MATLAB Vocabulary

Gerald Recktenwald*

Version 0.965, 25 February 2017

MATLAB is a software application for scientific computing developed by the Mathworks. MATLAB runs on Windows, Macintosh and Unix operating systems and can be controlled with standard mouse and keyboard inputs. However, to take full advantage of MATLAB's capabilities you must write programs in the MATLAB language.

This document lists some important MATLAB commands and programming constructs organized by the context in which those commands and constructs are used. This document is not an exhaustive guide to MATLAB as a computer language, and neither is it a tutorial on programming. The MATLAB vocabulary guide was created as a reference to users learning to program MATLAB, as well as a cheat sheet for users trying to remember how to perform simple tasks with MATLAB.

Create Vectors and Matrices

There are many commands to create vectors and matrices. One can always use direct, manual entry such as

```
x = [1 3 5]
y = [9; 8; 6; 4]
A = [1 2; 3 4; 5 6]
```

but this only makes sense for small vectors and matrices because of the tedium of typing values and the potential for introducing typos. The following functions return vectors and matrices with some sort of pattern, order or mathematical properties. The following functions also allow vectors and matrices to be created of any size with no need to type out the specific values in those vectors or matrices.

`linspace` `linspace(a,b)` creates a vector of 100 linearly spaced elements between `a` and `b`. `linspace(a,b,n)` creates a vector of `n` linearly spaced elements between `a` and `b`.

`logspace` `logspace(a,b)` creates 50 logarithmically spaced elements between 10^{a} and 10^{b}. `logspace(a,b,n)` creates `n` logarithmically spaced elements between 10^{a} and 10^{b}.

`zeros` `zeros(m,n)` creates an $m \times n$ matrix filled with zeros. Often, a statement like `A = zeros(m,n)` is used to pre-allocated a variable

*gerry@pdx.edu, Mechanical and Materials Engineering Department, Portland State University, Portland, Oregon.

so that values of `A` can be assigned in a loop without having to (inefficiently) resize `A` dynamically in that loop.

`ones` `ones(m,n)` creates an $m \times n$ matrix filled with ones.

`rand` `rand(m,n)` creates an $m \times n$ matrix filled with values obtained from a pseudo-random, uniform distribution on $[0, 1]$, i.e., all values are equally likely.

`randn` `randn(m,n)` creates an $m \times n$ matrix filled with values obtained from a pseudo-random, normal distribution with mean of zero and standard deviation of one. To create a normal distribution with mean of `xbar` and standard deviation of `sd` use `x = xbar + sd*randn(m,n)`.

There are many more matrix and vector-creating commands for specialized applications.

Getting Information about Vectors and Matrices

The following commands provide information about MATLAB variables. These commands behave differently if the argument is a vector or matrix. First, consider the case of input arguments being vectors.

`abs` `abs(x)` returns a variable with the absolute value of all elements in `x`.

`length` `length(x)` returns the number of elements in vector `x`. Use `size(A)`, not `length(A)` to determine the number of elements in a matrix, `A`.

See also `end` as the last element in a vector, and other subscript operations described in the "Subscripts" section on page 4.

`max` `max(x)` returns the "largest" element of `x`. If `x` has only positive elements, then `max(x)` returns the element with the largest magnitude. If `x` has positive and negative values, `max(x)` returns the *most positive* value.

If you want the value in `x` with the largest magnitude, regardless of sign, use `max(abs(x))`.

`min` `min(x)` returns the "smallest" element of `x`. If `x` has only positive elements, then `min(x)` returns the element with the smallest magnitude. If `x` has positive and negative values, `min(x)` returns the *most negative* value.

If you want the value in `x` with the smallest magnitude, regardless of sign, use `min(abs(x))`.

`size` `size(A)` returns the number of rows and columns in `A`. `size(A)` works whether `A` is a vector or matrix.

`size(A,1)` returns the number of rows in `A`.

`size(A,2)` returns the number of columns in `A`.

`sum` `sum(x)` returns the sum of elements in `x`.

If `x` is a vector, `sum(x)` is the scalar equal to the arithmetic sum of all values in `x`.

Operations on matrices that return vectors

The commands in the preceding section the perform some mathematical operations on vectors, e.g., `max`, `min`, `sum`, can also be applied to matrices. The operations are *applied by column* so that the result is a row vector. Consider the following examples.

```
>> A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

>> max(A)
ans =
    16    14    15    13

>> min(A)
ans =
     4     2     3     1

>> sum(A)
ans =
    34    34    34    34
```

Since `min(A)`, `max(A)` and `sum(A)` return vectors, those functions can be applied twice to obtain the minimum, maximum and sum of elements in a matrix.

```
>> max(max(A))
ans =
    16

>> min(min(A))
ans =
     1

>> sum(sum(A))
ans =
   136
```

An alternative approach that is marginally more efficient is to use a trick with colon notation to convert `A` to a column vector as it is passed as input to these functions.

```
>> max(A(:))
ans =
    16

>> min(A(:))
ans =
     1

>> sum(A(:))
ans =
   136
```

Note that using `A(:)` as the argument of a function, or in the right hand side of an expression does not permanently convert `A` to a column vector.

Working with Vectors and Matrices

All numerical variables in MATLAB are matrices. A scalar is a matrix with one row and one column. A vector is a matrix with either one column and m rows, or one row and n columns. Individual elements in a matrix are referenced by subscripts. An entire row or column of a matrix is referenced with the colon as a wildcard.

In this section, MATLAB variables are treated as arrays in typical programming languages. Refer to the separate section on *Linear Algebra* for linear algebra operations on MATLAB variables.

Subscripts

Element `i` of vector `x` is `x(i)`. For a matrix `A`, the element in row `i` and column `j` is `A(i,j)`.

Just looking at a single line of MATLAB code, you cannot tell whether `foo(i)` is subscript `i` of vector `foo`, or whether `foo` is a function and `i` is the input to that function. Therefore, you need to read the lines of code before `foo(i)` appears. Ask yourself, has `foo` been previously defined by appearing on the left side of an equals sign? If yes, then `foo` is a variable and `foo(i)` is an attempt to reference element `i` of the vector or matrix `foo`.

Loop Over Elements of a Vector

A `for` loop is commonly used to modify or access elements in a vector or matrix one element at a time. The following is an example of combining `length(x)` to determine the number of elements in `x`, a `for` loop to operate on each element of `x`, and `x(i)` to refer to element `i` in the body of the loop.

```
x = linspace(0,100);   %  Just an example of how x might be defined

for i = 1:length(x)    %  Loop over all elements of x

   % do something with x(i)

end
```

Loop Over Elements of a Matrix

For a matrix `A`, the `size` command returns the number of rows and columns.

```
A = ...                %  A is defined somehow
[m,n] = size(A);

for i = 1:m    %  Loop over rows in A

  for j = 1:n   %  Loop over columns in A, for row i

   % do something with A(i,j)

end
```

Using `end` to extract the last element

`end` is a shortcut for the last index in a vector, or as the last element along any dimension of a matrix.

```
xlast = x(end)                  %  value of last element in x
y = x(1:2:end)                  %  y consists of every-other element in x
z = 0.5*( x(end-1) + x(end) )   %  average of last two elements in x
```

In this context, `end` is just a convenience. The last three statements could be replaced with

```
n = length(x)
xlast = x(n)                    %  value of last element in x
y = x(1:2:n)                    %  y consists of every-other element in x
z = 0.5*( x(n-1) + x(n) )       %  average of last two elements in x
```

For matrices, `end` can be used to select the last element along any one dimension.

```
>> A = magic(8);
>> s = A(end,end)
s =
     1

>> t = A(end-1, end-2)
t =
    11

>> u = A(end,:)
u =
     8    58    59     5     4    62    63     1
```

To use `end` as the last index, `end` must appear as a subscript. `end` cannot appear alone on the right hand side of an expression.

Do not forget that `end` is a reserved word that appears in several other contexts. Use `end` to close an `if`, `if...else`, or `if...elseif` construct.

```
if x<0
   disp('x is negative')
elseif x>0
   disp('x is positive')
else
   disp('x is EXACTLY zero')
end
```

Use `end` to close a `for` loop or a `while` loop.

```
for i = 1:length(x)
  fprintf('x(%d) = %f\n',x(i))
end
```

Use `end` to close a `for` loop or a `while` loop.

```
for i = 1:length(x)
  fprintf('x(%d) = %f\n',x(i))
end
```

Use `end` to indicate the last line in a function.

```
function s = addtwo(a,b)
s = a + b
end
```

Colon Operator

The colon operator is used in several contexts. Generally, the colon operator is used to create or represent ranges of integers. Sometimes the goal is to create a vector of integers that is reused. Many times the colon operator is used as a subscript in a vector or matrix to select a range of indices.

Colon Notation to Create Stand Alone Vectors

To construct vectors, either a two parameter or three parameter form can be used

```
x = startValue:endValue

x = startValue:increment:endValue
```

Examples:

```
x = 1:10
y = 1:2:10
t = 0:pi/4:2*pi
```

Note that in the three parameter form, the *endValue* is not guaranteed to be an element of the vector, as demonstrated by the following examples

```
y = 1:2:10
v = 1:3:10
```

If a vector of values that includes the endpoints is required, consider using the `linspace` command instead.

Colon Notation for Ranges of Subscripts

The expression `4:7` is equivalent to the vector of integers `[4 5 6 7]`. Expressions with colons can be used to select ranges of elements in a vector or matrix. For example, we can create `y` from the first three elements of `x` with

```
y = x(1:3)
```

or we can create `z` from the last two elements of `x` with

```
n = length(x);
z = x([n-1, n])
```

or

```
n = length(x);
z = x( (n-1):n )
```

or

```
z = x( (end-1):end )
```

We can create `w` from every other element of `x` with

```
w = x(1:2:length(x))
```

or

```
w = x(1:2:end)
```

Colon Operator as a Wildcard

The colon operator can be used as a wildcard to select an entire row or column of a matrix. Thus, if `A` is a matrix, the following expressions use the wildcard

`A(:,1)` is the first column of `A`
`A(:,j)` is column `j` of `A`
`A(:,end)` is the last column of `A`

and

`A(1,:)` is the first row of `A`
`A(i,:)` is row `i` of `A`
`A(end,:)` is the last row of `A`

Colon Converts any Vector or Matrix to a Column Vector

When the colon operator is used as the sole index of a vector or matrix, the result is a column vector.

```
>> x = randperm(5)
x =
     1     4     5     3     2

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

This trick also works on matrices

```
>> B = magic(3)
B =
     8     1     6
     3     5     7
     4     9     2

>> x = B(:)
x =
     8
     3
     4
     1
     5
     9
     6
     7
     2
```

Note that using `x(:)` or `B(:)` as the argument of a function, or in the right hand side of an expression does not permanently convert `x` or `B` to a column vector.

Vectorized Calculations with Array Operators

MATLAB provides `.*`, `./` and `.^` *array operators* that perform element-by-element operations on vectors and matrices. These operators complement the standard `*` and `^` operators that obey the rules of linear algebra.

Examples

```
t = linspace(0,4*pi);
y = exp(-0.4*t) .*  sin(2*t) .* cos(t);

u = linspace(-3,3);
v = 3*u.^2 - 2*u + 5;
```

Language constructs

`for` A `for` loop repeats a block of code a pre-defined number of times. For example, the following code adds up the elements of the `x` vector.

```
s = x(1);
for i = 2:length(x)
   s = s + x(i);
end
```

The end of the loop block is indicated with an `end` statement.

`while` A `while` loop repeats a block of code as long as a test condition is true. For example, the following code samples the uniform random distribution until a value greater than 0.375 is obtained.

```
x = 0.0;
while x < 0.375
   x = rand(1,1);
   fprintf('%8.4f\n',x)
end
```

This is just a compact and convenient example of the `while` construct, not a model of useful code. The extent of the `while` loop code is terminated with an `end` statement.

`if` An `if` construct allows a block of code to be executed only when a condition is met.

```
if x<0
   fprintf('x = %8.4f is negative\n',x)
end
```

The extent of the `if` block is terminated with an `end` statement.

`else` The `else` statement designates an exclusive alternative code block for an `if` construct

```
x = rand(1,1);
if x < 0.4
   fprintf('x = %8.4f is less than 0.4\n',x)
else
   fprintf('x = %8.4f is greater than 0.4\n',x)
end
```

The `if...else` construct implements an either/or choice.

`elseif` The `elseif` statement designates one alternative code block in in an `if` construct. Unlike an `else` block, which is simply an alternative to other possibilities in an `if` construct, the `elseif` block has its own condition.

```
x = rand(1,1);
if x < 0.4
   fprintf('x = %8.4f is less than 0.4\n',x)
elseif x < 0.8
   fprintf('x = %8.4f is greater than or equal to 0.4 AND less than 0.8\n',x)
else
   fprintf('x = %8.4f is greater than or equal to 0.8\n',x)
end
```

`end` `end` is used to terminate a `for` loop, a `while` loop, a `if` block, and a function. Refer to the examples on page 5 for the different uses of `end`.

`:` The colon operator is used to construct vectors. The colon can also be used as a wildcard. Refer to the more extensive discussion beginning on page 6.

`[ ]` Brackets are used to define vectors and matrices

```
x = [ 1 17 -22]
```

`[ ; ]` When brackets are used to define vectors and matrices the semicolon (inside the brackets) is used to delineate rows

```
r = [1; 2; 3]
A = [10 20; 30 40; 50 60]
```

A semicolon at the end of a complete expression is used to suppress output from the statement, a behavior that is unrelated to the use of a semicolon *inside* the bracket.

`[ ; ]'` The transpose of a vector or matrix is obtained by appending a single quote (or apostrophe) to the vector or matrix.

```
x = [1 2 3]'
y = x'
A = [10 20; 30 40; 50 60];
B = A'
v = (1:5)'
s = v'*v
```

`...` Long lines of MATLAB code can be broken into separate lines by including `...` at the end of the line. The `...` sequence is called a *continuation character* or `continuation symbol`

```
y = exp(-alpha*zeta*t)*( cos(omegan*t) + sin(omegan*t) ) ...
    + 25*mass*g*sin(omegad*t);
```

`nargin` `nargin` is the number of input values supplied to a function at the time it was called.

`nargout` `nargout` is the number of output values that are expected to be returned by a function when it is called.

`feval` The `feval` function evaluates a function when that function is given by a text string or function handle. For ME 350, you don't have to write code with this, but `feval` is used by code that expects you to supply a problem-defining function. Examples are `bisect`, `fzero`, `odeEuler`, and `ode45`.

2D Plots

`plot` Create 2D plots on linearly scaled axes

Examples: Plot (x,y) pairs, connect points with lines

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

Plot (x,y) pairs with dots at each point

```
>> plot(x,y,'.')
```

Plot (x,y) pairs connected w/ lines and (v,w) pairs connected with lines

```
>> plot(x,y,v,w)
```

Plot (x,y) pairs with the points connected by dashed black lines; and plot (v,w) pairs as points identified with blue circles

```
>> plot(x,y,'k--',v,w,'bo')
```

`semilogx, semilogy, loglog:`
Create 2D plots on logarithmically scaled axes. With the exception of the axis scaling, and the requirements that variables on a logarithmic axis are positive, these functions are analogous to, and use the same syntax of, the `plot` command.

`xlabel, ylabel:`
Add text labels to the axes of a 2D plot

`legend` Add a legend to a 2D plot with more than one data series.

`xlim, ylim` Independently set the axis limits on a 2D plot

`axis` Set the x and y limits on a 2D plot.

The single command

```
axis([xmin xmax ymin ymax])
```

is equivalent to the two separate statements

```
xlim([xmin xmax])
ylim([ymin ymax])
```

Linear Algebra

All numerical variables in MATLAB are matrices. All mathematical operations with the *, +, - and ^ operators obey the rules of linear algebra. There is no division operator in linear algebra.

Matrix Multiplication

```
A = [1 2; 4 6];
B = [0 1; 1 0];
C = A*B
```

Matrix-Vector Multiplication

```
D = [2 5; 1 4];
x = [1; -1]
y = D*x
```

Vector Inner Product

```
u = [-1; 1]
v = [4; 6]
s = u'*v
```

Operations with Scalars

Multiplication and division by scalars also obeys the rules of linear algebra.

```
x = [1 2 3]
y = 4*x
A = [10 20; 30 40; 50 60]
B = A/10
```

The rigorous interpretation of the + and - operators is relaxed when a scalar is added to or subtracted from a vector or matrix in MATLAB.

```
u = [1 2 3]
v = 4 + u
C = [10 20; 30 40; 50 60]
D = C - 10
```

Linear Algebra Commands

The following commands perform linear algebra calculations.

`\` Solve $n \times n$ linear systems or obtain the least squares solution to a $m \times n$ system when $m > n$.

Use \, not `inv` to solve linear systems of equations.

In almost all situations where the mathematical operation A^{-1} appears, the most efficient and most accurate MATLAB implementation uses \ instead of `inv`. In other words, *avoid using* `inv` *unless you have a very good technical reason why you need the inverse.*

`condest` `condest(A)` computes an estimate of the condition number of matrix `A`. See `help condest` for details.

`inv` `inv(A)` computes the matrix inverse of `A`. There are very, very few situations where using `inv` is necessary. Although you should avoid using `inv` *you should* understand the relationship between A^{-1} and the \ operator.

`norm` `norm(x)` is the L_2 norm of `x`, where `x` can be a vector or matrix. `norm(x,1)` computes the L_1 norm of `x`, where `x` can be a vector or matrix. If `x` is a vector, `norm(x,Inf)` computes the L_∞ norm of `x`. Note that computing the L_2 norm of a *matrix* is computationally expensive. The `normest`, `condest` and `rcond` functions provide complementary information without the computational cost of computing a matrix norm.

`normest` `normest(A)` computes an *estimate* of the L_2 norm of matrix `A`.

`rank` `rank(A)` is the rank of matrix `A`. If `A` is an $n \times n$ matrix, `rank(A)` is the number of linearly independent columns in `A`, which is the same as the number of linearly independent rows in `A`. We very rarely need to compute `rank(A)` during numerical analysis. However, the `rank` function is useful when studying linear algebra.

`rcond` `rcond(A)` computes an estimate of the *inverse* of the condition number of matrix `A`. If you solve a system of equations with `x = A\b` and `rcond(A)` is $\mathcal{O}(10^{-16})$ or smaller (actually, when `rcond(A)<eps`), then MATLAB will give a warning that `A` is "badly conditioned".

Numerical Methods

`bisect` This is a simple, robust, but not very efficient tool for finding the roots of $f(x)$. `bisect` was used to explain how root-finding algorithms work. For everyday root-finding, use `fzero` instead. For simple problems on modern computers, the inefficiency of `bisect` is not a problem.

`fzero` This is an efficient root-finding tool that is superior to `bisect`. Use `fzero` for everyday root-finding.

`brackPlot` `brackPlot` is just a tool, not a method. Use it to quickly plot a function $f(x)$ and look for brackets to the roots (and singularities) of $f(x)$.

`odeEuler` `odeEuler` is a toy code used to introduce the basic procedure for integrating initial value problems specified by ODEs. Use `odeEuler` only for demonstrations. For everyday engineering work, use `ode45` or one of the other built-in programs for solving ODEs with MATLAB.

Know how to create nested a function for use with `odeEuler`.

`ode45` `ode45` is an adaptive-stepsize function for integrating systems of ordinary differential equations. Use `ode45` as a first choice for solving ODEs. Other built-in routines such as `ode113` and `ode15s` may be necessary for difficult, e.g. stiff, ODEs.