

## Chapter 6

# Operators and Flow Control

### 6.1. Relational and Logical Operators

MATLAB has a logical data type, with the possible values 1, representing true, and 0, representing false. Logicals are produced by relational and logical operators/functions and by the functions `true` and `false`:

```
>> a = true
a =
    1

>> b = false
b =
    0

>> c = 1
c =
    1

>> whos
  Name      Size      Bytes  Class
-----
  a         1x1         1  logical array
  b         1x1         1  logical array
  c         1x1         8  double array
```

Grand total is 3 elements using 10 bytes

As this example shows, logicals occupy one byte, rather than the eight bytes needed by a double.

MATLAB's relational operators are

```
== equal
~= not equal
< less than
> greater than
<= less than or equal
>= greater than or equal
```

Note that a single = denotes assignment and never a test for equality in MATLAB.

Comparisons between scalars produce logical 1 if the relation is true and logical 0 if it is false. Comparisons are also defined between matrices of the same dimension

and between a matrix and a scalar, the result being a matrix of logicals in both cases. For matrix–matrix comparisons corresponding pairs of elements are compared, while for matrix–scalar comparisons the scalar is compared with each matrix element. For example:

```
>> A = [1 2; 3 4]; B = 2*ones(2);
```

```
>> A == B
ans =
     0     1
     0     0
```

```
>> A > 2
ans =
     0     0
     1     1
```

To test whether arrays *A* and *B* are equal, that is, of the same size with identical elements, the expression `isequal(A,B)` can be used:

```
>> isequal(A,B)
ans =
     0
```

The function `isequal` is one of many useful logical functions whose names begin with `is`, a selection of which is listed in Table 6.1; for a full list type `doc is`. For example, `isinf(A)` returns a logical array of the same size as *A* containing true where the elements of *A* are plus or minus `inf` and false where they are not:

```
>> A = [1 inf; -inf NaN];
>> isinf(A)
ans =
     0         1
     1         0
```

The function `isnan` is particularly important because the test `x == NaN` always produces the result 0 (false), even if *x* is a NaN! (A NaN is defined to compare as unequal and unordered with everything.)

Note that an array can be real in the mathematical sense, but not real as reported by `isreal`. For `isreal(A)` is true if *A* has *no* imaginary part. Mathematically, *A* is real if every component has *zero* imaginary part. How a mathematically real *A* is formed can determine whether it has an imaginary part or not in MATLAB. The distinction can be seen as follows:

```
>> a = 1;
>> b = complex(1,0);
>> c = 1 + 0i;

>> [a b c]
ans =
     1     1     1
```

Table 6.1. *Selected logical is\* functions.*

<code>ischar</code>	Test for char array (string)
<code>isempty</code>	Test for empty array
<code>isequal</code>	Test if arrays are equal
<code>isequalwithequalnans</code>	Test if arrays are equal, treating NaNs as equal
<code>isfinite</code>	Detect finite array elements
<code>isfloat</code>	Test for floating point array (single or double)
<code>isinf</code>	Detect infinite array elements
<code>isinteger</code>	Test for integer array
<code>islogical</code>	Test for logical array
<code>isnan</code>	Detect NaN array elements
<code>isnumeric</code>	Test for numeric array (integer or floating point)
<code>isreal</code>	Test for real array
<code>isscalar</code>	Test for scalar array
<code>issorted</code>	Test for sorted vector
<code>isvector</code>	Test for vector array

```
>> whos a b c
  Name      Size      Bytes  Class

  a         1x1         8  double array
  b         1x1        16  double array (complex)
  c         1x1         8  double array
```

Grand total is 3 elements using 32 bytes

```
>> [isreal(a), isreal(b), isreal(c)]
ans =
     1     0     1
```

MATLAB's logical operators are

```
&  logical and
&& logical and (for scalars) with short-circuiting
|  logical or
|| logical or (for scalars) with short-circuiting
~  logical not
xor logical exclusive or
all true if all elements of vector are nonzero
any true if any element of vector is nonzero
```

Like the relational operators, the `&`, `|`, and `~` operators produce matrices of logical 0s and 1s when one of the arguments is a matrix. When applied to a vector, the `all` function returns 1 if all the elements of the vector are nonzero and 0 otherwise. The `any` function is defined in the same way, with “any” replacing “all”. Examples:

```
>> x = [-1 1 1]; y = [1 2 -3];
```

```

>> x>0 & y>0
ans =
     0     1     0

>> x>0 | y>0
ans =
     1     1     1

>> xor(x>0,y>0)
ans =
     1     0     1

>> any(x>0)
ans =
     1

>> all(x>0)
ans =
     0

```

Note that `xor` must be called as a function: `xor(a,b)`. The `and`, `or`, and `not` operators and the relational operators can also be called in functional form as `and(a,b)`, `...`, `eq(a,b)`, `...` (see `help ops`).

The operators `&&` and `||` are special in two ways. First, they work with scalar expressions only, and should be used in preference to `&` and `|` for scalar expressions. Continuing the previous example, compare

```

>> any(x>0) && any(y>0)
ans =
     1

>> x>0 && y>0
??? Operands to the || and && operators must be convertible to
    logical scalar values.

```

The second feature of these “double barreled” operators is that they short-circuit the evaluation of the logical expressions, where possible. In the compound expression `expr1 && expr2`, if `expr1` evaluates to false then `expr2` is not evaluated. Similarly, in `expr1 || expr2`, if `expr1` evaluates to true then `expr2` is not evaluated. Short-circuiting saves computation, but it also enables warnings and errors to be avoided. For example, a statement beginning

```
if x > 0 && sin(1/x) < 0.5
```

avoids a division by zero.

The precedence of arithmetic, relational, and logical operators is summarized in Table 6.2 (which is based on the information provided by `help precedence`). For operators of equal precedence MATLAB evaluates from left to right. Precedence can be overridden by using parentheses. Note, in particular, that `and` has higher precedence than `or`, so a logical expression of the form

```
x | y & z
```

is equivalent to

Table 6.2. *Operator precedence.*

Precedence level	Operator
1 (highest)	Parentheses ( )
2	Transpose (.'), power (.^), complex conjugate transpose ('), matrix power (^)
3	Unary plus (+), unary minus (-), logical negation (~)
4	Multiplication (.*), right division (./), left division (.\), matrix multiplication (*), matrix right division (/), matrix left division (\)
5	Addition (+), subtraction (-)
6	Colon operator (:)
7	Less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), equal to (==), not equal to (~=)
8	Logical and (&)
9	Logical or ( )
10	Logical short-circuit and (&&)
11 (lowest)	Logical short-circuit or (  )

```
x | (y & z)
```

It is good practice to insert parentheses to make the intention completely clear.

For matrices, `all` returns a row vector containing the result of `all` applied to each column. Therefore `all(all(A==B))` is another way of testing equality of the matrices `A` and `B`. The `any` function works in the corresponding way. Thus, for example, `any(any(A==B))` has the value 1 if `A` and `B` have any equal elements and 0 otherwise. Alternatives are `all(A(:)==B(:))` and `any(A(:)==B(:))`.

The `find` command returns the indices corresponding to the nonzero elements of a vector. For example,

```
>> x = [-3 1 0 -inf 0];
>> f = find(x)
f =
     1     2     4
```

The result of `find` can then be used to extract just those elements of the vector:

```
>> x(f)
ans =
    -3     1   -Inf
```

With `x` as above, we can use `find` to obtain the finite elements of `x`,

```
>> x(find(isfinite(x)))
ans =
    -3     1     0     0
```

and to replace negative components of `x` by zero:

```
>> x(find(x < 0)) = 0
x =
    0     1     0     0     0
```

When `find` is applied to a matrix `A`, the index vector corresponds to `A` regarded as a vector of the columns stacked one on top of the other (that is, `A(:)`), and this vector can be used to index into `A`. In the following example we use `find` to set to zero those elements of `A` that are less than the corresponding elements of `B`:

```
>> A = [4 2 16; 12 4 3], B = [12 3 1; 10 -1 7]
A =
     4     2    16
    12     4     3
B =
    12     3     1
    10    -1     7

>> f = find(A<B)
f =
     1
     3
     6

>> A(f) = 0
A =
     0     0    16
    12     4     0
```

An alternative usage of `find` for matrices is `[i,j] = find(A)`, which returns vectors `i` and `j` containing the row and column indices of the nonzero elements.

The results of MATLAB's logical operators and logical functions are logical arrays of 0s and 1s. Logical arrays can also be created by applying the function `logical` to a numeric array; nonzero values other than 1 that are converted to 1 result in a warning message. Logical arrays and numeric arrays can both be used for subscripting, but with an important difference: logical arrays pick out elements where the subscript is true, whereas numeric arrays pick out elements indexed by the subscript. An example should make this distinction clear.

```
>> clear
>> y = [1 2 0 -3 0]
y =
     1     2     0    -3     0

>> i1 = (y ~= 0)
i1 =
     1     1     0     1     0

>> i2 = [1 1 0 1 0]
i2 =
     1     1     0     1     0
```

```

>> y(i1)
ans =
     1     2    -3

>> y(i2)
??? Subscript indices must either be real positive integers or
    logicals.

>> whos i1 i2
    Name      Size      Bytes  Class

    i1        1x5         5  logical array
    i2        1x5        40  double array

>> isequal(i1,i2)
ans =
     1

>> i3 = [1 2 4]; y(i3)
ans =
     1     2    -3

```

Although the numeric array `i2` has the same elements as the logical array `i1` (and compares as equal with it), only `i1` can be used for subscripting. To achieve the required subscripting effect with a numerical array, `i3` must be used.

A call to `find` can sometimes be avoided when its argument is a logical array. In our example on p. 67, `x(find(isfinite(x)))` can be replaced by `x(isfinite(x))`.

Addition and multiplication can be done on logicals, and they can be used in arithmetic expressions containing doubles. The result is always a double:

```

>> a = true; b = false; c = 2*a+b, class(c)
c =
     2
ans =
double

```

However, many other arithmetic operations fail:

```

>> b/a
??? Function 'mrdivide' is not defined for values of class 'logical'.

Error in ==> mrdivide at 16
    builtin('mrdivide', varargin{:});

```

## 6.2. Flow Control

MATLAB has four flow control structures: the `if` statement, the `for` loop, the `while` loop, and the `switch` statement. The simplest form of the `if` statement is

```

if expression
    statements
end

```

where the statements are executed if the elements of *expression* are all nonzero. For example, the following code swaps *x* and *y* if *x* is greater than *y*:

```
if x > y
    temp = y;
    y = x;
    x = temp;
end
```

When an *if* statement is followed on its line by further statements, a comma is needed to separate the *if* from the next statement:

```
if x > 0, x = sqrt(x); end
```

Statements to be executed only if *expression* is false can be placed after *else*, as in the example

```
e = exp(1);
if 2^e > e^2
    disp('2^e is bigger')
else
    disp('e^2 is bigger')
end
```

Finally, one or more further tests can be added with *elseif* (note that there must be no space between *else* and *if*):

```
if isnan(x)
    disp('Not a Number')
elseif isinf(x)
    disp('Plus or minus infinity')
else
    disp('A ''regular'' floating point number')
end
```

In the third *disp*, '' prints as a single quote '.

The *for* loop is one of the most useful MATLAB constructs although, as discussed in Section 20.1, experienced programmers who are concerned with producing compact and fast code try to avoid *for* loops wherever possible. The syntax is

```
for variable = expression
    statements
end
```

Usually, *expression* is a vector of the form *i:s:j* (see Section 5.2). The statements are executed with *variable* equal to each element of *expression* in turn. For example, the sum of the first 25 terms of the harmonic series  $1/i$  is computed by

```
>> s = 0;
>> for i = 1:25, s = s + 1/i; end, s
s =
    3.8160
```

Another way to define *expression* is using the square bracket notation:

```
>> for x = [pi/6 pi/4 pi/3], disp([x, sin(x)]), end
    0.5236    0.5000
    0.7854    0.7071
    1.0472    0.8660
```

Multiple `for` loops can of course be nested, in which case indentation helps to improve the readability. The following code forms the 5-by-5 symmetric matrix `A` with  $(i, j)$  element  $i/j$  for  $j \geq i$ :

```
n = 5; A = eye(n);
for j = 2:n
    for i = 1:j-1
        A(i,j) = i/j;
        A(j,i) = i/j;
    end
end
```

The *expression* in the `for` loop can be a matrix, in which case *variable* is assigned the columns of *expression* from first to last. For example, to set `x` to each of the unit vectors in turn, we can write `for x=eye(n), ..., end`.

The `while` loop has the form

```
while expression
    statements
end
```

The *statements* are executed as long as *expression* is true. The following example approximates the smallest nonzero floating point number:

```
>> x = 1; while x>0, xmin = x; x = x/2; end, xmin
xmin =
    4.9407e-324
```

A `while` loop can be terminated with the `break` statement, which passes control to the first statement after the corresponding `end`. An infinite loop can be constructed using `while 1, ..., end`, which is useful when it is not convenient to put the exit test at the top of the loop. (Note that, unlike some other languages, MATLAB does not have a “repeat-until” loop.) We can rewrite the previous example less concisely as

```
x = 1;
while 1
    xmin = x;
    x = x/2;
    if x == 0, break, end
end
xmin
```

The `break` statement can also be used to exit a `for` loop. In a nested loop a `break` exits to the loop at the next higher level.

The `continue` statement causes execution of a `for` or `while` loop to pass immediately to the next iteration of the loop, skipping the remaining statements in the loop. As a trivial example,

```

for i=1:10
    if i < 5, continue, end
    disp(i)
end

```

displays the integers 5 to 10. In more complicated loops the `continue` statement can be useful to avoid long-bodied `if` statements.

The final control structure is the `switch` statement. It consists of “`switch expression`” followed by a list of “`case expression statements`”, optionally ending with “`otherwise statements`” and followed by `end`. The switch expression is evaluated and the statements following the first matching `case` expression are executed. If none of the cases produces a match then the statements following `otherwise` are executed. The next example evaluates the  $p$ -norm of a vector  $x$  (i.e., `norm(x,p)`) for just three values of  $p$ :

```

switch p
    case 1
        y = sum(abs(x));
    case 2
        y = sqrt(x'*x);
    case inf
        y = max(abs(x));
    otherwise
        error('p must be 1, 2 or inf.')
end

```

(The `error` function is described in Section 14.1.) The expression following `case` can be a list of values enclosed in parentheses (a cell array—see Section 18.3). The switch expression then matches any value in the list:

```

x = input('Enter a real number: ');
switch x
    case {inf,-inf}
        disp('Plus or minus infinity')
    case 0
        disp('Zero')
    otherwise
        disp('Nonzero and finite')
end

```

C programmers should note that MATLAB’s `switch` construct behaves differently from that in C: once a MATLAB `case` group expression has been matched and its statements executed, control is passed to the first statement after the `switch`, with no need for `break` statements.

*Kirk: "Well, Spock, here we are.  
Thanks to your restored memory, a little bit of good luck,  
we're walking the streets of San Francisco,  
looking for a couple of humpback whales.  
How do you propose to solve this minor problem?"  
Spock: "Simple logic will suffice."  
— Star Trek IV: The Voyage Home (Stardate 8390)*

*Things equally high on the pecking order get evaluated from left to right.  
When in doubt, throw in some parentheses and be sure.  
Only use good quality parentheses with nice round sides.  
— ROGER EMANUEL KAUFMAN, A FORTRAN Coloring Book (1978)*

Copyright ©2005 by the Society for Industrial and Applied Mathematics

This electronic version is for personal use and may not be duplicated or distributed.

From "MATLAB Guide, Second Edition" by Desmond J. Higham and Nicholas J. Higham.

Buy this book from SIAM at [www.ec-securehost.com/SIAM/ot92.html](http://www.ec-securehost.com/SIAM/ot92.html)