


MATLAB Course
For Engineers
lecture 3


Matrix Methods for Linear Equations

Matrix notation enables us to represent multiple equations as a single matrix equation. For example, consider the following set:

$$2x_1 + 9x_2 = 5$$

$$3x_1 - 4x_2 = 7$$

This set can be expressed in vector-matrix form as

$$\begin{bmatrix} 2 & 9 \\ 3 & -4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 7 \end{bmatrix}$$

Which can be represented in the following compact form

$$Ax=b \rightarrow x=b/A$$

Solution of Linear Algebraic Equations

1- The Matrix Inverse Method

The procedure for doing this is developed from the concept of a matrix inverse. The inverse of a matrix A is denoted by A^{-1} and has the property that

$$A^{-1}A = AA^{-1} = I$$

Where I is the identity matrix. Using this property, we multiply both sides of equation from the left by A^{-1} to obtain

$$A^{-1}Ax = A^{-1}b \longrightarrow Ix = A^{-1}b \longrightarrow x = A^{-1}b$$

$$x = inv(A)*b$$

```
>> A=[ 2,9;3,-4];      >> b = [5;7];      >> x = inv(A)*b
```

Note: If A (det(A) = 0) is singular, then a unique solution to equation does not exist.

Solution of Linear Algebraic Equations

2- The Left Division Method

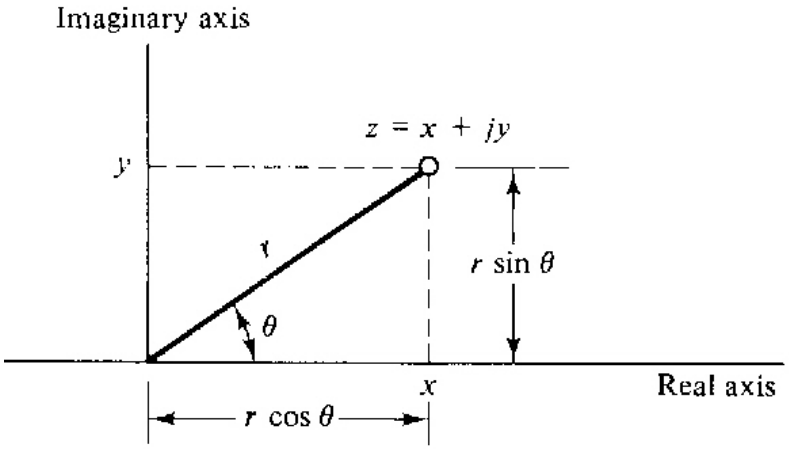
Matlab provides the left division method for solving the equation set Ax = b.

This method is based on Gauss elimination.
To use the left division method to solve for x, you type
x = A\b

```
>> A = [2,9;3,-4];
>> b = [5,7]';
>> x = A\b
x =
    2.3714
    0.0286
```

Complex Number

A complex number consists of the real part and the imaginary part.



➢**Imaginary number**

In Matlab, i and j are variable names that default to the imaginary number

This imaginary number is defined to be the square root of −1

$$z = x + jy \quad \text{or} \quad z = x + iy$$

where x is real part y is imaginary part

Complex Number Function

For Example:

The number z = 1-2i is entered as follows:

>> z = 1 – 2i or z = 1 – 2j

An asterisk (*) is not needed between i or j

abs(z)	**Absolute value**
angle(z)	**Angle of a complex number**
conj(z)	**Complex conjugate**
imag(z)	**Imaginary part of a complex number**
real(z)	**Real part of a complex number**

Script File (M-file)

You can perform operations in Matlab in two ways:

1. **In the interactive mode (Similar to using a calculator), in which all commands are entered directly in the Command window.**
2. **By running a Matlab program stored in script file. This type of file contains Matlab Commands, so running it is equivalent to typing all the commands- one at a time- at the Command window prompt(>>).**

   You can run the file by tying its name at the Command window prompt (>>).

Rules For Naming a Script File

1- The name of a script file must begin with a letter, and may include digits and the underscore character, up to 31 characters.

2- Do not give a script file the same name as a variable.

3- Do not give a script file the same name as a MATLAB command or function. You can check to see if a command, function or file name already exists by using the exist command.

Diary

Another way of capturing output is with the diary command. The command

>> diary filename % your name

copies every thing that subsequently appears in the Command Window to the text file filename. You can then edit the resulting file with any text editor (including the Matlab Editor).

Stop recording the session with

>> diary off

Creating and Using Script File

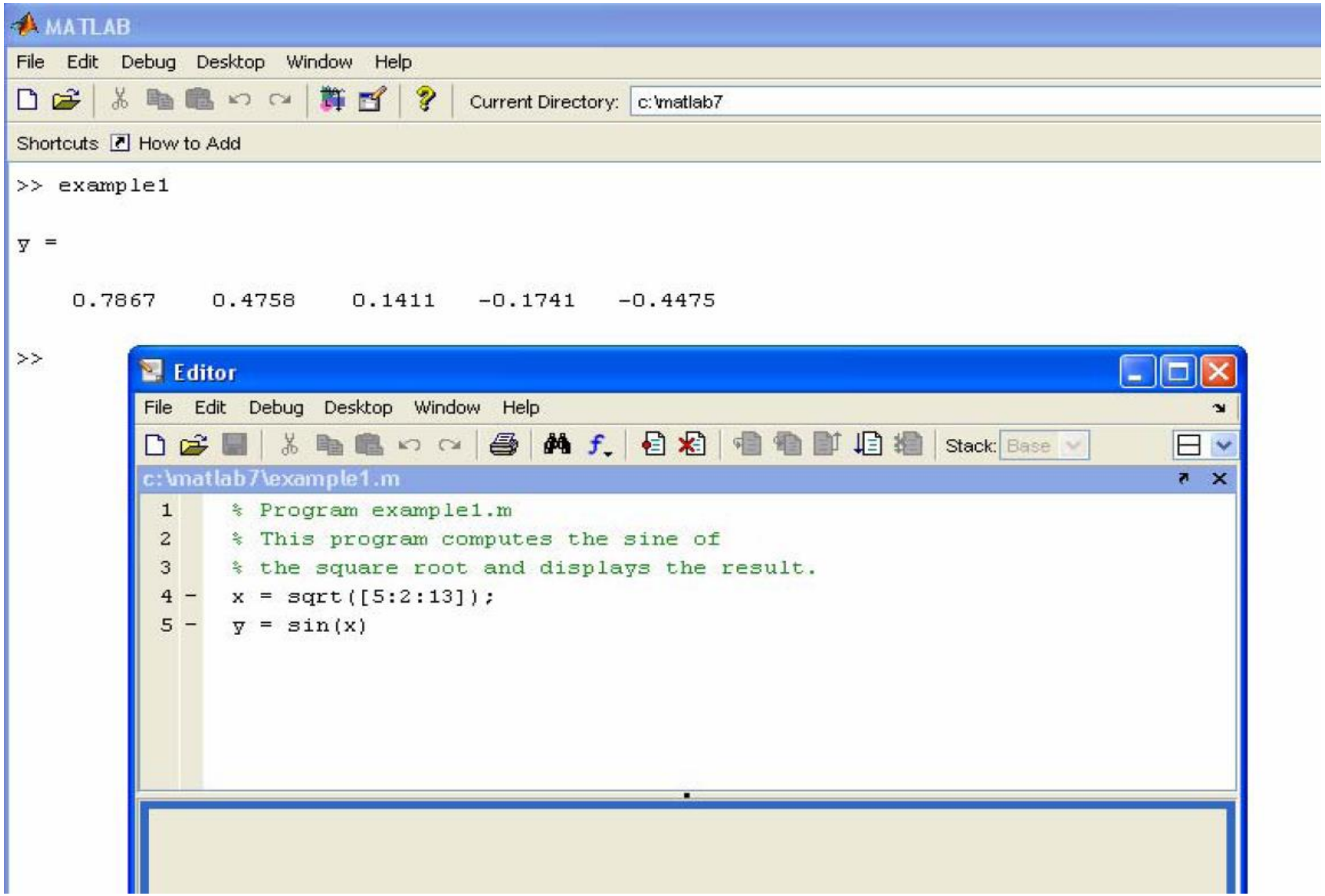
COMMENTS

The Comment symbol (%) may be put anywhere in the line. Matlab ignores everything to the right of the % symbol. For example,

```
>> % This is a comment.
>> x = 2+3 % So is this.
x =
     5
```

Note that the portion of the line before the % sign executed to compute x.

The Matlab Command window with the Editor/Dedugger open



Debugging Script Files

Program errors usually fall into one of the following categories.

1. **Syntax errors such as omitting a parenthesis or comma, or spelling a command name incorrectly. MATLAB usually detects the more obvious errors and displays a message describing the error and its location.**
2. **Errors due to an incorrect mathematical procedure, called *runtime errors*. Their occurrence often depends on the particular input data. A common example is division by zero.**

To locate program errors, try the following:

1. **Test your program with a simple version of the problem which can be checked by hand.**
2. **Display any intermediate calculations by removing semicolons at (;) the end of statements.**
3. **Use the debugging features of the Editor/Debugger.**

Programming Style

1. *Comments section %*

   ***a*. The name of the program and any key words in the first line.**

   ***b*. The date created, and the creators' names in the second line.**

   c*. The definitions of the variable names for every input and output variable. Include definitions of variables used in the calculations and *units of measurement for all input and all output variables!

   ***d*. The name of every user-defined function called by the program.**

Programming Style

2. **Input Section**

   Include input data and/or the input functions and comments for documentation.

3. **Calculation Section**

4. **Output Section**

   This section might contain functions for displaying the output on the screen.

Input/output commands

Command	Description
disp(A)	Displays the contents, but not the name, of the array A.
disp('text')	Displays the text string enclosed within quotes.
x = input('text')	Displays the text in quotes, waits for user input from the keyboard, and stores the value in x.
x = input('text', 's')	Displays the text in quotes, waits for user input from the keyboard, and stores the input as a string in x.

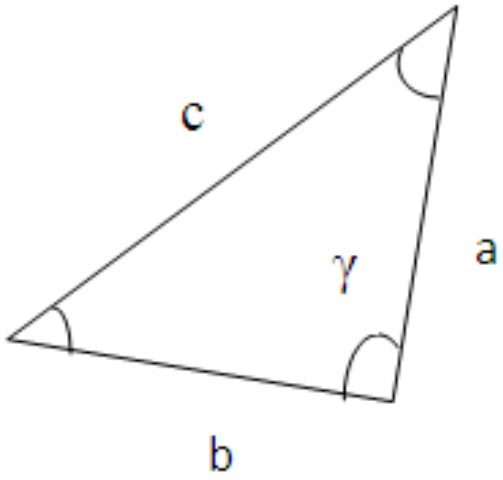
Example of a Script File

Problem:

In the triangle shown has a three sides a,b,c and three angle α,β,γ, write a script file to calculate the side c , defined a, b,γ (in degree) as input variables by substituting the variables in the law of cosines.

$$c^2=a^2+b^2-2ab\cos(\gamma)$$

Use the program to find c at a=21, b=45, γ=60°



Solution

```
%Programming for law of cosines
%Calculate the one side of a triangle
%Created in 3 January 2012 by Eng.Bani yaseen
%Input variables length of two sides of triangle
... and angle.
%Output Variable compute length of the third
... side of the triangle
a=input('please enter side a:');
b=input('please enter side b:');
gamma=input('please enter the angle gamma:');

c=sqrt(a^2+b^2-2*a*b*cosd(gamma));

disp(c)
```

User-Defined Function (Function File)

Another type of M-file is a function file. Unlike a script file, all the variables in a function file are local variables, Which means their values are available only within the function.

Function file are useful when you need to repeat a set of commands several times. They are the building blocks of a large programs.

To create a function file, open M-file from / file/ tools bar.

Function Definition Line

The first line in a function file must begin with a *function definition line* that has a list of inputs and outputs. This line distinguishes a function M-file from a script M-file. Its syntax is as follows:

```
function [output variables] = function_name(input variables)
```

Note that the output variables are enclosed in *square brackets* [], while the input variables must be enclosed with *parentheses* (). The function name (here, name) should be the same as the file name in which it is saved (with the .m extension).

Function Example

```
function z = fun(x,y)
u = 3*x;
z = u + 6*y.^2;
```

Note the use of a semicolon ; at the end of the lines.

This prevents the values of u and z from being displayed.

Note also the use of the array exponentiation operator (.^).
This enables the function to accept y as an array.

Call this function with its output argument:

```
>>fun(3,7)
ans =
      303
```

The function uses x = 3 and y = 7 to compute z.

Call this function without its output argument and try to access its value. You will see an error message.

```
>>fun(3,7)
ans =
     303
>>z
???  Undefined function or variable 'z'.
```

Assign the output argument to another variable q:

```
>>q = fun(3,7)
q =
   303
```

You can suppress the output by putting a semicolon ; after the function call. For example, if you type `q = fun(3,7);` the value of `q` will be computed but not displayed (*because of the semicolon*).

The variables x and y are local to the function fun, so unless you pass their values by naming them x and y, their values will not be available in the workspace outside the function. The variable u is also local to the function. For example,

```
>>x = 3;y = 7;
>>q = fun(x,y);
>>x
     x =
          3
>>y
     y =
          7
>>u
??? Undefined function or variable 'u'.
```

Only the order of the arguments is important, not the names of the arguments:

```
>>x = 7;y = 3;
>>z = fun(y,x)
z =
    303
```

The second line is equivalent to `z = fun(3,7)`.
You can use arrays as input arguments:

```
>>r = fun([2:4],[7:9])
r =
    300     393     498
```

A function may have more than one output. These are enclosed in square brackets.

For example, the function circle computes the area A and circumference C of a circle, given its radius as an input argument.

```
function [A, C] = circle(r)
A = pi*r.^2;
C = 2*pi*r;
```

The function is called as follows, if the radius is 4.

```
>>[A, C] = circle(4)
A =
   50.2655
C =
   25.1327
```

Examples of Function Definition Lines

1. One input, one output:

```
function [area_square] = square(side)
```

2. Brackets are optional for one input, one output:

```
function area_square = square(side)
```

3. Three inputs, one output:

```
function[volume_box]= box(height,width,length)
```

4. One input, two outputs:

```
function [area_circle,circumf]=circle(radius)
```

5. No named output: `function sqplot(side)`

Local Variables

The names of the input variables given in the function definition line are local to that function.

This means that other variable names can be used when you call the function.

All variables inside a function are erased after the function finishes executing, except when the same variable names appear in the output variable list used in the function call.

Global Variables

The global command declares certain variables global, and therefore their values are available to the basic workspace and to other functions that declare these variables global.

The syntax to declare the variables a, x, and q is

```
>>global a x q
```

Any assignment to those variables, in any function or in the base workspace, is available to all the other functions declaring them global.

Finding Zeros of a Function `fzero`

You can use the `fzero` function to find the zero of a function of a single variable, which is denoted by x. One form of its syntax is:

```
>>fzero('function', x0)
```

where function is a string containing the name of the function, and x_0 is a user-supplied guess for the zero.

The fzero function returns a value of x that is near x_0.
It identifies only points where the function crosses the *x*-axis, not points where the function just touches the axis.

For example, `fzero('cos',2)` returns the value 1.5708.

Example:

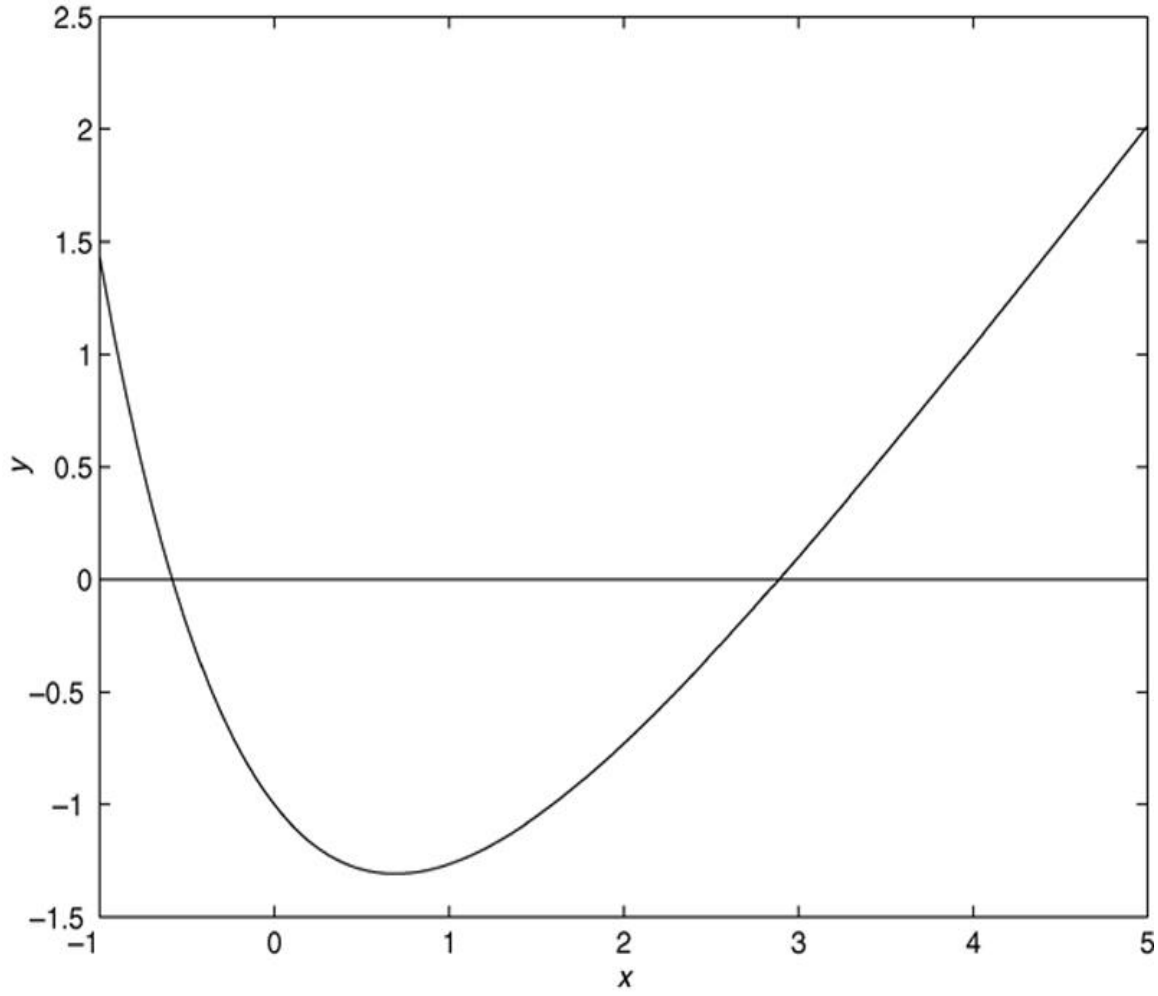
Plot of the function $y = x + 2e^{-x} - 3$.

There is a zero near

$x = -0.5$

and one near

$x = 3$



To use the `fzero` function to find the zeros of more complicated functions, it is more convenient to define the function in a function file.

For example, if $y = x + 2e^{-x} - 3$, define the following function file:

```
function y = f1(x)
y = x + 2*exp(-x) - 3;
```

To find a more precise value of the zero near x = -0.5, type

```
>>x1 = fzero('f1',-0.5)
```

The answer is x = -0.5881.

```
>>x2 = fzero('f1',3)
```

The answer is x = 2.8887.

Finding the Minimum of a Function (`fminbnd`)

The `fminbnd` function finds the minimum of a function of a single variable, which is denoted by `x`. One form of its syntax is:

```
fminbnd('function', x1, x2)
```

where function is a string containing the name of the function. The `fminbnd` function returns a value of `x` that minimizes the function in the interval `x1 ≤ x ≤ x2`.

For example,
`>>fminbnd('cos',0,4)` returns the value 3.1416.

When using `fminbnd` it is more convenient to define the function in a function file. For example, if $y = 1 - xe^{-x}$, define the following function file:

```
function y = f2(x)
y = 1-x.*exp(-x);
```

To find the value of x that gives a minimum of y for $0 \le x \le 5$, type

```
>>x = fminbnd('f2',0,5)
```

The answer is $x = 1$. To find the minimum value of y, type `>>y = f2(x)` . The result is y = 0.6321.

Function Handles @

You can create a function handle to any function by using the at sign, @, before the function name. You can then name the handle if you wish, and use the handle to reference the function. For example, to create a handle to the sine function, you type

```
>>sine_handle = @sin;
```

where sine_handle is a user-selected name for the handle.
A common use of a function handle is to pass the function as an argument to another function. For example, we can plot sin x over $0 \leq x \leq 6$ as follows:

```
>>plot([0:0.01:6],sine_handle,[0:0.01:6])?!
```

Methods for Calling Functions

There are four ways to invoke, or "call," a function into action. These are:

1. **As a character string identifying the appropriate function M-file,**
2. **As a function handle,**
3. **As an "inline" function object,**
4. **As a string expression.**

Examples of these ways follow for the fzero function used with the user-defined function fun1, which computes $y = x^2 - 4$.

1. As a character string identifying the appropriate function M-file, which is

```
function y = fun1(x)
y = x.^2-4;
```

The function may be called as follows, to compute the zero over the range $0 \le x \le 3$:

```
>>[x, value] = fzero('fun1',[0, 3])
```

2. As a function handle to an existing function M-file:

```
>>[x, value] = fzero(@fun1,[0, 3])
```

3. As an "inline" function object:

```
>>fun1 = 'x.^2-4';
>>fun_inline = inline(fun1);
>>[x, value] = fzero(fun_inline,[0, 3])
```

4. As a string expression:

```
>>fun1 = 'x.^2-4';
>>[x, value] = fzero(fun1,[0, 3])
```

or as

```
>>[x, value] = fzero('x.^2-4',[0, 3])
```

Anonymous Functions

Anonymous functions enable you to create a simple function without needing to create an M-file for it. You can construct an anonymous function either at the MATLAB command line or from within another function or script. The syntax for creating an anonymous function from an expression is

```
fhandle = @(arglist) expr
```

where arglist is a comma-separated list of input arguments to be passed to the function, and expr is any single, valid MATLAB expression.

To create a simple function called sq to calculate the square of a number, type

```
>>sq = @(x) x.^2;
```

To improve readability, you may enclose the expression in parentheses, as `sq = @(x)(x.^2);`

To execute the function, type the name of the function handle, followed by any input arguments enclosed in parentheses.

For example,

```
>>sq([5,7])
ans =
    25      49
```

Multiple Input Arguments

You can create anonymous functions having more than one input. For example, to define the function $\sqrt{(x^2 + y^2)}$**, type**

```
>>sqrtsum = @(x,y) sqrt(x.^2 + y.^2);
```

Then type

```
>>sqrtsum(3,4)
ans =
    5
```

As another example, consider the function defining a plane, $z = Ax + By$**. The scalar variables** A **and** B **must be assigned values before you create the function handle. For example,**

```
>>A = 6; B = 4:
>>plane = @(x,y) A*x + B*y;
>>z = plane(2,8)
z =
  44
```

Calling One Function within Another

One anonymous function can call another to implement function composition. Consider the function $5\sin(x^3)$. It is composed of the functions $g(y) = 5\sin(y)$ and $f(x) = x^3$. In the following session the function whose handle is h calls the functions whose handles are f and g.

```
>>f = @(x) x.^3;
>>g = @(x) 5*sin(x);
>>h = @(x) g(f(x));
>>h(2)
ans =
    4.9468
```

Importing Spreadsheet Files

Some spreadsheet programs store data in the `.wk1` format. You can use the command `M = wk1read('filename')` to import this data into MATLAB and store it in the matrix M.

The command `A = xlsread('filename')` imports the *Microsoft Excel* workbook file `filename.xls` into the array A. The command `[A,B]=xlsread('filename')` imports all *numeric* data into the array A and *all text* data into the cell array B.