Basic Concepts in Matlab

Michael G. Kay

Fitts Dept. of Industrial and Systems Engineering
North Carolina State University
Raleigh, NC 27695-7906, USA
`kay@ncsu.edu`

September 2010

Contents

1. The Matlab Environment

After launching Matlab, a multi-panel window appears containing Command Window, Workspace, Current Directory, and Command History panels, among others. This, along with windows for the Editor/Debugger, Array Editor, Help Browser, etc., that can be invoked as needed, is the Matlab environment.

Matlab has an extensive set of built-in functions as well as additional toolboxes that consist of functions related to more specialized topics like fuzzy logic, neural networks, signal processing, etc. It can be used in two different ways: as a traditional programming environment and as an interactive calculator. In *calculator mode*, the built-in and toolbox functions provide a convenient means of performing one-off calculations and graphical plotting; in *programming mode*, it provides a programming environment (editor, debugger, and profiler) that enables the user to write their own functions and scripts.

Expressions consisting of operators and functions that operate on variables are executed at the command-line prompt `>>` in the Command Window. All of the variables that are created are stored in the workspace and are visible in the Workspace panel.

Information about any function or toolbox is available via the command-line `help` function (or from the `doc` command that provides the same information in the Help Browser):

```
help sum
```

In Matlab, everything that can be done using the GUI interface (e.g., plotting) can also be accomplished using a command-line equivalent. The command-line equivalent is useful because it can be placed into scripts that can be executed automatically.

2. Creating Arrays

The basic data structure in Matlab is the two-dimensional *array*. Array variables can be scalars, vectors, or matrices:

- Scalar $n = 1$ is represented as a 1×1 array
- Vector $\mathbf{a} = [1\ 2\ 3]$ is a 1×3 array
- Matrix $\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$ is a 2×4 array.

Arrays can be created on the command line as follows:

```
n = 1
n =

     1
a = [1 2 3]
a =
     1     2     3
A = [1 2 3 4; 5 6 7 8]
A =
     1     2     3     4
     5     6     7     8
```

In Matlab, the case of a variable matters; e.g., the arrays **a** and **A** are different variables.

To recall the last command executed, press the up-arrow key (↑). To suppress the output from a command, end an expression with a semicolon (`;`):

```
A = [1 2 3 4; 5 6 7 8];
```

An empty array is considered a 0 × 0 matrix:

```
a = []
a =
     []
```

The following operators and functions can be used to automatically create basic structured arrays:

```
a = 1:5
a =
     1     2     3     4     5
a = 1:2:5
a =
     1     3     5
a = 10:-2:1
a =
    10     8     6     4     2
a = ones(1,5)
a =
     1     1     1     1     1
a = ones(5,1)
a =
     1
     1
     1
     1
     1
a = zeros(1,5)
a =
     0     0     0     0     0
```

The `rand` function generates random numbers between 0 and 1. (Each time it is run, it generates different numbers.)

```
a = rand(1,3)
a =
    0.6086    0.2497    0.8154
b = rand(1,3)
b =
    0.2618    0.4760    0.3661
```

A random permutation of the integers 1 to *n* can be generates using the `randperm(n)` function:

```
randperm(5)
ans =
     1     3     4     5     2
```

To create a 3 × 3 identity matrix:

```
eye(3)
ans =
     1     0     0
     0     1     0
     0     0     1
```

The variable **ans** is used as a default name for any expression that is not assigned to a variable.

3. Saving and Loading Variables

The variables currently in the workspace are visible in the Workspace panel, or through the `whos` command:

```
whos
  Name      Size         Bytes  Class
   A        2x4             64  double array
   a        1x3             24  double array
   ans      3x3             72  double array
   b        1x3             24  double array
   n        1x1              8  double array
Grand total is 24 elements using 192
bytes
```

To save arrays **A** and **a** to a file:

```
save myvar A a
```

Data files in Matlab have the extension `.mat`, so the file `myvar.mat` would be saved in the current directory (see Current Directory panel). To remove the variables from the workspace:

```
clear
whos
```

To restore the variables:

```
load myvar
```

Data files can also be saved and loaded using the File menu.

4. Selecting Array Elements

In Matlab, *index arrays* inside of parenthesis are used to select elements of an array. The colon operator is used to select an entire row or column.

```
A
A =
     1     2     3     4
     5     6     7     8
A(:,:)
ans =
     1     2     3     4
     5     6     7     8
A(1,2)
ans =
     2
A(1,:)
ans =
     1     2     3     4
A(:,1)
ans =
     1
     5
A(:,[1 3])
ans =
     1     3
     5     7
```

The vector [1 3] is an index array, where each element corresponds to a column index number of the original matrix **A**.

The keyword **end** can be used to indicate the last row or column:

```
A(:,end)
ans =
     4
     8
A(:,end-1)
ans =
     3
     7
```

The selected portion of the one array can be assigned to a new array:

```
B = A(:,3:end)
B =
     3     4
     7     8
```

To select the elements along the main diagonal:

```
diag(B)
ans =
     3
     8
```

5. Changing and Deleting Array Elements

In calculator mode, the easiest way to change the elements of an array is to double click on the array's name in the Workspace panel to launch the Array Editor, which allows manual editing of the array. (The editor can also be used to create an array by first generating an array of zeros at the command line and using the editor to fill in the nonzero values of the array.)

In programming mode, elements can be changed by selecting a portion of an array as the left-hand-side target of an assignment statement:

```
a = 1:5
a =
     1     2     3     4     5
a(2) = 6
a =
     1     6     3     4     5
a([1 3]) = 0
a =
     0     6     0     4     5
a([1 3]) = [7 8]
a =
     7     6     8     4     5
```

```
A(1,2) = 100
A =
     1   100     3     4
     5     6     7     8
```

To delete selected elements of a vector:

```
a(3) = []
a =
     7     6     4     5
a([1 4]) = []
a =
     6     4
```

A row or column of a matrix can be deleted by assigning it to an empty array:

```
A(:,2) = []
A =
     1     3     4
     5     7     8
```

6. Manipulating Arrays

The following operators and functions can be used to manipulate arrays:

```
A
A =
     1     3     4
     5     7     8
A'                                   (Transpose)
ans =
     1     5
     3     7
     4     8
fliplr(A)                            (Flip left/right)
ans =
     4     3     1
     8     7     5
flipud(A)                            (Flip up/down)
ans =
     5     7     8
     1     3     4
[A B]                                (Concatenate matrices)
ans =
     1     3     4     3     4
     5     7     8     7     8
[A [10 20]'; 30:10:60]
ans =
     1     3     4    10
     5     7     8    20
    30    40    50    60
A(:)                                 (Convert matrix to column vector)
ans =
     1
     5
     3
     7
     4
     8
A = A(:)'                            (Convert matrix to row vector)
A =
     1     5     3     7     4     8
A = reshape(A,2,3)                   (Convert back to matrix)
A =
     1     3     4
     5     7     8
```

7. Multiplication and Addition

Scalars

A scalar can be added to or multiplied with each element of an array; e.g.,

```
A
A =
     1     3     4
     5     7     8
2 + A
ans =
     3     5     6
     7     9    10
```

```
B = 2 * A
B =
     2     6     8
    10    14    16
```

Multiplication

Matrices are multiplied together in two different ways: element-by-element multiplication, indicated by the use of a *dot* (`.`) along with the operator, and matrix multiplication, where the inner dimensions of the matrices must be the same; i.e.,

Element-by-element multiplication: $\mathbf{A}_{m\times n} \,.\!*\, \mathbf{B}_{m\times n} = \mathbf{C}_{m\times n}$

Matrix multiplication: $\mathbf{A}_{m\times n} * \mathbf{B}_{n\times p} = \mathbf{C}_{m\times p}$

```
C = A .* B                       (2×3 .* 2×3 = 2×3)
C =
     2    18    32
    50    98   128
A * B                            (Error: 2×3 * 2×3 ≠ 2×3)
??? Error using ==> *
Inner matrix dimensions must agree.
C = A * B'                       (2×3 * 3×2 = 2×2)
C =
    52   116
   116   276
C = A' * B                       (3×2 * 2×3 = 3×3)
C =
    52    76    88
    76   116   136
    88   136   160
a = 1:3
a =
     1     2     3
a * a'                           (1×3 * 3×1 = 1×1)
ans =
    14
A * a'                           (2×3 * 3×1 = 2×1)
ans =
    19
    43
```

Division (`./`) and power (`.^`) operators can also be preformed element by element.

Addition

Matrices are added together in Matlab element by element; thus, each matrix must be the same size; i.e.,

Addition: $\mathbf{A}_{m\times n} + \mathbf{B}_{m\times n} = \mathbf{C}_{m\times n}$

```
C = A + B
C =
     3     9    12
    15    21    24
```

To add vector **a** to each row of **A**, **a** can be converted into a matrix with the same dimensions as **A**. This can be accomplished in several different ways:

```
ones(2,1) * a                    (Matrix multiplication)
ans =
     1     2     3
     1     2     3
ones(2,1) * a + A
ans =
     2     5     7
     6     9    11
a(ones(2,1),:)                   (Tony's Trick)
ans =
     1     2     3
     1     2     3
repmat(a,2,1)                    (Replicate array)
ans =
     1     2     3
     1     2     3
```

Using `repmat` is the fastest approach when **a** is a scalar, while Tony's Trick is not possible if **a** does not yet exist and is being created in the expression for the first time.

Summation

The elements of a single array can be added together using the `sum` and `cumsum` functions:

```
a = 1:5
a =
     1     2     3     4     5
sum(a)                              (Array summation)
ans =
    15
cumsum(a)                           (Cumulative summation)
ans =
     1     3     6    10    15
```

By default, Matlab is column oriented; thus, for matrices, summation is performed along each column (the 1st dimension) unless summation along each row (the 2nd dimension) is explicitly specified:

```
A
A =
     1     3     4
     5     7     8
sum(A)
ans =
     6    10    12
sum(A,2)                            (Sum along rows)
ans =
     8
    20
sum(A,1)                            (Sum along columns)
ans =
     6    10    12
sum(sum(A))                         (Sum entire matrix)
ans =
    28
```

Forcing column summation: Even if the default column summation is desired, it is useful (in programming mode) to explicitly specify this in case the matrix has only a single row, in which case it would be treated as a vector and sum to a single scalar value (an error):

```
A = A(1,:)                (Convert A to single-row matrix)
A =
     1     3     4
sum(A)                              (Incorrect)
ans =
     8
sum(A, 1)                           (Correct)
ans =
     1     3     4
```

8. Functions and Scripts

Functions and scripts in Matlab are just text files with a `.m` extension. User-defined functions can be used to extend the capabilities of Matlab beyond its basic functions. A user-defined function is treated just like any other function. Scripts are sequences of expressions that are automatically executed instead of being manually executed one at a time at the command-line. A scripts uses variables in the (base) workspace, while each function has its own (local) workspace for its variables that is isolated from other workspaces. Functions communicate only through their input and output arguments. A function is distinguished from a script by placing the keyword **function** as the first term of the first line in the file.

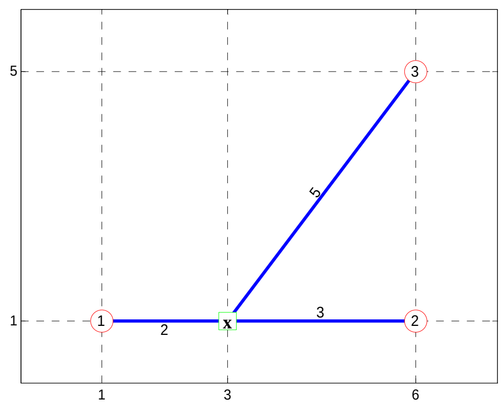
Although developing a set of functions to solve a particular problem is at the heart of using Matlab in the programming mode, the easiest way to create a function is to do it in an incremental, calculator mode by writing each line at the command-line, executing it, and, if it works, copying it to the function's text file.

Example: Given a 2-D point **x** and *m* other 2-D points in **P**, create a function `mydist.m` to determine the Euclidean (i.e., straight-line) distance **d** from **x** to each of the *m* points in **P**:

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}, \quad \mathbf{P} = \begin{bmatrix} p_{1,1} & p_{1,2} \\ \vdots & \vdots \\ p_{m,1} & p_{m,2} \end{bmatrix}$$

$$\mathbf{d} = \begin{bmatrix} \sqrt{(x_1 - p_{1,1})^2 + (x_2 - p_{1,2})^2} \\ \vdots \\ \sqrt{(x_1 - p_{m,1})^2 + (x_2 - p_{m,2})^2} \end{bmatrix}$$

The best way to start is to create some example data for which you know the correct answer:



```
x = [3 1];
P = [1 1; 6 1; 6 5]
P =
     1     1
     6     1
     6     5
```

The first step is to subtract **x** from each row in **P**:

```
ones(3,1) * x
ans =
     3     1
     3     1
     3     1
ones(3,1)*x - P
ans =
     2     0
    -3     0
    -3    -4
```

Square each element of the result:

```
(ones(3,1)*x - P) .^ 2
ans =
     4     0
     9     0
     9    16
```

Add the elements in each row:

```
sum((ones(3,1)*x - P).^2, 2)
ans =
     4
     9
    25
```

Then take the square root and assign the result to **d**:

```
d = sqrt(sum((ones(3,1)*x - P).^2,2))
d =
     2
     3
     5
```

The M-File Editor can be used to create the text file for the function `mydist`. Either select New, M-File from the File menu, or

```
edit mydist
```

where the editor should appear. Type the following two lines (adding a semicolon to the end of the command-line expression to suppress output from inside the function):

```
function d = mydist(x,P)
d = sqrt(sum((ones(3,1)*x - P).^2,2));
```

Save the file, then check to see if Matlab can find the file by using the `type` command to print the function at the command line, and then check to see that it works as desired:

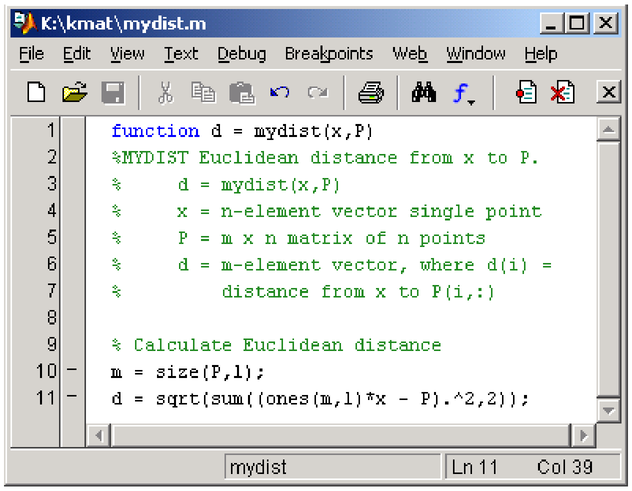
```
type mydist
function d = mydist(x,P)
d = sqrt(sum((ones(3,1)*x - P).^2,2));
d = mydist(x,P)
d =
     2
     3
     5
```

As it is written, the function will work for points of any dimension, not just 2-D points. For *n*-dimensional points, **x** would be a *n*-element vector and **P** a $m \times n$ matrix; e.g., for 4-D points:

```
d = mydist([x x],[P P])
d =
    2.8284
    4.2426
    7.0711
```

The only thing "hard-coded" in the function is *m*. The `size` function can be used inside the function to determine the number of rows (dimension 1) or columns (dimension 2) in **P**:

```
m = size(P,1)
m =
     3
n = size(P,2)
n =
     2
```



```
function d = mydist(x,P)
%MYDIST Euclidean distance from x to P.
%     d = mydist(x,P)
%     x = n-element vector single point
%     P = m x n matrix of n points
%     d = m-element vector, where d(i) =
%         distance from x to P(i,:)

% Calculate Euclidean distance
m = size(P,1);
d = sqrt(sum((ones(m,1)*x - P).^2,2));
```

The last thing that should be added to the function is some help information. All of the first block of contiguous comment lines in a function is returned when the `help` command is executed for the function. Comment lines are indicated by an asterisk (%).

To get help:

```
help mydist
 MYDIST Euclidean distance from x to P.
     d = mydist(x,P)
     x = n-element vector single point
     P = m x n matrix of n points
     d = m-element vector, where d(i) =
         distance from x to P(i,:)
```

The function `mydist` can now be used inside of any expression; e.g.,

```
sumd = sum(mydist(x,P))
sumd =
    10
```

If other people will be using your function, it is a good idea to include some error checking to make sure the values input to the function are correct; e.g., checking to make sure the points in **x** and **P** have the same dimension.

9. Anonymous Functions

Anonymous functions provide a means of creating simple functions without having to create M-files. Given

```
x = [3 1];
P = [1 1; 6 1; 6 5];
```

the sum of the distances from **x** to each of the points in **P** can be determined by creating a *function handle* `sumMydist` to an anonymous function:

```
sumMydist = @() sum(mydist(x,P));
sumMydist
sumMydist =
    @() sum(mydist(x,P))
sumMydist()
ans =
    10
```

The values of **x** and **P** are fixed at the time `sumMydist` is created. To make it possible to use different values for **x**:

```
sumMydist = @(x) sum(mydist(x,P));
sumMydist([6 1])
ans =
     9
sumMydist([4 3])
ans =
    9.2624
```

10. Example: Minimum-Distance Location

Anonymous functions can be used as input arguments to other functions. For example, fminsearch performs general-purpose minimization. Starting from an initial location $\mathbf{x}_0$, it can be used to determine the location **x** that minimizes the sum of distances to each point in **P**:

```
x0 = [0 0];
[x,sumd] = fminsearch(sumMydist,x0)
x =
    5.0895    1.9664
sumd =
    8.6972
```

For this particular location problem, any initial location can be used to find the optimal location because the objective is convex:

```
[x,sumd] = fminsearch(sumMydist,[10 5])
```

```
x =
    5.0895    1.9663
sumd =
    8.6972
```

For many (non-convex) problems, different initial starting values will result in different (locally optimal) solutions.

11. Logical Expressions

A *logical array* of 1 (true) and 0 (false) values is returned as a result of applying logical operators to arrays; e.g.,

```
a = [4 0 -2 7 0]
a =
     4     0    -2     7     0
a > 0                                   (Greater than)
ans =
     1     0     0     1     0
a == 7                                  (Equal to)
ans =
     0     0     0     1     0
a ~= 0                                  (Not equal to)
ans =
     1     0     1     1     0
(a >= 0) & (a <= 4)                     (Logical AND)
ans =
     1     1     0     0     1
(a < 0) | (a > 4)                       (Logical OR)
ans =
     0     0     1     1     0
~((a < 0) | (a > 4))                    (Logical NOT)
ans =
     1     1     0     0     1
```

A logical array can be used just like an index array to select and change the elements of an array; e.g.,

```
a(a > 0)
ans =
     4     7
a(a == 7) = 8
a =
     4     0    -2     8     0
a(a ~= 0) = a(a ~= 0) + 1
a =
     5     0    -1     9     0
```

12. Cell Arrays, Structures, and N-D Arrays

Cell Arrays

A cell array is a generalization of the basic array in Matlab that can have as its elements any type of Matlab data structure. Curly braces, `{ }`, are used to distinguish a cell array from a basic array.

Unlike a regular matrix, a cell array can be used to store rows of different lengths:

```
c = {[10 20 30],[40],[50 60]}
c =
   [1x3 double]    [40]    [1x2 double]
```

The individual elements in the cell array are each a row vector that can be accessed as follows:

```
c{1}
ans =
     10     20     30
```

To access the elements within each row:

```
c{1}(1)
ans =
    10
```

To add an element to the end of the first row:

```
c{1}(end+1) = 35
c =
   [1x4 double]    [40]    [1x2 double]
c{1}
ans =
    10    20    30    35
```

To add another row to the end of the cell array:

```
c(end+1) = {1:2:10}
c =
   [1x4 double]    [40]    [1x2 double]
[1x5 double]
c{end}
ans =
     1     3     5     7     9
```

A common use for cell arrays is to store text strings:

```
s = {'Miami','Detroit','Boston'}
s =
    'Miami'    'Detroit'    'Boston'
```

Some functions can accept a cell array as input:

```
s = sort(s)
s =
    'Boston'    'Detroit'    'Miami'
```

A cell array can be used to store any type of data, including other cell arrays. One use of a cell array is to store all of the input arguments for a function:

```
xP = {x, P}
xP =
    [1x2 double]    [3x2 double]
```

The arguments can then be passed to a function by generating a *comma-separated list* from the cell array:

```
d = mydist(xP{:})
d =
     2
     3
     5
```

Cell arrays of can be created using the `cell` function:

```
c = cell(1,3)
c =
     []      []      []
```

Non-empty values can then be assigned to each element using a FOR Loop (see Section 13 below).

A cell array can be both created and assigned non-empty values by using the `deal` function:

```
[c2{1:3}] = deal(0)
c2 =
    [0]    [0]    [0]
[c3{1:3}] = deal(1,2,3)
c3 =
    [1]    [2]    [3]
```

Structures

Structures are used to group together related information. Each element of a structure is a *field*:

```
s.Name = 'Mike'
s =
    Name: 'Mike'
s.Age = 44
s =
    Name: 'Mike'
     Age: 44
```

Structures can be combines into structure arrays:

```
s(2).Name = 'Bill'
s =
1x2 struct array with fields:
    Name
    Age
s(2).Age = 40;
s(2)
ans =
    Name: 'Bill'
     Age: 40
s.Name
ans =
Mike
ans =
Bill
```

An alternate way to construct a structure array is to use the `struct` function:

```
s =
struct('Name',{'Mike','Bill'},'Age',{44,40})
s =
1x2 struct array with fields:
    Name
    Age
```

When needed, the elements in each field in the array can be assigned to separate arrays:

```
names = {s.Name}
names =
    'Mike'    'Bill'
ages = [s.Age]
ages =
    44    40
```

An alternate way to do the same thing that is sometimes more efficient is to use a single structure instead of an array of structures and use an array for each field of the structure:

```
t.Name = {'Mike','Bill'}
t =
    Name: {'Mike'  'Bill'}
t.Age = [44 40]
t =
    Name: {'Mike'  'Bill'}
     Age: [44 40]
```

N-D Arrays

Multidimensional, or N-D, arrays extend the basic 2-D array used in Matlab. N-D arrays cannot be stored as sparse arrays, so they should only be used for dense arrays. N-D array can be created by extending a 2-D array or by directly creating the array using functions like zeros and ones:

```
D3 = [1 2 3; 4 5 6]
D3 =
     1     2     3
     4     5     6
D3(:,:,2) = [7 8 9; 10 11 12]
D3(:,:,1) =
     1     2     3
     4     5     6
D3(:,:,2) =
     7     8     9
    10    11    12
```

To access individual elements and cross-sections:

```
D3(1,3,2)
ans =
     9
D3(:,1,:)
ans(:,:,1) =
     1
     4
ans(:,:,2) =
     7
    10
```

To convert the 3-D answer to a 2-D array:

```
D2 = squeeze(D3(:,1,:))
D2 =
     1     7
     4    10
```

13. Control Structures

In Matlab, FOR loops iterate over the columns of an array, and logical expressions are used for conditional evaluation in IF statements and WHILE loops.

FOR Loop

```
for i = 1:3
   i
end
i =
     1
i =
     2
i =
     3
for i = 5:-2:1, i, end
i =
     5
i =
     3
i =
     1
```

Any type of array can be used; e.g., a *character array*:

```
chararray = 'abc'

chararray =
abc
for i = chararray
   i
end
i =
a
i =
b
```

```
i =
c
```

Because any type of array can be used in a FOR loop, using FOR loops can greatly slow down the execution speed of Matlab as compared to the equivalent array operation (although FOR loops with only scalar arithmetic inside have been optimized in Release 13 of Matlab so that there is no performance penalty).

Most of the standard arithmetic operations and functions in Matlab cannot be directly applied to an entire cell array; instead, a FOR loop can used to apply the operation or function to each element of the cell array:

```
c = {[10 20 30],[40],[50 60]}
c =
    [1x3 double]    [40]    [1x2 double]
c = c + 1
??? Error using ==> +
Function '+' is not defined for values
of class 'cell'.
for i = 1:length(c), c{i} = c{i} + 1;
end
```

The function `length` is equal to `max(size(c))`. To see the contents of each element in cell array:

```
c{:}
ans =
    11    21    31
ans =
    41
ans =
    51    61
```

IF Statement

```
n = 3;
if n > 0
   disp('Positive value.')
elseif n < 0
   disp('Negative value.')
else
   disp('Zero.')
end
Positive value.
```

WHILE Loop

```
while n > 0
   n = n - 1
end
n =
     2
n =
     1
n =
     0
```

DO-WHILE Loop

```
done = false;
while ~done
   n = n + 1
   if n >= 3, done = true; end
end
n =
     1
n =
     2
n =
     3
```

The DO-WHILE loop is used to ensure that the statements inside the loop are evaluated at least once even if the logical expression is not true.

The logical expression in a conditional statement must evaluate to a single logical scalar value. To convert a logical vector to a scalar, the functions `any` and `all` can be used:

```
a = [5 0 -1 9 0];
a > 0
ans =
     1     0     0     1     0
any(a > 0)
ans =
     1
all(a > 0)
ans =
     0
```

14. Example: Random Walk Simulation

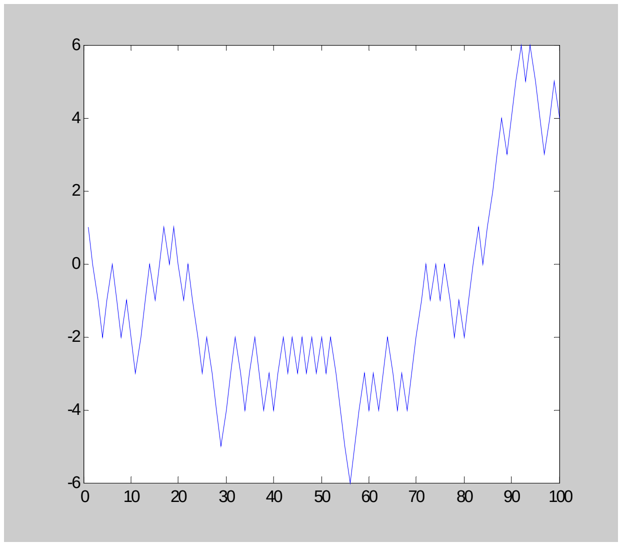
The following example simulates a random walk. Starting from the origin at 0, there is a 50–50 chance that the next step is up one unit of distance (+1) or down one unit of distance (–1). The vector **d** represents the cumulative distance from the origin at each step:

Start by using only 5 steps (and, to get the same random numbers as show below, the *state* of the random number generator can first be set to some arbitrary number like 123).

```
rand('state',123)
s = rand(1,5)
s =
    0.0697    0.2332    0.7374
0.7585    0.6368
s > 0.5
ans =
     0     0     1     1     1
(s > 0.5)*2
ans =
     0     0     2     2     2
((s > 0.5)*2) - 1
ans =
    -1    -1     1     1     1
d = cumsum(((s > 0.5)*2) - 1)
d =
    -1    -2    -1     0     1
```

Now increase to 100 steps (and use semicolon so output is suppressed):

```
s = rand(1,100);
d = cumsum(((s > 0.5)*2) - 1);
plot(d)
```



Multiple runs of the simulation can be used to verify the theoretical estimate of the expected distance from the origin after t steps, where $d(t)$ is the last element of **d**:

$$\mathbf{E}\left[\left|d(t)\right|\right] \to \sqrt{\frac{2t}{\pi}}$$

A FOR-loop can be used iterate through 100 runs of a 1,000 step random walk:

```
for i = 1:100
 d = cumsum(((rand(1,1000)>0.5)*2)-1);
 dt(i) = d(end);
end
mean(abs(dt))                    (Sample mean)
ans =
   24.2200
```

This compares with the theoretical estimate:

$$\sqrt{\frac{2(1{,}000)}{\pi}} = 25.2313$$

15. Logical vs. Index Arrays

Logical arrays and index arrays provide alternative means of selecting and changing the elements of an array. The `find` function converts a logical array into an index array:

```
a
a =
     5     0    -1     9     0
ispos = a > 0                    (Logical array)
ispos =
     1     0     0     1     0
a(ispos)
ans =
     5     9
idxpos = find(a > 0)
idxpos =
     1     4
a(idxpos)
ans =
     5     9
```

(Index array)

Some functions return logical or index arrays:

```
s = {'Miami','Detroit','Boston'};
idxDetroit = strmatch('Detroit',s)
idxDetroit =
     2
isDetroit = strcmpi('detroit',s)
isDetroit =
     0     1     0
```

Although similar, the use of logical and index arrays have different advantages:

Advantages of Logical Arrays

1. *Direct addressing*: It is easy to determine if individual elements of the target array satisfy the logical expression; e.g., a value of 1 (true) for `ispos(4)` directly determines that `a(4)` is positive, while it would be necessary to search through each element of `idxpos` to determine if 4 is an element of the array (i.e., `any(idxpos == 4)`).
2. *Use of logical vs. set operators*: When comparing multiple logical arrays, logical operators like `&` (AND), `|` (OR), and `~` (NOT) can be used instead of the more cumbersome set operator functions like `intersect`, `union`, and `setdiff` that are necessary if index arrays are combined.

Advantages of Index Arrays

1. *Order information*: Unlike logical arrays, the order of the elements in an index array provides useful information; e.g., the index array `idxa` returned by the function `sort` indicates the sorted order of the original unsorted array **a**:

```
a
a =
  5     0    -1     9     0
[sa, idxa] = sort(a)
sa =
    -1     0     0     5     9
idxa =
     3     2     5     1     4
```

2. *Duplicate values*: An index array can have multiple elements with the same index value, allowing arrays to be easily manipulated; e.g.,

```
idx = [1 2 1];
a(idx)
ans =
  5     0     5
```

3. *Space-saving*: If only a few elements of the target array are being considered, then an index array need only store these elements, instead of a logical array that is equal in size to the target array; e.g., the index array idxmina has only a single element:

```
[mina, idxmina] = min(a)
mina =
    -1
idxmina =
  3
```

16. Example: The Monti Carlo Method

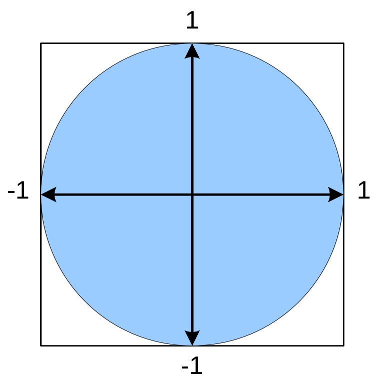
The Monti Carlo method is a general-purpose simulation technique that uses random numbers to estimate the solutions to problems that are too difficult to solve analytically or by using other, more specialized, approximation techniques. It differs from other types of simulation because it is used for static problems where time is not involved.

In this example[*], the value of pi will be estimated by determining the number m out of n points that fall within a unit circle ($r = 1$). The probability that a point (x, y) randomly generated inside a square is also inside the circle is equal to the ratio of area of the circle and the square:

$$P(x^2 + y^2 < 1) = \frac{A_{\text{circle}}}{A_{\text{square}}} = \frac{\pi r^2}{4} = \frac{\pi}{4} \approx \frac{m}{n}$$

[*] Adapted from A. Csete, http://www.daimi.au.dk/~u951581/pi/MonteCarlo/pi.MonteCarlo.html.

Pi can then be estimated as $\pi = \frac{4m}{n}$.



The Monti Carlo method can be implemented in Matlab as follows, starting with a small number of points (n = 3) while the code is being developed and then switching to a larger number to actually estimate pi (to get the same random numbers as show below, the state of the random number generator can first be set to some arbitrary number like 123):

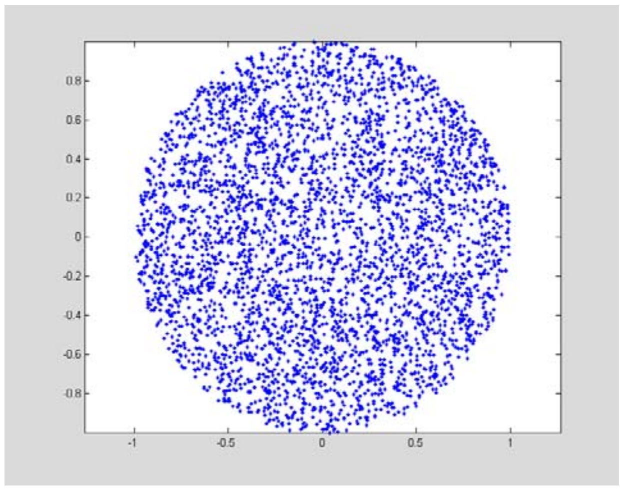
```
rand('state',123)
n = 3;
XY = rand(n,2)
XY =
    0.0697    0.7585
    0.2332    0.6368
    0.7374    0.6129
XY = XY * 2 - 1
XY =
   -0.8607    0.5171
   -0.5335    0.2737
    0.4749    0.2257
isin = sum(XY .^ 2, 2) < 1
isin =
     0
     1
     1
m = sum(isin)
m =
     2
piestimate = 4 * m/n
piestimate =
    2.6667
pi
ans =
    3.1416
```

Now that it is working, n can be increased (along with adding semicolons to the end of statements) and the results plotted:

```
n = 5000;
XY = rand(n,2) * 2 - 1;
isin = sum(XY .^ 2, 2) < 1;
m = sum(isin)
m =
        3967
piestimate = 4 * m/n
piestimate =
    3.1736
pi
ans =
    3.1416
```

To plot results:

```
plot(XY(isin,1),XY(isin,2),'b.')
axis equal
```



The commands used in this example could be copied to the M-File Editor and saved as a script, e.g., `montipi.m`. The script could then be executed by typing `montipi` at the command prompt.

17. Full vs. Sparse Matrices

In many applications, only a small number of the elements of a matrix will have nonzero values. For efficiency, these matrices can be stored as sparse matrices instead of as normal full matrices; e.g.,

```
A = [0 0 0 30; 10 0 20 0]
A =
     0     0     0    30
    10     0    20     0
sparse(A)
ans =
   (2,1)        10
   (2,3)        20
   (1,4)        30
full(A)
ans =
     0     0     0    30
    10     0    20     0
```

18. Inserting and Extracting Array Elements

Inserting Into a New Matrix

The sparse function can be used to create a 2 × 3 matrix **A** with nonzero values $A(1,4) = 30$, $A(2,1) = 10$, and $A(2,3) = 20$:

```
A = sparse([1 2 2], [4 1 3], [30 10
20], 2, 4)
A =
   (2,1)        10
   (2,3)        20
   (1,4)        30
```

Inserting Into an Existing Matrix

If a 2 × 3 matrix **B** exists, new values can be inserted into **B** by first creating a sparse matrix with the same size as **B** with nonzero values corresponding to the elements to be inserted in **B**; e.g., to insert the nonzero values of **A** into **B**:

```
B = [1:4; 5:8]
B =
     1     2     3     4
     5     6     7     8
B(A ~= 0) = A(A ~= 0)
B =
     1     2     3    30
    10     6    20     8
```

Extracting From a Matrix

The (1,4), (2,1), and (2,3) elements of **B** can be extracted to the vector **b** as follows:

```
B([1 2 2],[4 1 3])
ans =
    30     1     3
     8    10    20
     8    10    20
b = diag(B([1 2 2],[4 1 3]))
b =
    30
    10
    20
```

19. List of Functions Used

The following basic Matlab functions were used in this paper:

`deal`	`length`	`sort`
`diag`	`load`	`sparse`
`disp`	`mean`	`sqrt`
`doc`	`min`	`squeeze`
`eye`	`ones`	`strcmpi`
`find`	`rand`	`strmatch`
`fliplr`	`randperm`	`struct`
`flipud`	`repmat`	`sum`
`fminsearch`	`reshape`	`type`
`full`	`save`	`whos`
`help`	`size`	`zeros`