlab Commands	**What's Happening**
>> d = b^2 - 4*a*c;	
>> if d<0	%Begin conditional ***if***
>> disp('warning: discriminant negative, roots imag');	
>> else	%single alternative (else) clause
>> disp('OK: roots are real, but may be repeated');	
>> end	%End ***if***

Matlab Commands	**What's Happening**
>>d = b^2 - 4*a*c;	
>>if d<0	%Begin conditional ***if***
>>disp('warning: discriminant negative, roots imag');	
>>elseif d==0	%1st alternative (elseif) clause
>>disp('discriminant is zero, roots are repeated');	
>>else	%Last alternative (else) clause
>>disp('OK: roots are real and distinct');	
>>end	%End ***if***

Note

- no semicolon is needed to suppress output at the end of lines containing *if, else, elseif* or *end.*
- The operator == (is equal to) is used for logical comparison. It is quite distinct from the single = sign *which assigns a value* to a variable.
- Indentation of ***if*** blocks is not required, but considered good style.

Primer Exercise 2.6
Write a Matlab function which takes a single 2-D matrix input argument (an image) and

- Scales the matrix so that the maximum pixel value is = 1.
- Calculates the mean pixel value of the scaled matrix
- Conditionally prints the following strings to the monitor -
  "Image is dark" if the mean x<0.5
  "Image is normal" if the mean x=0.5
  "Image is bright" if the mean x>0.5

Test your function on one or two of the Matlab images to make sure it operates properly.

Switch-case constructs

A switch-case construct is similar in basic purpose to an ***if*** construct. If a sequence of commands must be conditionally evaluated based on the equality of a single common argument, a switch-case construct is often easier. Here is an example :–

Matlab Commands	**What's Happening**
info=imfinfo('cameraman.tif');	%assign image information to structure *info*
switch info.Format	%switch construct depends on *Format* field
case {'tif'}	%execute statement if Format ==’tif’
disp('Queried file is TIF format')	
case {'jpg','jpeg'}	
disp('Queried file is JPG format')	%execute statement if Format ==’jpg’
end	

Comments

- The Matlab function imfinfo attempts to infer the contents of the image *cameraman.tif*. It assigns the results to a *structure* called *info* (a structure is essentially a collection of numeric and string variables each of which can be accessed using the *same basic name* but a different specific field).
  Type >> *help struct* at the Matlab prompt for further information
- One of the fields in this particular structure is the *Format* field. The switch statement effectively registers/records the string contained in the format field.
- This is compared for *equality* with each of the strings contained within the curly braces { } that are part of the case clauses.
- If equality occurs, the commands following the case clause are executed.

In the example above, Matlab responds with
>> *Queried file is TIF format*

Thus the generic form for a switch statement is

```
switch the switch expression
case a case expression
     corresponding statements….
case another case expression
otherwise
     corresponding statements….
end
```

Thus the corresponding group of legal Matlab statements will be executed if the case expression matches the switch expression.

Loop Constructs *for* and *while*

for loops allow a set of statements to be executed a fixed number of times. The syntax is :–

```
for x = array
     statements..
end
```

The for loop basically works by assigning the loop variable x to the next column of the vector *array* at each iteration. Consider the following simple ***for*** loop to sum all the integers from 1 to 10 :–

Matlab Commands	**What's Happening**
sum=0;	%Initialise sum
for i = 1:10	%Create 10 element array i=[1 2 3…10]. Begin loop
sum=sum + i;	%Calculate cumulative sum
end	%End Loop
cumsum(1:10)	%NB. This is the easy way to do this in Matlab…

Thus, in this ***for*** loop the variable i takes on the next value in the array (1,2,..10) on each successive iteration. The next example shows a slightly more advanced piece of code involving some new Matlab functions - we include it now to show that the basic construction of the ***for*** loop is exactly the same :–

Matlab Commands	**What's Happening**
A=imread('cameraman.tif'); B=imread('moon.tif');	%Read in two images assign to arrays ay
B=imresize(B,size(A),'bilinear');	%Resize B to match dimensions of A
for i=1:size(A,1)	%Begin looping through columns of A
rvals=rand(size(A,1),1); indices=find(rvals>0.5);	%Randomly sample for pixel locations %in columns
A(i,indices)=B(i,indices);	%At selected pixel locations, copy B %values into A
end	%End loop
imshow(A);	%Display modified image

The following example creates a vector with the first 16 terms of the Fibonacci series $x_{n+1} = x_n + x_{n-1}$ using a *for* loop. Another *for* loop is then used to cyclically permute the terms to form the rows of the image :–

Matlab Commands	What's Happening
vec=ones(1,16);	%16 element vector all =1
for i=2:15	%Begin Loop
vec(i+1)=vec(i)+vec(i-1)	%Build Fibonacci terms
end	%End Loop
A(1,:)=vec;	%First column of A Fibonacci series
for i=1:15	%Begin Loop
A(i+1,:)=[vec(17-i:16) vec(1:16-i)]	%Cycle elements for each new col
End	%End Loop
imagesc(log(A))	%Display on log scale

while loops
The syntax of the while loop is very simple -

```
while expression
     statements
end
```

In the while loop, *expression* is a logical expression and the *statements* following are repeatedly executed until the *expression* in the while statement becomes logically false. Here are a couple of simple examples :-

Matlab Commands	What's Happening
>>i = 2;	%Initialise i=2
>>while i<8	%logically true if i<8
>>i = i + 2	%increment I by 2 each iteration
>>end	%End loop

This example successively generates sets of 4 random numbers in the interval 0-1. The loop stops when we obtain a set of 4 numbers all of which are less than 0.25.

Matlab Commands	What's Happening
A=ones(4,1); k=0;	%Initialise A and counter k
while any(A)	%False if ALL elements of A =0
A=rand(4,1)>0.25	%A is 4 element logical array
	%Element = 1 if rand > 0.5
k=k+1;	%Increment counter
end	%End loop
k	%Show final number of trials

More on Arrays and Indexing

We end this primer with an overview of some of the more advanced aspects of indexing into arrays. Indexing into an array is a means of selecting a subset of elements from the array. Indexing is also closely related to another term we mentioned in passing early on: *vectorization*. Vectorization means using Matlab language constructs to eliminate program loops, usually resulting in programs that run faster and are more readable. Try out the following examples which illustrate a number of powerful indexing techniques.

Indexing Vectors

Let's start with the simple case of a vector and a single subscript. The vector is:-

```
>> v = [16 5 9 4 2 11 7 14];
>> v(3)                          % The subscript can be a single value –
                                 %"Extract the % third element"  ans =  9
```

Or the subscript can itself be another vector :–

```
>> v([1 5 6])                    % Extract the first, fifth, and sixth elements
                                 % ans = 16   2   11
```

Matlab's colon notation (**:**) provides an easy way to extract a range of elements from v:-

```
>> v(3:7)                        % Extract elements 3 to 7 inclusive.
                                 % ans =   9   4   2   11   7
```

You can swap the two halves of v to make a new vector :–

```
>> v2 = v([5:8 1:4])             % Extract and swap the halves of v
                                 % v2 =    2   11   7   14   16   5   9   4
```

The special *end* operator is an easy short-hand way to refer to the last element of v -

```
>> v(end)                        % Extract the last element
                                 % ans =   14
```

The end operator can be used in a range –

```
v(5:end)                         % Extract elements 5 to last inclusive
                                 % ans =   2   11   7   14
```

You can even do arithmetic using end –

```
v(2:end-1)                       % Extract second to penultimate element
                                 % ans = 5   9   4   2   11   7
```

The colon operator is very powerful and can be used to achieve a variety of effects :–

```
>> v(1:2:end)                    %extract every k-th element – k=2 here
                                 % ans = 16   9   2   7
```

```
   v(end:-1:1)                   % Reverse the order of elements
                                 %ans = 14   7   11   2   4   9   5   16
```

By using an indexing expression on the left side of the equal sign, you can replace certain elements of the vector.:-

```
>> v([2 3 4]) = [10 15 20]       % Replace selected elements of v
                                 %v =   16    10    15    20    2    11    7    14
```

Usually the number of elements on the right must be the same as the number of elements referred to by the indexing expression on the left. You can always, however, use a scalar on the right side.

```
>>v([2 3]) = 30                   % Replace second and third elements by 30
                                 %v =  16   30   30   20   2   11   7 1  4
```

This form of indexed assignment is called scalar expansion.

Indexing Arrays with Two Subscripts

Now consider indexing into an array. We'll use a magic square for our experiments.

```
>>A = magic(4)                    %Create a 4 x %4 magic square using in-built
                                 %function magic
```

Most often, indexing in arrays is done using two subscripts - one for the rows and one for the columns. The simplest form just picks out a single element :–

```
>>A(2,4)                          % Extract the element in row 2, column 4
                                 % ans =  8
```

More generally, one or both of the row and column subscripts can be vectors.

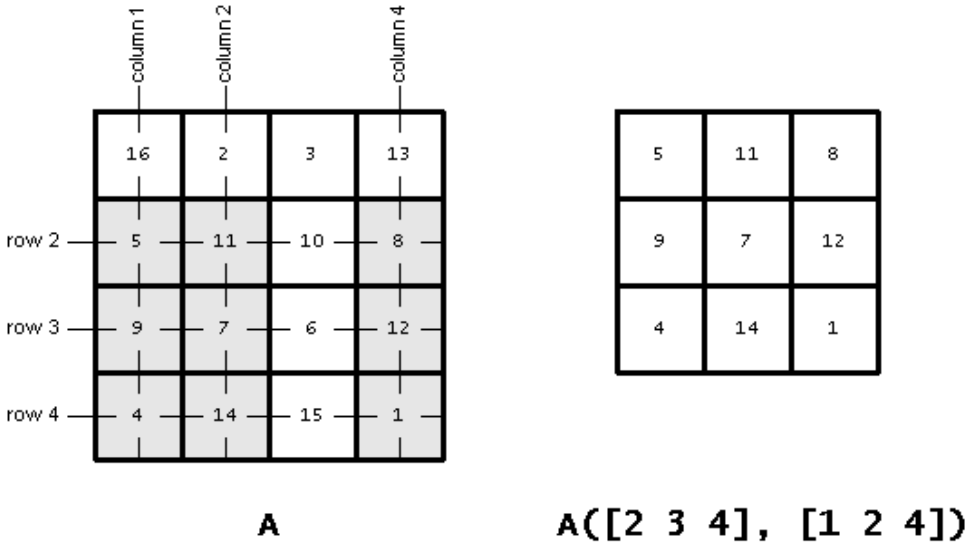
```
>>A(2:4,1:2)                      %Extract rows 2 to 4, columns 1 to 2
```

A single semi-colon (**:**) in a subscript position is short-hand notation for *"1:end*" and is often used to select entire rows or columns.

```
>> A(3,:)                    % Extract third row
                             % ans =      9   7   6   12
```

```
>> A(:,end)                  % Extract last column
                             % ans = 13 8 12 1
```

The diagram below illustrates a trickier use of *two-subscript indexing*. The expression A([2 3 4], [1 2 4]) does NOT extract elements A(2,1), A(3,2) and A(4,4) as one might, at first glance think, but rather extracts the elements shown on the right :–



Selecting scattered elements from an array

Suppose you do want to extract the (2,1), (3,2), and (4,4) elements from A ? As we have just shown, the expression A([2 3 4], [1 2 4]) doesn't do this.

There is often confusion over how to select *scattered elements* from an array. To do this requires the use of *linear indexing* and that brings us to our next topic.

Linear Indexing

We begin with a question. What exactly does this expression A(14) do ?

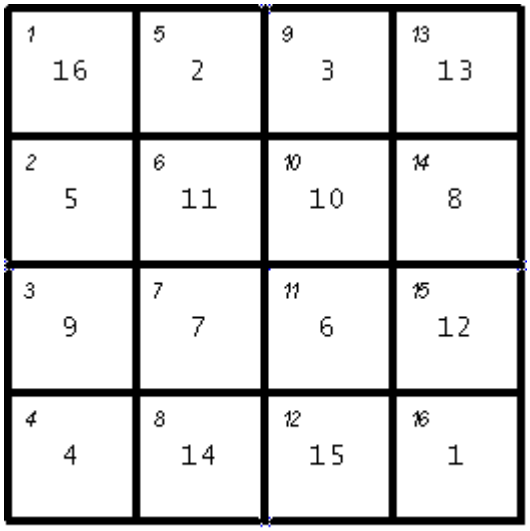
When you index into the array A using only one subscript, ***MATLAB*** treats A *as if its elements were strung out in a long column vector by going down the columns consecutively,* as in :–

16
5
9
…..

8
12
1

The expression A(14) simply extracts the 14th element of the implicit column vector.

Here are the elements of the matrix A along with their linear indices in the top left corner of each square:-



A

The linear index of each element is shown in the upper left. From the diagram you can see that A(14) is the same as A(2,4) – i.e. *Linear index = (column number – 1) x (number of rows in array) + row number*

The single subscript can be a vector containing more than one linear index, as in:

```
>>A([6 12 15])                    %Extract "linear" elements 6,12,15
                                  %ans = 11   15   12
```

Let's consider again the problem of extracting just the (2,1), (3,2), and (4,4) elements of A. You can use linear indexing to extract those elements:

```
>>A([2 7 16])                     %ans = 5   7   1
```

It's easy to see the corresponding linear indices for this example, but how do we compute linear indices in general ? provides a function called ***sub2ind*** that provides the appropriate linear indices corresponding to the given row and column subscripts –

```
>>idx = sub2ind(size(A), [2 3 4], [1 2 4])        % idx =  2   7   16
```

```
>>A(idx)                                          %display values referenced by these indices
                                                  %ans =  5   7   1
```

The in-built Matlab function ***ind2sub*** does precisely the reverse, providing the row and column indices given the linear index and the size of the array.

The *find* function

The find function was introduced earlier but is very useful and so deserves special mention. It can be used to extract the indices of all those elements of an array satisfying a stated condition. For example, consider first creating a vector :-

```
>> clear; A=1:10;  i=find(A>5)                    % i = [6 7 8 9 10]
```

It can also be used on 2-D arrays to return the *row-column* indices :–

```
>> clear; A=[1 2 3; 0 1 2; -1 1 1]; [i,j]=find(A<=0)    % i=[2 3] j=[1 1]
```

Or it can extract the *linear indices* of the 2-D array :-

```
>> i=find(A<=0)                                   % i=[2 3]
```

In the following example we take two images A and B (of equal size) and compare them on a pixel by pixel basis to see whether image A has the greater intensity. If so, we copy the value in A to B.

```
A=imread('cameraman.tif'); B=imread('rice.png');    %Read in images
I=find(A>B); B(I)=A(I); imshow(B);                  %get linear indices for A>B. Copy over original.
```

Logical Indexing

Another indexing variation, logical indexing, has proved to be both useful and expressive. In logical indexing, you use a single, logical array for the matrix subscript. Matlab extracts the matrix elements corresponding to the nonzero values of the logical array. The output is always in the form of a column vector. For example, A(A > 12) extracts all the elements of A that are greater than 12.

```
>>A(A > 12)                                    %ans = 16 14 15 13 (as column vector)
```

Many Matlab functions that start with "*is*" (***isnan, isreal, isempty..)*** return logical arrays and are very useful for logical indexing. For example, you could replace all the NaNs[4] in an array with another value by using a combination of isnan, logical indexing, and scalar expansion. To replace all NaN elements of the matrix B with zero, use:-

```
>> B=0./linspace(5,0,6)
>> B(isnan(B)) = 0
```

Or you could replace all the spaces in a string matrix str with underscores:-

```
>>str='A stitch in time'
>>str(isspace(str)) = '_'
```

Logical indexing is closely related to the ***find*** function. The expression A(A > 5) is equivalent to A(find(A > 5)). Which form you use is mostly a matter of style and your sense of the readability of your code, but it also depends on whether or not you need the actual index values for something else in the computation. For example, suppose you want to temporarily replace NaN values with zeros, perform some computation, and then put the NaN values back in their original locations. In this example, the computation is two-dimensional filtering using ***filter2***. You do it like this :-

```
>>nan_locations = find(isnan(A));
>>A(nan_locations) = 0;
>>A = filter2(ones(3,3), A);
>>A(nan_locations) = NaN;
```

The Matlab indexing variants illustrated above give you a feel for ways we can create compact and efficient code. You will certainly not need them all right away but learning to include these techniques and related functions in your Matlab programs helps to write and create efficient, readable, vectorized code.

[4] NaN is the I.E.E.E. representation for Not-a-number. A NaN results from mathematically undefined operations such as 0/0 or inf/inf. Any arithmetic operation involving a NaN and a legal mathematical entity results in a NaN. They ca, however, sometimes be useful to identify and/or label locations at which data is missing.

Further Help in Learning Matlab

In this brief introduction, we have tried to give an indication and some simple examples of some of Matlab's key programming constructs – hopefully enough to get you off the ground. There are now many good core Matlab books (non image processing specific) and a significant amount of web-based material specifically aimed at helping users develop their knowledge of the Matlab language and we refer the reader to a selected sub-set of this material on the book website (with these primers).

We must also make mention of the excellent on-line help resources and comprehensive documentation available to Matlab users. In the authors' experience it is clearly written, well organized and both quick and easy to use – it almost (but we hope not quite) makes the these two supporting Matlab primers for this book superfluous. We hasten to recommend this material to the reader.

As a rather keen DIYer, I cannot help but compare programming in Matlab with undertaking some task of physical construction such as building a house. It is a simple but accurate analogy. Building a basic house *can* actually be done with a bare minimum of essential tools but it tends to be long-winded and hard work. I have often had the experience of beginning or even completing a particular task in building or carpentry only then to discover that there is actually a much easier way to do it, given the right tool and approach. If only one had known beforehand, how much time and effort could have been saved !
I look upon the basic knowledge of variable types, arrays and indexing and key programming constructs such as *for* and *while* loops, conditional *if* clauses etc that we have discussed as the ***essential tools***. However, many of Matlab's in-built functions are very much akin to ***specialist tools*** – they make easy that which is much harder if you try to do it all yourself. These days, when I am in a hardware store, I often just browse the tool section to see what's available just in case I should ever encounter a future job which could make good use of some specialist tool I encounter. In this spirit, we finish this very brief introduction by providing the reader with a selected but large list (by category) of Matlab functions together with a brief description of their purpose. Please browse at your leisure. The completely comprehensive list is, of course, available in Matlab's documentation and the help facility.

This is a book about image processing so, finally, note that the specialist functions associated with the *Matlab image processing toolbox* are ***not yet*** given here. The use of many of these specialist image processing functions is discussed and demonstrated in the remaining chapters of this book. A comprehensive list of functions available is available in the *Matlab image processing toolbox* documentation.

ELEMENTARY MATRICES	
Matlab function	**Description**
zeros	Zeros array
ones	Ones array
eye	Identity matrix
repmat	Replicate and tile array
rand	Uniformly distributed random numbers
randn	Normally distributed random numbers
linspace	Linearly spaced vector
logspace	Log spaced vector
freqspace	Frequency spacing for frequency response
meshgrid	X and Y arrays for 3-D plots
accumarray	Construct an array with accumulation
BASIC ARRAY INFORMATION	
size	Size of array
length	Length of vector
ndims	Number of dimensions
numel	Number of elements
disp	Display matrix or text
isempty	True for empty array
isequal	True if arrays numerically equal
isequalwithequalnans	True if arrays numerically equal
MULTI-DIMENSIONAL ARRAY FUNCTIONS	
ndgrid	Generate arrays for N-D functions and interpolation
permute	Permute array dimensions
ipermute	Inverse permute array dimensions
shiftdim	Shift dimensions
circshift	Shift array circularly
squeeze	Remove singleton dimensions

MATRIX MANIPULATION	
cat	Concatenate arrays
reshape	Change size
diag	Diagonal matrices and diagonals of matrix
blkdiag	Block diagonal concatenation
tril	Extract lower triangular part
triu	Extract upper triangular part
fliplr	Flip matrix in left/right direction
flipud	Flip matrix in up/down direction
flipdim	Flip matrix along specified dimension
rot90	Rotate matrix 90 degrees
:	Regularly spaced vector and index into matrix
find	Find indices of nonzero elements
end	
sub2ind	Last index
ind2sub	Linear index from multiple subscripts
ARRAY UTILITY FUNCTIONS	
compan	Companion matrix
gallery	Higham test matrices.
hadamard	Hadamard matrix.
hankel	Hankel matrix.
hilb	Hilbert matrix.
invhilb	Inverse Hilbert matrix.
magic	Magic square.
pascal	Pascal matrix.
rosser	eigenvalue test matrix.
toeplitz	Toeplitz matrix.
vander	Vandermonde matrix.
wilkinson	Wilkinson's eigenvalue test matrix

SPECIAL VARIABLES AND CONSTANTS	
ans	Most recent answer
eps	Floating point relative accuracy
realmax	Largest positive floating point number
realmi	Smallest positive floating point number
pi	3.1415926535897
i, j	Imaginary unit
inf	Infinity
NaN	Not-a-Number
INTERPOLATION AND GRIDDING	
pchip	1-D interpolation (table lookup).
interp1	Quick 1-D linear interpolation
interp1q	1-D interpolation using FFT method
interpft	2-D interpolation (table lookup).
interp2	3-D interpolation (table lookup).
interp3	N-D interpolation (table lookup).
interpn	Data gridding and surface fitting.
griddata	Data gridding and hyper-surface fitting for 3-dimensional data.
griddata3	Data gridding and hyper-surface fitting (dimension >= 2).
griddatan	1-D interpolation (table lookup).
SPLINE INTERPOLATION	
spline	Cubic spline interpolation.
ppval	Evaluate piecewise polynomial.

GEOMETRIC ANALYSIS	
delaunay	Delaunay triangulation.
delaunay3	3_D Delaunay tessellation.
delaunayn	N_D Delaunay tessellation.
dsearch	Search Delaunay triangulation for nearest point.
dsearchn	Search N_D Delaunay tessellation for nearest point.
tsearch	Closest triangle search.
tsearchn	N_D closest triangle search.
convhull	Convex hull.
convhulln	N_D convex hull.
voronoi	Voronoi diagram.
voronoin	N_D Voronoi diagram.
inpolygon	True for points inside polygonal region.
rectint	Rectangle intersection area.
polyarea	Area of polygon.
POLYNOMIALS	
roots	Find polynomial roots.
poly	Convert roots to polynomial.
polyval	Evaluate polynomial.
polyvalm	Evaluate polynomial with matrix argument.
residue	Partial-fraction expansion (residues).
polyfit	Fit polynomial to data.
polyder	Differentiate polynomial.
polyint	Integrate polynomial analytically.
conv	Multiply polynomials.
deconv	Divide polynomials.

MATRIX ANALYSIS	
norm	Matrix or vector norm.
normest	Estimate the matrix 2 norm.
rank	Matrix rank.
det	Determinant.
trace	Sum of diagonal elements.
null	Null space.
orth	Orthogonalization.
rref	Reduced row echelon form.
subspace	Angle between two subspaces.
LINEAR EQUATIONS	
\ and /	Linear equation solution; use "help slash".
linsolve	Linear equation solution with extra control.
inv	Matrix inverse.
rcond	LAPACK reciprocal condition estimator
cond	Condition number with respect to inversion.
condest	1_norm condition number estimate.
normest1	1_norm estimate.
chol	Cholesky factorization.
cholinc	Incomplete Cholesky factorization.
lu	LU factorization.
luinc	Incomplete LU factorization.
qr	Orthogonal
lsqnonneg	Linear least squares with nonnegativity constraints.
pinv	Pseudoinverse.
lscov	Least squares with known covariance.

EIGENVALUES AND SINGULAR VALUES	
eig	Eigenvalues and eigenvectors.
svd	Singular value decomposition.
gsvd	Generalized singular value decomposition.
eigs	A few eigenvalues.
svds	A few singular vales.
poly	Characteristic polynomial.
polyeig	Polynomial eigenvalue problem.
condeig	Condition number with respect to eigenvalues.
hess	Hessenberg form.
qz	QZ factorization for generalized eigenvalues.
ordqz	Reordering of eigenvalues in QZ factorization.
schur	Schur decomposition.
ordschur	Reordering of eigenvalues in Schur decomposition.
FACTORIZATION UTILITIES	
expm	Matrix exponential.
logm	Matrix logarithm.
sqrtm	Matrix square root.
funm	Evaluate general matrix function.
qrdelete	Delete a column or row from QR factorization.
qrinsert	Insert a column or row into QR factorization.
rsf2csf	Real block diagonal form to complex diagonal form.
cdf2rdf	Complex diagonal form to real block diagonal form.
balance	Diagonal scaling to improve eigenvalue accuracy.
planerot	Givens plane rotation.
cholupdate	rank 1 update to Cholesky factorization.
qrupdate	rank 1 update to QR factorization.