
Matlab Tutorial and Reference

December 9, 2009

© Barry G Adams

Contents

1	Quick Tour of the Matlab Environment	2
1.1	Command window	2
1.2	Current directory window	3
1.3	Workspace window	3
1.4	Command history window	3
1.5	Configuring the windows	3
2	Using Matlab as a Super Calculator	3
2.1	Scalar variables and arithmetic expressions	3
2.1.1	Standard scalar arithmetic operations	4
2.1.2	Variables	5
2.1.3	Built-in mathematical functions	6
2.2	Anonymous functions	8
2.3	Vector operations and expressions	9
2.3.1	Row and column vectors	9
2.3.2	Arithmetic operations with vectors	12
2.4	Matrix and array operations and expressions	14
2.4.1	Matrix operations from linear algebra	16
2.4.2	Array (element by element) operations	18
2.5	Vectorization	19
2.6	Plotting tables and functions	21
2.6.1	Using fplot	21
2.6.2	Plotting tables of values	22
2.6.3	Plotting two functions on one graph	23
3	Interactive Scripts and Function M-files	24
3.1	Interactive scripts	24
3.1.1	Script for solving a quadratic equation	24
3.1.2	An algorithm for square roots	26
3.2	Function M-files	29
3.2.1	Factorial example	29
3.2.2	Example: A falling object	30
3.2.3	Local variables	32
4	Some Applications	33
4.1	Root finding example for a falling object	33
5	Command and Function Reference	34
5.1	Commands	34
5.2	Elementary Math Functions	34
5.2.1	Trigonometric functions	34
5.2.2	Exponential Functions	35

5.2.3	Complex Functions	36
5.2.4	Rounding and Remainder Funtions	36
5.3	Arithmetic Operators	36
5.4	Relational Operators	36
5.5	Logical Operators	37
5.6	Special variables and constants	37
5.7	Elementary matrices and matrix manipulation	37
5.7.1	Elementary matrices	37
5.7.2	Basic array information	38
5.7.3	Matrix manipulation	38
5.7.4	Multi-dimensional array functions	38
5.7.5	Array utility functions	38
5.7.6	Specialized matrices	39
5.8	Operators and special characters	39
5.8.1	Arithmetic operators	39
5.8.2	Relational operators	39
5.8.3	Logical operators	40
5.8.4	Special characters	40

1 QUICK TOUR OF THE MATLAB ENVIRONMENT

Before starting MATLAB for the first time make yourself a directory (folder) that will contain your MATLAB scripts. When you have done this run MATLAB.

1.1 Command window

Commands are typed here after the `>>` prompt. To make your command window look like the one in this tutorial enter the two commands (press enter after each one)

```
>> format compact
>> format short
```

Enter the command

```
>> pi + 1
ans =
    4.1416
>>
```

Here `ans` is a variable whose value is the last quantity calculated. If you terminate a command or statement with a semi-colon the command output is not shown. Try this:

```
>> pi + 1;
>>
```

1.2 Current directory window

Later we will write scripts that are stored in files and can be run from the command window. Such scripts are assumed to be in the current directory.

To make the directory you created the current directory on the MATLAB toolbar there is box that you can use to navigate to the directory you have made to hold your scripts. This can also be done using the toolbar at the top of the current directory window itself.

1.3 Workspace window

Variables you have defined are shown in this window. Their values can be displayed and/or edited in this window. You should see your `ans` variable. To edit a variable double click on it.

1.4 Command history window

This window contains a list of commands that have been entered. One of these commands can be executed in the command window by double clicking on it. To copy a command from the history window to the command window without executing it simply select the command and drag it to the command window.

1.5 Configuring the windows

The windows can be configured by choosing various entries from the Desktop menu. Try some choices to see what happens and then return to the default option.

2 USING MATLAB AS A SUPER CALCULATOR

Before writing scripts we show how to use MATLAB as a super calculator.

2.1 Scalar variables and arithmetic expressions

Unless stated otherwise in this tutorial we assume the following statements have been executed

```
format short
format compact
```

The first displays only 4 digits after the decimal point and the second does not leave blank lines in the output.

If you want to leave blank lines and show the maximum digits in a floating point number you can use the commands

```
format long
format loose
```

2.1.1 Standard scalar arithmetic operations

The usual arithmetic operators $+$, $-$, $*$ and $/$ are available. Enter the following statements and note that division is a floating point division not an integer division.

```
>> 1 + 2*3 - 5.23
ans =
    1.7700
>> (2.5 - 3.2)*(6.2 + 3.27) / 4
ans =
   -1.6573
>> 2/3
ans =
    0.6667
>>
```

Scientific notation can also be used:

```
>> 1.5e-15 + 2.3e-12
ans =
   2.3015e-012
>>
```

The exponentiation operator is the caret (\wedge) but it associates from left to right (unconventional):

```
>> 2^3^4
ans =
    4096
>> (2^3)^4
ans =
    4096
>> -4 ^ 2
ans =
   -16
>>
```

The last result shows that exponentiation takes precedence over negation. A complete list of operations can be obtained using either of the help commands

```
>> help ops
>> helpwin ops
```

2.1.2 Variables

Variable names begin with a letter which can be followed by 0 or more letters or digits. The underscore character is considered to be a letter. Each variable can hold a value of any type. For example try the following statements.

```
>> x = 'hello'
x =
hello
>> x = 3
x =
    3
>> y = 4
y =
    4
>> z = x^2 + y^2
z =
    25
>>
```

Note how variable `x` first holds a string value and then holds a numeric value.

If you look at the workspace window you will see that `x`, `y` and `z` appear as workspace variables. To remove the variable `x` from the workspace use the command

```
>> clear x
```

To see the value of any variable, say `z`, simply type its name and press enter:

```
>> z
z =
    5
>>
```

To remove all variables use the command

```
>> clear all
```

If you try to access a variable that was removed you get an error message:

```
>> x
??? Undefined function or variable 'x'.
```

2.1.3 Built-in mathematical functions

MATLAB has a large collection of elementary mathematical functions including trigonometric, inverse trigonometric, hyperbolic, inverse hyperbolic, logarithmic and exponential functions. The complete list can be obtained with either of the help commands

```
>> help elfun
>> helpwin elfun
```

You can also use the help browser from the main Help menu. Try these examples noting that `log`, not `ln`, is the logarithm to base e .

```
>> x = 3;
>> y = 4;
>> z = sqrt(x^2 + y^2)
z =
     5
>> sin(pi/2)^2 + cos(pi/2)^2
ans =
     1
>> log(1.12)
ans =
     0.1133
>> exp(1.0)
ans =
     2.7183
>>
```

Here `pi` is the value of π and there is also `e` for the value of e^1 . Angles for trigonometric functions must be in radians and the inverse trigonometric functions return an angle in radians. Sometimes it is more convenient to use angles in degrees. MATLAB provides degree versions `sind(x)` and similarly for the other 5 trigonometric functions. For the inverse functions the value returned from `asind(x)` is in degrees. Try the following statements.

```
>> sind(90)
ans =
     1
>> asind(1)
ans =
     90
>>
```

Example Evaluate the expression

$$v = \sqrt{\frac{gL}{2\pi} \tanh\left(\frac{2\pi d}{L}\right)}$$

where v m/s is the velocity of a water wave, $d = 50$ m is the depth of water without the wave, $L = 100$ m is the wave length of the wave, and $g = 9.81$ m/s² is the acceleration due to gravity.

```
>> g = 9.81;
>> L = 100;
>> d = 50;
>> v = sqrt( (g*L)/(2*pi)*tanh(2*pi*d/L) )
v =
    12.4719
>>
```

Example Evaluate the expression

$$h = \frac{v^2 \sin 2\theta}{2g}$$

where h m is the maximum height of a projectile moving in a vertical plane at an angle $\theta = 50$ degrees from the horizontal and having speed $v = 20$ m/s, and $g = 9.81$ m/s² is the acceleration due to gravity.

```
>> g = 9.81;
>> v = 20;
>> theta = 50*pi/180;
>> h = v^2*sin(2*theta)/(2*g)
h =
    20.0776
>>
```

To convert from degrees to radians we have multiplied 50 by $\pi/180$. As mentioned earlier MATLAB has the usual trig functions `sin`, `cos`, and `tan` for angles in radians and `sind`, `cosd`, and `tand` for angles in degrees. Here is the previous calculation using `sind`

```
>> g = 9.81;
>> v = 20;
>> theta = 50;
>> h = v^2*sind(2*theta)/(2*g)
h =
    20.0776
>>
```

2.2 Anonymous functions

The format of an anonymous function of one variable is

$$f\text{-name} = @(var\text{-name}) \textit{expression}$$

where *f-name* is the name of the function, *var-name* is the name of the independent variable and *expression* is the expression defining the function in terms of the independent variable. The function name is also called a **function handle**.

Example The formula $49(t + 5e^{-t/5}) - 245$ is an expression that defines a function of t . In MATLAB this function can be expressed as the anonymous function of t .

```
>> f = @(t) 49*(t + 5*exp(-t/5)) - 245;
```

This function can be evaluated at $t = 1.5$ and $t = 11$ as follows

```
>> f(1.5)
ans =
    10.0005
>> f(11)
ans =
   321.1468
```

It is also possible to define anonymous functions of more than one variable. For example, the formula

$$v = \sqrt{\frac{gL}{2\pi} \tanh\left(\frac{2\pi d}{L}\right)}$$

can be considered as an anonymous function of the three variables g , d , and L as follows.

```
>> v = @(g,d,L) sqrt((g*L)/(2*pi)*tanh(2*pi*d/L));
>> v(9.81,50,100)
ans =
    12.4719
>>
```

It is also possible to have anonymous functions inside other anonymous functions. For example, if $f(x) = x^2$, $g(x) = 3x$, the composition is $h(x) = (f \circ g)(x) = f(g(x))$. This can be expressed in MATLAB as

```
>> f = @(x) x^2;
>> g = @(x) 3*x;
>> h = @(x) f(g(x));
```

2.3 Vector operations and expressions

So far our variables have been scalars. The fundamental structure in MATLAB is the $m \times n$ matrix (m rows and n columns). A vector is a special case since it is either a $1 \times n$ matrix (row vector) or an $m \times 1$ matrix (column vector). In fact a scalar is just a 1×1 matrix. The power of MATLAB comes from its ability to work easily with vectors and matrices.

2.3.1 Row and column vectors

A row vector $x = [x_1, x_2, \dots, x_n]$ can be entered in two ways, with or without commas:

```
>> x = [6 5 4]
x =
     6     5     4
>> x = [6,5,4]
x =
     6     5     4
```

Each element has an index and indices always begin at 1. Thus x has 3 elements. Parentheses are used to access an individual element using its index. Thus, the three elements of x can be obtained as $x(1)$, $x(2)$, and $x(3)$. The following statement accesses the second element of the vector x .

```
>> x(2)
ans =
     5
```

The following statements show that the elements of a vector can also be changed.

```
>> x = [6 5 4]
x =
     6     5     4
>> x(2) = 99
x =
     6    99     4
```

Similarly column vectors can be expressed in two ways. The first way is to use a semi-colon to terminate each row and the second way is to press the enter key at the end of each row. Try the following statements.

```
>> x = [6;5;4]
x =
     6
     5
     4
```

```
>> x = [  
6  
5  
4  
]  
x =  
    6  
    5  
    4  
>>
```

Length of a vector The number of elements in a row or column vector is given by the `length` function. Try the following statements.

```
>> x = [6,5,4]  
x =  
    6    5    4  
>> length(x)  
ans =  
    3
```

Constructing zero vectors The `zeros` function can be used to construct vectors and matrices whose elements are initialized to zero: The first argument of the `zeros` function gives the number of rows and the second argument gives the number of columns.

```
>> x = zeros(1,4)  
x =  
    0    0    0    0  
>> y = zeros(4,1)  
y =  
    0  
    0  
    0  
    0  
>>
```

Arithmetic sequence vectors These important sequences can be constructed using the colon operator. The expression `a:b` gives a row vector whose elements go from `a` up to `b` inclusive in steps of 1, the expression `a:c:b` creates a vector whose elements go from `a` up to `b` inclusive in steps of `c` if `c` is positive, or from `a` down to `b` in steps of `|c|` if `c` is negative. Try the following examples.

```
>> x = 1:4
```

```
x =
    1     2     3     4
>> y = 1:2:10
y =
    1     3     5     7     9
>> z = 10:-1:2
z =
   10     9     8     7     6     5     4     3     2
>>
```

The linspace function This function can be used to divide an interval into equally spaced subintervals and put the results into a row vector. For example `linspace(0,1,11)` divides the interval 0 to 1 using 11 points. The width of each subinterval is 0.1. If the third argument is not present then 100 points are used. Try the following statements (some output is omitted to save space).

```
>> x1 = linspace(0,1,11)
x1 =
Columns 1 through 6
    0    0.1000    0.2000    0.3000    0.4000    0.5000
Columns 7 through 11
    0.6000    0.7000    0.8000    0.9000    1.0000
>> x2 = linspace(0,1)
x2 =
Columns 1 through 6
    0    0.0101    0.0202    0.0303    0.0404    0.0505
Columns 7 through 12
    0.0606    0.0707    0.0808    0.0909    0.1010    0.1111
Columns 13 through 18
    0.1212    0.1313    0.1414    0.1515    0.1616    0.1717
.
.
.

Columns 91 through 96
    0.9091    0.9192    0.9293    0.9394    0.9495    0.9596
Columns 97 through 100
    0.9697    0.9798    0.9899    1.0000
>>
```

Subvectors The colon operator can be used to make a subvector. Try the following statements.

```
>> x = [5,4,3,2]
```

```
x =  
    5    4    3    2  
>> y = x(2:3)  
y =  
    4    3  
>> y(1)  
ans =  
    4  
>>
```

Here `2:3` is used to make a vector `y` from the second and third elements of the vector `x`. The special value `end` refers to the last element of a vector. Try the following statements.

```
>> x = [5 6 7 3 4 8 9]  
x =  
    5    6    7    3    4    8    9  
>> y = x(4:end)  
y =  
    3    4    8    9  
>>
```

Transpose of a vector A row vector can be converted to a corresponding column vector and vice versa using the transpose operation which is the single quote character. Try the following statements:

```
>> x = [6, 5, 4]  
x =  
    6    5    4  
>> y = x'  
y =  
    6  
    5  
    4  
>>
```

2.3.2 Arithmetic operations with vectors

There are two kinds of arithmetic operations with vectors: the ones from linear algebra and the element by element operations that are not used in linear algebra.

Vector operations from linear algebra In linear algebra vectors of the same length and type (row or column) can be added and subtracted and multiplied by a scalar. This is easily done in MATLAB. We illustrate the operations using row vectors but the same rules also apply to column vectors. Try the following statements.

```
>> x = [6,5,3];
>> y = [2,4,3];
>> x + y
ans =
     8     9     6
>> x - y
ans =
     4     1     0
>> 2*x + 4*y
ans =
    20    26    18
>>
```

Adding scalars and vectors In linear algebra it is not possible to add or subtract a scalar and a vector but this is possible and useful in MATLAB. Such operations are called **array** operations or **element-by-element** operations as opposed to matrix operations. For example $3 + x$ is obtained by adding 3 to each element of x , $x - 1$ is obtained by subtracting 1 from each element of x , and $2 - x$ is obtained by subtracting each element of x from 2. Try the following statements.

```
>> x = [6,5,4];
>> 3 + x
ans =
     9     8     7
>> x - 1
ans =
     5     4     3
>> 2 - x
ans =
    -4    -3    -2
>>
```

Element by element exponentiation Exponentiation of each element in a vector can be done using an element-by-element version of the exponentiation operator $^$, denoted by $.^$, and pronounced “dot hat”. Try the following statements.

```
>> x = [6,5,4]
x =
     6     5     4
>> x^2
??? Error using ==> mpower
Matrix must be square.
```

```
>> x.^2
ans =
    36    25    16
>>
```

This shows why the operator \wedge cannot be used. It is a matrix operation so $x \wedge 2$ is trying to multiply x by itself and this operation is only defined for square matrices ($n \times n$) matrices. Therefore we need a new operator to do element by element exponentiation (don't put a space between $.$ and \wedge).

Example The square of the length of a vector $x = [x_1, x_2, \dots, x_n]$ is given by $x_1^2 + x_2^2 + \dots + x_n^2$. This can be done in MATLAB two ways. Try the following

```
>> x = [6, 5, 4]
x =
     6     5     4
>> x * x'
ans =
     77
>> sum(x.^2)
ans =
     77
>>
```

Here we multiply x by its transpose x' . This is a matrix multiplication of a 1×3 column vector and a 3×1 row vector to give a 1×1 scalar to get the sum of squares, This can also be done using the `sum` function which adds together all the elements of a vector.

2.4 Matrix and array operations and expressions

An $m \times n$ matrix can be expressed in MATLAB by using spaces or commas to separate elements in a row and a semi-colon to end each row. Try the following statements which construct a 2×3 matrix A, a 3×3 square matrix B, and another 2×3 matrix C.

```
>> A = [1, 2, 9; 3, 6, 8]
A =
     1     2     9
     3     6     8
>> B = [6, 9, 0; 6, 4, 6; 1, 8, 4]
B =
     6     9     0
     6     4     6
     1     8     4
>> C = [6, 7, 2; 9, 2, 3]
```

```
C =
     6     7     2
     9     2     3
>>
```

The size function The `size` function can be used to determine the number of rows and columns in a matrix. This function returns the row and column dimensions as a two element row vector. Try the following statements.

```
>> A = [1,2,9; 3,6,8];
>> size(A)
ans =
     2     3
>> [rows,cols] = size(A)
rows =
     2
cols =
     3
>>
```

Subscript notation To reference the element of A in row *i* and column *j* the notation $A(i, j)$ is used. Try the following statements.

```
>> A = [1,2,9; 3,6,8];
>> A(1,1)
ans =
     1
>> A(2,3)
ans =
     8
>>
```

Submatrices and the colon operator The colon operator can be used to refer to individual rows and columns and submatrices. Try the following statements.

```
>> A = [1,2,9; 3,6,8];
>> B = [6,9,0; 6,4,6; 1,8,4];
>> row2 = A(2,:)
row2 =
     3     6     8
>> col2 = A(:,2)
col2 =
```

```
    2
    6
>> B2 = B(1:2,1:2)
B2 =
    6    9
    6    4
>>
```

Here $A(2, :)$ refers to all columns of row 2, $A(:, 2)$ refers to all rows of column 2, and $B(1:2, 1:2)$ refers to the first two rows and columns of B which is just the top left 2×2 matrix.

2.4.1 Matrix operations from linear algebra

In linear algebra you learn how to multiply a matrix by a scalar, add or subtract two matrices having the same size, multiply two compatible matrices, and solve linear systems of equations. These operations are available in MATLAB.

Scalar multiplication To multiply a matrix by 3 you multiply each element of the matrix by 3. Try the statement

```
>> A = [1, 2, 9; 3, 6, 8];
>> 3*A
ans =
    3    6   27
    9   18   24
>>
```

Adding and subtracting two matrices Try the following statements.

```
>> A = [1, 2, 9; 3, 6, 8];
>> C = [6, 7, 2; 9, 2, 3];
>> A - C
ans =
   -5   -5    7
   -6    4    5
>> -A + C
ans =
    5    5   -7
    6   -4   -5
>> 3*A - 5*C
ans =
  -27  -29   17
  -36    8    9
>>
```

Matrix multiplication Two matrices A and B can be multiplied to give the product AB only if the number of columns of A is the same as the number of rows of B . In MATLAB the matrix product is given by $A*B$. Try the following statements using the matrices defined above.

```
>> A = [1,2,9; 3,6,8];
>> B = [6,9,0; 6,4,6; 1,8,4];
>> A*B
ans =
    27    89    48
    62   115    68
>> B^2
ans =
    90    90    54
    66   118    48
    58    73    64
>> B*A
??? Error using ==> mtimes
Inner matrix dimensions must agree.
>>
```

Here $B*A$ does not exist since B has 3 columns and A has only 2 rows.

Solving a linear system An $n \times n$ system of linear equations has the form $\mathbf{Ax} = \mathbf{b}$ where \mathbf{A} is the $n \times n$ coefficient matrix, \mathbf{b} is the right hand side column vector and \mathbf{x} is the column vector for the solution.

In MATLAB the backslash operator is used to solve the linear system. For example try the following system.

```
>> A = [5,1,1; 1,4,1; 1,1,3]
A =
     5     1     1
     1     4     1
     1     1     3
>> b = [5;4;3] % column vector
b =
     5
     4
     3
>> x = A\b
x =
    0.7600
    0.6800
    0.5200
```

2.4.2 Array (element by element) operations

The element-by-element, or array, operations are not used in linear algebra but are very useful in MATLAB.

Element-by-element multiplication This multiplication is denoted by `.*` and is defined for matrices having the same size. It is quite different than matrix multiplication in linear algebra. If $P = A .* C$ is the element-by-element product of A and C then the matrix element $P(i, j)$ is $A(i, j) * C(i, j)$.

```
>> A = [1,2,9; 3,6,8];
>> C = [6,7,2; 9,2,3];
>> A .* C
ans =
     6    14    18
    27    12    24
>>
```

Element-by-element division Similarly there is an element-by-element divide operation which is denoted by `./` for matrices having the same size. If $P = A ./ C$ is the element-by-element quotient of A and C then the matrix element $P(i, j)$ is $A(i, j) / C(i, j)$. Try the following statements.

```
>> A = [1,2,9; 3,6,8];
>> C = [6,7,2; 9,2,3];
>> A ./ C
ans =
    0.1667    0.2857    4.5000
    0.3333    3.0000    2.6667
>>
```

Element-by-element exponentiation Similarly there is an element-by-element exponentiation operation which is defined by `.^` for matrices having the same size. If $P = A .^ C$ is A to the element-by-element power C then the matrix element $P(i, j)$ is $A(i, j) ^ C(i, j)$. Try the following statements.

```
>> A = [1,2,9; 3,6,8];
>> C = [6,7,2; 9,2,3];
>> A .^ C
ans =
         1        128         81
    19683         36        512
>>
```

2.5 Vectorization

Instead of writing loops to process vectors and matrices one element at a time it is often possible to work directly with the vectors and matrices, avoiding loops altogether. This is called **vectorization**.

Example As a simple example suppose we have the five element vector $d = [0, 30, 45, 60, 90]$ of angles in degrees and we want to convert it to a vector r with angles in radians by multiplying each element of d by $\pi/180$. Then we want to convert r to a vector y whose elements are the sines of these angles. The components of the vectors r and y are given for $k = 1, 2, \dots, 5$ by

$$r_k = \pi d_k / 180,$$
$$y_k = \sin(r_k)$$

Then we would need to use a loop to compute y_1, y_2, \dots, y_5 . Try the following statements that compute the vector r and y using a `for` loop over the values k from 1 to 5:

```
>> d = [0, 30, 45, 60, 90]
d =
    0    30    45    60    90
>> for k = 1:5
    r(k) = d(k)*pi/180;
    y(k) = sin(r(k));
end
>> r
r =
    0    0.5236    0.7854    1.0472    1.5708
>> y
y =
    0    0.5000    0.7071    0.8660    1.0000
```

The vectorized way to do this has no loops:

```
>> d = [0, 30, 45, 60, 90]
d =
    0    30    45    60    90
>> r = d*pi/180
r =
    0    0.5236    0.7854    1.0472    1.5708
>> y = sin(r)
y =
    0    0.5000    0.7071    0.8660    1.0000
```

Here the factor `pi/180` automatically multiplies every element of the vector d to give the vector r . Then the `sin` function is designed to work on vectors in the same way by taking the sine of each element and producing a vector of sines.

If we don't need the vector r we can obtain y in one statement

```
>> y = sin(d*pi/180)
y =
    0    0.5000    0.7071    0.8660    1.0000
>>
```

Example: Displaying a table of values With vectorization it is easy to produce a table of values of a function. Suppose we want to compute a table of the sine function on the interval 0 to 1 in steps of 0.1. First we use the colon operator to construct the vector x of points. Then we apply the sine function to this vector to get a vector y of function values. Then we construct a two column table (matrix) by transposing the row vectors x and y to column vectors x' and y' and concatenating them together using $[x', y']$. Try the following statements to see the results

```
>> x = 0:0.1:1;
>> y = sin(x);
>> [x', y']
ans =
    0    0
    0.1000    0.0998
    0.2000    0.1987
    0.3000    0.2955
    0.4000    0.3894
    0.5000    0.4794
    0.6000    0.5646
    0.7000    0.6442
    0.8000    0.7174
    0.9000    0.7833
    1.0000    0.8415
>>
```

Example: Vectorized anonymous function Try the following statement which defines the function:

$$y = \frac{1}{(x-0.2)^2+0.01} + \frac{1}{(x-0.8)^2+0.04} - 6$$

```
>> y = @(x) 1 ./ ((x-0.2).^2 + 0.01) + 1 ./ ((x-0.8).^2 + 0.04) - 6
```

Note how vectorization is used in this function. Let's trace what happens. First 0.2 is subtracted from each element of the vector x . Then each element of the resulting vector is squared, using $.^2$. Then 0.01 is added to each element of the resulting vector. Then $./$ is applied to 1 and this vector. Call the result $x1$.

Then for the second term 0.8 is subtracted from each element of the vector x . Then each element of the resulting vector is squared, using $.^2$. Then 0.04 is added to each element of the resulting vector. Then $./$ is applied to 1 and this vector. Call the result $x2$.

Finally x_1 and x_2 are added and 6 is subtracted from each element of the resulting vector to get the final vector.

2.6 Plotting tables and functions

MATLAB has two functions for plotting, `plot` to plot a table of values, and `fplot` to plot a given function. The `plot` function is more general since it can plot a table or a table obtained from a function, whereas `fplot` requires an explicit function.

2.6.1 Using `fplot`

The basic syntax for `fplot` is

```
fplot(func, [xmin, xmax])
fplot(func, [xmin, xmax, ymin, ymax])
```

where `func` is a function handle (reference to a function). In the first form `xmin` and `xmax` are the horizontal limits for the graph. In the second form the vertical limits are also specified

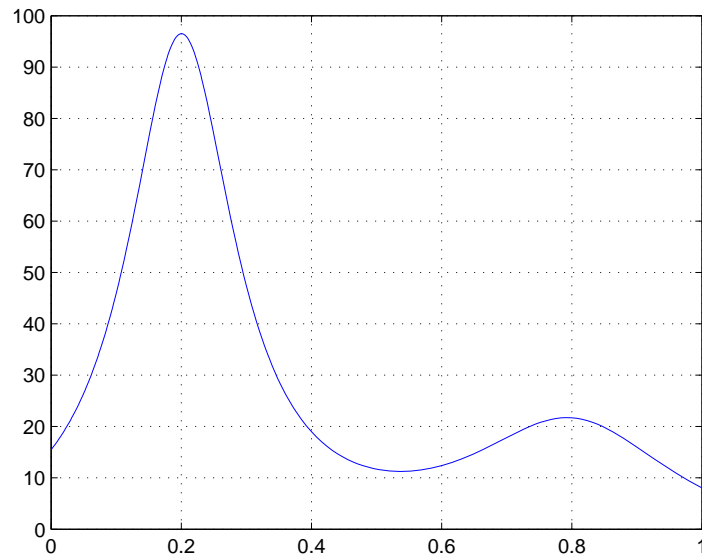
Example: plotting trig functions To plot one of the built-in functions you need to prefix the function name with the `@` character. This creates a function handle. Try the following plots (**WARNING:** plot windows have a nasty habit of hiding behind other windows, so if you don't see your plot window, its probably hiding).

```
>> fplot(@sin, [0, 2*pi])
>> fplot(@cos, [0, 2*pi])
>> fplot(@tan, [0, 2*pi, -5, 5])
```

Since the `tan` function has vertical asymptotes it is necessary to specify vertical limits.

Example: plotting an anonymous function

```
>> y = @(x) 1 ./ ((x-0.2).^2 + 0.01) + 1 ./ ((x-0.8).^2 + 0.04) - 6
>> fplot(y, [0 1])
>> grid on % displays a grid over the graph
```



Note that in the call to the `fplot` function it is not necessary to use the ampersand character since the right hand side of the definition of `y` already has the ampersand. In other words `f` is already a function handle. On the other hand, when using built-in functions such as `sin` it is necessary to use `@sin` to get a function handle.

2.6.2 Plotting tables of values

The `plot` function can be used to plot a table of values or a table of values constructed from a function. The basic syntax is

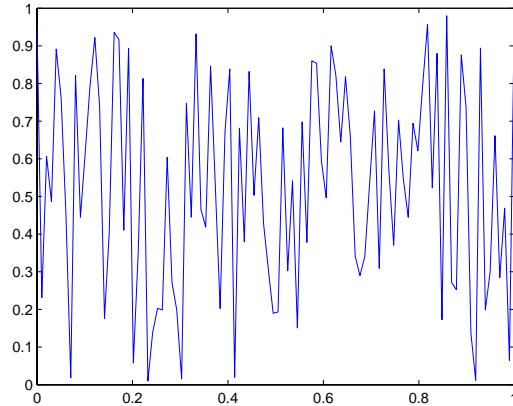
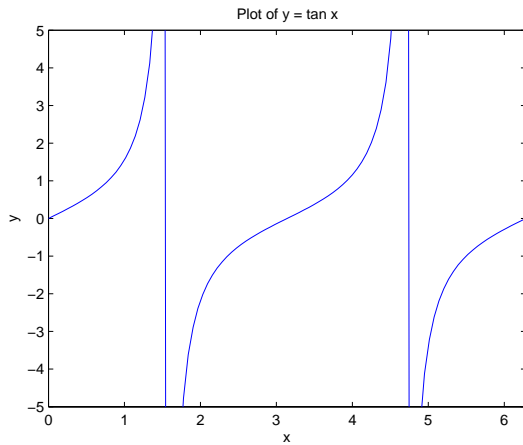
```
plot(x,y)
```

where `x` is the vector of points to use for the x-axis and `y` is the vector of points for the y-axis. The points in the plane are given by (x_k, y_k) for $k = 1, \dots, n$. The `plot` function joins the points (x_1, y_1) , (x_2, y_2) , \dots , (x_n, y_n) by straight line segments. It follows that the more points you use the smoother the graph. Try the following statements, and be sure to terminate the first two statements with a semi-colon.

```
>> x = linspace(0,2*pi,100);  
>> y = tan(x);  
>> plot(x,y)  
>> axis([0,2*pi,-5,5])  
>> xlabel('x')  
>> ylabel('y')  
>> title('Plot of y = tan x')
```

We can also use `plot` for a table of values not obtained from a function. Try the following statements which generate random values in the range 0 to 1.

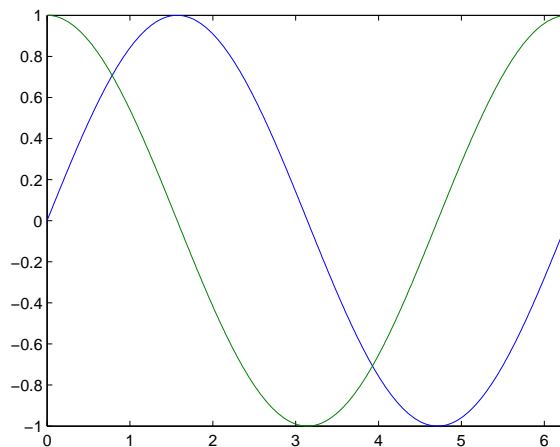
```
>> x = linspace(0,1,100);  
>> y = rand(1,100);  
>> plot(x,y)
```



2.6.3 Plotting two functions on one graph

To plot $y = \sin x$ and $y = \cos x$ on the same graph try the statements

```
x = linspace(0,2*pi,1000);  
y1 = sin(x);  
y2 = cos(x);  
plot(x,y1,'-', x,y2,'-')  
axis([0,2*pi,-1,1])
```



Here each graph will be drawn with a solid line as indicated by the hyphens. They will have different colors. To plot $\cos x$ using a dashed line replace the second hyphen by two hyphens.

3 INTERACTIVE SCRIPTS AND FUNCTION M-FILES

There are two kinds of M-files (files with extension `m`) in MATLAB that can be used to save MATLAB programs.

- **Script M-Files:** They contain MATLAB statements that can be executed by typing the script name in the command window (without the file extension). MATLAB executes the statements as though they were entered one by one in the command window.
- **Function M-files:** They contain MATLAB functions that are executed in the same way as the built-in functions and function M-files.

It is important to change the current directory, if necessary, to the one containing your M-files. MATLAB will look there for scripts we are going to write.

3.1 Interactive scripts

We want to create an M-file for a script that calculates the roots of a quadratic equation. To begin make sure your current directory is set to the one you created at the beginning of the tutorial.

3.1.1 Script for solving a quadratic equation

Use an editor and choose a name for the script MATLAB has an editor for creating M-files. From the MATLAB “File” menu, select “New” and choose “M-file”. This brings up a blank editor window. To set the file name choose “Save As” from the editor “File” menu and save the file in your directory with the name `quad_script.m`.

Enter the script Now enter the following script (if you hate typing and you are using the online version of this PDF file you can cut and paste it from the PDF file to the MATLAB editor).

```
% Finding the roots of a quadratic equation
% ax^2 + bx + c = 0

a = input('Enter coeff of x^2 ');
b = input('Enter coeff of x ');
c = input('Enter constant coeff ');

d = b*b - 4*a*c;
```

```
if d > 0
    root1 = (-b + sqrt(d)) / (2*a);
    root2 = (-b - sqrt(d)) / (2*a);
    disp('Real unequal roots');
    disp(root1);
    disp(root2);
elseif d < 0
    root_real = -b / (2*a);
    root_imag = sqrt(abs(d));
    disp('Complex roots');
    disp(complex(root_real, root_imag));
    disp(complex(root_real, -root_imag));
else
    root = -b / (2*a);
    disp('Real equal roots');
    disp(root);
end
```

Explanation of the script Here is a brief explanation of the script.

- Comments begin with the % character.
- Make sure you terminate lines by semi-colons if you don't want to see their output.
- The `input` command is used to prompt for and enter 3 floating point numbers. To enter a string use a statement such as `name = input('Enter your name ', 's');`.
- The `disp` function is used to display an array or a string, a string in our case. The display occurs whether or not the statement is terminated by a semi-colon.
- MATLAB can do arithmetic on complex numbers. The `complex` function takes two arguments, for the real and imaginary parts of the complex number and constructs a complex number from them. Then `disp` displays this complex number.

Although not illustrated in this example script, if you want to break a long line over several lines it is necessary to use `...` at the end of a line to be continued. For example, the two lines defining `root1` and `root2` could be expressed as

```
root1 = ...
    (-b + sqrt(d)) / (2*a);
root2 = ...
    (-b - sqrt(d)) / (2*a);
```

Running the script To run the script enter its name, without the file extension, in the command window (if you get a “not found” error you have saved the file in the wrong directory or the current directory has not been set to the directory containing your file). Try the following

```
>> quad_script
Enter coeff of x^2  1
Enter coeff of x    1
Enter constant coeff 1
Complex roots
  -0.5000 + 1.7321i
  -0.5000 - 1.7321i
>> quad_script
Enter coeff of x^2  1
Enter coeff of x    2
Enter constant coeff 1
Real equal roots
  -1
>> quad_script
Enter coeff of x^2  1
Enter coeff of x    2
Enter constant coeff -3
Real unequal roots
  1
  -3
```

3.1.2 An algorithm for square roots

To illustrate the for loop lets write a script called `sq_root1.m` that uses a simple iterative algorithm for finding the square root of a number: To find \sqrt{a} let $x_0 = a/2$ and perform the iterations defined by

$$x_k = \frac{1}{2} \left(x_{k-1} + \frac{a}{x_{k-1}} \right), k = 1, 2, \dots$$

Here is the script that inputs a and the maximum number of iterations and displays the results.

```
% Simple algorithm for square roots

a = input('Number to find square root of: ');
max_iter = input('Maximum iterations: ');

x = a/2;
for k = 1:max_iter
    x = 0.5*(x + a/x);
    disp(x);
end
```

Run it in the command window:

```
>> format long % to display full precision
>> sq_root1
Number to find square root of: 3
Maximum iterations: 5
 1.750000000000000
 1.73214285714286
 1.73205081001473
 1.73205080756888
 1.73205080756888
>> x*x % Check the result
ans =
 3.000000000000000
>> format short % return to short display
```

Note that we have squared the result as a check. The `for` loop uses the colon operator to define the range of the loop index. A more general range could be specified by the arithmetic sequence `a:c:b` which goes from `a` up to `b` in steps of `c` if `c > 0` and from `a` down to `b` in steps of `|c|` if `c < 0`.

Using a while loop The following while loop version is equivalent. Try it and save it in the file `sq_root2.m`:

```
% Simple algorithm for square roots
% This version uses a while loop

a = input('Number to find square root of: ');
max_iter = input('Maximum iterations: ');

x = a/2;
k = 1;
while k <= max_iter
    x = 0.5*(x + a/x);
    disp(x);
    k = k + 1; % don't forget this
end
```

A third version We can write a version of the square root algorithm that terminates if a given tolerance (maximum error) is obtained or if a maximum number of iterations is exceeded. We will stop the algorithm if the relative error of successive iterations is met. This means that we stop iterating if

$$\left| \frac{x_k - x_{k-1}}{x_k} \right| < \text{tol}$$

Try this version and save it in the file [sq_root3.m](#).

```
% Simple algorithm for square roots
% This version uses a for loop with a break

a = input('Number to find square root of: ');
max_iter = input('Maximum iterations: ');
tol = input('Enter tolerance: ');

x_old = a/2;

for k = 1:max_iter

    x_new = 0.5*(x_old + a/x_old);
    disp(x_new);

    if abs(x_new - x_old) < x_new*tol
        break;
    end

    x_old = x_new;
end

if k == max_iter
    fprintf('Failure to converge in %d iterations\n', max_iter)
end
```

Note that the for loop makes sure the loop is not infinite and the break statement causes the loop to exit if the given tolerance is met. When the loop exits we can check whether the break occurred.

This script also illustrates the fprintf statement. See help on sprintf and fprintf. In our case the string is displayed with the format code %d replaced by the value of max_iter. Here is some output in the command window.

```
>> format long % to display full precision
>> sq_root3
Number to find square root of: 1001
Maximum iterations: 5
Enter tolerance: 1e-8
    2.5125000000000000e+002
    1.276170398009950e+002
    67.73040996142100
    41.25479556911950
    32.75932070637978
```

```
Failure to converge in 5 iterations
>> sq_root3
Number to find square root of: 1001
Maximum iterations: 10
Enter tolerance: 1e-8
    2.5125000000000000e+002
    1.276170398009950e+002
    67.73040996142100
    41.25479556911950
    32.75932070637978
    31.65775492926360
    31.63858984374151
    31.63858403911328
    31.63858403911275
>> format short
```

3.2 Function M-files

A function M-file is a file that contains the definition of a function. The simplest syntax for a function is

```
function [output_args] = f_name(input_args)
    statements
end
```

Here `f_name` is the function name, `input_args` is a list of values supplied to the function when it is called and `output_args` is a vector of the values output when the function finishes execution. The function is stored in an M-file called `f_name.m` and the file name must be the same as the function name.

There are other types of functions (e.g., subfunctions, nested functions) but we won't consider them here.

3.2.1 Factorial example

The factorial of a number n is denoted by $n!$ and is defined as $n! = 1 \times 2 \times \dots \times n$ if $n > 0$ and 1 if $n = 0$. Use the MATLAB editor to create the file `fact.m`.

```
function [product] = fact(n)
product = 1;
for k = 2:n
    product = product * k;
end
```

Here the final value of `product` is returned as the output value of the function. Note that the for loop is never executed in case $n < 2$.

Testing the factorial function In the command window the fact function can be executed as follows

```
>> fact(10)
ans =
    3628800
>> fact(12)
ans =
  479001600
>> fact(13)
ans =
  6.2270e+009
```

Note that 12 is the last value of n whose factorial can be computed exactly using the MATLAB double precision arithmetic. The factorial function is a built-in function with name `factorial`.

3.2.2 Example: A falling object

The position and velocity of a falling object with air resistance proportional to velocity are

$$x(t) = x_0 + \frac{mg}{k}t + \frac{m}{k} \left(v_0 - \frac{mg}{k} \right) \left(1 - e^{-kt/m} \right)$$

$$v(t) = \frac{mg}{k} + \left(v_0 - \frac{mg}{k} \right) e^{-kt/m}$$

where g is the acceleration due to gravity, m is the mass, k is the drag coefficient and the positive direction is downward.

If we assume that the initial conditions are $x_0 = 0$ and $v_0 = 0$ corresponding to an object falling from rest then the following function M-File called `falling_object.m`, can calculate the position and velocity

```
function [x,v] = falling_object(g,m,k,t)
% A function to calculate the position and velocity of
% a falling object with air resistance proportional to velocity.
% It is assumed that the initial position is 0 with positive
% direction downward and that the initial velocity is also 0
%
% Input:
% g: the acceleration due to gravity (for example 9.81 m/s^2)
% m: the mass of the object (for example, 10 kg)
% k: the drag coefficient (for example, 2 kg/s)
% t: the time in seconds since object was dropped
% Output:
% x: the position at time t
% v: the velocity at time t
```

```
c1 = m*g/k;
c2 = m/k;

% Initial conditions at time 0

x0 = 0;
v0 = 0;

% Formulas for velocity and position

v = c1 + (v0 - c1)*exp(-t/c2);
x = x0 + c1*t + c2*(v0 - c1)*(1 - exp(-t/c2));
end
```

Some output Here is some output from the command window.

```
>> [x,v] = falling_object(9.81,10,2,10.1)
x =
 282.6888
v =
 42.5432
```

The `falling_object` function is also a vectorized function so the value of `t` can be supplied as a vector. For example try

```
>> t = 10:0.1:12;
>> [x,v] = falling_object(9.81, 10, 2, t)
x =
Columns 1 through 6
278.4410 282.6888 286.9495 291.2231 295.5091 299.8074
Columns 7 through 12
304.1178 308.4399 312.7735 317.1184 321.4745 325.8414
Columns 13 through 18
330.2189 334.6070 339.0052 343.4135 347.8316 352.2594
Columns 19 through 21
356.6966 361.1430 365.5986
v =
Columns 1 through 6
42.4118 42.5432 42.6721 42.7984 42.9222 43.0435
Columns 7 through 12
43.1624 43.2790 43.3933 43.5053 43.6151 43.7227
Columns 13 through 18
```

```
    43.8282    43.9316    44.0330    44.1323    44.2297    44.3251
Columns 19 through 21
    44.4187    44.5104    44.6003
```

To obtain a table of the t , x , and v values try

```
>> [t',x',v']
ans =
    10.0000    278.4410    42.4118
    10.1000    282.6888    42.5432
    10.2000    286.9495    42.6721
    10.3000    291.2231    42.7984
    10.4000    295.5091    42.9222
    10.5000    299.8074    43.0435
    10.6000    304.1178    43.1624
    10.7000    308.4399    43.2790
    10.8000    312.7735    43.3933
    10.9000    317.1184    43.5053
    11.0000    321.4745    43.6151
    11.1000    325.8414    43.7227
    11.2000    330.2189    43.8282
    11.3000    334.6070    43.9316
    11.4000    339.0052    44.0330
    11.5000    343.4135    44.1323
    11.6000    347.8316    44.2297
    11.7000    352.2594    44.3251
    11.8000    356.6966    44.4187
    11.9000    361.1430    44.5104
    12.0000    365.5986    44.6003
>>
```

Note that an initial height of 300 metres is obtained for $t \approx 10.5$ seconds

3.2.3 Local variables

The variables defined in the body of a function are **local variables**. This means that they come into existence when the function is called but disappear when the function finishes execution.

On the other hand the variables defined in a script are available in the workspace after the script finishes execution. You can verify this by looking at the workspace window when you run a script M-file or a function M-file.

4 SOME APPLICATIONS

4.1 Root finding example for a falling object

Drop an object with a mass of 10 kg from the top of a 300 m high building. Assume that air resistance is $-kv$ where v m/s is the velocity, $k = 2$ kg/s is the air resistance and $g = 9.81$ is the acceleration due to gravity.

How long does it take for the object to reach the ground? What will be the velocity at that time?

Assuming that the positive direction is downward it can be shown that the differential equation for the velocity is (from Newton's second law)

$$m \frac{dv}{dt} = mg - kv$$

This equation can be solved to give the velocity v and the position x below the top of the building:

$$\begin{aligned} x(t) &= 49.05(t + 5e^{-t/5}) - 245.25 \\ v(t) &= 49.05(1 - e^{-t/5}) \end{aligned}$$

The object hits the ground when $x(t) = 300$ so we need to find this value of t . Note the $x(t)$ is almost a linear function since $e^{-t/5} \rightarrow 0$ very fast as $t \rightarrow \infty$. If we neglect the exponential term we get the linear function

$$x_{\text{linear}}(t) = 49.05t - 245.25$$

The solution of $x_{\text{linear}}(t) = 300$ is given by $t = (300 + 245.25)/49.05 = 11.1162$. We can use this as an initial guess to find the root of $x(t) = 300$. The following MATLAB statements define x and v as anonymous functions

```
>> xt = @(t) 49.05*(t + 5*exp(-t/5)) - 245.25;
>> vt = @(t) 49.05*(1 - exp(-t/5));
```

The root finding problem is $x(t) - 300 = 0$ so we look for a zero of the function

```
>> xt_root = @(t) xt(t) - 300;
```

We can use the linear approximation $t = 11.1162$ and plot the function `xt_root` on the interval from $t = 10$ to $t = 12$ to verify that there is a root in this interval:

```
>> fplot(xt_root, [10,12])
```

The MATLAB root finding function `fzero` can be used to find the root when $x(t) - 300 = 0$ in the interval from 10 to 12 using

```
>> t_ground = fzero(xt_root, [10,12])
t_ground =
    10.5045
```

Finally the corresponding velocity is

```
>> v_ground = vt(t_ground)
v_ground =
    43.0489
```

5 **COMMAND AND FUNCTION REFERENCE**

Most of the commands and functions listed here and others can be obtained using the help commands

```
helpwin elfun      Help on elementary functions
helpwin ops        Help on operations
helpwin elmat      Help on elementary matrix functions
helpwin strfun     help on string functions
helpwin graph2d    help on 2d graphics functions
```

5.1 **Commands**

```
clc                Clear the command window
clear x            Clear (remove) variable x from workspace
clear all          Clear all variables from workspace
doc                Display documentation
help               Displays help text in the Command window
helpwin           Displays help text in separate window
who                gives a list of current variables
whos               longer version of who
```

5.2 **Elementary Math Functions**

5.2.1 **Trigonometric functions**

```
sin                Sine.
sind               Sine of argument in degrees.
sinh               Hyperbolic sine.
asin               Inverse sine.
asind              Inverse sine, result in degrees.
asinh              Inverse hyperbolic sine.
cos                Cosine.
cosd               Cosine of argument in degrees.
cosh               Hyperbolic cosine.
acos               Inverse cosine.
acod               Inverse cosine, result in degrees.
```

acosh	Inverse hyperbolic cosine.
tan	Tangent.
tand	Tangent of argument in degrees.
tanh	Hyperbolic tangent.
atan	Inverse tangent.
atand	Inverse tangent, result in degrees.
atan2	Four quadrant inverse tangent.
atanh	Inverse hyperbolic tangent.
sec	Secant.
secd	Secant of argument in degrees.
sech	Hyperbolic secant.
asec	Inverse secant.
asecd	Inverse secant, result in degrees.
asech	Inverse hyperbolic secant.
csc	Cosecant.
cscd	Cosecant of argument in degrees.
sch	Hyperbolic cosecant.
acsc	Inverse cosecant.
acscd	Inverse cosecant, result in degrees.
acsch	Inverse hyperbolic cosecant.
cot	Cotangent.
cotd	Cotangent of argument in degrees.
coth	Hyperbolic cotangent.
acot	Inverse cotangent.
acotd	Inverse cotangent, result in degrees.
acoth	Inverse hyperbolic cotangent.

5.2.2 Exponential Functions

exp	Exponential.
expm1	Compute $\exp(x)-1$ accurately.
log	Natural logarithm.
log1p	Compute $\log(1+x)$ accurately.
log10	Common (base 10) logarithm.
log2	Base 2 logarithm and dissect floating point number.
pow2	Base 2 power and scale floating point number.
realpow	Power that will error out on complex result.
reallog	Natural logarithm of real number.
realsqrt	Square root of number greater than or equal to zero.
sqrt	Square root.
nthroot	Real n-th root of real numbers.
nextpow2	Next higher power of 2.

5.2.3 Complex Functions

abs	Absolute value.
angle	Phase angle.
complex	Construct complex data from real and imaginary parts.
conj	Complex conjugate.
imag	Complex imaginary part.
real	Complex real part.
unwrap	Unwrap phase angle.
isreal	True for real array.
cplxpair	Sort numbers into complex conjugate pairs.

5.2.4 Rounding and Remainder Functions

fix(x)	Round towards zero.
floor(x)	Round towards minus infinity.
ceil(x)	Round towards plus infinity.
round(x)	Round towards nearest integer.
mod(x)	Modulus (signed remainder after division).
rem(x)	Remainder after division.
sign(x)	Signum: returns 1 if $x > 0$, 0 if $x = 0$ and -1 if $x < 0$

5.3 Arithmetic Operators

+	Plus (addition)
-	Minus
*	Matrix multiply
^	Matrix power
.*	Array multiply (element-by-element)
.^	Array power (element-by-element)
./	Array divide (element-by-element)

5.4 Relational Operators

==	Equal
~=	Not equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

5.5 Logical Operators

<code>&&</code>	Short circuit logical AND
<code> </code>	Short circuit logical OR
<code>&</code>	Element-wise logical AND
<code> </code>	Element-wise logical OR
<code>~</code>	Logical NOT
<code>xor</code>	Logical EXCLUSIVE OR
<code>any(x)</code>	True if any element of vector <code>x</code> is non zero
<code>all(x)</code>	True if all elements of vector <code>x</code> are non zero

5.6 Special variables and constants

<code>ans</code>	Most recent answer
<code>eps</code>	Floating point relative accuracy
<code>realmax</code>	Largest positive floating point number
<code>realmin</code>	Smallest positive floating point number
<code>pi</code>	3.1415926535897...
<code>i, j</code>	Imaginary unit
<code>inf</code>	Infinity
<code>NaN</code>	Not-a-Number
<code>isnan</code>	True for Not-a-Number
<code>isinf</code>	True for infinite elements
<code>isfinite</code>	True for finite elements
<code>why</code>	Succinct answer

5.7 Elementary matrices and matrix manipulation

5.7.1 Elementary matrices

<code>zeros</code>	Zeros array
<code>ones</code>	Ones array
<code>eye</code>	Identity matrix
<code>repmat</code>	Replicate and tile array
<code>rand</code>	Uniformly distributed random numbers
<code>randn</code>	Normally distributed random numbers
<code>linspace</code>	Linearly spaced vector
<code>logspace</code>	Logarithmically spaced vector
<code>freqspace</code>	Frequency spacing for frequency response
<code>meshgrid</code>	X and Y arrays for 3-D plots
<code>accumarray</code>	Construct an array with accumulation
<code>:</code>	Regularly spaced vector and index into matrix

5.7.2 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 are numerically equal

5.7.3 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	Last index
sub2ind	Linear index from multiple subscripts
ind2sub	Multiple subscripts from linear index

5.7.4 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

5.7.5 Array utility functions

isscalar	True for scalar
isvector	True for vector

5.7.6 Specialized matrices

companion	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	Classic symmetric eigenvalue test problem
toeplitz	Toeplitz matrix
vander	Vandermonde matrix
wilkinson	Wilkinson's eigenvalue test matrix

5.8 Operators and special characters

5.8.1 Arithmetic operators

plus	Plus	+
uplus	Unary plus	+
minus	Minus	-
uminus	Unary minus	-
mtimes	Matrix multiply	*
times	Array multiply	.*
mpower	Matrix power	^
power	Array power	.^
mldivide	Backslash or left matrix divide	\
mrdivide	Slash or right matrix divide	/
ldivide	Left array divide	.\
rdivide	Right array divide	./
kron	Kronecker tensor product	kron

5.8.2 Relational operators

eq	Equal	==
ne	Not equal	~=
lt	Less than	<
gt	Greater than	>
le	Less than or equal	<=
ge	Greater than or equal	>=

5.8.3 Logical operators

	Short-circuit logical AND	&&
	Short-circuit logical OR	
and	Element-wise logical AND	&
or	Element-wise logical OR	
not	Logical NOT	~
xor	Logical EXCLUSIVE OR	
any	True if any element of vector is nonzero	
all	True if all elements of vector are nonzero	

5.8.4 Special characters

colon	Colon	:
paren	Parentheses and subscripting	()
paren	Brackets	[]
paren	Braces and subscripting	{ }
punct	Function handle creation	@
punct	Decimal point	.
punct	Structure field access	.
punct	Parent directory	..
punct	Continuation	...
punct	Separator	,
punct	Semicolon	;
punct	Comment	%
punct	Invoke operating system command	!
punct	Assignment	=
punct	Quote	'
transpose	Transpose	.'
ctranspose	Complex conjugate transpose	'
horzcat	Horizontal concatenation	[,]
vertcat	Vertical concatenation	[;]
subsasgn	Subscripted assignment	(), { }, .
subsref	Subscripted reference	(), { }, .
subsindex	Subscript index	