MATLAB Primer

John F. Patzer II

MATLAB: Getting Started

- MATLAB™, which stands for MATrix LABoratory, is a product of The Mathworks, Inc.

MATLAB is a platform independent (meaning it should work equally well on a PC, a MAC, or a mainframe system), interactive environment for computation, visualization, and animation. MATLAB has a "natural language" interface that is easy to use. It also has built-in functions and toolboxes that that help facilitate problem solution and graphical visualization of problems. MATLAB is especially powerful in the solution of linear algebra (matrix) problems.

- The mathematics that you typically learn in math class is called symbolic because it typically involves rearrangement and manipulation of symbols to find solutions. MATLAB, however, uses numerical methods to solve the same types of problems. The solutions generally end up being the same, but the method of arriving at them differs.

- Windows - MATLAB uses three main windows for communication
  - Command Window - main communication screen

    You can scroll up and down through this screen, which records your commands and the results of your commands

    » MATLAB prompt - where the next command is typed

    ↑, ↓ MATLAB command buffer, scroll through previous commands

    `help` type `help topic` to access online help on a topic

    `lookfor` `lookfor topic` will search the MATLAB libraries for matching phrases

    `%` comment delimiter; MATLAB ignores everything on a line after the `%` symbol

    `;` suppress immediate output display
  - Graphics Window - displays plots, animation, etc

    More than one graphics window can be open at a time.
  - Edit Window - simple text editor

    Used to create, edit, and store your own programs (called scripts).

- MATLAB `diary` command

A useful command that will create a text log of your command window screen session in a file that you specify. The log can then be accessed using your favorite text editor. It can be edited to remove errors. The edited file can be saved to archive your work. The edited version can also be made into script files (more later) or function files (more later). The diary can be turned on and off at will during a session.

```
diary  filename   % set and open filename (with path) for archive file
     example:  » diary  c:\temp\prob4_3.txt
diary off    % turn off diary record
diary on     % turn diary record back on with previous filename
```

- MATLAB `save filename` command

Occasionally, you might want to pause in the middle of a MATLAB session. To save the variables you have generated and not have to reenter them, you could use the `save filename` command. MATLAB will save your variables for later use. You can reload the variables with the `load filename` command. **Note:** this command pair only saves and loads the variables that are in the workspace - not the dialogue. The workspace is saved in binary (not text) format so that you cannot use a text editor to examine the contents. If you want to review the dialogue, you must use the `diary` command.

- MATLAB mathematics
  - *Variable names*: variable names must begin with a letter. Variable names can have any combination of letters, numbers, and underscores, _ (the shift- key). Although they can be any length, MATLAB only retains and recognizes the first 19 characters. Finally, MATLAB is case sensitive. That means that **a** is not the same as **A**. Variable names should be meaningful, reflecting its use in the problem you are solving.

  - *Data type*: all MATLAB values are complex and double precision. Double precision means that, depending upon the computer, 15-18 significant figures are retained by the computer and used in computations. Complex values are generically represented as **(a+b*i)** where **a** is the real part, **b** is the imaginary part, and **i** is $\sqrt{-1}$. Values entered without the **b*i** component are treated as real numbers (MATLAB recognizes that **b = 0**).

- MATLAB screen display
  - *Result display*: MATLAB displays the results of every command (data input, declaration, or calculation) immediately after each time the return (or enter) key is used. Sometimes, such immediate display is not desired. The immediate display can be suppressed by using the semicolon, `;`, at the end of a line before pressing the return key.
  - *Display format*: MATLAB uses double precision arithmetic internally (retaining all significant digits to the double precision limit). The display is controlled by the `format` command as follows. The default display (if you do nothing) is `short`.

`format`	Default. Same as `short`.
`format short`	Scaled fixed point format with 5 digits.
`format long`	Scaled fixed point format with 15 digits.
`format short e`	Floating point format with 5 digits.
`format long e`	Floating point format with 15 digits.
`format short g`	Best of fixed or floating point format with 5 digits.
`format long g`	Best of fixed or floating point format with 15 digits.

- MATLAB help facility

MATLAB has a reasonably good help facility. Typing `help` on the command line produces the following display

```
HELP topics:

workarea\matlab      -  (No table of contents file)
matlab\general       -  General purpose commands.
matlab\ops           -  Operators and special characters.
matlab\lang          -  Programming language constructs.
matlab\elmat         -  Elementary matrices and matrix manipulation.
matlab\elfun         -  Elementary math functions.
matlab\specfun       -  Specialized math functions.
matlab\matfun        -  Matrix functions - numerical linear algebra.
matlab\datafun       -  Data analysis and Fourier transforms.
matlab\polyfun       -  Interpolation and polynomials.
matlab\funfun        -  Function functions and ODE solvers.
matlab\sparfun       -  Sparse matrices.
matlab\graph2d       -  Two dimensional graphs.
```

```
matlab\graph3d       -  Three dimensional graphs.
matlab\specgraph     -  Specialized graphs.
matlab\graphics      -  Handle Graphics.
matlab\uitools       -  Graphical user interface tools.
matlab\strfun        -  Character strings.
matlab\iofun         -  File input/output.
matlab\timefun       -  Time and dates.
matlab\datatypes     -  Data types and structures.
matlab\winfun        -  Windows Operating System Interface Files
matlab\demos         -  Examples and demonstrations.
images\images        -  Image Processing Toolbox.
images\imdemos       -  Image Processing Toolbox -demos/sample images
toolbox\rtw          -  Real-Time Workshop
rtw\rtwdemos         -  (No table of contents file)
signal\signal        -  Signal Processing Toolbox.
signal\siggui        -  Signal Processing Toolbox GUI
signal\sigdemos      -  Signal Processing Toolbox Demonstrations
toolbox\optim        -  Optimization Toolbox.
toolbox\control      -  Control System Toolbox.
control\ctrlguis     -  Control System Toolbox - GUI support functions.
control\obsolete     -  Control System Toolbox -- obsolete commands.
toolbox\sb2sl        -  SystemBuild to Simulink Translator
stateflow\sfdemos    -  Stateflow demonstrations and samples.
stateflow\stateflow  -  Stateflow
simulink\simulink    -  Simulink
simulink\blocks      -  Simulink block library.
simulink\simdemos    -  Simulink 3 demonstrations and samples.
simulink\dee         -  Differential Equation Editor
toolbox\tour         -  MATLAB Tour
MATLABR11\work       -  (No table of contents file)
toolbox\local        -  Preferences.

For more help on directory/topic, type "help topic".
```

More information about any of the topics can be obtained by typing `help topic` on the command line. For example, `help elfun` produces the elementary math functions display

```
  Elementary math functions.

  Trigonometric.
    sin         - Sine.
    sinh        - Hyperbolic sine.
    asin        - Inverse sine.
    asinh       - Inverse hyperbolic sine.
    cos         - Cosine.
    cosh        - Hyperbolic cosine.
    acos        - Inverse cosine.
    acosh       - Inverse hyperbolic cosine.
    tan         - Tangent.
    tanh        - Hyperbolic tangent.
    atan        - Inverse tangent.
    atan2       - Four quadrant inverse tangent.
    atanh       - Inverse hyperbolic tangent.
```

```
    sec         - Secant.
    sech        - Hyperbolic secant.
    asec        - Inverse secant.
    asech       - Inverse hyperbolic secant.
    csc         - Cosecant.
    csch        - Hyperbolic cosecant.
    acsc        - Inverse cosecant.
    acsch       - Inverse hyperbolic cosecant.
    cot         - Cotangent.
    coth        - Hyperbolic cotangent.
    acot        - Inverse cotangent.
    acoth       - Inverse hyperbolic cotangent.

  Exponential.
    exp         - Exponential.
    log         - Natural logarithm.
    log10       - Common (base 10) logarithm.
    log2        - Base 2 logarithm and dissect floating point number.
    pow2        - Base 2 power and scale floating point number.
    sqrt        - Square root.
    nextpow2    - Next higher power of 2.

  Complex.
    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.

  Rounding and remainder.
    fix         - Round towards zero.
    floor       - Round towards minus infinity.
    ceil        - Round towards plus infinity.
    round       - Round towards nearest integer.
    mod         - Modulus (signed remainder after division).
    rem         - Remainder after division.
    sign        - Signum.
```

Information about how to use any individual function can also be obtained. For example, `help atan` produces the display

```
 ATAN   Inverse tangent.
    ATAN(X) is the arctangent of the elements of X.

    See also ATAN2.
```

which tells us that the `atan` function takes one argument (either scalar, vector, or matrix) and returns the value of the arctangent. It also tells us that there is a closely related function, `atan2`, that may be useful.

MATLAB Workshop 1

Objectives: Start MATLAB, do some calculations, use some basic functions, change display format, quit MATLAB.

command line »

arithmetic operators

`+`	addition	`-`	subtraction
`*`	multiplication	`/`	division (left)
`^`	exponentiation		

output formats (shown with 10e)

`format short`	27.1828	`format short e`	2.7183e+01
`format long`	27.18281828459045	`format long e`	2.718281828459045e+01

basic trig functions

`sin( )`	`asin( )`	`sinh( )`	`asinh( )`
`cos( )`	`acos( )`	`cosh( )`	`acosh( )`
`tan( )`	`atan( ), atan2( )`	`tanh( )`	`atanh( )`
`cot( )`	`acot( )`	`coth( )`	`acoth( )`
`sec( )`	`asec( )`	`sech( )`	`asech( )`
`csc( )`	`acsc( )`	`csch( )`	`acsch( )`

logarithmic/exponential functions

`exp( )`	`log( )`	`log10( )`	`sqrt( )`

MATLAB constants

`pi`	π (=3.14159...)	`inf`	∞ (infinity)
`i or j`	imaginary unit ($\sqrt{-1}$)	`eps`	machine (computer) precision
`realmax`	largest real number	`realmin`	smallest real number

- **Start MATLAB and perform the following**

```
» 1+3
ans =
    4
```

Enter 1+3 and hit return (enter key). The result is calculated and displayed. If the calculation is unassigned, the result is saved in the default variable `ans`.

```
»
» a = 4-3
a =
    1
```

You can choose the variable name to which a value is assigned.

```
»
» % calculate area of circle
»    radius = 5;
»    area = pi*radius^2
area =
```

```
      78.5398
```

% can be used to add comments to your work. Display can be suppressed with a `;` at the end of a line to create a more readable screen display. `pi` is a MATLAB defined constant (3.14159...).

```
»
» angle_deg  = 30;
» sin1 = sin(angle_deg)
sin1 =
    -0.9880
»
» sin2 = sin(pi*angle_deg/180)
sin2 =
    0.5000
```

Trig functions are in radians (not degrees). Using an angle in degrees will produce a wrong result.

```
»
» format long
» angle_deg
angle_deg =
    30
»
» sin2
sin2 =
    0.500000000000000
```

The `format` command controls the display of values (5 digits under `short`, 15 under `long`). Integers are displayed as integers under either. Variable values are display simply by typing the variable name.

```
»
» format short e
» sin1
sin1 =
    -9.8803e-001
```

Exponential (scientific) notation is denoted by `e`. `e-001` means 10^{-001}.

```
» quit
```

End MATLAB session.

Exercises: Perform the following operations in MATLAB.

1. *Arithmetic operations*. Compute the following. Display 5 digits.

   a. $\dfrac{3^{2.5}}{3^{2.5}-1}$ compare with $\left(1-\dfrac{1}{3^{2.5}}\right)^{-1}$

   b. $7\dfrac{\sqrt{7}-2}{\left(\sqrt{5}+1\right)^2}-3$ √x can be represented by `sqrt(x)` or `x^0.5` in MATLAB

c. Volume = $\frac{4}{3}\pi\ radius^3$ with radius = $\pi^{\frac{1}{5}} - 1$ π is `pi` in MATLAB.

2. *Exponential and logarithmic expressions*. Evaluate each of the following. Display in 5-digit scientific notation.

> e^x is written as `exp(x)`, 10^x is written as `10^x`, ln(x) is written as `log(x)`, and $\log_{10}(x)$ is written as `log10(x)` in MATLAB.

a. e^5, $\ln(e^5)$, $\log_{10}(e^5)$, and $\log_{10}(10^5)$

b. $e^{\pi\sqrt{33}}$

c. Solve $5^x = 23$. (Note, by taking logarithms, the solution is $x = \frac{\ln(23)}{\ln(5)}$. Compute this and verify that it is the correct answer by direct substitution into the original equation. What happens if you use base 10 logarithms instead of natural logarithms?

3. *Trigonometric functions*. Evaluate each of the following. Display in 15-digit scientific notation.

> MATLAB trig functions are provided at the start of this Workshop. Remember: angle measurements are in radians!!

a. $\sin\left(\frac{\pi}{5}\right)$, $\cos(\pi)$, $\tan(45°)$

b. $\sin^2\left(\frac{\pi}{5}\right)+\cos^2\left(\frac{\pi}{5}\right)$ (What do you think the result should be? Why?)

c. Solve $t^2 = \cos^2(\omega) - \sin^2(\omega)$ with $\omega = 1.5\pi$.

4. *Machine limits*. Evaluate each of the following. Display in 15-digit scientific notation.

a. `>> max = realmax` greatest magnitude real value permitted
b. `>> min = realmin` least magnitude real value permitted
c. `>> precision = eps` machine accuracy (significant digits)

Recap: You should have learned

- How to do simple arithmetic and function calculations
- How to assign values to variables
- How to control screen display (no display and appearance of floating point numbers)
- Limits of machine precision, size of allowable numbers

MATLAB Workshop 2

Objectives: Use MATLAB to solve engineering problems and archive results.

diary archive capability

```
» diary  filename  %set and open filename (with path) for archive file
```

example: `>> diary  c:\temp\prob4_3.txt`

```
» diary off    %turn off diary record
```

turns diary record off

```
» diary on     %turn on diary record
```

turns diary record back on with previous filename

Example 1: Pressure at the bottom of a tank

Problem Statement: The absolute pressure at the bottom of a liquid storage tank that is vented to the atmosphere is given by the relation, $P_{abs} = \rho\, gh + P_{atm}$, where P_{abs} is the absolute pressure, ρ is the liquid density, g is gravitational acceleration, h is the height of the liquid, and P_{atm} is the outside atmospheric pressure. Find P_{abs} in SI units if ρ = 1000 kg/m^3, g = 32.2 ft/s^2, h = 7 yd, and P_{atm} = 1 atm.

Background: This is a problem in units conversion.

Solution strategy: Convert all units to SI before performing calculation

ρ is already in SI units.

need factors: ft_to_m = 0.3048
yd_to_m = 0.9144
atm_to_Pa = $1.013 \bullet 10^5$

Calculate P_{abs}

- **Start MATLAB and perform the following**

```
» diary  c:\temp\wkshop2 example 1.txt (or choose your own path)
»
» % Your Name
» % Your Class
» % Today's Date
»
» % solution to Workshop 2, Example 1
»    format short e
»
» % conversion factors
»    ft_to_m = 0.3048; yd_to_m = 0.9144; atm_to_Pa = 1.013e5;
»
» % convert all parameters to SI
»    rho = 1000; g = ft_to_m*32.2; h = yd_to_m*7;
»    Patm = atm_to_Pa*1;
»
» % calculate Pabs in Pa
»    Pabs = rho*g*h + Patm
Pabs =
     1.6412e+005
»
» diary off
```

An empty return adds white space to your dialogue and makes it easier to read. Indenting (using the spacebar) also adds whitespace and makes your work easier to read. Comments help inform you and others of what is happening. More than one statement can be placed on a line! If separated by a semicolon, `;`, the display is suppressed. If separated by a comma, `,`, the display follows the first return.

Be sure to use the `diary off` command to close the diary (otherwise, you may lose the diary file). Use whatever means and text editor with which you are familiar to open the file c:\temp\wkshop2 example 1.txt (or whatever you called it). Note that it contains a listing of all screen display (including white space and comments) that was entered from the time that the diary was initiated. This record can be edited (if necessary to remove errors and error statements that might occur), printed, and attached to a homework, report, or work file as an excellent record of your solution.

Example 2: Spring mechanics

Problem Statement: A spring has a spring constant of 25 lb_f/in. What force is required to stretch the spring 3 in? How much work is done by the force in stretching the spring 3 in?

Background: From physics and conservation of momentum, the force required to stretch the spring is given by

$$F = kd$$

and the work performed by the spring is

$$W = kd^2 / 2$$

where F is the force, k is the spring constant, d is the distance, and W is the work.

Solution strategy: Convert all units to SI before performing calculation

in_to_m = 0.0254
lbf_to_N = 4.448

Find F and W by straightforward substitution into the equations.

- **Start MATLAB and perform the following**

```
» diary  c:\temp\wkshop2 example 2.txt (or choose your own path)
»
» % Your Name
» % Your Class
» % Today's Date
»
» % solution to Workshop 2, Example 2
»    format short e
»
» % conversion factors
»    in_to_m = 0.0254; lbf_to_N = 4.448;
»
» % convert k and d to SI
»    k = 25*lbf_to_N/in_to_m;
»     d = 3*in_to_m;
»
» % calculate F in N and W in J (SI units)
»    F = k*d
F =
     3.3360e+002
»
»    W = k*d^2/2
W =
     1.2710e+001
»
» diary off
```

Example 3: Light bulb life expectancy

Problem Statement: The life of an incandescent light bulb has been experimentally determined to vary inversely as the 12th power of the applied voltage. A rated life of a bulb is 800 hours at 115 V. What is the life expectancy at 120 V? What is the life expectancy at 110 V?

Background: From the problem statement

$$L = AV^{-12}$$

where L is the expected life of the light bulb (hr), V is the applied voltage (V), and A is the proportionality constant (hr•V^{12}).

Solution strategy: Find A from L = 800 hours at 115 V. Calculate L at V = 120V. Calculate L at V = 110 V.

- **Start MATLAB and perform the following**

```
» diary  c:\temp\wkshop2 example 3.txt (or choose your own path)
»
» % Your Name
» % Your Class
» % Today's Date
»
» % solution to Workshop 2, Example 3
»      format short e
»
» % calculate proportionality constant (hr.V^12)
»    A = 800*115^12
A =
     4.2802e+027
»
» % expected lifetime at 120 V, hr
»    L120 = A*120^-12
L120 =
     4.8005e+002
»
» % expected lifetime at 110 V, hr
»    L110 = A*110^-12
L110 =
     1.3638e+003
»
» diary off
```

Exercises

1. Use MATLAB to solve the following problem.

   Problem Statement: A piece of cast iron, which has a density of 450 lb_m/ft^3, has a very irregular shape. You need to determine its volume in SI units. To do so, you submerge the specimen in a cylindrical water tank (d = 0.5 yd). The water rises 8.64 cm above its original level.

Background: The volume of the specimen is equal to the volume of displaced water. The volume of displace water is equal to

$$V = (\frac{\pi d^2}{4})h$$

where h is the rise in water level.

Need to convert everything to SI units for consistency.

Solution strategy: Convert everything to SI units before calculating V

Need conversion factors

yd_to_m = 0.9144

cm_to_m = 0.01

(Note: density conversion not needed to find volume!!!!)

Calculate volume, V

2. Use MATLAB to solve the following problem.

Problem Statement: A pipeline in an oil refinery is carrying oil to a large storage tank. The pipe has a 20 in internal diameter. The oil is flowing at 5 ft/s. The density of the oil is 57 lb_m/ft^3. What is the mass flow rate of oil in SI units? What is the mass and volume of oil, in SI units, that flows in a 24-hour time period.

Background: Need to be careful about units. By dimensional analysis, the mass flow rate of oil, $\dot{M}$ (kg/s) is

$$\dot{M} = \rho\, vA$$

where ρ is the density (kg/m^3), v is the flow speed (m/s), and A is the cross-sectional area of the pipe (m^2). The flows in any time period, T (s), are given by

$$M = \dot{M}\, T \quad \text{and} \quad V = M / \rho$$

where M is the mass (kg) and V is the volume (m^3).

Solution strategy: Convert everything to SI units before proceeding.

Need conversion factors

in_to_m = 0.0254

ft_to_m = 0.3048

lbm_to_kg = 0.4535

hr_to_s = 3600

Make conversions for ρ, v, and d.

Calculate cross-sectional area, $A = \pi d^2/4$

Calculate mass flow rate, $\dot{M}$.

Calculate total mass in 24 hours, M.

Calculate equivalent volume, V.

3. Use MATLAB to solve the following problem.

Problem Statement: A researcher proposes to use a hollow wrought aluminum alloy sphere, 500 cm outside diameter with a wall thickness of 3 mm, as a buoy to mark the location of an underwater research site. Will the sphere float? If so, how high does the sphere rise above the water?

Background: This is buoyancy problem. The sphere will float if its average density is less than that of water (assume to be 1 kg/L = 1000 kg/m^3). Need to calculate mass of aluminum used

$$M = \rho_{Al}\left(\frac{\pi}{6}\right)\left(d_0^3 - d_i^3\right)$$

where M is the mass of aluminum (kg), ρ_{Al} is the density of aluminum (= 2800 kg/m^3), d_o is the outside wall diameter (m), and d_i is the inner wall diameter (m). The average density, ρ_{ave} (kg/m^3), is then

$$\rho_{ave} = \frac{M}{V_s} = \frac{6M}{\pi d_o^3}$$

where V_s is the volume of the sphere. If the sphere floats, the volume that it displaces is equivalent to a volume of water of equal mass, i.e.,

$$V_d = M / \rho_{H_2O}$$

where V_d is the submerged volume (m^3) and ρ_{H_20} is the density of water. This needs to be used in conjunction with the mensuration formula for the volume of a spherical sector,

$$V_h = \frac{\pi}{6} R^2 h$$

where V_h is the volume (m^3) of a sector of depth h (m). If $V_d > V_s/2$, need to think carefully about how to use the V_h relation (Why?).

Solution strategy: Make sure all parameters are in SI units.

Specify parameters: d_o, d_i, ρ_{Al}, ρ_{H_20}
Calculate sphere volume, V_s.
Calculate mass, M.
Calculate average density, ρ_{ave}.
If $\rho_{ave} > \rho_{H_20}$, sphere sinks; else
Calculate h.
Calculate height above water, H = d_o - h.

4. Use MATLAB to solve the following problem.

Problem Statement: A 10 m, medium carbon steel cable is needed to support a load of 100,000 N. The deflection (elongation) of the cable under the load must be less than 1 cm. Ignoring the mass of the cable, what cable mass is required to just support the load without permanent deformation? (E = 207,000 MPa, S_y = 552 MPa, S_t = 690 MPa, ρ = 7900 kg/m^3).

Background: This is a stress-strain problem (S = Ee)

Stress: $S = \frac{T}{A_0}$ where $A_0 = \frac{\pi d^2}{4}$

S: stress, T: tension, A_0: cable cross-sectional area, d: cable diameter

Strain: $e = \frac{\Delta l}{l_0}$ where $\Delta l = l - l_0$

e: strain, l_0: initial cable length, Δl: change in cable length, l: cable length under load

Mass: $m = \rho V$ where $V = A_0 l = \pi d^2 l_0 / 4$

m: mass, V: cable volume

Solution strategy: Make sure all units are SI. Need to check two limits for stress

permanent deformation ==> $T/A_0 < S_y$ or $A_0 > T/S_y$

maximum deflection ==> $S = T/A_0 < Ee_{max}$ or $A_0 > T/(Ee_{max})$

strategy: calculate A_0 by both methods

use larger A_0 to compute mass

Recap: You should have learned

- How to create a diary to record your MATLAB session.
- The use of white space and comments to make your sessions more readable.
- How to set up and solve problems with MATLAB

MATLAB Workshop 3

Objectives: Define your working directory, create and use a script file, create and use a function file.

directory commands

`pwd`	show current (active) directory
`cd directory_name`	change current (active) directory to specified directory
`dir, ls`	list current (active) directory contents
`delete filename`	delete indicated file from current (active) directory
`what`	list files in current (active) directory

```
» edit
```
invoke MATLAB editor

Script files

A *script file* is an external file that contains a sequence of MATLAB statements that perform a *task*. Typing the filename executes the task. **The purpose of a script file is to group commonly occurring MATLAB program lines that perform a task under a single name so that they can be executed by typing a simple command rather than constantly retyping the entire set of lines.** Script files are generally headed by some comment statements that describe the contents of the file. Typing `help filename` will display the initial comment lines.

Script files have a filename extension of ".m" and are often called "M-files". MATLAB will search the current directory for user-defined (i.e., your) M-files.

Function files

A user-defined *function file* can be added to MATLAB's file vocabulary and invoked (used) in the same manner as any intrinsic MATLAB function (such as sin and cos). **The purpose of a function is to compute (and retain in defined variables) values.** The top line of the file defines the syntax for the function (more on this later). The next few lines are generally comment lines that describe the function and how to use it. These lines are displayed when `help filename` is typed.

Function files have a filename extension of ".m" and are often called "M-files". MATLAB will search the current directory for user-defined (i.e., your) M-files.

There is no way to distinguish between a script file and a function file simply by examining the file name.

Example 1: Setting the current directory

Generally, you will want to specify the directory where you are keeping your M-files as the current directory. This is also a good place to keep the diary files that you create from running MATLAB. Diary files are a good starting place to use MATLAB commands that you have already tested to create appropriate script files and function files without having to retype the commands.

- **Start MATLAB and perform the following**

```
» cd  c:\temp\     (or choose your own directory - possibly a: if you are using a floppy)
» pwd
ans =
c:\temp
```

Change current directory. Check to see result.

```
» diary  wkshop3.txt
```

Start diary. Note that the full path is not provided, the file will be placed in the current directory.

You can use the directory commands shown at the start of this workshop to examine or alter the contents of the directory. Try them!

Example 2: Creating a script header file

Script files can be generated within MATLAB using the MATLAB header file, as shown here, or with any text editor. If you use another text editor, be sure to save the file as ASCII text with a .m extension in your MATLAB file directory.

- **Continuing in the same MATLAB session, perform the following**

```
» edit
```

Invokes MATLAB text editor which pops up in a new window.

Type the following lines in the text editor window:

```
% header - a script file to provide a diary header
% file written by your name.  last modified today's date.

disp('  ')
disp('your name')
disp('engr 0111')
disp(date)
disp('  ')
```

`disp` is a MATLAB function that displays the text contained within the parens. Note that the text is contained within single quotes. `date` is a MATLAB function that returns the current date.

After entering the above lines, save the file with the filename header. The MATLAB editor automatically provides the file extension .m. If you are using a different editor, such as Notepad or Wordpad, you will need to supply the extension.

After saving the file, exit the editor (click on the X in the upper right-hand corner).

- **Back in the MATLAB Command Window**

```
» help header
header - a script file to provide a diary header
file written by your name.  last modified today's date.
```

`help header` causes the comment lines up to the first non-comment line in the designated script/function file to display. You should always identify the purpose of the file, who wrote the file, and when the file was last modified in your comment section.

```
»
» header

your name
engr 0111
current date
```

`header` alone causes the script file to execute as shown. Now, rather than type in your name, class, and date each time you run a MATLAB session, you only need type `header` to accomplish the same purpose.

Example 3: Creating a (constant) function file

A common activity in solving engineering problems is unit conversions so that all variables and parameters are in consistent (SI?) units. Remember how many times you had to do that in Workshop 2? Creating MATLAB function files for the conversion factors would make life a little easier and would help avoid errors from possibly mistyping the value of a conversion factor (and not noticing it when reviewing your solution).

- **Continuing in the same MATLAB session, perform the following**

```
» edit
```

Invokes MATLAB text editor which pops up in a new window.

Type the following lines in the text editor window:

```
function [ft_to_m] = ft_to_m;
% ft_to_m - a function file to for ft to m conversion
% file written by your name.  last modified today's date.

ft_to_m = 0.3048;
```

Function files are always headed by a line with `function function_name;`. You will learn various possibilities and formats for `function_name` as we progress. `function` will always remain the same. The next few lines describe the function. This particular function assigns the conversion factor to the name as shown.

After entering the above lines, save the file with the filename ft_to_m.m. After saving the file, exit the editor (click on the X in the upper right-hand corner).

Back in the MATLAB Command Window

```
» help ft_to_m
ft_to_m - a function file to for ft to m conversion
file written by your name.  last modified today's date.
```

Displays the comment lines in your function.

```
»
» length = 2*ft_to_m
length =
    0.6096
```

Use of the function name retrieves the appropriate conversion factor. The conversion factor is thus always available if the function is saved in your MATLAB working directory.

```
» diary off
```

Exercises

Create functions for all of the conversion factors that you think you might want to use. Save them in your MATLAB working directory.

Recap: You should have learned

- How to specify your MATLAB working directory (where you keep MATLAB files).
- How to invoke and use the MATLAB editor.
- How to create, save, and use a simple script file.
- How to create, save, and use a (constant) function file.

MATLAB Workshop 4

Objectives: Learn how to declare arrays, do simple array arithmetic, and use array functions in MATLAB.

array operations

- `.*` term by term multiplication
- `./` term by term division
- `.^` term by term exponentiation

array functions

- `size(x)` determines size of **x**
- `linspace(a,b,c)`
- `length(y)` determines # of elements in **y**

- **Start MATLAB and perform the following**

```
» cd  your MATLAB directory
» diary  wkshop4.txt
» header
```

Display of your header output

```
»
» p = [2 4 6]
p =
    2     4     6
»
» q = [1, 3, 5, 7]
q =
    1     3     5     7
```

p is a *row vector* with three *elements* (values, members). **q** is a *row vector* with four elements. Row vectors are declared by giving them a name and enclosing the elements in square brackets, `[ ]`. The elements can be separated by spaces or commas.

```
»
» s = [1; 2; 3]
s =
    1
    2
    3
```

s is a *column vector* with three *elements*. The elements of a column vector are separated by semicolons.

```
»
» size(p)
ans =
    1     3

» size(q)
ans =
    3     1
```

`size()` is a MATLAB array function that tells how big an array is. **p** is an array with 1 row and 3 columns, i.e., a row vector with three elements. **q** is an array with 3 rows and 1 column, i.e., a column vector with three elements.

```
»
» t = [1 3 5];
» u = p + t
u =
    3     7     11
```

Arrays of the same *size* can be added term by term. The result is $z_i = x_i + y_i$ for each element, i.

```
»
» v = p + s
??? Error using ==> +
Matrix dimensions must agree
```

A row vector and a column vector have different *size* even if they have the same number of elements. They cannot be added (or subtracted).

```
»
» p = 0.5*p
p =
    1     2     3
```

A vector can be multiplied by a scalar (the result is multiplying each element by the scalar). Remember, **p** was originally [2 4 6] until halved in this step. The result is $z_i = s * y_i$ for each element, i.

```
»
» v = p.*t
v =
    1     6     15
```

Two vectors of the same size can be multiplied (or divided) term by term using the array arithmetic operators `.*` or `./`. The result is $z_i = x_i * y_i$ for each element, i.

```
»
» w = sin(p)
w =
    0.8415    0.9093    0.1411
```

Functions of vectors are applied term by term, resulting in a vector of the same size. The result is $z_i = \sin(x_i)$ for each element, i.

```
»
» a = sin(p).*cos(p)
a =
    0.4546   -0.3784   -0.1397
```

Since functions of vectors are also vectors, functions of vectors can be multiplied term by term also. The result is $z_i = \sin(x_i)\cos(x_i)$ for each element, i.

```
»
» c = linspace(1,9,5)
c =
    1     3     5     7     9
```

A vector with 5 **lin**early **space**d elements between 1 and 9 is easily created using the MATLAB function `linspace`.

```
»
» d = 1:5
d =
    1     2     3     4     5
```

Contrast `linspace` with this method of creating a vector. How do they differ?

```
»
» e = 3*sin(2*p).*log10(t.*u)
e =
    1.3015   -3.0020   -1.4589
```

Complex functions and arithmetic operations with vectors are possible. The result of this one is the same as $e_i = 3\sin(2p_i)\log_{10}(t_i * u_i)$.

```
» diary off
» quit
```

Exercises: Perform the following operations using array arithmetic where appropriate.

1. *Equation of a line*. The equation of a line is given by $y = mx + b$ where m is the slope (a scalar) and b is the intercept (also a scalar).

   a. Compute the corresponding y-coordinates for the following x-coordinates if m = 3 and b = -1
   ```
   x = [1, 2, 3, 5, 8, 13, 21]
   ```

   b. Compute the corresponding x-coordinates for the following y-coordinates if m = 5 and b = 2.3
   ```
   y = [0, 1, 1, 2, 3, 5, 8]
   ```

   c. Compute the corresponding y-coordinates for the following x-coordinates if m = π/2 and b = 2/π
   ```
   x = linspace(0,30,8)
   ```

   d. Given the equation $d = 3\sin(t) + t/2$, compute the corresponding d-coordinates for
   ```
   t = 0:10
   ```

2. *Vector multiplication, division, exponentiation*. Create a vector, **g**, with 10 evenly spaced elements starting at 1 and ending at 10. Compute the following with vector operations:

   a. $h = g\cos(g)$

   b. $z = \dfrac{g-1}{h+1}$

   c. $s = \dfrac{\cos(g^2)}{hg}$ (try this by squaring **g** and then by multiplying **g** times **g**)

   d. Given the equation $q = 5\sin(\pi t / 2.5)e^{-t/2}$, compute the corresponding q-coordinates for
   ```
   t = 0:100
   ```

3. *Parametric equation for a circle*. The parametric equation for a circle is $x = r\cos(\theta)$ and $y = r\sin(\theta)$ where r is the radius and θ is the angle of rotation counter-clockwise from the positive x-axis. Defined this way, x and y satisfy the equation $x^2 + y^2 = r^2$. Show this using MATLAB. Use `linspace` to create an angle vector, **theta**, with values (0, π/3, 2π/3, π, 4π/3, 5π/3, 2π). Compute the corresponding **x**- and **y**-vectors for r = 5. Show that the **x**- and **y**-vectors satisfy the equation of a circle.

Recap: You should have learned
- How to declare a vector
- How to declare a vector with evenly spaced elements (two methods)

- Arithmetic operations between a scalar and a vector
- Arithmetic operations between two vectors
- Simple function operations with a vector
- Arithmetic operations between functions of vectors

MATLAB Workshop 5

Objectives: Create a simple graph using MATLAB, save your workspace, create a script for plots.

Frequently, we want to visually examine the behavior of a function (equation) or look at a graphical display of data that we have acquired. MATLAB provides some useful plotting/graphing tools for this purpose.

MATLAB plot functions

`plot(x,y,s)`	Plot with linear (x,y) axes; line type specified by string `s`
`semilogy(x,y,s)`	Plot with linear x, logarithmic y axes; line type specified by string `s`
`semilogx(x,y,s)`	Plot with logarithmic x, linear y axes; line type specified by string `s`
`loglog(x,y,s)`	Plot with logarithmic (x,y) axes; line type specified by string `s`

Example 1: Plotting a simple graph

- **Start MATLAB and perform the following**

```
» cd  your MATLAB directory
» diary  wkshop5a.txt
» header
     Display of your header output
»
» x = linspace(0,4*pi,1000);
» y = sin(x);
» plot(x,y)
```

Create 1000 evenly spaced (x,y) coordinates in the range [0,4π]. `plot` launches the *Graphics Window*, showing an (x,y) plot of the vectors, with default scaling and limits on the axis.

```
»
» xlabel('x')
» ylabel('y')
» title('plot of sin(x) vs x')
```

Basic annotation of your plot.

```
»
» diary off
```

Congratulations! You have just created your first graph of a function using MATLAB. Simple graphs are easy! How about plotting some data? Unlike an equation or function, which we want to display as a continuous line, data needs to be displayed as discrete points.

Example 2: Plotting data

An environmental engineer has obtained the laboratory measurements shown in the table below for settling of solids in a holding pond. As a first step in analyzing the information, she would like to display the data in a simple graph.

Solids settling: agglomeration as a function of time										
Time, min	1	5	10	15	20	1	5	10	15	20
Mass, kg	0.12	4	16	33	61	0.19	3.5	14	35	58

- **Back in MATLAB**

```
» diary  wkshop5b.txt
» header
     Display of your header output
»
» time = [1,5,10,15,20,1,5,10,15,20];
» mass = [0.12,4.0,16,33,61,0.19,3.5,14,35,58];
» plot(time,mass,'*')
```

Enter the data as vectors (note the meaningful names). This time, specify the type of "line" `plot` uses. What must the default line type for `plot` be? **Note**: your prior graph has been replaced!!!

```
»
» xlabel('time, min')
» ylabel('mass, kg')
» title('mass settled as a function of time')
```

Annotate your plot.

Example 3: Saving your workspace

You are now starting to invest some "effort" in creating a MATLAB solution to your problem. What happens if you need to leave the computer for a while? Do you need to reenter all your data from scratch before proceeding? The answer is no. When you come back to the computer, you could edit your diary file and run it as a script. Another option is to use the `save` command.

- **Back in MATLAB**

```
» who
Your variables are:
mass      time      x         y
```

Check to see what variables are active in your workspace.

```
»
» save agglom_data
» diary off
» quit
```

The `save` command saved your variables and values in your current directory in a *binary* file called agglom_data.mat . Check to see that the file is there. If you were to open it with a text editor, such as Notepad, you would see garbage. That is because the information is stored in a binary, not text, format. However, MATLAB knows how to read the file.

- **Restart MATLAB and perform the following**

```
» cd  your MATLAB directory
```

```
» load agglom_data
» who
Your variables are:
mass      time      x         y
```

MATLAB has reloaded your workspace variables and values. Check to see that the values are there by typing `mass` or `time` at the command prompt. Note that the "text" associated with your session has not been saved - only the variables and their values. If you want the text, you need to review your session diary.

Example 4: Creating a script for graphs

The need to visualize functions and data occurs so frequently that having a script file available to accomplish the task would be advantageous. The problem is that vectors, axis labels, and plot titles are unique to a given plot - that is, they change with each plot. How can a single script file handle such varied needs? By asking the user!

- **Back in MATLAB**

```
» diary wkshop5c
» edit
```

When the MATLAB Editor pops up, enter the following script file

```
% myplot - create an (x,y) plot with labels and title
% created by your name.  last modified today's date.

% get plot information
   disp('  ')
   xvector = input('what is the x-vector? ==> ');
   yvector = input('what is the y-vector? ==> ');
   xlabelname = input('what label for the x-axis? ==> ','s');
   ylabelname = input('what label for the y-axis? ==> ','s');
   titlename = input('what title? ==> ','s');
   disp('what line type?')
   disp('   1 = solid')
   disp('   2 = dashed')
   disp('   3 = *' )
   linetype = input('line type? ==> ');

% assign line type
   if  linetype == 1
      plotline = '-';
   elseif  linetype == 2
      plotline = '--';
   else
      plotline = '*';
   end

% create plot
   plot(xvector,yvector,plotline)
   xlabel(xlabelname)
   ylabel(ylabelname)
```

```
title(titlename)
```

Note the indenting and use of white space in creating this file. Comments head sections of commands that are indented under the comment. Using white space and comments make your script files easy to read, understand, and edit, if necessary. The script file uses some "programming language" constructs, e.g., the *if...elseif* construct, that you will learn more about. It also uses a MATLAB specific function, `input`, that allows the user to define exactly what vectors and labels to use in the plot. Save the script file as myplot.m .

Before continuing, close the MATLAB Graphics Window (click the X in the upper right corner).

- **Back in MATLAB**

```
» help myplot
myplot - create an (x,y) plot with labels and title
created by your name.  last modified today's date.
```

`help filename` will display the beginning comment lines in a file.

```
»
» myplot
what is the x-vector? ==> time
what is the y-vector? ==> mass
what label for the x-axis? ==> time, min
what label for the y-axis? ==> mass, kg
what title? ==> agglomeration mass vs time
what line type?
  1 = solid
  2 = dashed
  3 = *
line type? ==> 3
```

The MATLAB graphics window reappears with the desired graph.

```
»
» diary off
```

If you made errors in creating the script, MATLAB will respond with (ambiguous) error messages. Simply edit the script file with the text editor, resave it, and then rerun the file.

Exercises: Perform the following operations using array arithmetic where appropriate.

1. *Parametric equation for a circle*. The parametric equation for a circle is $x = r\cos(\theta)$ and $y = r\sin(\theta)$ where r is the radius and θ is the angle of rotation counter-clockwise from the positive x-axis. Use `linspace` to create an angle vector, **theta**, with 200 equally spaced values in the range (0, 2π). Compute the corresponding **x**- and **y**-vectors for r = 2. Use `myplot` to display the resulting (x,y) plot.

   Doesn't look much like a circle does it? Why? (Hint: look at the scales on the x & y axes.) Correct the problem by typing the command

   ```
   axis('equal')
   ```

This forces the scales on the x and y axes to be the same.

2. Frequently, plots of data do not yield straight lines (what shape "curve" did you get from the agglomeration vs time data?). The reason we plot data is to obtain a visual idea of the relation between the dependent and independent variables. The table at the right identifies which relations have straight-line plots with the corresponding MATLAB function.

Relation	MATLAB plot
$y = mx + b$	`plot`
$y = ae^{bx}$	`semilogy`
$y = a\ln(bx)$	`semilogx`
$y = ax^b$	`loglog`

   a. Create a script file called `mysemilogy` that will plot data with a linearly-scaled x-axis and logarithmically-scaled y-axis. (Hint: edit `myplot` appropriately and save it with the new name.)
   b. Create a script file called `mysemilogx` that will plot data with a logarithmically-scaled x-axis and linearly-scaled y-axis.
   c. Create a script file called `myloglog` that will plot data with a logarithmically-scaled x- and y-axes.
   d. Plot the agglomeration vs time data with each of these functions. Which one produces a straight-line plot? What is the functional relationship?

3. Plot each of the following relations first with `myplot` and then with the appropriate function to obtain a straight-line result.
   a. $y = 1.25e^{-x/25}$ $(0 \le x \le 100)$
   b. $x = 1.25e^{2.5y}$ $(10 \le x \le 50)$
   c. $y = 2x^{1.5}$ $(0 < x \le 100)$
   d. $y = 3x^2 - 2x + 1$ $(-10 \le x \le 10)$
   e. $y = 2\sin(\pi x/2)e^{-2x}$ $(0 \le x \le 10)$

4. *Multiple (overlay) plots*. Display $h = \cos(1.5\pi t/2)$ and $g = \cos(1.5\pi t/2)e^{-0.15t}$ in the range $(0 \le t \le 10)$ on the same plot. Hint: use `myplot` to plot the first followed by the command `hold on` followed by `plot` for the second. You can learn more advanced plotting techniques by referring to MATLAB Basic Graphics.

5. As part of a mechanical engineering laboratory, you obtained the following stress-strain data on a new material. Use MATLAB to plot the data with a green solid line connecting red triangles.

Strain, (in/in)	Load, lb_f	Strain, (in/in)	Load, lb_f	Strain, (in/in)	Load, lb_f
0.0002	900	0.0024	7350	0.020	9150
0.0004	1600	0.0028	7850	0.040	8950
0.0006	2350	0.0032	8200	0.060	8100
0.0008	3150	0.0036	8400	0.080	7250
0.0010	3900	0.0040	8600	0.10	6300
0.0012	4650	0.0044	8700	0.12	5150

0.0014	5200	0.0048	8800	0.14	3950
0.0016	5750	0.0052	8900	0.16	2100
0.0018	6300	0.0070	9100	0.17	500
0.0020	6700	0.0100	9200	Fracture	

Recap: You should have learned

- How to create simple (x,y), semilogarithmic, and log-log plots.
- How to save and load your workspace.
- How to create a more complex script with screen display and input capability.

MATLAB Workshop 6

Objectives: Understand user-defined MATLAB functions. Use functions for linear regression. Plot graph with data shown as points and a "best fit" line from linear regression..

User-defined MATLAB functions

Although MATLAB has a large variety of useful mathematical, matrix manipulation, and graphics visualization functions, we frequently need to design our own function for a special purpose. MATLAB provides the capability to do so through its user-defined function.

The purpose of a user-defined function is to compute one or more values that are required for solving your problem. Contrast this with a user-defined *script*, which is a set of instructions to accomplish a task. Functions are much more restricted in scope because they are designed only to compute a value(s) - not accomplish a task.

The generic format for the first line of a MATLAB function file (a .m file), i.e., the function header, is

```
function [out1, out2, out3, ...] = fcn_name(in1, in2, ...);
```

where `out1`, `out2`, `out3`, etc, are the *output values* begin calculated by the function. The output values can be *scalars*, *vectors*, or *matrices*. `in1`, `in2`, etc, are *input values* that are required by the function to compute the output values. The input values can also be *scalars*, *vectors*, or *matrices*. `function` must be lower case; it tells MATLAB that this is indeed a function. The function is saved in a file named fcn_name.m.

The lines immediately following the function header are comment lines, started with the comment delimiter `%`, that are displayed when `help fcn_name` is input in the Command Window. These lines should describe what the function computes, the meaning of the output values list, and the required input values.

After a blank line, the remainder of the function script, properly annotated, is directed toward computing the desired values. Any variable named in the function header is available for use. Additional variable names can be used for intermediate steps in the calculations. However, these are *local variables* that are not available back in the command window. The only values that return to the command window are those that are in the output values list.

Functions are accessed from the command window by the statement

```
[myout1, myout2, myout3, ...] = fcn_name(myin1, myin2, ...);
```

where `myout1`, `myout2`, `myout3`, etc correspond to the names you are using for the output values and `myin1`, `myin2`, etc, are the names you are using for the input values. This means that you can use meaningful names for your current problem in the command window and MATLAB will take care to make sure that values are calculated properly.

Example 1: Least-squares, linear regression data analysis

A common activity in developing understanding of experimental results is the use of *linear regression* to find a "best fit" relationship between the independent and dependent variables in a data

set. Given a data set of *n* experimental points, (x_i,y_i), $1 \le i \le n$, the *least squares, linear regression* algorithm seeks to find the straight-line fit to the data set that minimizes the sum of the square of the vertical distances between the data and the straight-line given by

$$y = mx + b\,.$$

The resulting equations for the slope, m, and the intercept, b, are

$$m = \frac{n\sum_{i=1}^{n} x_i y_i - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2} \quad \text{and} \quad b = \bar{y} - \left(\frac{m}{n}\right)\sum_{i=1}^{n} x_i \quad \text{where} \quad \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

The correlation coefficient, r^2, given by

$$r^2 = 1 - \frac{SSE}{SST} \quad \text{where} \quad SSE = \sum_{i=1}^{n} (y_i - mx_i - b)^2 \quad \text{and} \quad SST = \sum_{i=1}^{n} (y_i - \bar{y})^2$$

is another commonly used statistic. The correlation coefficient represents the proportion of total variability in the data that is explained (accounted for) by the linear fit to the data.

The following MATLAB function implements the linear regression calculations

```
function  [m,b,rsq] = linreg(x,y);
% linreg - [m,b,rsq] = linreg(x,y)
% determine straight-line fit
%           y = mx + b
% and correlation coefficient, rsq, for
% input vectors (x,y)

% local variable definitions
%   SX is sum(x), SY is sum(y), SXY is sum(xy)
%   SXsq is sum(xsq), Yave is SY/n
%   SSE is sum(y-mx-b)_sq, SST is sum(y-yave)

% determine m and b
   n = length(x);
   SX = sum(x);
   SY = sum(y);
   SXY = sum(x.*y);
   SXsq = sum(x.*x);
   Yave = SY/n;

   m = (n*SXY-SX*SY)/(n*SXsq-SX*SX);
   b = Yave-(m/n)*SX;

% determine rsq
   SSE = sum((y-m*x-b).^2);
   SST = sum((y-Yave).^2);
```

```
    rsq = 1 - SSE/SST;
```

The data vectors, **x** and **y**, are the required input data. The function returns **m**, **b**, and **rsq**. The first few lines following the `function` definition are comments that describe the function (note they describe both input data and returned data). These will be displayed when the `help linreg` command is used. After a blank line, local variables are defined in comments. Finally, the calculations are performed, with appropriate comments. **Note:** Good programming practice is to make liberal use of comments to identify both variables and actions. Use plenty of whitespace. This will prove invaluable to understanding what you meant a function to do when you try to use it months after creating it.

An engineer obtained the (time, height) data shown below for the height of a column of water from duplicate experiments. The engineer wants to create a figure showing the data together with a linear regression, "best-fit" line for the data.

Column Height as a function of time

Time, min	0	1	3	5	10	15	20	0	1	3	5	10	15	20
Height, cm	0.0	3.2	8.6	15.6	31.2	43.3	57.6	0.0	2.8	9.4	14.1	28.7	46.2	61.1

- **Start MATLAB and perform the following**

```
» cd  your MATLAB directory
» diary  wkshop6a.txt
» header
```

Display of your header output

```
»
» time = [ enter time data here ];
» height = [ enter height data here ];
» plot(time,height,'*')
» hold on
» edit
```

Enter data, create basic plot, hold plot. When Edit Window appears, enter **`function linreg`** as shown above. Save function as **`linreq.m`** in your MATLAB directory before continuing.

```
»
» help linreg
linreg - [m,b,rsq] = linreg(x,y)
determine straight-line fit
          y = mx + b
and correlation coefficient, rsq, for
input vectors (x,y)
```

Note that, by placing the "calling" in the definition, you can see how to "call" the function (if you forgot how) simply by using the help command.

```
»
» [slope, intcpt, cor] = linreg(time,height)
slope =
        2.9873
intcpt =
        -0.0023
cor =
       0.9983
```

Call `linreg`. Note that you can use variable names of your choice as input and output (you do not need to use the same names as in the definition. MATLAB will make a one-to-one correspondence between the names you use and the corresponding use in the function. Placing a suppress output display marker, `;`, at the end of the call would suppress the display (the values would still be calculated and available by variable name).

```
»
» fit = slope*time + intcpt;
» plot(time,fit,'-')
» legend('data','regression fit')
```

Add the regression line to the plot. Add a legend. You can now use the menus in the graphics window to add axis labels, use the text command to add the equation and rsq value, and adjust font displays.

```
»
» diary off
» quit
```

Example 2: Least squares regression of non-linear functions

Functions can call other functions to help do their job. For example, we want to find the "best fit" for data that seems to follow the exponential relation

$y = Ae^{By}$ which can be transformed to $\ln(y) = Bx + \ln(A)$

by taking logarithms of each side. This corresponds to the linear equation $y = mx + b$ if we substitute ln(y) for y, B for m, and ln(A) for b. Thus, we should be able to design a function, `semilogy_fit`, that will input data for x and y and return the "best-fit" values for A and B. Such a function is listed here.

```
function  [A,B,rsq] = semilogy_fit(x,y);
% semilogy_fit - [A,B,rsq] = semilogy_fit(x,y)
% determine A and B parameters for semilog fit to
%      y = A*exp(Bx)
% and correlation coefficient, rsq, for
% input vectors (x,y)

% local variable definitions
%   log_y = log(y)

% algorithm
   log_y = log(y);
   [B,lnA,rsq] = linreg(x,log_y);
   A = exp(lnA);
```

Note that the log transformation (taking logs of the input y data) is done inside the function before calling `linreg` to find the best-fit values. `linreg` returns the value of B directly. However, it returns the value of ln(A) (why?), so that that value needs to be transformed back to A before `semilogy_fit` is finished.

Exercises: Perform the following operations using array arithmetic where appropriate.

1. *Power law data fit*. Sometimes data is best represented by a power law equation of the form

   $y = Ax^B$ which can be transformed to $\ln(y) = B\ln(x) + \ln(A)$

   Design a function, `loglog_fit`, that will compute the best-fit values for A and B given an (x,y) data set.

2. *Exponential (x) data fit.* Sometimes data is best represented by a power law equation of the form

$$x = Ae^{By} \quad \text{which can be transformed to} \quad \ln(x) = B\,y + \ln(A)$$

Design a function, `semilogx_fit`, that will compute the best-fit values for A and B given an (x,y) data set.

3. An environmental engineer has obtained the laboratory measurements shown at the right for settling of solids in a holding pond. Use your graphing and curve fitting capabilities to find a reasonable "best-fit" for the data. You should create a well-annotated graph, with appropriate legend, that shows the data and best fit to the data. Put the best fit equation and rsq values on the graph.

Agglomeration as a function of time

Time, min	1	5	10	15	20	1	5	10	15	20
Mass, kg	0.12	4	16	33	61	0.19	3.5	14	35	58

Hint: make a basic plot of the data first. Use the axes properties menu in the graphics window to change the axes type (linear or log) to find which equation appears to best represent the data before using your curve fitting routines.

4. An bioengineer has obtained the data shown below for bacterial growth in a culture. Use your graphing and curve fitting capabilities to find a reasonable "best-fit" for the data. You should create a well-annotated graph, with appropriate legend, that shows the data and best fit to the data. Put the best fit equation and rsq values on the graph.

Bacterial count as a function of time

Time, hr	1	3	5	7	9	12	16	20	30	50
Count, 10^3/mL	1	1.4	1.5	1.9	2	2.5	3.6	4.9	11.1	54.4

Hint: make a basic plot of the data first. Use the axes properties menu in the graphics window to change the axes type (linear or log) to find which equation appears to best represent the data before using your curve fitting routines.

5. The Stefan-Boltzmann radiation law states that the energy radiation flux, R (J/s), from a hot object varies as

$$R = \sigma A(T^4 - T_o^4)$$

where σ is the Stefan-Boltzmann constant, A (m^2) is the object surface area, T (K) is the object temperature, and T_o (K) is the surrounding (room) temperature. An engineer collected the data at the right for a 0.01 m^2 object. Use the data to determine a value for σ and the surrounding room temperature. (The accepted value for σ is $5.670 \pm 0.003 \cdot 10^{-8}$ W/(K•m^2).)

Energy radiated as a function of temperature

T, K	300	350	400	450	500	550
R, W	0.4	4.3	10.5	19.2	31.5	47.5

Recap: You should have learned

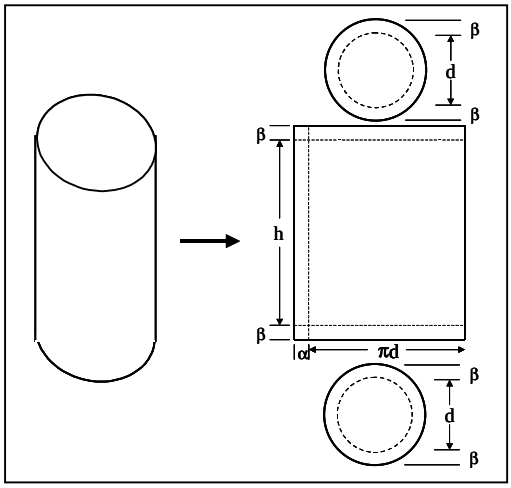
- How to define functions in MATLAB.
- How to “call” functions in MATLAB.
- How to use linear regression in MATLAB to find the “best fit” for data.
- How to create a figure showing data and the best fit curve in MATLAB.

MATLAB Workshop 7

Objectives: Use MATLAB for functional analysis - graphing of equations, finding minima and maxima, finding roots of equations, solving first order ordinary differential equations.

Example 1: Functional Analysis - Minimum or Maximum

Consider the common tin can, as illustrated at the right. The tin can is comprised of two circular ends capping a right circular cylinder. The individual pieces are made from flat sheet materials as shown in the exploded view. α represents the additional material needed to make the vertical seam on the right cylinder. β represents the additional material needed to make the end welds or seals. The amount of material needed to make a can is given by the equation



$$M = (\pi d + \alpha)(h + 2\beta)(\rho t) + 2\pi\,(\frac{d}{2} + \beta)^2(\rho t)$$

where

M	:	mass of material, kg
d	:	can diameter, m
α	:	excess for vertical seam, m
h	:	can height, m
β	:	excess for end seams, m
ρ	:	material density, kg m^{-3}
t	:	material thickness, m

The first term on the right represents the mass of the right cylinder section. The second term represents the contribution of the end caps (hence the factor 2).

The objective in can design is to use the minimum amount of material to enclose a specified volume. The volume enclosed by a right circular cylinder is

$$V = \frac{\pi d^2 h}{4}$$

Substituting this into the equation for M yields

$$M = (\pi d + \alpha)(\frac{4V}{\pi d^2} + 2\beta)(\rho t) + 2\pi\,(\frac{d}{2} + \beta)^2(\rho t)$$

which is an equation in d for a specified M.

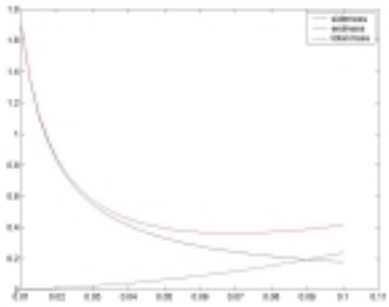
What is the minimum mass of wrought aluminum alloy (ρ = 2800 kg/m^3) needed to make a can from a sheet 0.5 mm thick that holds 285 mL fluid. What are the corresponding diameter and height of the can? Assume α = 0.2 cm and β = 0.2 cm.

The first question is what are the relative contributions of the side and two ends to the total mass. To answer this visually, use MATLAB to create a graph with three lines. The first is the total mass of the can, the second is the mass of the side, the third is the mass of the two ends. Consider the following script

- **Start MATLAB and perform the following**

```
» cd  your MATLAB directory
» diary  wkshop7a.txt
» header
```
Display of your header output
```
»
» % set parameters - make sure all SI units
»    rho = 2800; vol = 285e-6; thick = 0.005;
»    alpha = 0.002; beta = 0.002;
»
» % use diameter as independent variable - range from experience
»    diam = 0.01:0.0025:0.10;
»    sidemass = ???; (enter appropriate equation - first term of total mass eqn)
»    endmass = ???; (enter appropriate equation - second term of total mass eqn)
»    totmass = sidemass + endmass;
»
» % create plot
»    graph = [sidemass; endmass; totmass];
»    plot( diam, graph )
»    legend( 'sidemass', 'endmass', 'total mass' )
```

This will create a figure similar to the one at the right. Perusal of the figure shows that the side mass is a decreasing function of diameter and that the end mass is an increasing function of diameter. The total mass, which is the sum of the two, has a minimum. So, the challenge now is to find the minimum. MATLAB provides the capability to search the vectors for the minimum or maximum and to determine which element holds the minimum or maximum. Use `help min` or `help max` to learn more about this capability.



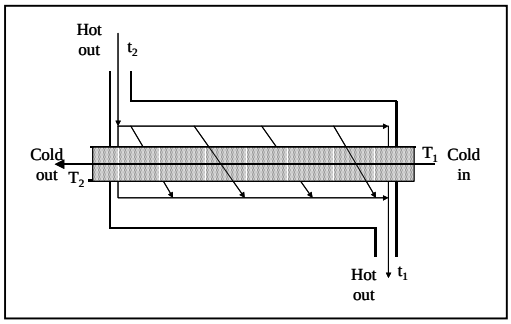
- **Back in MATLAB**

```
» % find minimum mass (kg) & index
»    [minmass,loc] = min(totmass)
minmass =
    0.3629
loc =
    25
»
» % determine diameter (m) at minmass
»    min_d = diam(loc)
min_d =
    0.0700
»
» % calculate height (m)
»    height = (4*vol)/(pi*min_d^2)
height =
    0.0741
» diary off
```

MATLAB has several functions such as `min` and `max` that will search a vector for specified elements. returns from the function can include both a value and index (location) of the value.

Example 2: Functional Analysis - Roots of Equations

A counter-current heat exchanger that consists of a tube inserted inside another tube (called the shell) is a common industrial means to exchange heat between two process streams. As depicted in the figure, a "cold" fluid, at temperature T_1, enters the central tube at one end of the heat exchanger. A "hot" fluid, at temperature t_2, enters the shell-side at the opposite end of the heat exchanger.



As the cold fluid flows through the inner tube, it gains heat from the hot fluid by convective and conductive heat transfer mechanisms. The cold fluid then exits the heat exchanger at a higher temperature, T_2. The hot fluid, which has lost some heat, exits the heat exchanger at temperature t_1. Because the two fluids are flowing in opposite directions through the heat exchanger, it is called a counter-current heat exchanger.

The design equation for counter-current heat exchange is

$$q = UA\left[\frac{\Delta T_2 - \Delta T_1}{\ln(\Delta T_2 / \Delta T_1)}\right]$$

where

q : heat transfer rate, Js^{-1}
U : overall heat transfer coefficient, $Jm^{-2}s^{-1}C^{-1}$
A : outer area of inner tube, m^2
ΔT_1 : (t_1-T_1), temperature difference at cold fluid entrance, C
ΔT_2 : (t_2-T_2), temperature difference at hot fluid entrance, C

A process engineer needs to reduce the temperature of a "hot" treated waste water stream from 85 °C to 30 °C prior to discharge from the plant. The heat transfer rate required to achieve this reduction is 10,000 Js^{-1}. Cooling water is available at 20 °C. The engineer has found an idle heat exchanger that can be made available for her use. The heat exchanger design equation for the process stream combination is

$$q = 650\left[\frac{\Delta T_2 - \Delta T_1}{\ln(\Delta T_2 / \Delta T_1)}\right]$$

Can the heat exchanger be used for this purpose and, if yes, what is the cold stream exit temperature?

Since three of the four temperatures are known, we are seeking to determine whether there is some temperature for T_2 between 20 and 84 °C that provides 10,000 Js^{-1} heat removal. The lower bound, 20 °C, corresponds to so much cooling water moving through the heat exchanger that its temperature remains essentially unchanged despite gaining heat from the hot stream. The upper bound, 84 °C, corresponds to the cooling stream coming into near thermal equilibrium with the entering hot stream. Why not use 85 °C?

- **Back in MATLAB**

```
» diary wkshop7b.txt
» header
```

Display of your header output

```
»
» % set parameters - make sure all SI units
»    T1 = 20; t1 = 30; t2 = 85; UA = 650; DelT1 = t1-T1;
»
» % define heat exchange function - use inline
»    heatex = inline('UA*((t2-T2)-DelT1)/log((t2-T2)/DelT1)', ...
       'T2','UA','t2','DelT1');
heatex =
     Inline function:
     heatex(T2,UA,t2,DelT1) = UA*((t2-T2)-DelT1)/log((t2-T2)/DelT1)
```

The `inline` function can be used to create a function in the current workspace rather than write a function and save it as an .m file. Note that the function and its arguments (parameters) are all contained in single quotes (strings). The independent variable is the first argument listed. The other arguments can be in any order. MATLAB responds by showing the function definition. Learn more by typing `help inline` on the command line.

```
»
» % set up graph
»    T2range = 20:84;
»    i = 1;
»      for T2=20:84
       q(i) = heatex(T2, UA, t2, DelT1);
       i = i+1;
     end
```

Example of using a `for` loop to create a vector. Run it without the `;` to see term-by-term calculations. The "call" to `heatex` looks identical to the call that would be made if `heatex` were a function in a .m file rather than one defined by the `inline` function.

```
Warning: Divide by zero.
> In D:\MATLABR11\toolbox\matlab\funfun\@inline\subsref.m at line 25
```

A warning about division by zero. WHY?

```
»
» % create plot
»    plot(T2range,q)
»    xlabel('Cold Outlet Temp, C')
»    ylabel('Heat flow, J/s)'
»
```

Looking at the graph, the heat exchanger is capable of removing between about 3,000 and 19,000 Js^{-1} for this process. So, the question now becomes, what is the outlet cold temperature for the process if it removes 10,000 Js^{-1}? That is , we are seeking the solution to

$$q = 650\left[\frac{\Delta T_2 - \Delta T_1}{\ln(\Delta T_2 / \Delta T_1)}\right] - 10000 = 0$$

Before proceeding, use your favorite text editor to create the following *residual* function

```
function [resid] = qfcn(T2,UA,t2,DelT1,heatflow);
% qfcn evaluates residual of counter-current heat exchange equation
%    resid = UA*((t2-T2)-DelT1)/log((t2-T2)/DelT1) - heatflow
% requires T2, UA, t2, DelT1, heatflow
% returns residual
```

```
% calculate residual
   resid = UA*((t2-T2)-DelT1)/log((t2-T2)/DelT1) - heatflow;
```

and save it as qfcn.m in your MATLAB directory.

- **Back in MATLAB**

```
» % find root of equation using .m file function
»    T2out = fzero('qfcn',50,optimset,UA,t2,DelT1,10000)
Zero found in the interval: [34, 66].
T2out =
    62.5767
```

The root of the equation **f(x) = 0** is found with `fzero`. The arguments inside the parentheses are, in order, the function name (enclosed in single quotes if an .m file), an initial guess for the root, `optimset`, and the parameters required by the function, in the same order as the function definition. **Note**: if the function has no parameters, e.g. f(x) = sin(x) - x, a simpler form of `fzero` can be used:
`root = fzero(fcn_name, initial_guess)`.
More information can be found by typing `help inline` at the command line.

```
»
» % find root of equation using inline function definition
»  qfcn2 = inline('UA*((t2-T2)-DelT1)/log((t2-T2)/DelT1)- heatflow',...
    'T2','UA','t2','DelT1','heatflow')
qfcn2 =
     Inline function:
     qfcn2(T2,UA,t2,DelT1,heatflow) = UA*((t2-T2)-DelT1)/log((t2-
           T2)/DelT1)-heatflow
»
»    T2out = fzero(qfcn2,50,optimset,UA,t2,DelT1,10000)
Zero found in the interval: [34, 66].
T2out =
    62.5767
```

If an `inline` function is used, the function name is not enclosed in quotes.

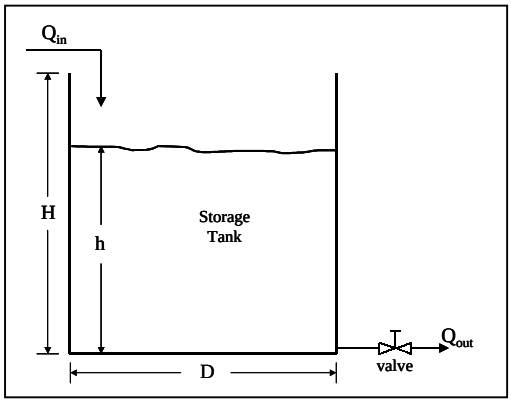
```
» diary off
```

Roots of equations are found by putting the equation into the form

$$f(x) = 0$$

and searching for the value of x that makes the equation true. The MATLAB function `fzero` can be used. `fzero` requires that you have an initial guess for the location of the root. The initial guess is most accurate if you have a graph of the equation and can see about where it crosses the axis. This technique can be extended to finding minimums and maximums of equations because of the relation

$$\frac{df}{dx} = 0 \qquad \text{at a minimum or maximum.}$$



Example 3: ODE integration

The diagram to the right depicts a common industrial situation that involves flow into and out of a storage tank. The storage tank has diameter D (m) and height H (m). The

fluid level inside the tank is h (m). Fluid is flowing into the tank at rate Q_{in} (m^3/s). Fluid is flowing out of the tank through an outlet valve at rate Q_{out} (m^3/s). The question is what is the fluid level, h, as a function of time?

Application of the principle of conservation of mass produces the ordinary differential equation

$$\frac{dh}{dt} = \frac{Q_{in} - C_v \rho g h}{A} \qquad i.c.: \; h = h_0 \;\; at \;\; t = t_0$$

which describes the height as a function of time. C_v is the valve coefficient which describes flow through the valve, ρ is the fluid density (kg/m^3), g is gravitational acceleration (m/s^2), and A is the cross-sectional area of the tank (m^2).

MATLAB can be used to solve the equation and find the height as a function of time in the following manner.

1) Write a function that returns the derivative as a function of time and height (independent and dependent variables). Note that both the independent and independent variables, in that order, must be in the argument list. The following script is suitable for the present problem (it should be saved as tankflow.m in your MATLAB directory):

```
function dhdt = tankflow(t,h);
% tankflow: flow into and out of a tank
%    dhdt = (Qin - Cv*rho*g*h)/A

% specify parameters
   Qin = 0.25;  Cv = 3.2e-6; rho = 1000;
   g = 9.8; A = pi*6*6;

% calculate derivative
   dhdt = (Qin - Cv*rho*g*h)/A;
```

2) Select one of the many MATLAB functions for integration (`ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`). MATLAB has functions that implement different numerical approaches to ODE integration. The choice of which function to use will depend upon particular aspects of your problem when rendered into a numerical approach. All of the functions require specification of the function name (as text or string), a vector containing the initial and final values for the independent variable, and the initial value of the dependent variable. The following script can be entered in the command window:

- **Back in MATLAB**

```
» diary wkshop7c.txt
» header
     Display of your header output
»
» % specify independent var range, init value
»    tspan = [0 15000]; h0 = 6;
»    [t,h] = ode23('tankflow',tspan,h0);
```

`ode23` will generate the independent variable vector **t** and through integration, the corresponding values of height in the vector **h**. Omitting the display suppression command on the last statement will cause the **t** and **h** vectors to display.

3) Interpret the results. This is most easily done graphically, e.g.,

```
»    plot(t,h);
```

You can annotate the graph using methods described in the MATLAB Graphics section.

You should see what differences, if any, result from using one of the other ODE integration functions. If you simply want to see a graph of the integrated function *without* capturing the independent and dependent variable information, you could simply type

```
» % specify independent var range, init value
»    tspan = [0 15000]; h0 = 6;
»    ode23('tankflow',tspan,h0);
```

at the command line (leave off the `[t,h]` part). A graph showing the integrated function will appear with the integration points highlighted.

Exercises

1. *Problem Statement*: A sealed storage tank is partially filled with a solution of polymer in a solvent. The vapor pressure inside the tank is measured to be 5 atm (absolute). The pure solvent exerts a vapor pressure of 6 atm (absolute), so the polymer is acting to depress the pressure. Laboratory experiments have found the following relation between the volume fraction of polymer, x, and the vapor pressure of the solvent, P,

$$\ln(\frac{P}{P_0}) = \ln(1-x) + x + 0.4x^2$$

where P_0 is the vapor pressure of the pure solvent. (Note: volume fraction is defined as (volume of polymer)/(volume of polymer+volume of solvent), dimensionless).

What is the volume fraction of polymer in the tank.

Background: This is a simple root finding problem for a value of x given (P/P_0). No unit conversions are necessary.

Solution strategy:

Specify pressure ratio parameter (P/P_0).

Rearrange equation to $f(x) = \ln(\frac{P}{P_0}) - \ln(1-x) - x - 0.4x^2 = 0$

Create a graph of $f(x)$.

Use `fzero` to find x.

2. *Problem Statement*: Fluid flow in a pipe of length, *l* (m), and diameter, d (m), is characterized by two regimes: laminar flow, Re < 2000, and turbulent flow, Re > 2000, where Re is the Reynolds number defined as

$$\mathrm{Re} = \frac{d\,v\rho}{\mu}$$

Re is dimensionless, v is the fluid velocity (m/s), ρ is the fluid density (kg/m^3), and μ is the fluid viscosity (Pa·s). In turbulent flow, the equation

$$\frac{1}{\sqrt{f}} = 2\log_{10}(\mathrm{Re}\sqrt{f}) - 0.802$$

defines the relation between the friction factor, f, and Re. The friction factor is defined as

$$f = \frac{d\,\Delta P}{\frac{1}{2}\rho\, v^2\, l}$$

where ΔP (Pa) is the pressure drop from the beginning to the end of the pipe.

Use the friction factor to find the pressure drop per unit length, Δ*P*/*l*, for water flowing through a 5 cm diameter pipe at 5 m/s. (ρ_{water} = 1000 kg/m^3; μ_{water} = 0.001 Pa•s).

Background: This is a root finding problem for a value of f given Re. Since units are metric, no unit conversions other than diameter (0.05 m) are necessary.

Solution strategy:

Specify parameters and givens
Calculate Re
Rearrange friction factor equation to $f(f) = \frac{1}{\sqrt{f}} - 2\log_{10}(\mathrm{Re}\sqrt{f}) - 0.802 = 0$
Create a graph of $f(f)$.
Use `fzero` to find f.

3. The van der Waals equation of state (more accurate than the ideal gas law) is given by

$$(P + \frac{a}{\overline{V}^2})(\overline{V} - b) = RT$$

where P is the absolute pressure (Pa), $\overline{V}$ is the specific or molar volume (m^3/mol), R is the gas constant (= 8.3144 J/(mol•K)), T is the absolute temperature (K), and a (m^6•Pa/mol^2) and b (m^3/mol) are parameters different for and specific to each gas. Note that $\overline{V}$ is not simply the volume; rather it is the volume that would be occupied by one mole of gas at a specified temperature and pressure.

Find the specific volume for carbon dioxide (a = 3.592 L^2•atm/mol^2 and b = 42.67 cm^3/mol) at 300 K and 73 atm.

Background: This problem requires units conversion. Also, the equation cannot be rearranged to a form $\overline{V}$ = something (try it!), so a root-finding technique must be used to find $\overline{V}$.

Solution strategy: Convert all units to SI before performing calculation

L_to_m3 = 1.0•10^{-3}
atm_to_Pa = 1.013•10^5
cm3_to_m3 = 1.0•10^{-6}

Rearrange equation to form f(x) = 0 for finding root.

$$f(\overline{V}) = (P + \frac{a}{\overline{V}^2})(\overline{V} - b) - RT = 0$$

Create a graph of $f(\overline{V})$.

Use `fzero` to find $\overline{V}$.

4. Finding the trajectory that maximizes the range of a rocket is a topic near and dear to many aerospace engineers. The equations that describe a rocket trajectory are rather complex and depend upon the type of engine used (constant thrust or constant acceleration). Instead of solving the complex equations, consider a simpler problem in which an object of mass m is given an initial momentum p at an angle θ relative to the horizontal. Neglecting air resistance and curvature of the earth, from elementary physics

$V_x = (p/m)\cos(\theta)$ horizontal velocity

$x_{\max} = V_x(t_{\max})$ horizontal distance or trajectory

$t_{\max} = (2/g)(p/m)\sin(\theta)$ time of flight

where g is gravitational acceleration.

What is the maximum trajectory for a 10 kg object given an initial momentum of 2000 kg m s^{-1} ? Create a graph that illustrates the trajectory.
(Hint: find the θ that maximizes the trajectory. Ans: x_{max} = 4082 m)

5. The city of Lower Podunk needs a new water tank reservoir for its water supply system. The city manager asks you to estimate the minimum cost of material required to construct a cylindrical tank that will hold 5000 m^3 of water. The reservoir requires a steel floor and side wall, but does not need a top. After consulting with a materials specialist, you determine that 2.0 cm-thick steel plate will withstand the water pressure inside the tank without collapsing. You also have the following information: ρ_{steel} = 0.284 $lb_m in^{-3}$, steel cost = 0.40 \$ kg^{-1}

What are the tank dimensions (height, diameter) that will minimize the cost of material? What is the minimum cost? Make a graph that illustrates the contributions of the bottom and side costs to the overall costs as a function of diameter. Choose a display range that highlights the information the graph is intended to convey.

Recap: You should have learned

- How to graph several lines on a single graph.
- How to fine the minimum and maximum elements in a vector.
- How to use `inline` to define a function
- How to use `fzero` to find the roots of equations
- How to solve an ordinary differential equation using MATLAB routines

MATLAB Workshop 8

Objectives: Learn how to declare matrices, do simple matrix arithmetic, use matrix functions, use the *colon operator*, `:`, and save workspaces in MATLAB.

Recall: A *row vector* is an array with one row and n columns (i.e., a *1 by n* array or matrix). A *column vector* is an array with n rows and one column (i.e., an *n by 1* array or matrix). More generally, a matrix is an *n by m* (rows by columns) array of elements.

array term by term operators

`.*`	term by term multiplication
`./`	term by term division
`.^`	term by term exponentiation

array functions

`size(X)`	determines size of X
`inv(A)`	inverts array A

- **Start MATLAB and perform the following**

```
>> cd  your MATLAB directory
>> diary  wkshop8a.txt
>> header
```

Display of your header output

- ***Declaring matrices (arrays)***

```
»
» P = [2 4 6; 6 4 2]
P =
     2     4     6
     6     4     2
» size(P)
ans =
     2     3
```

P is an array (matrix) with 2 rows and 3 columns. The elements $P_{i,j}$ are entered row-by-row with rows separated by a semicolon, `;`. `size` returns the number of rows and columns.

```
»
» Q = [1, 3, 5; 0, -1, -3; 1, 1, 1]
Q =
     1      3      5
     0     -1     -3
     1      1      1
» size(Q)
ans =
     3     3
```

Q is an array (matrix) with 3 rows and 3 columns. The elements $Q_{i,j}$ are entered row-by-row with rows separated by a semicolon, `;`. Elements in a row can be separated by commas, `,`.

```
»
» R = [1 3 5
1 1 1
5 3 1]
R =
     1     3     5
```

```
     1     1     1
     5     3     1
» size(R)
ans =
     3     3
```

R is an array (matrix) with 3 rows and 3 columns. Rows can be separated by a semi-colon, `;`, or, as shown here, using the return key. **Note**: matrices will typically (not always) be denoted by an upper case letter, vectors by lower case.

```
»
» S = [1, 3, 5; 0; 1, 1, 1]
??? , 3, 5; 0; 1, 1, 1]
All rows in the bracketed expression must have the same
number of columns.
```

A matrix must have the same number of columns in every row.

```
»
» d = [1 0 0];
» e = [0 1 0];
» f = [0 0 1];
» length(d)
ans =
     3
»
» I = [d; e; f]
I =
     1     0     0
     0     1     0
     0     0     1
» size(I)
ans =
     3     3
```

A matrix can also be formed as rows of equal length vectors (separated by a semi-colon, `;`, or the return key.).

```
»
» f = [d e f]
f =
     1     0     0     0     1     0     0     0     1
» size(f)
ans =
     1     9
```

Not using semi-colons or the return key to separate rows creates a row vector!

```
»
» Q(1,1)
ans =
     1
» Q(2,3)
ans =
    -3
```

Individual elements of a matrix are referenced by their (row, column) numbers (or subscripts). Note that the subscripts are inside normal parentheses. *Using square brackets will redefine the matrix !*

- ***Matrix mathematics***

MATLAB is uniquely equipped to handle matrix mathematics and operations (remember "MATrix LABoratory?").

```
» J = sin((pi/4)*I)
J =
    0.7071          0          0
         0     0.7071          0
         0          0     0.7071
```

"Scalar" operations on matrices are similar to scalar operations on vectors. The above statement was equivalent to $J_{i,j} = \sin((\pi/4)I_{i,j})$ for every element in **I**. The result is a matrix with the same size as **I**.

```
»
» K = J + I
K =
    1.7071          0          0
         0     1.7071          0
         0          0     1.7071
»
» L = J + 2*I
L =
    2.7071          0          0
         0     2.7071          0
         0          0     2.7071
»
» M = P + I
??? Error using ==> +
Matrix dimensions must agree
```

Two matrices of the same size can be added term by term. The result is $R_{i,j} = A_{i,j} + B_{i,j}$.

```
»
» A = P*I
A =
    2     4     6
    6     4     2
»
» B = I*P
??? Error using ==> *
Matrix dimensions must agree
```

Two matrices can be multiplied *if, and only if*, the number of columns in the first (left hand) matrix are the same as the number of rows in the second (right hand) matrix. In the preceding example, **P** has dimension (2x3) and **I** has dimension (3x3). Thus, the first multiplication is allowed and the reverse multiplication is not. The result of matrix multiplication, **R** = **A***__B__, is $R_{i,j} = \Sigma(A_{i,k} B_{k,j})$ where the sum is over the subscript k.

```
»
» C = Q*R
C =
     29      21     13
    -16     -10     -4
      7       7      7
»
```

```
» D = R*Q
D =
     6      5      1
     2      3      3
     6     13     17
```

Matrix multiplication is not symmetric! A*B ≠ B*A.

```
»
» Q
Q =
     1      3      5
     0     -1     -3
     1      1      1
»
» E = Q'
E =
     1      0     1
     3     -1     1
     5     -3     1
```

The MATLAB single quote operator, `'`, is used to *transpose* a matrix (exchange rows and columns).

```
»
» F = inv(Q)
F =
    -1.0000     -1.0000      2.0000
     1.5000      2.0000     -1.5000
    -0.5000     -1.0000      0.5000
```

The MATLAB `inv` function finds a matrix *inverse*.

```
»
» G = F*Q
G =
     1     0     0
     0     1     0
     0     0     1
»
» H = F.*Q
H =
    -1.0000     -3.0000     10.0000
          0     -2.0000      4.5000
    -0.5000     -1.0000      0.5000
```

MATLAB also provides the term by term operators (`.*`, `./`, and `.^`) that perform the indicated operation the indicated operation term by term. These operators should not be confused with the whole matrix operators (without the `.` before the operator): note the difference between **G** and **H**.

- ***The Colon Operator, :***

The *colon operator* is used to specify a range. For example, you have already seen

`S = 1:5` with the result `S = [1 2 3 4 5]`

and

`T = 1:2:10` with the result `T = [1 3 5 7 9]`

Used in this manner, the colon operator specifies a range of numbers delimited by the outside values and a step size denoted by the middle number. If the middle number is missing, the step size is one.

The colon operator can also be used to identify elements in a matrix as follows.

```
» I = H(:,1)
I =
    -1.0000
          0
    -0.5000
```

The *colon operator*, `:`, is used to specify the range of all rows. The result is the column vector represented by the first column of **H** is assigned to **I**. A row of **H** could be specified similarly.

```
»
» J = H(1:2,2:3)
J =
    -3.0000    10.0000
    -2.0000     4.5000
```

The *colon operator* is used to assign the elements in rows 1 through 2, columns 2 through 3 of **H** to **J**.

- ***The save Command***

The amount of information and results of computations that come with matrix algebra problems balloons considerably above that for other types of problems. The `save` command is a convenient feature for saving your work for later access or for import into another application. The `save` command has several variations. For a complete listing, type `help save` at the command line.

```
» who
your variables are:
     A listing of all variables you have created or used in this session
»
» save wkshop8a
» diary off
» quit
```

List the variables in your workspace. Save the workspace. MATLAB creates a *binary* file with the designated name with the file extension .mat and saves it in the current directory. Because the file is saved in *binary* format, it cannot be read by other text editors (try it!).

- **Restart MATLAB and perform the following**

```
» cd  your MATLAB directory
» diary  wkshop8b.txt
» header
     Display of your header output
»
» load wkshop8a
» who
your variables are:
     A listing of all variables you have created or used in this session
»
» H
```

```
H =
    -1.0000    -3.0000    10.0000
          0    -2.0000     4.5000
    -0.5000    -1.0000     0.5000
```

Restart MATLAB. `load` workspace just saved. The variable list should be identical. The values associated with the variables are accessed as before.

If you are not interested in saving all the variables in your workspace, you could save only those of interest by typing

```
save filename variable_list
```

MATLAB will save only those variables listed.

Finally, if you want to save the information in ASCII format, which can be read by other text editors, you could use any of the commands

`save fname.ext X Y Z -ASCII` uses 8-digit ASCII form instead of binary.

`save fname.ext X Y Z -ASCII -DOUBLE` uses 16-digit ASCII form.

`save fname.ext X Y Z -ASCII -DOUBLE -TABS` delimits (spaces) with tabs.

Note that you have the option of specifying a file type (extension) when saving in ASCII format. Omitting the .ext will create an ASCII file without a file type. The file can still be read by a text editor. *Caution*: The variable names will not be saved with these options - only the values associated with the variables in the order specified.

Recap: You should have learned

- How to declare a matrix
- How to perform simple matrix arithmetic
- How to use matrix functions and matrix element-by-element operations
- How to use the *colon operator* to specify ranges within a matrix
- How to save workspaces and/or variables using the `save` command

MATLAB Workshop 9

Objectives: Develop a script to solve linear algebraic systems of equations. Use the script to solve various problems involving linear systems of equations.

Solving a system of linear algebraic equations

MATLAB is a convenient tool for a very common engineering problem: solution of a system of linear algebraic equations, e.g.,

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_1$$
$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_1$$

or

$$\mathbf{Ax} = \mathbf{b}$$

where

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \qquad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \qquad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

A is the coefficient matrix, **x** is the unknowns vector, and **b** is the right hand side or forcing vector. The solution can be obtained by applying a numerical technique known as Gaussian elimination to the matrix equation

$$\mathbf{Ax} = \mathbf{b}$$

or by finding the inverse of **A** and computing

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

Creating a data file with matrix coefficients

The first step in the solution is to create a data file that contains the augmented or extended coefficient matrix

$$\mathbf{E} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{bmatrix}$$

which is the coefficient matrix with the column vector of right hand side elements appended. This matrix contains all of the known information about your problem. This can be created by using a text editor of your choice or by using the `edit` command in MATLAB to invoke the MATLAB text editor. The individual elements of the matrix should be entered row-by-row, i.e.,

$$a_{11} \quad a_{12} \quad a_{13} \quad b_1$$
$$a_{21} \quad a_{22} \quad a_{23} \quad b_2$$
$$a_{31} \quad a_{32} \quad a_{33} \quad b_3$$

with spaces or tabs separating individual elements and the return key used to begin a new row. The structure of the matrix and individual elements should be easily discerned when looking at the file. Save your file as a text file with a meaningful name and with a .dat extension. The extension tells you that the file contains data.

Example 1: Quarry problem

A civil engineer needs a gravel mix consisting of 25% fine gravel, 50% gravel, and 25% coarse gravel. Three quarries supply gravel mix. Quarry A produces 20% fine gravel, 50 % gravel, and 30%

coarse gravel. Quarry B produces 30% fine gravel, 55% gravel, and 15% coarse gravel. Quarry C produces 40% fine gravel, 40% gravel, and 20% coarse gravel. What fraction of gravel for the required mix should come from each quarry?

The equations that describe this problem are

$$\begin{aligned} &\text{fine gravel:} && 0.20A + 0.30B + 0.40C = 0.25 \\ &\text{gravel} && 0.50A + 0.55B + 0.40C = 0.50 \\ &\text{coarse gravel} && 0.30A + 0.15B + 0.20C = 0.25 \end{aligned}$$

where A, B, and C represent the fractions of gravel from quarries A, B, and C, respectively. Each row of coefficients represents the individual fractions of gravel in each quarry for the stated kind of gravel. The right hand side represents the fraction of overall product represented by each kind of gravel.

The extended coefficient matrix for this problem is thus

$$\mathbf{E} = \begin{bmatrix} 0.20 & 0.30 & 0.40 & 0.25 \\ 0.50 & 0.55 & 0.40 & 0.50 \\ 0.30 & 0.15 & 0.20 & 0.25 \end{bmatrix}$$

Create a text file with the rows and columns of the E-matrix. Save it in your MATLAB directory as quarry.dat.

Creating a script file to solve linear systems of algebraic equations

If the problem on which you are working is a unique, one-time problem, MATLAB provides an excellent interactive interface for solution. MATLAB also provides the ability to interactively develop *script files* for use in solving commonly recurrent problems such as solution of a set of linear algebraic equations. Working through the problem once will produce a `diary` file that can be easily edited to produce a script file for use the next time you need to solve the problem.

When working through a problem and creating a script file, some simple planning and forethought will ease the process. With respect to linear algebraic systems of equations, the general process involves (a) defining the equations to be solved based upon the problem at hand (a thought and paper and pencil task - and actually the most difficult stage of the process because you will have a script file to use for the solution), (b) creating a text file containing the extended coefficient matrix, (c) solving the system of equations with your script file. The script file will need to do the following: (a) ask you for the extended coefficient matrix file name, (b) extract the coefficient matrix; (c) extract the right hand side matrix; (d) solve the set of equations; and (e) display the answer vector.

- **Start MATLAB and perform the following**

```
» cd  your MATLAB directory
» diary  wkshop9.txt
» header
     Display of your header output
»
» disp('Linear algebraic equation solver')
Linear algebraic equation solver
»
» % get augmented coefficient file name
```

```
»    filename = input('What filename? ==> ','s')
What filename? ==>
filename =
    quarry.dat
```

Display information when script file is invoked. Needs file name for extended coefficient matrix.

```
»
» % load extended coefficient matrix
»    E = load(filename)
E =
    0.2000    0.3000    0.4000    0.2500
    0.5000    0.5500    0.4000    0.5000
    0.3000    0.1500    0.2000    0.2500
```

The `load` command is used to bring in a text file, row by row. Entries in a row are treated as a column.

```
»
» % determine size of matrix
»    [rows,cols] = size(E)
rows =
    3
cols =
    4
```

The `size` function is used to determine the size of a matrix. It returns the number of `rows` and `cols`.

```
»
» % extract coefficient matrix, rhs vector
»    A = E(:,1:rows)
A =
    0.2000    0.3000    0.4000
    0.5000    0.5500    0.4000
    0.3000    0.1500    0.2000
»
»    b = E(:,cols)
b =
    0.2500
    0.5000
    0.2500
```

Use the colon operator, `:`, to extract appropriate parts of **E** into **A** and **b**.

```
»
» % solve using Gaussian elimination
» %    x is the solution vector
»    x = A\b
x =
    0.6250
    0.2500
    0.1250
```

Solve using Gaussian elimination. (See discussion below).

```
»
» %    to see coefficient matrix, type A at command line
» %    to see rhs vector, type b at command line
```

```
» diary off
```

Final comments to user to display coefficient matrix and rhs vector, if desired.

You now have a basic script file that will solve linear systems of equations. Notice that comments were entered as the work progressed to identify what was happening (just as you should be doing in solving a "regular" problem). These comments are also required in the script file to provide a narrative of what the script file is doing. You will appreciate this if you need to review or revise the script file at a later date.

Using the text editor of your choice, open the file wkshop9.txt and edit it to produce a script file for generic use in solving systems of linear algebraic equations. Your script file should look something like this:

```
% lin_eq_solver - linear algebraic equation solver
% finds solution to system of equations Ax=b  where
%    A is the coefficient matrix
%    b is the right hand side vector
%    x is the unknowns vector
% requires user to input a text file with extended
% coefficient matrix, E

% get extended coefficient matrix file name
      disp(' ')
   disp('Linear algebraic equation solver')
   filename = input('   What filename? ==> ','s');

% load extended coefficient matrix, determine size
   E = load(filename);
   [rows,cols] = size(E);

% extract coefficient matrix, rhs vector
   A = E(:,1:rows);
   b = E(:,cols);

% solve using Gaussian elimination
%    x is the solution vector
   disp(' ')
   x = A\b
   disp(' ')

% inform user about viewing A and b
   disp('to see coefficient matrix, type A')
   disp('to see rhs vector, type b')
   disp(' ')
```

Save your script file in your MATLAB directory as lin_eq_solver.m. Note that in editing the `diary` file comments were added at the top of the file to explain what the script does. It identifies not only the purpose, but what the variables are and what is required by the script to properly operate. These lines are displayed when `help lin_eq_solver` is typed at the command line. Once created, you should check your work. Also note that the display suppression command, `;`, was used to suppress

display of commands in the script file (once it is running properly, we are only interested in the results of running the script file - not the process).

- **Back in the MATLAB command window**

```
» help lin_eq_solver
 lin_eq_solver - linear algebraic equation solver
 finds solution to system of equations Ax=b  where
    A is the coefficient matrix
    b is the right hand side vector
    x is the unknowns vector
 requires user to input a text file with extended
 coefficient matrix, E
»
» lin_eq_solver

Linear algebraic equation solver
   What filename? ==> quarry.dat

x =
     0.6250
     0.2500
     0.1250

to see coefficient matrix, type A
to see rhs vector, type b

» A
A =
     0.2000    0.3000    0.4000
     0.5000    0.5500    0.4000
     0.3000    0.1500    0.2000
»
» b
b =
     0.2500
     0.5000
     0.2500
```

Check the script by running the problem that was used to develop it. If you don't like the display, edit the script file until you have a display that you like.

Saving your results

You worked hard to get the solution to your set of equations. You probably now need to export the results to a file for use in the word processor that you are using to create the report of your efforts. This is most easily done with the `save` command.

- **Back in the MATLAB command window**

```
» save quarry_results.txt x -ascii
```

`save filename.ext variable_list -ascii` will create a file in the active directory with the indicated filename. It will place the indicated variables into the file. The `-ascii` command tells MATLAB to save the variables as ASCII text rather than binary code. For more information, type `help save`.

Gaussian elimination versus matrix inversion

The preceding script file made use of the *slash* operator, \, to instruct MATLAB to use a numerical algorithm called Gaussian elimination to solve the system of equations. Type `help slash` if you want more information on the command. The problem could also be solved by using the set of commands

```
» Ainv = inv(A)
» x = Ainv*b
```

which is the more traditional method, i.e., finding the inverse of the coefficient matrix, `Ainv`, and multiplying the rhs vector, `b`, by `Ainv`. Try it!

However, the "traditional method" requires more computational time and, because computers are subject to round-off and truncation errors in representing numbers, more prone to instability and errors than the Gaussian elimination method. Thus, Gaussian elimination (or one of its variants) is the preferred computational method to solve systems of equations.

Exercises: Use your script file, `lin_eq_solver` to solve the following problems.

1. Six people, Alice, Barb, Carol, Dean, Eric, and Fred, each have a bag of money. Find the amount in each bag if:
   - The total of all six bags is $33.56;
   - Carol has twice as much money as Alice;
   - Barb and Carol together have as much as Dean;
   - Alice and Barb together have as much as Eric;
   - Eric's amount subtracted from Fred's amount is twice Alice's amount; and
   - $1.15 is left after subtracting Barb's amount and Eric's amount from Fred's amount.

2. Find the five-digit number that satisfies the following properties:
   - The sum of all the digits is 18;
   - The third digit is the sum of the first and second digits;
   - Subtracting the third digit from the fourth digit yields 4;
   - The first and third digits added together give the fifth digit; and
   - The first digit is twice the second digit.

3. Find the six-digit number that satisfies the following properties:
   - The first and third digit sum to 10;
   - The sum of the second, third, and fifth digits equals the fourth digit;
   - The third and last digits are the same;
   - The sum of the second and third digit equals the fifth digit;
   - The fourth digit is the same as the last digit minus the first digit; and
   - The sum of all the digits is 22.

4. Five people, Alice, Ben, Cindy, Dean, and Evan, decide to invest in the stock market. What is each person's profit after one year if:
   - The sum of all their profits is $299.25;

- Evan's profit is three times as large as Dean's profit;
- Ben's and Cindy's profits together total to Dean's profit;
- Alice and Cindy together made $89.20 profit; and
- Alice's profit was $101.55 more than Dean's profit.

Recap: You should have learned

- How to create a *script file* to solve linear systems of equations.
- How to use the `load` command.
- How to use the `size` command.
- How to extract the coefficient matrix and rhs vector from the extended coefficient matrix.
- How to solve the system of equations using Gaussian elimination.
- How to solve the system of equations using matrix inversion.
- How to use the `save` command.

MATLAB Basic Graphics

MATLAB has numerous graphics techniques that allow simple, easy visualization of functions and data. Basic 2D plotting capabilities are addressed in this chapter. If you are interested in more information than is provided here, you can obtain it directly from the MATLAB help facility by typing

`help graphics`	General description of graphics functions
`help graph2d`	More specialized description of 2D graphics functions
`help graph3d`	More specialized description of 3D graphics functions
`help specgraph`	Description of special graphic styles and displays
`help winfun`	Description of Windows™ interface functions

MATLAB contains a complete *graphics toolbox* called *Handle Graphics* that allows one to have complete control over the look and feel of any graphics display. This includes font type and font size for text, location of text on the screen, embedding graphs within other graphs, etc, that the advanced graphics designer might want to control. If you are not satisfied with the basic graphics functions and capabilities described here, you might want to learn more about *Handle Graphics*.

Saving and/or printing plots

The MATLAB Graphics Window has the typical menu and shortcut icons associated with Windows™ applications. To save a graphics figure, use the File-Save or File-Save As menu commands. If you have previously saved the figure, it will simply be saved under whatever name you are using for the figure. If you have not yet saved the figure, or you want to change the name under which you are saving the figure, a typical *Save As* window will appear. Save your file with file type Fig-files (.fig extension). The MATLAB graphics window will then be able to open the figure for further work at your convenience.

To *export* your figure for use in another application, such as a word processor, use the File-Export menu. Figures can be saved in the file format of your choice, e.g., .emg, .bmp, .eps, .jpg, or .tif. The figures in this chapter were saved in .jpg format and imported into a Microsoft Word™ document.

Your figure can be sent to the printer either by using the *printer icon* on the tool bar or by using the File-Print menu command. Note that you can specify the printing format using the File-Page Setup and File-Print Setup menu commands.

Multiple open graphics windows

You can have more than one open graphics window by using the command

```
» figure(N)
```

in the command window. MATLAB will either open a new graphics window, number **N**, and make it the current graphics window when this command is invoked or, if the command was used previously, make graphics window **N** the current graphics window. The number of graphics windows you can have open at one time is limited only by the capacity of your computer.

2D plot graphics

The basic 2D plot functions are

`plot(x,y,s)`	Plot with linear (x,y) axes; line type specified by string **s**
`semilogy(x,y,s)`	Plot with linear x, logarithmic y axes; line type specified by string **s**

`semilogx(x,y,s)`	Plot with logarithmic x, linear y axes; line type specified by string **s**
`loglog(x,y,s)`	Plot with logarithmic (x,y) axes; line type specified by string **s**
`plot(y,s)`	Plot **y** versus its row index; line type specified by string **s**

The first four of these require two vectors of equal length, **x** and y, and the line-style string, s, that defines how the vectors are displayed on the plot (color, symbols, line type). `plot(y,s)`, in contrast, will plot the elements of **y** versus their row index. If **s** is omitted, MATLAB will use default options for the plot.

- Line-style options. The string, **s**, is optional when the plotting command is invoked. If not provided, MATLAB will use the next default option. A line-style is defined by choosing an entry from each column of the line-style options table. Because all of the codes for the various style options are different, the options in the string can be defined in any order, e.g., `'m^-.'` is the same as `'^m-.'` which is the same as `'-.m^'`, etc.

Line-style Options					
Line Color		Line Symbol		Line Type	
y	yellow	.	point	-	solid
m	magenta	o	circle	:	dotted
c	cyan	x	x-mark	-.	dash dot
r	red	+	plus	--	dashed
g	green blue	*	star		
b	blue	s	square		
w	white	d	diamond		
k	black	v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

Examples

`plot(x,y)`	plots **x** vs **y** with default line specifications
`plot(x,y,'m<')`	plots **x** vs **y** with magenta left-triangles (no line)
`plot(x,y,'d-.b')`	plots **x** vs **y** with blue diamonds and blue dash-dot line

Once a plot is displayed, you can change the line-style at will in the graphics window. One method is to select a line by clicking on it with the *left* mouse button and use the Tools-Line Properties menu, which will bring up a window that allows you to specify properties for the currently selected line. Another method is to click on the line with the *right* mouse button to bring up a pop-up menu that offers choices of what to change.

- Text objects. Once a basic plot has been created using one of the various `plot` commands, you can alter text the text objects that are used to annotate the graph at will.

- *Axes labels, plot title*. The three most basic text objects are the x-axis label, the y-axis label, and the title, which are implemented after the basic plot has been made with the commands

```
» xlabel(xaxis_string_name)
```

```
» ylabel(yaxis_string_name)
» title(title_string_name)
```

respectively. `string_name` can be supplied as text enclosed within single quotes, e.g.,

```
» xlabel('time, s')
» ylabel('displacement, m')
» title('spring position vs time')
```

Alternatively, it could be supplied through a string variable, e.g.,

```
» xlabelname = 'time, s';
» ylabelname = 'displacement, m';
» titlename = 'spring position vs time';
» xlabel(xlabelname)
» ylabel(ylabelname)
» title(titlename)
```

If you do not want to use any of these, you can add them by using the Tools-Axes Properties menu in the graphics window. This will bring up a window where you could specify them.

- *Legend*. If more than one function or set of data are displayed in a plot (see Plot Overlay, following), you might need a legend to help the viewer distinguish which curve is which. The `legend` command produces a boxed legend on the graph.

The basic format for the legend command is

```
legend( string_name1, string_name2, …, tol)
```

where `string_nameX` is the name by which you want to refer to the X^{th} data set that you have plotted in the figure. MATLAB will stack the legend entries vertically with the current line style associated with legend entries displayed to the left of the entry.

`tol` is the legend placement: **-1** for outside the plot and **0** for inside the plot. If `tol` is omitted, MATLAB will seek to place the legend inside the plot where minimal information will be obscured by the legend. If MATLAB cannot do so, the legend will be placed outside the plot. If you do not like where the legend has been placed, you can grab the legend with the mouse and drag it to a location more to your liking. `legend off` will delete the legend from the plot.

- *Text*. Occasionally some additional text in the plot, highlighting a feature of the plot such a local maximum or minimum, will add more power to the graphic. MATLAB uses the command

```
text(x-coord,y-coord,'text string')
```

to place text starting at the indicated (x,y) coordinates on the plot. Perhaps more convenient and useful is the command

```
gtext('text string')
```

which lets the user specify where to place the text by clicking the mouse at the desired location.

Alternatively, and easier, you could use the *text icon* in the graphics window tool bar to place text wherever you desire in the graphics window.

- *Font type, size, and color*. With the exception of the legend window, you can control the font type, size, and color of any text (axis labels, title, etc) by clicking on the text with the *right* mouse button to activate a pop-up menu with font choices.

- Axis control. MATLAB has internal algorithms that control the default appearance of the axes when the plot first appears. Axis control commands must come after the plot is generated in order to have the desired effect. Some predefined, useful axis string commands are

`axis('equal')`	Sets displays scale lengths equal on both axes
`axis('square')`	Sets the default rectangular axes frame to a square
`axis('normal')`	Restores default axes
`axis('off')`	Removes axes frame (and tick marks) from the plot
`axis('axis')`	Freezes current axes limits

More generally, the display ranges can be controlled by using the command

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

which sets the ranges for the display. Note that `[ xmin xmax ymin ymax ]` is a vector and, thus, can be satisfied by any means that provides a vector. Thus

```
axis([ 0  10  10  100 ])
```
(sets ranges $0 \le x \le 10$ and $10 \le y \le 100$)

```
axislimits = [ 0 10 10 100 ]);
axis( axislimits )

x_axis = [ 0 10 ]; y_axis = [10 100];
axis([ x_axis y_axis ])
```

are all equivalent.

Partial specification of limits with MATLAB providing the other limit(s) is also possible by using `inf` where you would like MATLAB to provide the limit.

`axis([ 0 inf -inf inf ])`	sets xmin to 0, lets MATLAB determine other limits
`axis([ -inf 10 1 10 ])`	sets xmax, ymin, ymax, lets MATLAB determine xmin

The axis limits on a plot can be manually changed once the display is in the graphics window by using the Tools-Axes Properties menu command. This will bring up a window that allows you to specify not only the axis range but also whether to use linear or logarithmic scaling on the axes and/or whether to use normal (least to greatest) or reverse (greatest to least) number order on the axes. This window can also be brought up by clicking on an axis with the *right* mouse button and selecting Properties from the pop-up menu that appears.

- Plot overlay: Multiple display on one plot. MATLAB has several methods for displaying more then one curve on a single graph.

- *Method 1: `hold` command*. The `hold` command is used to maintain the current plot in the Graphics Window. `hold on` freezes the current plot in the Graphics Window. All subsequent plots are superimposed on the existing plot. The following script shows one use of the `hold` command to produce multiple curves on the same graph. It is available in the file TwoDplot1.m.

```
% TwoDplot1 : hold command example for multiple lines in same plot
% script to produce multiple curves on a single plot

% generate & plot 1st curve
   t = linspace(0, 4*pi, 200);
   curve1 = sin(t);
   plot(t,curve1,'r')
   hold on

% generate and plot 2nd curve
   curve2 = sin(2*t);
   plot(t,curve2,'g')

% generate and plot 3rd curve
   curve3 = sin(t/2);
   plot(t,curve3,'b')

% annotate
   xlabel('time, s')
   ylabel('sine function')
   title('sine function behavior')
   legend( 'sin(t)', 'sin(2*t)', 'sin(2/t)' )
   hold off
```

Since the `hold` command freezes the plot window, not all data need be generated before the first plot is made.

- *Method 2: `line` command*. The `line` command is and alternative method to plot more than one curve in a Graphics Window. The general form of the `line` command is

```
line(xdata,ydata,parameter_name,parameter_value)
```

where the `parameter_name / parameter_value` could be any graphics parameter. Most typically, we will use `'linestyle' / 'linestyle option'` as in the following example (available in the file TwoDplot2.m).

```
% TwoDplot2 : line command example for multiple lines in same plot
% script to produce multiple curves on a single plot

% generate & plot 1st curve
   t = linspace(0, 4*pi, 200);
   curve1 = sin(t);
   plot(t,curve1,'-')
```

```
% generate and plot 2nd curve
   t = linspace(0, 4*pi, 100);
   curve2 = sin(2*t);
   line(t,curve2,'linestyle','--')

% generate and plot 3rd curve
   t = linspace(0, 4*pi, 150);
   curve3 = sin(t/2);
   line(t,curve3,'linestyle','-.')

% annotate
   xlabel('time, s')
   ylabel('sine function')
   title('sine function behavior')
   legend( 'sin(t)', 'sin(2*t)', 'sin(2/t)' )
```

As with the `hold` command, not all data need be generated before the first plot is made when using the `line` command. Note that the **t** vectors, while encompassing the same range, can be of different length. The same is true when the `hold` command is used.

- *Method 3: extended `plot` command*. The extended `plot` command

```
plot(x1,y1,linestyle1,x2,y2,linestyle2,…)
```

can also be used as shown in the following script (available in the file TwoDplot3.m).

```
% TwoDplot3 : extended plot command example for multiple lines
% script to produce multiple curves on a single plot

% generate curves
   t = linspace(0, 4*pi, 200);
   curve1 = sin(t);
   curve2 = sin(2*t);
   curve3 = sin(t/2);
   plot(t,curve1,'-',t,curve2,'--',t,curve3,'-.')

% annotate
   xlabel('time, s')
   ylabel('sine function')
   title('sine function behavior')
   legend( 'sin(t)', 'sin(2*t)', 'sin(2/t)' )
```

The most obvious difference with the first two methods is that all data must be available before plotting. A more subtle difference is that all of the plot vectors **t** and **curve** must be the same length as well as the same range on **t**. The first two methods only require that the range of the x-variable vector be the same.

- *Method 4: `plot` command using matrices*. The `plot` command can also be used with the y-data held in a matrix as demonstrated in the following script (available in the file TwoDplot4.m).

```
% TwoDplot4 : plot command using matrices example for multiple lines
```

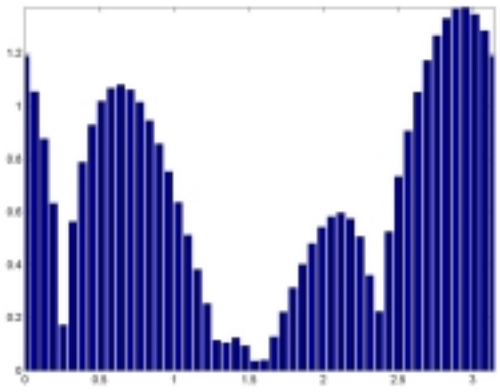
```
% script to produce multiple curves on a single plot

% generate curves
   t = linspace(0, 4*pi, 200);
   curves = [ sin(t); sin(2*t); sin(t/2)];

   plot(t,curves)

% annotate
   xlabel('time, s')
   ylabel('sine function')
   title('sine function behavior')
   legend( 'sin(t)', 'sin(2*t)', 'sin(2/t)' )
```

The three rows of the matrix `curves` each have the same number of elements as the independent variable vector, **`t`** (a requirement). Each row of `curves` is plotted against the corresponding value in **t**, just as in Method 3. However, MATLAB chooses default linestyles for the plot. Although you can change the default linestyles to whatever you wish in the graphics window, you need to know which is which!

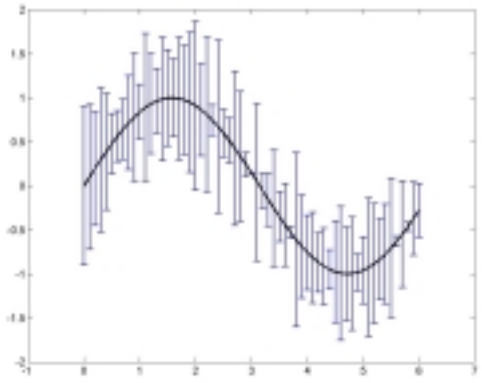
- Special 2D plots. MATLAB has a variety of special 2D plots that can be used. Some of them are shown in the following Table 1 along with the scripts that generated them. The scripts are available as the corresponding `.m` file. More information on any of the plotting functions can be obtained by using the `help` command followed by the function name. More special functions can be found by typing `help graph2d`.

Table 1. Examples of selected MATLAB special 2D plot functions.

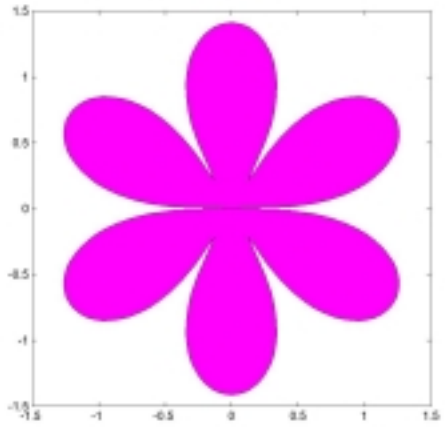
```
% bar_demo - example of 2D bar plot
%    r*r = 2cos(3t+pi/4),  0 <= t <= pi
%    x = rcos(t)

   t = linspace(0, pi, 50);
   r = sqrt(abs(2*cos(3*t+pi/4)));
   x = r.*abs(cos(t));

   bar(t,x)
   axis([0 pi 0 inf])
```



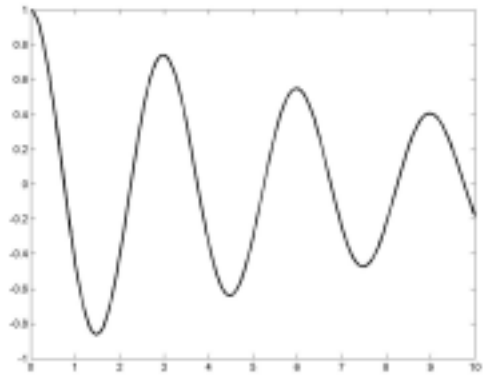
```
% errorbar_demo - example of 2D
                  errorbar plot
%    fcn = sin(x),  0 <= x <= 6
%    error = fcn + random number

   x = 0:0.1:6;
   fcn = sin(x);
   random = rand(1,61);

   errorbar(x,fcn,random)
```



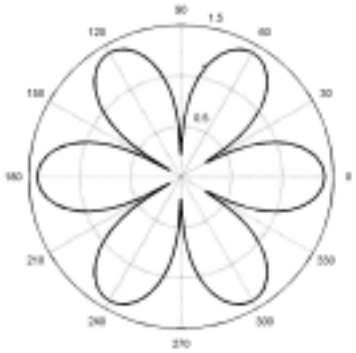
```
% fill_demo - example of 2D fill plot
%    r*r = 2sin(3t),  -pi <= t <= pi
%    x = rcos(t),  y = rsin(t)

   t = -pi:pi/100:pi;
   r = sqrt(abs(2*sin(3*t)));
   x = r.*cos(t);
   y = r.*sin(t);

   fill(x,y,'m')
   axis('square')
```



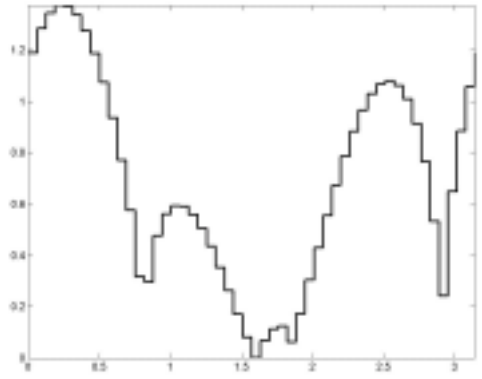
```
% fplot_demo - example of 2D fplot
%    function to be plotted can be a
%    string, defined by inline,
%    or a .m function

   fcn = inline('cos(2*pi*x/3)*...
         exp(-0.1*x)','x');

   fplot(fcn,[0 10])
```



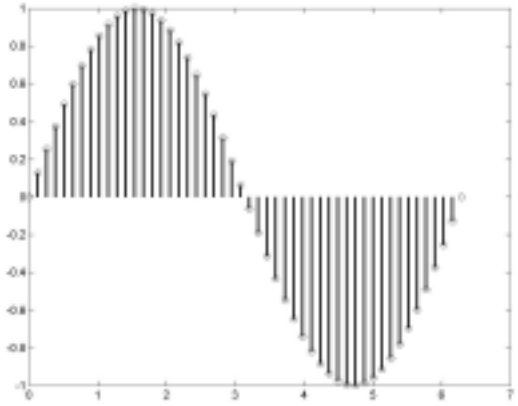
```
% polar_demo - example of 2D polar plot
%    r*r = 2cos(3t),  0 <= t <= 2*pi

   t = linspace(0, 2*pi, 200);
   r = sqrt(abs(2*cos(3*t)));

   polar(t, r)
```



```
% stairs_demo - example of 2D
                 stairs plot
%    r*r = 2cos(3t-pi/4),  0 <= t <= pi
%    x = rcos(t)

   t = 0:pi/50:pi;
   r = sqrt(abs(2*cos(3*t-pi/4)));
   x = r.*abs(cos(t));

   stairs(t,x)
   axis([0 pi 0 inf])
```



```
% stem_demo - example of 2D stem plot
%    f = sin(x),  0 <= x <= 2*pi


   x = linspace(0, 2*pi, 50);
   f = sin(x);

   stem(x,f)
```



```
% comet_demo - example of 2D comet plot
%    f = x sin(x),  0 <= x <= 10pi


   x = linspace(0, 10*pi, 500);
   f = x.*sin(x);

   comet(x,f,'k-')
```

Run comet_demo to see plot in action.

Multiple Plots in One Graphics Window: **`subplot`**

The command

```
subplot(m,n,p)
```

will divide the current window into a grid with **m** rows and **n** columns. It will place the next plot in the **p**th window, counting successively by columns across rows. An example script (available in the file subplot_demo.m)

```
% subplot_demo : multiple plots in single window
% script to produce multiple plots in a graphics window

% plot 2 rows, 2 columns in single window
   subplot(2,2,1), polar_demo
```

```
subplot(2,2,2), fill_demo
subplot(2,2,3), stairs_demo
subplot(2,2,4), bar_demo
```

produces the graphic in Figure 1. (Note that two MATLAB commands, separated by a comma, are placed on the same line to enhance script readability.)

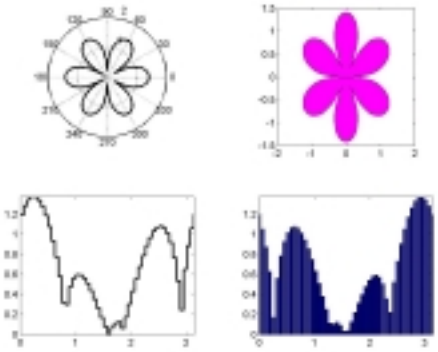


Fig 1. Example of multiple plots in a single graphics window using `subplot`.

3D plot graphics

MATLAB also provide 3D graphing capability. The basic commands are

`plot3(x,y,z)`	Plot 3D graph with linear scales
`view(azimuth, elevation)`	View from position indicated

The view command identifies the angles (in degrees!) of viewing vantage point, as illustrated in Fig 2. If omitted, the viewing angles are (-37.5, 30).

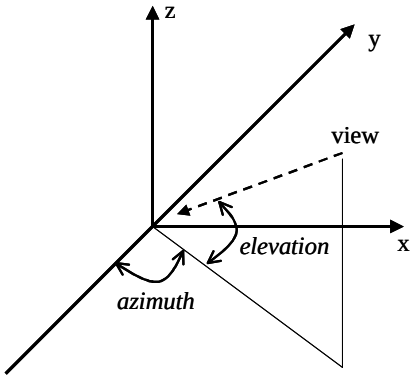


Fig 2. Definition of viewing angle in 3D plots.

The following script (available in the file ThreeDplot1.m) creates the traditional projections of a function in the various coordinate planes. The plot is shown in Fig 3.

```
% ThreeDplot1 : 3D plot example
% script to produce four views
% of the 3D function f(x,y,t)
%    r = sin(3t),  -pi <= t <= pi
%    x = rcos(t),  y = rsin(t)

% generate function points
   t = -pi:pi/100:pi;
   r = sin(3*t);
   x = r.*cos(t);
   y = r.*sin(t);

% place default view in first subwindow
   subplot(2,2,1)
   plot3(x,y,t), grid
   xlabel('rcos(t)'), ylabel('rsin(t)')
   zlabel('t'), title('default view')

% place x-y projection in second subwindow
   subplot(2,2,2)
   plot3(x,y,t), view(0,90)
   xlabel('rcos(t)'), ylabel('rsin(t)')
```

```
   zlabel('t'), title('x-y projection')

% place x-z projection view in third subwindow
   subplot(2,2,3)
   plot3(x,y,t), view(0,0)
   xlabel('rcos(t)'), ylabel('rsin(t)')
   zlabel('t'), title('x-z projection')

% place y-z projection in fourth subwindow
   subplot(2,2,4)
   plot3(x,y,t), view(90,0)
   xlabel('rcos(t)'), ylabel('rsin(t)')
   zlabel('t'), title('y-z projection')
```

- 3D surface/mesh plots.

Frequently, we are interested in examining a response surface rather than the linear behavior of a function. MATLAB provides two methods. `mesh` will create a mesh representation of the surface in which surface response points are connected by lines The lines are color-coded by height above the viewing plane. `surf` is essentially the same as `mesh` with the exception that the mesh space between lines is now also color-coded with fill to create a "smoother" surface. Both `mesh` and `surf` require mesh grid for calculating the surface response points. This is created by using `meshgrid` in conjunction with the **x** vector of x-points and **y**-vector of y-points. The following script (ThreeDplot2.m) illustrates the basics of mesh and surface plots.

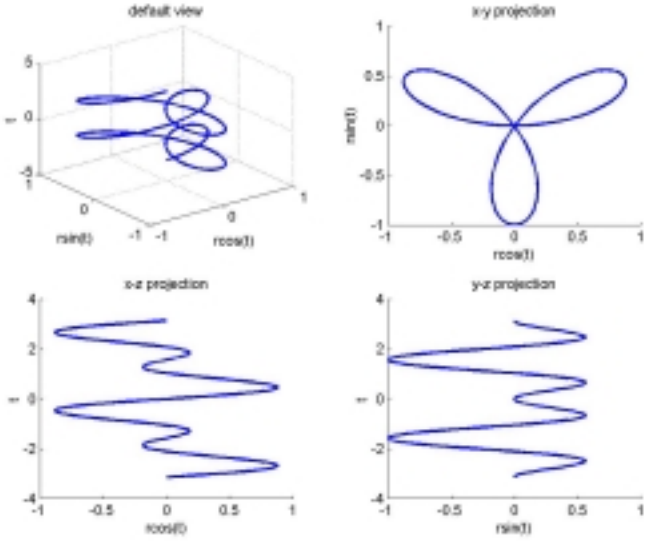


Fig 3. Three-dimensional projections using `plot3`.

```
% ThreeDplot2 : 3D surface plot example
% script to produce a 3D surface plot
%    z = 1/(x*x+y*y+10)

% generate function points
   x = linspace(-5,5,25);
   y = x;
   [X,Y] = meshgrid(x,y);
   Z = 1./(X.*X+Y.*Y+10);

% create mesh plot
   subplot(1,2,1)
   mesh(X,Y,Z)
   xlabel('x'), ylabel('y')
   zlabel('z'), title('mesh view')

% create surface plot
```

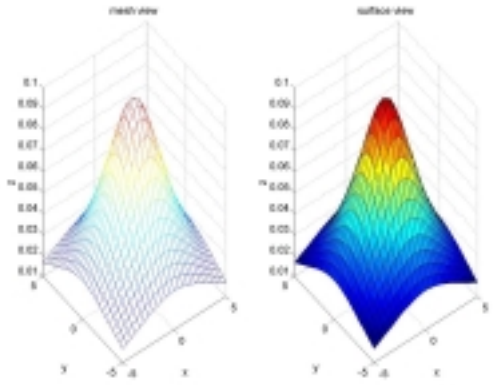


Fig 4. Difference between `mesh` and `surf`.

```
subplot(1,2,2)
surf(X,Y,Z)
xlabel('x'), ylabel('y')
zlabel('z'), title('surface view')
```

- 3D rotation. One of the most powerful features of MATLAB graphics is the ability to easily rotate any graph in three dimensions to obtain different views. Although of limited use, this applies also to 2D plots: they can be easily rotated into three dimensions. To rotate a plot, select Tools-Rotate 3D from the Figure Menu Bar. Then click on the figure with the right mouse button. The current view (azimuth, elevation) appears to the lower left. To rotate the figure, simply drag the figure in the direction you want to rotate. Fig 5 shows a rotation of the mesh representation of Fig 4 from the default projection to the (-28,-6) projection.

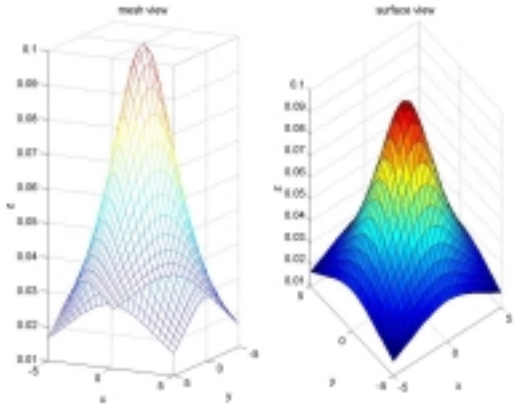


Fig 5. 3D rotation can highlight different aspects of a surface or mesh projection.

- Special 3D plots. Just as with 2D plots, MATLAB has a variety of special 3D plots. A complete listing of available plot types can be obtained by typing `help graph3D` and `help specgraph`.

MATLAB: Programming Features

Scripts and functions in MATLAB

In addition to the interactive "programming" available with the command window, MATLAB supports two kinds of *predefined* program elements: the *script* file and the *function* file. Both of these file types are most easily developed using the interactive mode in the command window to solve a known problem. The `diary` file resulting from the interactive problem solution can be easily edited, deleting unnecessary lines or errors and adding appropriate comments, to create the desired file.

Because *script* and *function* files use the same basic commands and program structure, they are similar in appearance. They are, however, very different in purpose. A *script* file is designed to perform a task, such as initial display of a graph or solving a system of linear algebraic equations. A *function* is designed to calculate a value, such as the sin or cos. A purist would maintain that a function should calculate only one value. If it were to be calculating more than one value, it would then be a task and should be more appropriately designed as a script file. However, many practical instances exist where more than one highly related value could conceivably be calculated by a single function, such as calculating the slope and intercept in linear regression.

Script and function files in MATLAB behave differently. The most important difference is that all variables declared in a script file are added to the workspace, regardless of whether the results display is suppressed with the semicolon operator, `;`. Thus, any variable used in a script is available after the script executes. Execution of a function file, however, does not add any variables to the workspace - the function's variables are "active" only while the function is executing. Thus, variables and/or values used in a function, other than those returned by the parameter list, are not available after the function executes.

- **Script Files**

The generic form of a script file is shown at the right. The script file starts with one or more comments that identify what task the script accomplishes, including values or information the script needs to perform its task and the values that result if the script operates successfully. The script then executes the task in sequential steps. The results of commands will display as the script executes unless the suppress display operator, `;`, is used at the end of each command line.

```
% script_name - script to do ??
%  requires:
%  results:

% perform first step of task
   step 1 commands

% perform second step of task
   step 2 commands
```

Generic script file structure.

The script is saved as script_name.m in your MATLAB directory. It can then be executed by simply typing `script_name` at the command line. Typing `help script_name` at the command prompt will display the all comment lines up to the first non-comment line.

Remember: variables created in the script file remain in your workspace. **Caution**: if you already have a variable with the same name as one in a script file, the script file changes the variable to have the values calculated in the script file! **Remedy**: use uncommon variable names within a script file to avoid overwriting other variables in your workspace.

- **Function Files**

The generic form of a script file is shown below. The first line in the function file has the general form

```
function  [out1, out2, … ] = function_name(parm1, parm2, …);
```

The *keyword* `function` is required and must be all lower case. The values calculated by the function are contained in square braces. The function name follows on the other side of the equals sign. The variables and/or parameters required by the function are then listed inside parentheses. If only one value is calculated by the function, which is the usual case, the first line can be abbreviated to

```
function  out1 = function_name(parm1, parm2, …);
```

```
function  [out1, out2, … ] = function_name(parm1, parm2, …);
% function_name - calculates ??
%   requires:
%   results:

% first step in calculation
   first step commands

% second step in calculations
   second step commands

% final values
   out1 =
   out2 =
```

Generic function file structure.

The function file then has one or more comments that describe what the function does, including values or information the function needs to perform its tasks and the value(s) that result if the function operates successfully. The function executes the value calculations sequentially. Each of the `out` variables must have a line in the function that *assigns* a value to the variable name, e.g.,

```
out1 = 2.0*pi;
```

The function is saved as function_name.m in your MATLAB directory. Typing `help function_name` at the command prompt will display the all comment lines up to the first non-comment line. Several methods are available for using a function file. Functions that return a *single* value can be used in any of the following ways:

```
alpha = function_name(parm list)        (assign the value to another variable)
[alpha] = function_name(parm list)      (assign the value to another variable)
alpha = 2.0*function_name(parm list)    (use the value in an arithmetic expression)
function_name(parm list)                (display the value)
```

Notice that assignment of the function value to a variable can be with or without square braces around the variable name. The following form is required to capture all of the values returned by a function that calculates more than one value:

```
[ans1, ans2, ...] = function_name(parm list)
```

The function will assign the values of `out1`, `out2`, etc, in sequence to `ans1`, `ans2`, etc, for further use.

Remember: variables and values created in the function file are not placed in your workspace. Because of this, you can have the same variable names in both your workspace and the function without changing the values of either by using the function. **Caution**: if you do not *assign* the values calculated by a function to variables in your workspace, you will lose the values. **Remedy**: be sure to properly assign all `out` values from the function to variables in your workspace.

MATLAB programming language features

MATLAB provides the same powerful programming capabilities for interactive computing, script files, and function files as most high level programming languages. Those familiar with another programming language will have little or no trouble in understanding and using the programming language structures available in MATLAB. Those who are less familiar with another programming language should also be able to master and use the following programming language structures.

- **MATLAB Program Control Structures**

Straight-line structures. The most common programming structure is straight-line execution of commands, such as

```
t = 1:2:20;
u = 1:10;
plot(t,s)
```

Each command is processed sequentially. (If you are not sure what will result from the commands, type them in the MATLAB command window.)

Branching or decision structures. Frequently, the decision of whether to execute some commands depends on the outcome of a comparison. For example, the sequence of commands

```
if (time < 0.0)
   velocity = 0.0;
end
```

asks that the value for time be compared with the value 0.0. If time is less than 0.0, then velocity is given the value 0.0. This is called a *one-sided decision structure*.

More frequently, decision structures have two or more sides. For example, the sequence of commands

```
if (time < 0.0)
   velocity = 0.0;
else
   velocity = 1.5*time;
end
```

is a *two-sided decision structure*. If the comparison on time is true, then the statement(s) between the `if` line and the `else` line are executed. If the comparison is false, the statement(s) between the `else` line and the `end` line are executed.

Sometimes, multiple comparisons are needed, e.g.,

```
if (time < 0.0)
   velocity = 0.0;
elseif (time >= 0.0) & (time <= 10.0)
   velocity = 1.5*time;
else
   velocity = 3.0*time;
end
```

The generic branching structure syntax is

```
if (comparison1)
   comparison1 commands;
elseif (comparison2)
   comparison2 commands;
elseif (comparison3)
   comparison3 commands;
...
else
   default commands;
end
```

Each structure starts with an `if`, may have `elseif` and `else` statements, and terminates with `end`. The relational and logical operators that can be used to construct a comparison are provided in the following table.

Relational Operators		Logical Operators	
<	less than	&	logical AND
<=	less than or equal to	|	logical OR
>	greater than	~	logical NOT
>=	greater than or equal to	xor	logical EXCLUSIVE OR
==	equal to (same as)		
~=	not equal to		

Looping or repetition structures. Loops are used to repeat a series of commands for a specified number of times or until a loop termination condition is met. MATLAB supports two types of loops. The `for` loop is used to repeat the specified commands a prescribed number of times. For example, the command sequence

```
% script to compute the sum and product of the first 10 integers
% initialize sum to zero, prod to one
   sum = 0;
   prod = 1;
% loop to find sum and product
for ( i = 1:10 )
```

```
   sum = sum + i;
   prod = prod*i;
end
```

will execute 10 times, producing the required values for `sum` and `prod`. The loop index, `i`, will start with the value 1, the two lines of code for `sum` and `prod` will execute, the loop index, `i`, will be incremented to the value 2, the two lines of code for `sum` and `prod` will executed, etc, until on the final repetition, the loop index, `i`, will have the value 10.

The general `for` statement is

```
for  index = start_value:increment_size:final_value
```

Any variable name can be used for the index. The index has a starting value, increment size, and final value. The loop will execute until the value of the index is the same as the final value. The display suppression operator, `;`, is used to suppress display of the repetitive commands while the loop is running. If omitted, the vectors are displayed in their entirety at each step. The `end` statement terminates the repetitive commands.

The `while` loop is used to execute a group of statements for an indefinite number of repetitions, terminating when the loop condition becomes false. An example similar to the preceding `for` loop is

```
% script to compute the sum and product integers
% until the sum exceeds 100
% initialize int to one, sum to zero, prod to one
   int = 1;
   sum = 0;
   prod = 1;
% loop to find sum and product
while ( sum <= 100 )
   sum = sum + i;
   prod = prod*i;
   int = int + 1;
end
```

As with the for loop, the while loop requires appropriate initialization conditions for the loop to operate. The repetitive commands inside the loop will execute while the loop entrance condition is true.

- **Other MATLAB Language Features**

MATLAB has some features that may be available in other high level programming languages, but have special formats for use in MATLAB.

Error messages. MATLAB has totally inadequate error message capability. Error messages tend to be terse, ambiguous, and uninformative.

Comments. The comment operator, `%`, is used to provide comments about the program. MATLAB will ignore everything on a line after the comment operator. Executable commands will resume with the next line. Use of comments at the start of your script and function files is a crucial

good programming practice to provide useful `help` information on what the script/function does, what variables/parameters it needs to do its job, and what will result if it does its job properly.

Continuation. Three consecutive periods, `...`, at the end of a line tells MATLAB that the next line is a continuation or part of the current line/command. For example

```
A = [1 1 1 1 1 1 1 1 ; 2 2 2 2 2 2 2 2; 3 3 ...
     3 3 3 3 3 3; 4 4 4 4 4 4 4 4; 5 5 5 5 5 5 5 5];
```

can be used to split the definition of matrix elements at any time.

`error` command. If MATLAB encounters the command `error( 'message' )` inside a function or script, the text message will display and the function or script will terminate. Control is passed back to the command line. For example, the code

```
% script error_test
divisor = 0;
if (divisor == 0)
   error( 'ERROR - About to divide by zero' )
else
   disp( 'Did not get this far' )
   division = 1/divisor;
end
```

inside a function or script could be used to provide a warning message about when a calculation will fail. If `divisor` equals zero, MATLAB will display the following

```
??? Error using ==> error_test
ERROR - About to divide by zero
```

The display and calculation in the `else` clause are not performed and control is passed immediately to the command line. This feature allows you to augment the MATLAB error message facility with something you can understand.

File handling. MATLAB supports the C-language input/output file handling functions. Primary among these are

`fopen`	Opens a file for further action
`fclose`	Closes an open file
`fscanf`	Reads data from a file
`fprintf`	Writes data to a file

Please consult a C-language reference for further details (the MATLAB `help` command will provide some information).

`menu` command. The MATLAB menu command is a convenient method to display a menu that asks for user input. For example, the script

```
% get user choice for line color
```

```
disp('Please enter choice for line color')
disp('   blue = 1 ')
disp('   green = 2')
disp('   red = 3')
linecolor = input(''Choice (1,2,3) ==> ');
```

can be replaced by the single command

```
menu('Please enter choice for line color','blue','green','red')
```

A pop-up window appears that allows the user to make a choice. The options are internally numbered as shown in the longer script.