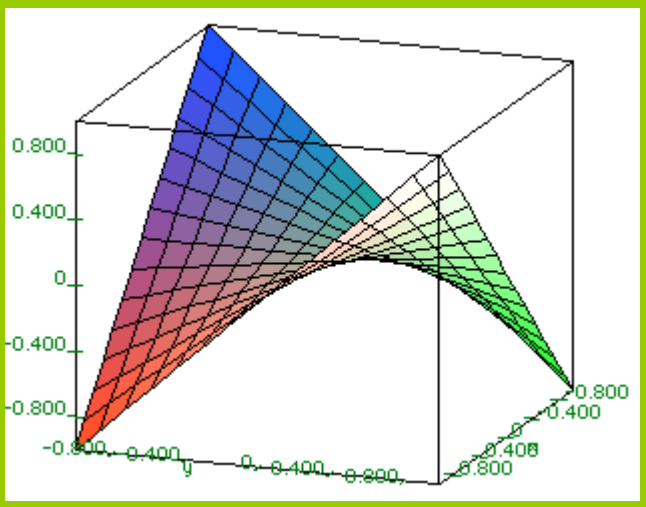
Lab Manual
Modeling & Simulation



Workbook

Modelling & Simulation
(SE-401)
Final Year

Name _____

Roll No _____

Batch _____

Year _____

Department _____

Department of Computer Science & Information Technology
NED University of Engineering & Technology

Workbook

Modelling & Simulation
(SE-401)
Final Year

Prepared by

Mr. Amjad Ali

Approved by

Chairman
Department of Computer Science & Information Technology

Department of Computer Science & Information Technology
NED University of Engineering & Technology

Index

Practical#	Description
01	Matrix Operations
02	Solving matrix equations using matrix division
03	Vectorized functions and operators; more on graphs
04(A)	Some miscellaneous commands
04(B)	Conditions and Loops
05	A nontrivial example
06	Creating a Sparse matrix
07	Putting several graphs in one window
08	3D Plots
09	Parametric Plots
10	Efficiency in Matlab
11	Structures
12	Cell arrays
13	Logicals
14	While Loop
15	Fibbonaci Sequence
16	Pair Dice
17	Find for Vectors
18	Find For Vectors
19	Command Summary
20	Flight Trajectory
21	Height of a building
22	Lake Distance Problem
23	Matrix Operations
24	Motion Under Gravity
25	Signals and Plots
26	Well Depth Problem
27	3D Plots
28(A)	Minimizing a Function of One Variable
28(B)	Tank Design Problem
29	Factorial
30	Coin Flipping
31	Generating Discontinuous Signals
32	Height and Speed of a Projectile
33	Soda Can Cooling Problem
34	Speech Recognition
35	Ship Sailing Problem

Practical # 01

Matrix Operations

```
>> A = [1 2 3;4 5 6;7 8 9];
>> B = [1 1 1;2 2 2;3 3 3];
>> C = [1 2;3 4;5 6];
>> whos
  Name      Size         Bytes  Class

  A         3x3             72  double array
  B         3x3             72  double array
  C         3x2             48  double array

Grand total is 24 elements using 192 bytes

>> A+B
ans =
     2     3     4
     6     7     8
    10    11    12

>> A+C

??? Error using ==> +
Matrix dimensions must agree.
```

Matrix multiplication is also defined:

```
>> A*C
ans =
    22    28
    49    64
    76   100

>> C*A

??? Error using ==> *

Inner matrix dimensions must agree.
```

If A is a square matrix and m is a positive integer, then A^m is the product of m factors of A.

However, no notion of multiplication is defined for multi-dimensional arrays with more than 2 dimensions:

```
>> C = cat(3,[1 2;3 4],[5 6;7 8])

C(:,:,1) =
     1     2
     3     4

C(:,:,2) =
     5     6
     7     8

>> D = [1;2]
D =
     1
     2

>> whos
  Name      Size         Bytes  Class

  C         2x2x2           64  double array
  D         2x1             16  double array

Grand total is 10 elements using 80 bytes

>> C*D

??? Error using ==> *

No functional support for matrix inputs.
```

Practical # 02

Solving matrix equations using matrix division

If *A* is a square, nonsingular matrix, then the solution of the equation $Ax=b$ is . Matlab implements this operation with the backslash operator:

```
>> A = rand(3,3)
A =
    0.2190    0.6793    0.5194
    0.0470    0.9347    0.8310
    0.6789    0.3835    0.0346

>> b = rand(3,1)
b =
    0.0535
    0.5297
    0.6711

>> x = A\b
x =
 -159.3380
  314.8625
 -344.5078

>> A*x-b
ans =
   1.0e-13 *
   -0.2602
   -0.1732
   -0.0322
```

Practical # 03

Vectorized functions and operators; more on graphs

Matlab has many commands to create special matrices; the following command creates a row vector whose components increase arithmetically:

```
>> t = 1:5
t =
     1     2     3     4     5
```

The components can change by non-unit steps:

```
>> x = 0:.1:1
x =
  Columns 1 through 7
         0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000
  Columns 8 through 11
    0.7000    0.8000    0.9000    1.0000
```

A negative step is also allowed. The command linspace has similar results; it creates a vector with linearly spaced entries. Specifically, linspace(a,b,n) creates a vector of length *n*

```
>> linspace(0,1,11)
ans =
  Columns 1 through 7
         0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000
  Columns 8 through 11
    0.7000    0.8000    0.9000    1.0000
```

There is a similar command logspace for creating vectors with logarithmically spaced entries:

```
>> logspace(0,1,11)
ans =
  Columns 1 through 7
    1.0000    1.2589    1.5849    1.9953    2.5119    3.1623    3.9811
  Columns 8 through 11
    5.0119    6.3096    7.9433   10.0000
```

Practical # 04(A)

Some miscellaneous commands

An important operator in Matlab is the single quote, which represents the (conjugate) transpose:

```
>> A = [1 2;3 4]
A =
    1    2
    3    4

>> A'
ans =
    1    3
    2    4

>> B = A + i*.5*A
B =
  1.0000 + 0.5000i   2.0000 + 1.0000i
  3.0000 + 1.5000i   4.0000 + 2.0000i

>> B'
ans =
  1.0000 - 0.5000i   3.0000 - 1.5000i
  2.0000 - 1.0000i   4.0000 - 2.0000i
```

In the rare event that the transpose, rather than the conjugate transpose, is needed, the ``.'" operator is used:

```
>> B.'
ans =
  1.0000 + 0.5000i   3.0000 + 1.5000i
  2.0000 + 1.0000i   4.0000 + 2.0000i
```

(note that ' and .' are equivalent for matrices with real entries).

The following commands are frequently useful; more information can be obtained from the on-line help system.

Creating matrices

- zeros(m,n) creates an mxn matrix of zeros;
- ones(m,n) creates an mxn matrix of ones;
- eye(n) creates the mxn identity matrix;

- diag(v) (assuming v is an n-vector) creates a diagonal matrix with v on the diagonal.

The commands zeros and ones can be given any number of integer arguments; with *k* arguments, they each create a *k*-dimensional array of the indicated size.

formatting display and graphics

- format
  - format short 3.1416
  - format short e 3.1416e+00
  - format long 3.14159265358979
  - format long e 3.141592653589793e+00
  - format compact suppresses extra line feeds (all of the output in this paper is in compact format).
- xlabel('string'), ylabel('string') label the horizontal and vertical axes, respectively, in the current plot;
- title('string') add a title to the current plot;
- grid adds a rectangular grid to the current plot;
- hold on freezes the current plot so that subsequent graphs will be displayed with the current;
- hold off releases the current plot; the next plot will erase the current before displaying;
- subplot puts multiple plots in one graphics window.

Miscellaneous

- max(x) returns the largest entry of x, if x is a vector; see help max for the result when x is a *k*-dimensional array;
- min(x) analogous to max;
- abs(x) returns an array of the same size as x whose entries are the magnitudes of the entries of x;
- size(A) returns a vector with the number of rows, columns, etc. of the *k*-dimensional array A;
- length(x) returns the ``length'' of the array, i.e. max(size(A)).
- save fname saves the current variables to the file named fname.mat;
- load fname load the variables from the file named fname.mat;
- quit exits Matlab

Practical # 04(B)

Conditions and Loops

Matlab has a standard if-elseif-else conditional; for example:

```
>> t = rand(1);
>> if t > 0.75
     s = 0;
   elseif t < 0.25
     s = 1;
   else
     s = 1-2*(t-0.25);
   end
>> s
s =
    0
>> t
t =
   0.7622
```

The logical operators in Matlab are <, >, <=, >=, == (logical equals), and ~= (not equal). These are binary operators which return the values 0 and 1 (for scalar arguments):

```
>> 5>3
ans =
    1
>> 5<3
ans =
    0
>> 5==3
ans =
    0
```

Thus the general form of the if statement is

```
if expr1
   statements
elseif expr2
   statements
 .
 .
else
   statements
end
```

The first block of statements following a nonzero *expr* executes.

Matlab provides two types of loops, a for-loop (comparable to a Fortran do-loop or a C for-loop) and a while-loop. A for-loop repeats the statements in the loop as the loop index takes on the values in a given row vector:

```
>> for i=[1,2,3,4]
     disp(i^2)
   end
    1
    4
    9
   16
```

(Note the use of the built-in function disp, which simply displays its argument.) The loop, like an if-block, must be terminated by end. This loop would more commonly be written as

```
>> for i=1:4
     disp(i^2)
   end
    1
    4
    9
   16
```

(recall that 1:4 is the same as [1,2,3,4]).

The while-loop repeats as long as the given expr is true (nonzero):

```
>> x=1;
>> while 1+x > 1
     x = x/2;
   end
>> x
x =
   1.1102e-16
```

Practical # 05

A nontrivial example

A general algorithm for nonlinear root-finding is the method of bisection, which takes a function and an interval on which function changes sign, and repeatedly bisects the interval until the root is trapped in a very small interval.

A function implementing the method of bisection illustrates many of the important techniques of programming in Matlab. The first important technique, without which a useful bisection routine cannot be written, is the ability to pass the name of one function to another function. In this case, bisect needs to know the name of the function whose root it is to find. This name can be passed as a string (the alternative is to ``hard-code" the name in bisect.m, which means that each time one wants to use bisect with a different function, the file bisect.m must be modified. This style of programming is to be avoided.).

The built-in function feval is needed to evaluate a function whose name is known (as a string). Thus, interactively

```
>> fcn(2)
ans =
   -0.7568
```

and

```
>> feval('fcn',2)
ans =
   -0.7568
```

are equivalent (notice that single quotes are used to delimit a string). A variable can also be assigned the value of a string:

```
>> str = 'fcn'
str =
f

>> feval(str,2)
ans =
   -0.7568
```

See help strings for information on how Matlab handles strings.

The following Matlab program uses the string facility to pass the name of a function to bisect. A % sign indicates that the rest of the line is a comment.

```
function c = bisect(fn,a,b,tol)
```

```
% c = bisect('fn',a,b,tol)
%
%   This function locates a root of the function fn on the interval
%   [a,b] to within a tolerance of tol.  It is assumed that the function
%   has opposite signs at a and b.

% Evaluate the function at the endpoints and check to see if it
% changes sign.

fa = feval(fn,a);
fb = feval(fn,b);

if fa*fb >= 0
  error('The function must have opposite signs at a and b')
end

% The flag done is used to flag the unlikely event that we find
% the root exactly before the interval has been sufficiently reduced.

done = 0;

% Bisect the interval

c = (a+b)/2;

% Main loop

while abs(a-b) > 2*tol & ~done

  % Evaluate the function at the midpoint

  fc = feval(fn,c);

  if fa*fc < 0       % The root is to the left of c
    b = c;
    fb = fc;
    c = (a+b)/2;
  elseif fc*fb < 0   % The root is to the right of c
    a = c;
    fa = fc;
    c = (a+b)/2;
  else               % We landed on the root
    done = 1;
  end

end
```

Assuming that this file is named bisect.m, it can be run as follows:

```
>> x = bisect('fcn',1,2,1e-6)
x =
    1.7725
>> sqrt(pi)-x
ans =
  -4.1087e-07
```

Not only can new Matlab commands be created with m-files, but the help system can be automatically extended. The help command will print the first comment block from an m-file:

```
>> help bisect

  c = bisect('fn',a,b,tol)

    This function locates a root of the function fn on the interval
    [a,b] to within a tolerance of tol.  It is assumed that the function
    has opposite signs at a and b.
```

(Something that may be confusing is the use of both fn and 'fn' in bisect.m. I put quotes around fn in the comment block to remind the user that a string must be passed. However, the variable fn is a string *variable* and does not need quotes in any command line.)

Notice the use of the error function near the beginning of the program. This function displays the string passed to it and exits the m-file.

At the risk of repeating myself, I want to re-emphasize a potentially troublesome point. In order to execute an m-file, Matlab must be able to find it, which means that it must be found in a directory in Matlab's path. The current working directory is always on the path; to display or change the path, use the path command. To display or change the working directory, use the cd command. As usual, help will provide more information.

Practical # 06

Creating a Sparse matrix

If a matrix A is stored in ordinary (dense) format, then the command S = sparse(A) creates a copy of the matrix stored in sparse format. For example:

```
>> A = [0 0 1;1 0 2;0 -3 0]
A =
     0     0     1
     1     0     2
     0    -3     0
>> S = sparse(A)
S =
   (2,1)        1
   (3,2)       -3
   (1,3)        1
   (2,3)        2

>> whos
  Name      Size         Bytes  Class

  A         3x3             72  double array
  S         3x3             64  sparse array

Grand total is 13 elements using 136 bytes
```

Unfortunately, this form of the sparse command is not particularly useful, since if A is large, it can be very time-consuming to first create it in dense format. The command S = sparse(m,n) creates an zero matrix in sparse format. Entries can then be added one-by-one:

```
>> A = sparse(3,2)
A =
   All zero sparse: 3-by-2
>> A(1,2)=1;
>> A(3,1)=4;
>> A(3,2)=-1;
>> A
A =
   (3,1)        4
   (1,2)        1
   (3,2)       -1
```

(Of course, for this to be truly useful, the nonzeros would be added in a loop.)

Another version of the sparse command is S = sparse(I,J,S,m,n,maxnz). This creates an sparse matrix with entry ($I(k)$,$J(k)$) equal to . The optional argument maxnz causes Matlab to pre-allocate storage for maxnz nonzero entries, which can increase efficiency in the case when more nonzeros will be added later to S.

There are still more versions of the sparse command. See help sparse for details.

The most common type of sparse matrix is a banded matrix, that is, a matrix with a few nonzero diagonals. Such a matrix can be created with the spdiags command. Consider the following matrix:

```
>> A
A =
    64   -16     0   -16     0     0     0     0     0
   -16    64   -16     0   -16     0     0     0     0
     0   -16    64     0     0   -16     0     0     0
   -16     0     0    64   -16     0   -16     0     0
     0   -16     0   -16    64   -16     0   -16     0
     0     0   -16     0   -16    64     0     0   -16
     0     0     0   -16     0     0    64   -16     0
     0     0     0     0   -16     0   -16    64   -16
     0     0     0     0     0   -16     0   -16    64
```

This is a matrix with 5 nonzero diagonals. In Matlab's indexing scheme, the nonzero diagonals of A are numbers -3, -1, 0, 1, and 3 (the main diagonal is number 0, the first subdiagonal is number -1, the first superdiagonal is number 1, and so forth). To create the same matrix in sparse format, it is first necessary to create a matrix containing the nonzero diagonals of A. Of course, the diagonals, regarded as column vectors, have different lengths; only the main diagonal has length 9. In order to gather the various diagonals in a single matrix, the shorter diagonals must be padded with zeros. The rule is that the extra zeros go at the bottom for subdiagonals and at the top for superdiagonals. Thus we create the following matrix:

```
>> B = [
   -16   -16    64     0     0
   -16   -16    64   -16     0
   -16     0    64   -16     0
   -16   -16    64     0   -16
   -16   -16    64   -16   -16
   -16     0    64   -16   -16
     0   -16    64     0   -16
     0   -16    64   -16   -16
     0     0    64   -16   -16
];
```

(notice the technique for entering the rows of a large matrix on several lines). The spdiags command also needs the indices of the diagonals:

```
>> d = [-3,-1,0,1,3];
```

The matrix is then created as follows:
S = spdiags(B,d,9,9);
The last two arguments give the size of S.

Perhaps the most common sparse matrix is the identity. Recall that an identity matrix can be created, in dense format, using the command eye. To create the identity matrix in sparse format, use I = speye(n).

Another useful command is spy, which creates a graphic displaying the sparsity pattern of a matrix. For example, the above penta-diagonal matrix A can be displayed by the following command;

>> spy(A)

Practical # 07

Putting several graphs in one window

The subplot command creates several plots in a single window. To be precise, subplot(m,n,i) creates *mn* plots, arranged in an array with *m* rows and *n* columns. It also sets the next plot command to go to the *i*th coordinate system (counting across the rows). Here is an example

```
>> t = (0:.1:2*pi)';
>> subplot(2,2,1)
>> plot(t,sin(t))
>> subplot(2,2,2)
>> plot(t,cos(t))
>> subplot(2,2,3)
>> plot(t,exp(t))
>> subplot(2,2,4)
>> plot(t,1./(1+t.^2))
```

Practical # 08

3D Plots

In order to create a graph of a surface in 3-space (or a contour plot of a surface), it is necessary to evaluate the function on a regular rectangular grid. This can be done using the meshgrid command. First, create 1D vectors describing the grids in the *x*- and *y*-directions:

```
>> x = (0:2*pi/20:2*pi)';
>> y = (0:4*pi/40:4*pi)';
```

Next, ``spread" these grids into two dimensions using meshgrid:

```
>> [X,Y] = meshgrid(x,y);
>> whos
```

```
  Name      Size         Bytes  Class

  X        41x21         6888  double array
  Y        41x21         6888  double array
  x        21x1           168  double array
  y        41x1           328  double array
```

Grand total is 1784 elements using 14272 bytes
The effect of meshgrid is to create a vector X with the *x*-grid along each row, and a vector Y with the *y*-grid along each column. Then, using vectorized functions and/or operators, it is easy to evaluate a function $z = f(x,y)$ of two variables on the rectangular grid:

```
>> z = cos(X).*cos(2*Y);
```

Having created the matrix containing the samples of the function, the surface can be graphed using either the mesh or the surf commands

```
>> mesh(x,y,z)
>> surf(x,y,z)
>> contour(x,y,z)
```

Practical # 09

Parametric Plots

It is easy to graph a curve ($f(t)$,$g(t)$) in 2-space. For example (see Figure 11):

```
>> t = (0:2*pi/100:2*pi)';
>> plot(cos(t),sin(t))
>> axis('square')
```

Practical # 10

Efficiency in Matlab

User-defined Matlab functions are interpreted, not compiled. This means roughly that when an m-file is executed, each statement is read and then executed, rather than the entire program being parsed and compiled into machine language. For this reason, Matlab programs can be much slower than programs written in a language such as Fortran or C.

In order to get the most out of Matlab, it is necessary to use built-in functions and operators whenever possible (so that compiled rather than interpreted code is executed). For example, the following two command sequences have the same effect:

```
>> t = (0:.001:1)';
>> y=sin(t);
```

and

```
>> t = (0:.001:1)';
>> for i=1:length(t)
     y(i) = sin(t(i));
   end
```

Practical #11

Structures

A structure is a data type which contains several values, possibly of different types, referenced by name. The simplest way to create a structure is by simple assignment. For example, consider the function

The following m-file f.m computes the value, gradient, and Hessian of *f* at a point *x*, and returns them in a structure:

```
function fx = f(x)

fx.Value = (x(1)-1)^2+x(1)*x(2);
fx.Gradient = [2*(x(1)-1)+x(2);x(1)];
fx.Hessian = [2 1;1 0];
```

We can now use the function as follows:

```
>> x = [2;1]
x =
     2
     1
>> fx = f(x)
fx =
       Value: 3
    Gradient: [2x1 double]
     Hessian: [2x2 double]

>> whos

  Name      Size         Bytes  Class

  fx        1x1            428  struct array
  x         2x1             16  double array

Grand total is 12 elements using 444 bytes
```

The potential of structures for organizing information in a program should be obvious.

Note that, in the previous example, Matlab reports fx as being a ``struct array". We can have multi-dimensional arrays of structs, but in this case, each struct must have the same field names:

```
>> gx.Value = 12;
>> gx.Gradient = [2;1];
>> A(1,1) = fx;
>> A(2,1) = gx;

??? Subscripted assignment between dissimilar structures.

>> fieldnames(fx)
ans =
    'Value'
    'Gradient'
    'Hessian'
>> fieldnames(gx)
ans =
    'Value'
    'Gradient'
```

(Note the use of the command fieldnames, which lists the field names of a structure.)

Beyond simple assignment, there is a command struct for creating structures. For information on this and other commands for manipulating structures, see help struct.

Practical # 12

Cell arrays

A *cell array* is an array whose entries can be data of any type. The index operator {} is used in place of () to indicate that an array is a cell array instead of an ordinary array:

```
>> x = [2;1];
>> fx = f(x);
>> whos

  Name      Size        Bytes  Class

  fx        1x1           428  struct array
  x         2x1            16  double array

Grand total is 12 elements using 444 bytes

>> A{1}=x
A =
    [2x1 double]
>> A{2}=fx
A =
    [2x1 double]    [1x1 struct]

>> whos

  Name      Size        Bytes  Class

  A         1x2           628  cell array
  fx        1x1           428  struct array
  x         2x1            16  double array

Grand total is 26 elements using 1072 bytes
```

Another way to create the same cell array is to place the entries inside of curly braces:

```
>> B = {x,fx}
B =
    [2x1 double]    [1x1 struct]
```

Practical # 13

Logicals

Matlab represents `true` and `false` by means of the integers 0 and 1.

`true = 1,  false = 0`

If at some point in a calculation a scalar `x`, say, has been assigned a value, we may make certain logical tests on it:

`x == 2`	is `x` equal to 2?
`x ~= 2`	is `x` **not** equal to 2?
`x > 2`	is `x` greater than 2?
`x < 2`	is `x` less than 2?
`x >= 2`	is `x` greater than or equal to 2?
`x <= 2`	is `x` less than or equal to 2?

Pay particular attention to the fact that the test for equality involves two equal signs `==`.

```
>> x = pi
x =
    3.1416
>> x ~= 3,  x ~= pi
ans =
     1
ans =
     0
```

When x is a vector or a matrix, these tests are performed elementwise:

```
x =
   -2.0000    3.1416    5.0000
   -1.0000         0    1.0000
>> x == 0
ans =
     0     0     0
     0     1     0
>> x > 1,  x >=-1
ans =
     0     1     1
     0     0     0
ans =
     0     1     1
     1     1     1
>> y = x>=-1, x > y
y =
     0     1     1
     1     1     1
ans =
     0     1     1
     0     0     0
```

We may combine logical tests, as in

```
>> x
x =
   -2.0000    3.1416    5.0000
   -5.0000   -3.0000   -1.0000
>> x > 3 & x < 4
ans =
     0     1     0
     0     0     0
>> x > 3 | x == -3
ans =
     0     1     1
     0     1     0
```

As one might expect, & represents **and** and (not so clearly) the vertical bar | means **or**; also ~ means **not** as in ~= (not equal), ~(x>0), etc.

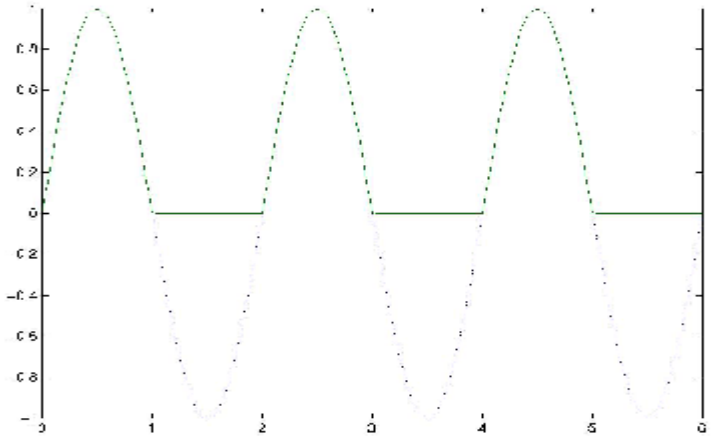
```
>> x > 3 | x == -3 | x <= -5
ans =
     0     1     1
     1     1     0
```

One of the uses of logical tests is to "mask out" certain elements of a matrix.

```
>> x, L =  x >= 0
x =
   -2.0000    3.1416    5.0000
   -5.0000   -3.0000   -1.0000
L =
     0     1     1
     0     1     1
>> pos = x.*L
pos =
         0    3.1416    5.0000
         0         0         0
```

so the matrix **pos** contains just those elements of **x** that are non–negative.

```
>> x = 0:0.05:6; y = sin(pi*x); Y = (y>=0).*y;
>> plot(x,y,':',x,Y,'-' )
```



Practical # 14

While Loop

There are some occasions when we want to repeat a section of Matlab code until some logical condition is satisfied, but we cannot tell in advance how many times we have to go around the loop. This we can do with a `while...end` construct.

Example 20.1 *What is the greatest value of n that can be used in the sum*

$$1^2 + 2^2 + \cdots + n^2$$

and get a value of less than 100?

```
>> S = 1; n = 1;
>> while S+ (n+1)^2 < 100
    n = n+1;    S = S + n^2;
   end
>> [n, S]
ans =
     6    91
```

The lines of code between `while` and `end` will only be executed if the condition `S+ (n+1)^2 < 100` is true.

Exercise

Find the approximate value of the root x *of the equation* $x = \cos x$.

```
>> x = zeros(1,20); x(1) = pi/4;
>> n = 1;  d = 1;
>> while d > 0.001
     n = n+1; x(n) = cos(x(n-1));
     d = abs( x(n) - x(n-1) );
   end
   n,x
n =
    14
x =
Columns 1 through 7
0.7854 0.7071 0.7602 0.7247 0.7487 0.7326 0.7435
Columns 8 through 14
0.7361 0.7411 0.7377 0.7400 0.7385 0.7395 0.7388
Columns 15 through 20
   0    0    0    0    0    0
```

Practical # 15

Fibbonaci Sequence

```
function f = Fib1(n)
% Returns the nth number in the
% Fibonacci sequence.
F=zeros(1,n+1);
F(2) = 1;
   for i = 3:n+1
     F(i) = F(i-1) + F(i-2);
   end
f = F(n);
```

Practical # 16

Pair Dice

Write a function–file that will simulate n throws of a pair of dice.

This requires random numbers that are integers in the range 1 to 6. Multiplying each random number by 6 will give a real number in the range 0 to 6; rounding these to whole numbers will not be correct since it will then be possible to get 0 as an answer. We need to use

```
floor(1 + 6*rand)
```

```
function  [d] = dice(n)
% simulates "n" throws of a pair of dice
% Input:     n, the number of throws
% Output:    an n times 2 matrix, each row
%            referring to one throw.
%
% Useage:  T = dice(3)
    d = floor(1 + 6*rand(n,2));
%% end of dice
```

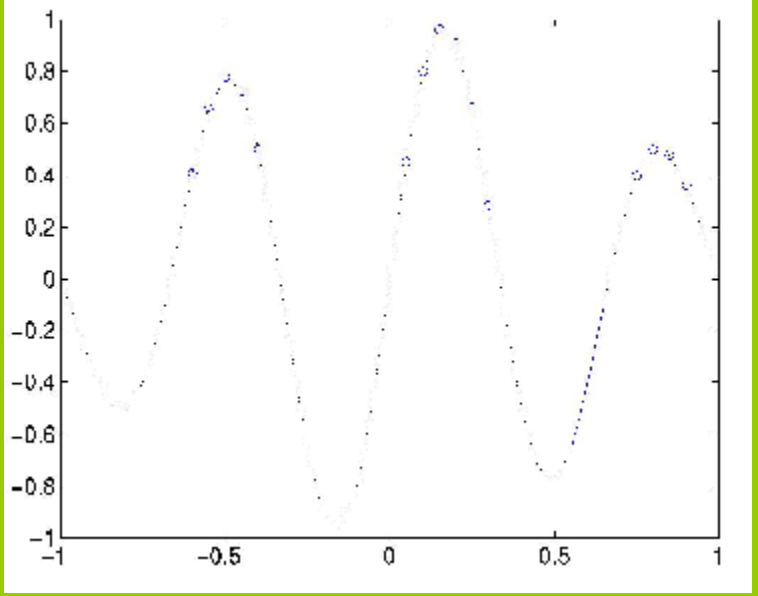
```
>> dice(3)
ans =
     6     1
     2     3
     4     1
>> sum(dice(100))/100
ans =
    3.8500    3.4300
```

Practical # 17

Find for Vectors

The function "find" returns a list of the positions (indices) of the elements of a vector satisfying a given condition. For example,

```
>> x = -1:.05:1;
>> y = sin(3*pi*x).*exp(-x.^2); plot(x,y,':')
>> k = find(y > 0.2)
 k =
  Columns 1 through 12
    9   10   11   12   13   22   23   24   25   26   27   36
  Columns 13 through 15
  37   38   39
>> hold on, plot(x(k),y(k),'o')
>> km = find( x>0.5 & y<0)
km =
    32    33    34
>> plot(x(km),y(km),'-')
```



Practical # 18

Find For Vectors

Example 27.1 *Suppose a sound pressure spectrum is to be plotted in a graph. There are four alternative plot formats; lin-lin, lin-log, log-lin and log-log. The graphic user interface below reads the pressure data stored on a binary file selected by the user, plots it in a lin-lin format as a function of frequency and lets the user switch between the four plot formats.*

We use two m–files. The first (`specplot.m`) is the main driver file which builds the graphics window. It calls the second file (`firstplot.m`) which allows the user to select among the possible `*.bin` files in the current directory.

```
figWindow = figure('Name','Plot alternatives');
      % Create file input selection button
fileinpBtn = uicontrol('Style','pushbutton',...
  'string','File','position',[5,395,40,20],...
  'callback','[fdat,pdat] = firstplot;');
      % Press 'File' calls function 'firstplot'
% Create pushbuttons for switching between four
% different plot formats. Set up the axis stings.
X = 'Frequency, [Hz]';
Y = 'Pressure amplitude, [Pa]';
linlinBtn = uicontrol('style','pushbutton',...
  'string','lin-lin',...
  'position',[200,395,40,20],'callback',...
  'plot(fdat,pdat);xlabel(X);ylabel(Y);');
linlogBtn = uicontrol('style','pushbutton',...
  'string','lin-log',...
  'position',[240,395,40,20],...
  'callback',...
  'semilogy(fdat,pdat);xlabel(X);ylabel(Y);');
loglinBtn = uicontrol('style','pushbutton',...
  'string','log-lin',...
  'position',[280,395,40,20],...
  'callback',...
  'semilogx(fdat,pdat);xlabel(X);ylabel(Y);');
```

```
loglogBtn = uicontrol('style','pushbutton',...
   'string','log-log',...
   'position',[320,395,40,20],...
   'callback',...
   'loglog(fdat,pdat);xlabel(X);    ylabel(Y);');

% Create exit pushbutton with red text.

exitBtn = uicontrol('Style','pushbutton',...
   'string','EXIT','position',[510,395,40,20],...
   'foregroundcolor',[1 0 0],'callback','close;');
```

```
% Script file: firstplot.m
% Brings template for file selection. Reads
% selected filename and path and plots
% spectrum in a  lin-lin diagram.
% Output data are frequency and pressure
% amplitude vectors: 'fdat' and  'pdat'.
% Author: U Carlsson, 2001-08-22

function [fdat,pdat] = firstplot

% Call Matlab function 'uigetfile' that
% brings file selction template.

[filename,pathname] = uigetfile('*.bin',...
   'Select binary data-file:');
% Change directory
cd(pathname);
% Open file for reading binary floating
% point numbers.
fid = fopen(filename,'rb');
data = fread(fid,'float32');
% Close file
fclose(fid);
% Partition data vector in frequency and
% pressure vectors
pdat = data(2:2:length(data));
fdat = data(1:2:length(data));
% Plot pressure signal in a lin-lin diagram
plot(fdat,pdat);
% Define suitable axis labels
xlabel('Frequency, [Hz]');
ylabel('Pressure amplitude, [Pa]');
```

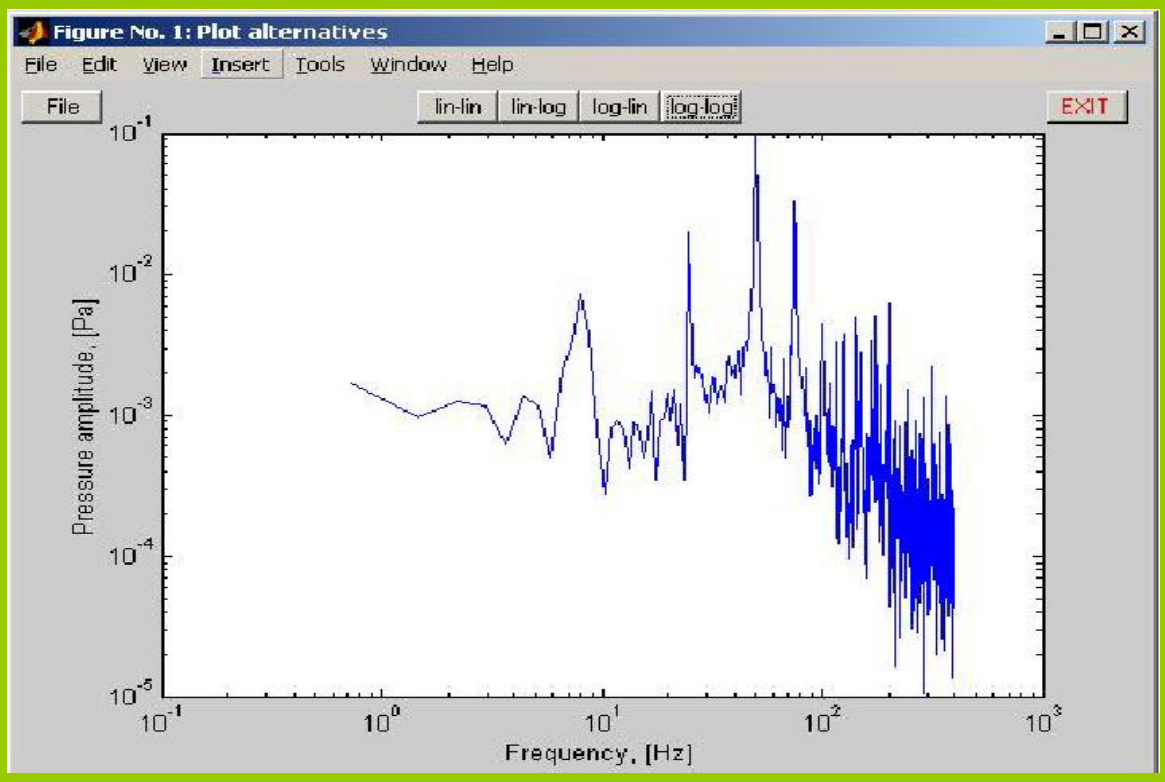


Figure No. 1: Plot alternatives
File Edit View Insert Tools Window Help
File
lin-lin
lin-log
log-lin
log-log
EXIT
Pressure amplitude, [Pa]
Frequency, [Hz]

Practical # 19

Command Summary

abs	Absolute value
sqrt	Square root function
sign	Signum function
conj	Conjugate of a complex number
imag	Imaginary part of a complex number
real	Real part of a complex number
angle	Phase angle of a complex number
cos	Cosine function
sin	Sine function
tan	Tangent function
exp	Exponential function
log	Natural logarithm
log10	Logarithm base 10
cosh	Hyperbolic cosine function
sinh	Hyperbolic sine function
tanh	Hyperbolic tangent function
acos	Inverse cosine
acosh	Inverse hyperbolic cosine
asin	Inverse sine
asinh	Inverse hyperbolic sine
atan	Inverse tan
atan2	Two–argument form of inverse tan
atanh	Inverse hyperbolic tan
round	Round to nearest integer
floor	Round towards minus infinity
fix	Round towards zero
ceil	Round towards plus infinity
rem	Remainder after division

bin.

Managing commands and functions.	
`help`	On-line documentation.
`doc`	Load hypertext documentation.
`what`	Directory listing of M-, MAT- and MEX-files.
`type`	List M-file.
`lookfor`	Keyword search through the HELP entries.
`which`	Locate functions and files.
`demo`	Run demos.
Managing variables and the workspace.	
`who`	List current variables.
`whos`	List current variables, long form.
`load`	Retrieve variables from disk.
`save`	Save workspace variables to disk.
`clear`	Clear variables and functions from memory.
`size`	Size of matrix.
`length`	Length of vector.
`disp`	Display matrix or text.
Controlling the command window.	
`cedit`	Set command line edit/recall facility parameters.
`clc`	Clear command window.
`home`	Send cursor home.
`format`	Set output format.
`echo`	Echo commands inside script files.
`more`	Control paged output in command window.
Quitting from MATLAB.	
`quit`	Terminate MATLAB.

Matrix analysis.	
`cond`	Matrix condition number.
`norm`	Matrix or vector norm.
`rcond`	LINPACK reciprocal condition estimator.
`rank`	Number of linearly independent rows or columns.
`det`	Determinant.
`trace`	Sum of diagonal elements.
`null`	Null space.
`orth`	Orthogonalization.
`rref`	Reduced row echelon form.
Linear equations.	
\ and /	Linear equation solution; use "help slash".
`chol`	Cholesky factorization.
`lu`	Factors from Gaussian elimination.
`inv`	Matrix inverse.
`qr`	Orthogonal- triangular decomposition.
`qrdelete`	Delete a column from the QR factorization.
`qrinsert`	Insert a column in the QR factorization.
`nnls`	Non–negative least- squares.
`pinv`	Pseudoinverse.
`lscov`	Least squares in the presence of known covariance.

Eigenvalues and singular values.	
`eig`	Eigenvalues and eigenvectors.
`poly`	Characteristic polynomial.
`polyeig`	Polynomial eigenvalue problem.
`hess`	Hessenberg form.
`qz`	Generalized eigenvalues.
`rsf2csf`	Real block diagonal form to complex diagonal form.
`cdf2rdf`	Complex diagonal form to real block diagonal form.
`schur`	Schur decomposition.
`balance`	Diagonal scaling to improve eigenvalue accuracy.
`svd`	Singular value decomposition.
Matrix functions.	
`expm`	Matrix exponential.
`expm1`	M- file implementation of expm.
`expm2`	Matrix exponential via Taylor series.
`expm3`	Matrix exponential via eigenvalues and eigenvectors.
`logm`	Matrix logarithm.
`sqrtm`	Matrix square root.
`funm`	Evaluate general matrix function.

Graphics & plotting.	
`figure`	Create Figure (graph window).
`clf`	Clear current figure.
`close`	Close figure.
`subplot`	Create axes in tiled positions.
`axis`	Control axis scaling and appearance.
`hold`	Hold current graph.
`figure`	Create figure window.
`text`	Create text.
`print`	Save graph to file.
`plot`	Linear plot.
`loglog`	Log-log scale plot.
`semilogx`	Semi-log scale plot.
`semilogy`	Semi-log scale plot.

Specialized X-Y graphs.	
`polar`	Polar coordinate plot.
`bar`	Bar graph.
`stem`	Discrete sequence or "stem" plot.
`stairs`	Stairstep plot.
`errorbar`	Error bar plot.
`hist`	Histogram plot.
`rose`	Angle histogram plot.
`compass`	Compass plot.
`feather`	Feather plot.
`fplot`	Plot function.
`comet`	Comet-like trajectory.

Graph annotation.	
`title`	Graph title.
`xlabel`	X-axis label.
`ylabel`	Y-axis label.
`text`	Text annotation.
`gtext`	Mouse placement of text.
`grid`	Grid lines.
`contour`	Contour plot.
`mesh`	3-D mesh surface.
`surf`	3-D shaded surface.
`waterfall`	Waterfall plot.
`view`	3-D graph viewpoint specification.
`zlabel`	Z-axis label for 3-D plots.
`gtext`	Mouse placement of text.
`grid`	Grid lines.

Practical # 20
Flight Trajectory

Computational Implementation:

The equations defined in the computational method can be readily implemented using MATLAB. The commands used in the following solution will be discussed in detail later in the course, but observe that the MATLAB steps match closely to the solution steps from the computational method.

```
% Flight trajectory computation
%
% Initial values
g = 32.2;                          % gravity, ft/s^2
v = 50 * 5280/3600;                % launch velocity, ft/s
theta = 30 * pi/180;               % launch angle, radians

% Compute and display results
disp('time of flight (s):')        % label for time of flight
tg = 2 * v * sin(theta)/g          % time to return to ground, s
disp('distance traveled (ft):')    % label for distance
xg = v * cos(theta) * tg          % distance traveled

% Compute and plot flight trajectory
t = linspace(0,tg,256);
x = v * cos(theta) * t;
y = v * sin(theta) * t - g/2 * t.^2;
plot(x,y), axis equal, axis([ 0 150 0 30 ]), grid, ...
   xlabel('Distance (ft)'), ylabel('Height (ft)'), title('Flight Trajectory')
```

- Words following percent signs (%) are comments to help in reading the MATLAB statements.
- A word on the left of an equals sign is known as a **variable name** and it will be assigned to the value or values on the right of the equals sign. Commands having this form are known as **assignment statements**.
- If a MATLAB command assigns or computes a value, it will display the value on the screen if the statement does not end with a semicolon (;). Thus, the values of `v`, `g`, and `theta` will not be displayed. The values of `tg` and `xg` will be computed and displayed, because the statements that computes these values does not end with a semicolon.
- The three dots (...) at the end of a line mean that the statement continues on the next line.
- More than one statement can be entered on the same line if the statements are separated by commas.
- The statement `t = linspace(0,tg,256);` creates a vector of length 256.
- The expression `t.^2` squares *each element* in `t`, making another vector.
- In addition to the required results, the flight trajectory has also been computed and plotted. This will be used below in assessing the solution.
- The `plot` statement generates the plot of height against distance, complete with a title and labels on the x and y axes.

Testing and Assessing the Solution:

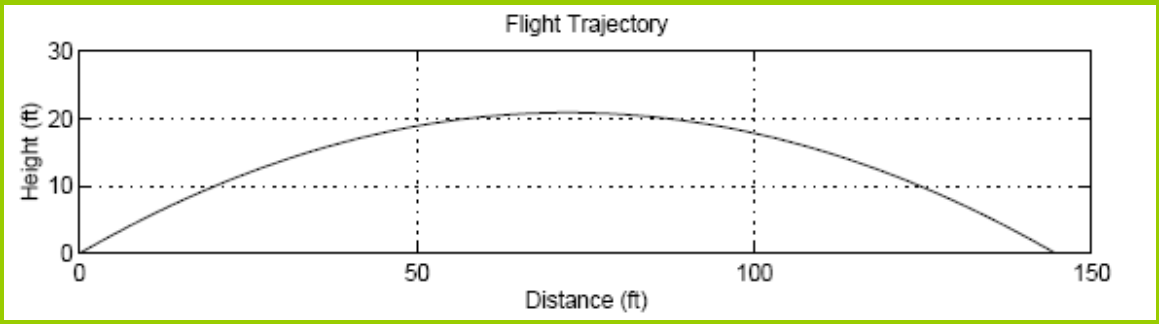
This problem is sufficiently simple that the results can be computed by hand with the use of a calculator. There is no need to generate test data to check the results. One could even question the need to use the power of MATLAB to solve this problem, since it is readily solved using a calculator. Of course, our objective here is to demonstrate the application of MATLAB to a problem with which you are familiar.

Executing the statements above provides the following displayed results:

```
time of flight (s):
tg =
    2.2774
distance traveled (ft):
xg =
  144.6364
```

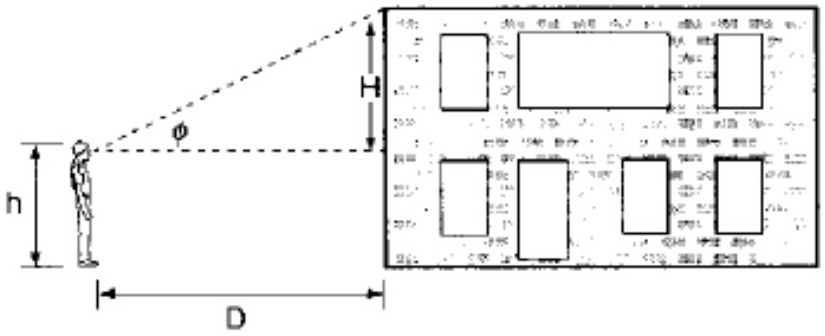
Computing these quantities with a calculator can be shown to produce the same results.

The flight trajectory plot that is generated is shown in Figure 1.2. This demonstrates another strength of MATLAB, as it has taken care of the details of the plot generation, including axis scaling, label placement, etc. This result can be used to assess the solution. Our physical intuition tell us that this result appears to be correct, as we know that we could throw a baseball moderately hard and have it travel a distance of nearly 150 feet and that maximum height of about 20 feet also seems reasonable. The shape of the trajectory also fits our observations.


Flight Trajectory
Height (ft)
30
20
10
0
0
50
100
150
Distance (ft)

Practical # 21
Height of a building

Problem: Consider the problem of estimating the height of a building, as illustrated in Fig. 4.1. If the observer is a distance D from the building, the angle from the observer to the top of the building is θ, and the height of the observer is h, what is the building height?



Solution: Draw a simple diagram as shown in Fig. 4.1. The building height B is

$$B = h + H$$

where H is the length of the triangle side opposite the observer, which can be found from the triangle relationship

$$\tan\theta = \frac{H}{D}$$

Therefore, the building height is

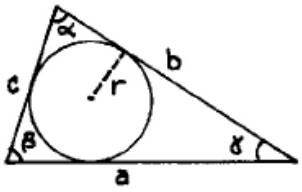
$$B = h + D \cdot \tan\theta$$

If h= 2 meters, D= 50 meters, and θ is 60 degrees, the MATLAB solution is

```
>> h=2;
>> theta=60;
>> D=50;
>> B = h+D*tan(theta*pi/180)
B =
   88.6025
```

Practical # 22
Lake Distance Problem

Solution of Triangles



Law of sines: $\frac{a}{\sin\alpha} = \frac{b}{\sin\beta} = \frac{c}{\sin\gamma}$

Law of cosines: $a^2 = b^2 + c^2 - 2bc\cos\alpha$

Law of tangents: $\frac{a-b}{a+b} = \frac{\tan\frac{1}{2}(\alpha-\beta)}{\tan\frac{1}{2}(\alpha+\beta)}$

Included angles: $\alpha + \beta + \gamma = \pi$ radians $= 180°$

In Figure 4.2, measurements of distances and angles around a lake have been indicated. The problem is to use these measurements to calculate the distance DE across the lake.

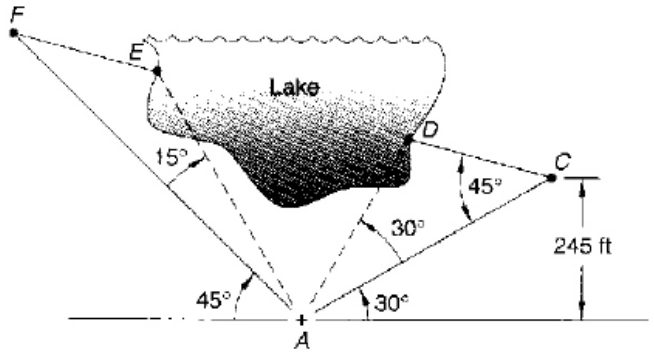


Figure 4.2: Distance and angle measurements near a lake

The solution is to sequentially apply the various triangle relations to determine the distances CF, CD, and EF, which can then be used to determine the specified distance DE.

First, determine the length of the sides of triangle ACF, beginning with the right triangle relationship on the bottom right:

$$\sin 30° = \frac{245}{AC}$$

Solving this equation for AC:

$$AC = \frac{245}{\sin 30^\circ}$$

From the included angles in triangle ACD:

$$D = 180^\circ - 30^\circ - 45^\circ$$

Applying the law of sines to triangle ACD:

$$\frac{CD}{\sin 30^\circ} = \frac{AC}{\sin D}$$

Solving for CD:

$$CD = AC\frac{\sin 30^\circ}{\sin D}$$

From the construction of angle A in triangle ACF:

$$A = 180^\circ - 30^\circ - 45^\circ$$

Triangles ACF and ACD are similar (have the same angles), which means that the lengths of corresponding sides are proportional:

$$\frac{AC}{CF} = \frac{CD}{AC}$$

Solving for CF:

$$CF = \frac{(AC)^2}{CD}$$

From the similar triangle, angle F must be the same as angle CAD, so $F = 30^\circ$.

Applying the Law of sines to triangle ACF:

$$\frac{AF}{\sin 45^\circ} = \frac{AC}{\sin F}$$

Solving for AF:

$$AF = AC\frac{\sin 45^\circ}{\sin F}$$

From the included angles in triangle AEF:

$$E = 180° - 15° - F$$

Applying the Law of sines to triangle AEF:

$$\frac{EF}{\sin 15°} = \frac{AF}{\sin E}$$

Solving for EF:

$$EF = AF\frac{\sin 15°}{\sin E}$$

Finally, the distance DE across the lake is:

$$DE = CF - CD - EF$$

A MATLAB script to compute DE, with angles in degrees and distances in feet:

A MATLAB script to compute DE, with angles in degrees and distances in feet:

```
% Lake distance problem
%
AC = 245/sin(30*pi/180)
D = 180-30-45
CD = AC*sin(30*pi/180)/sin(D*pi/180)
CF = AC^2/CD
F = 30
AF = AC*sin(45*pi/180)/sin(F*pi/180)
E = 180-15-F
EF = AF*sin(15*pi/180)/sin(E*pi/180)
DE = CF - CD - EF
```

The displayed results from MATLAB:

```
AC =
  490.0000
D =
   105
CD =
  253.6427
CF =
  946.6073
F =
   30
```

```
AF =
  692.9646
E =
   135
EF =
  253.6427
DE =
  439.3220
```

The final step in the solution to this problem is to carefully consider the results to determine if they make sense. This is done visually by comparing the computed lengths with those of Fig. 4.2.

Practical # 23
Matrix Operations

Submatrix creation: To create a submatrix using the colon operator, consider:

```
>> c = [1,0,0; 1,1,0; 1,-1,0; 0,0,2]
c =
     1     0     0
     1     1     0
     1    -1     0
     0     0     2
>> c_1 = c(:,2:3)
c_1 =
     0     0
     1     0
    -1     0
     0     2
>> c_2 = c(3:4,1:2)
c_2 =
     1    -1
     0     0
```

Thus, `c_1` contains all rows and columns 2 and 3 of `c` and `c_2` contains rows 3 and 4 of columns 1 and 2 of `c`.

Matrix size

To determine the size of a matrix array:

Command	Description
`s = size(A)`	For an `m` × `n` matrix `A`, returns the two-element row vector `s = [m, n]` containing the number of rows and columns in the matrix.
`[r,c] = size(A)`	Returns two scalars `r` and `c` containing the number of rows and columns in `A`, respectively.
`r = size(A,1)`	Returns the number of rows in `A` in the variable `r`.
`c = size(A,2)`	Returns the number of columns in `A` in the variable `c`.
`n=length(A)`	Returns the larger of the number of rows or columns in `A` in the variable `n`.
`whos`	Lists variables in the current workspace, together with information about their size, bytes, class, etc. (Long form of `who`.

Examples:

```
>> A = [1 2 3; 4 5 6]
A =
     1     2     3
     4     5     6
>> s = size(A)
s =
     2     3
>> [r,c] = size(A)
r =
     2
c =
     3
>> r = size(A,1)
r =
     2
>> c = size(A,2)
c =
     3
>> length(A)
ans =
     3
>> whos
  Name      Size          Bytes  Class

  A         2x3              48  double array
  ans       1x1               8  double array
  c         1x1               8  double array
  r         1x1               8  double array
  s         1x2              16  double array

Grand total is 11 elements using 88 bytes
```

Matrices containing zeros and ones

Two special functions in MATLAB can be used to generate new matrices containing all zeros or all ones.

`zeros(n)`	Returns an $n \times n$ array of zeros.
`zeros(m,n)`	Returns an $m \times n$ array of zeros.
`zeros(size(A))`	Returns an array the same size as `A` containing all zeros.
`ones(n)`	Returns an $n \times n$ array of ones.
`ones(m,n)`	Returns an $m \times n$ array of ones.
`ones(size(A)`	Returns an array the same size as `A` containing all ones.

Examples of the use of `zeros`:

```
>> A = zeros(3)
A =
     0     0     0
     0     0     0
     0     0     0
>> B = zeros(3,2)
B =
     0     0
     0     0
     0     0
>> C =[1 2 3; 4 2 5]
C =
     1     2     3
     4     2     5
 >> D = zeros(size(C))
 D =
     0     0     0
     0     0     0
```

Practical # 24
Motion Under Gravity

An object is thrown vertically upward with an initial speed v, under acceleration of gravity g. Neglecting air resistance, determine and plot the height of the object as a function of time, from time zero when the object is thrown until it returns to the ground.

The height of the object at time t is

$$h(t) = vt - \frac{1}{2}gt^2$$

The object returns to the ground at time t_g, when

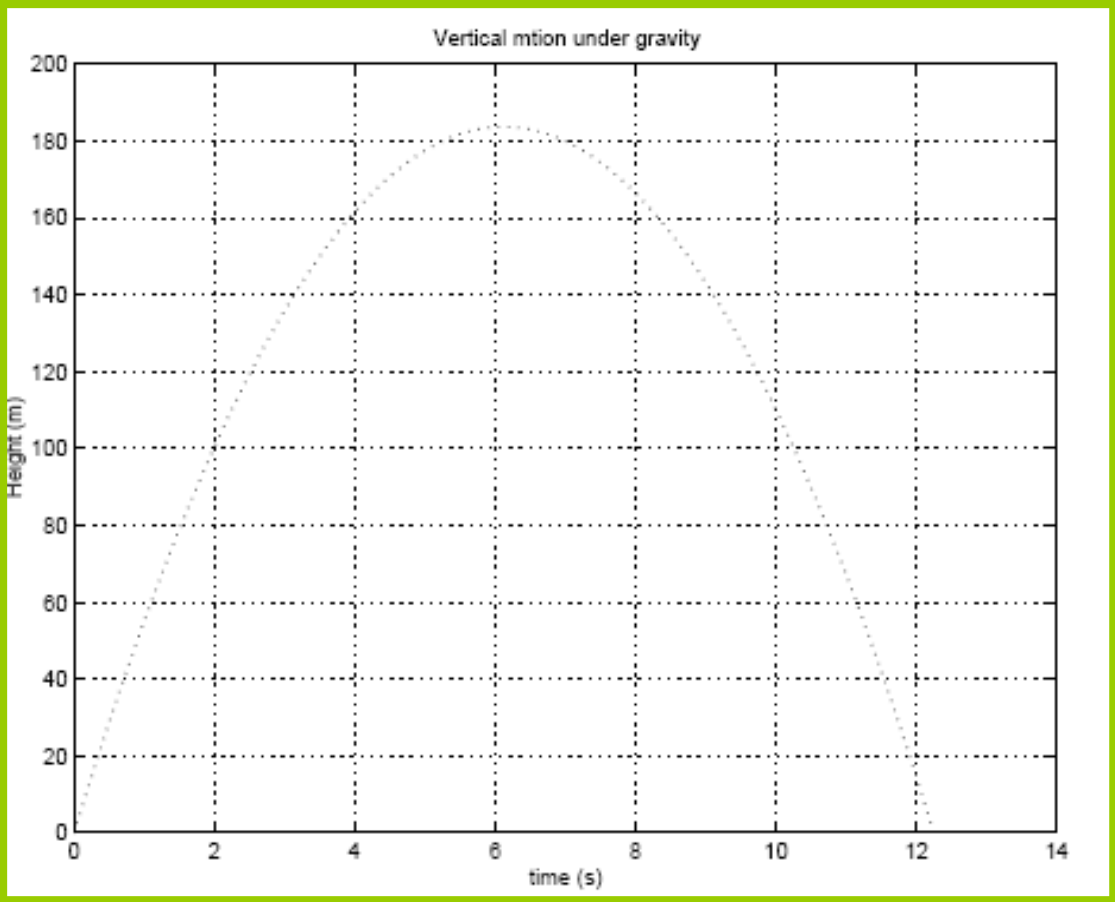
$$h(t_g) = vt_g - \frac{1}{2}gt_g^2 = 0$$

or

$$t_g = \frac{2v}{g}$$

If the initial velocity v is 60 m/s, the following script will compute and plot the vertical height as a function of the time of flight.

```
% Vertical motion under gravity

% Define input values
g = 9.8;                          % acceleration of gravity (m/s^2)
v = 60;                           % initial speed (m/s)

% Calculate times
t_g = 2*v/g;                      % time (s) to return to ground
t = linspace(0,t_g,256);          % 256 element time vector (s)

% Calculate values for h(t)
h = v * t - g/2 * t.^2;           % height (m)

% Plot h(t)
plot(t, h, ':'), title('Vertical mtion under gravity'), ...
     xlabel('time (s)'), ylabel('Height (m)'), grid
```


Vertical mtion under gravity
Height (m)
time (s)
0
20
40
60
80
100
120
140
160
180
200
0
2
4
6
8
10
12
14

Practical # 25
Signals and Plots

Sinusoidal Signal

A sinusoidal signal is a **periodic signal**, which satisfies the condition

$$x(t) = x(t + nT), \qquad n = 1, 2, 3, \ldots$$

where T is a positive constant known as the **period**. The smallest non-zero value of T for which this condition holds is the **fundamental period** T_0. Thus, the amplitude of the signal repeats every T_0.
Consider the signal

$$x(t) = A\cos(\omega t + \theta)$$

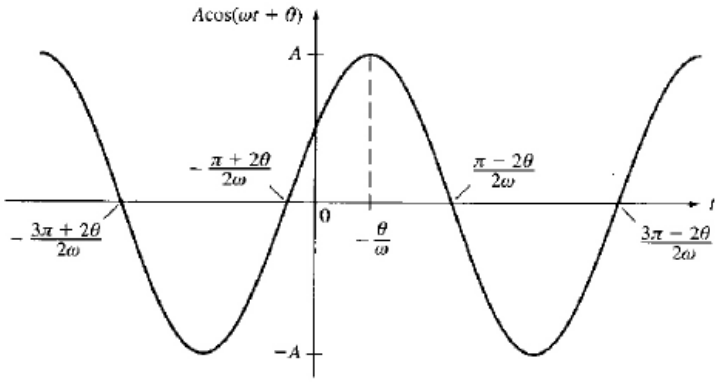
with **signal parameters**:

- A is the amplitude, which characterizes the peak-to-peak swing of $2A$, with units of the physical quantity being represented, such as volts.
- t is the independent time variable, with units of seconds (s).
- ω is the angular frequency, with units of radians per second (rad/s), which defines the fundamental period $T_0 = 2\pi/\omega$ between successive positive or negative peaks.
- θ is the initial phase angle with respect to the time origin, with units of radians, which defines the time shift $\tau = -\theta/\omega$ when the signal reaches its first peak.

With τ so defined, the signal $x(t)$ may also be written as

$$x(t) = A\cos\omega(t - \tau)$$

When τ is positive, it is a "time delay," that describes the time (greater than zero) when the first peak is reached. When τ is negative, it is a "time advance" that describes the time (less than zero) when the last peak was achieved. This sinusoidal signal is shown in Figure 6.1.



Consider computing and plotting the following cosine signal

$$x(t) = 2\cos 2\pi t$$

Identifying the parameters:

- Amplitude $A = 2$
- Frequency $\omega = 2\pi$. The fundamental period $T_0 = 2\pi/\omega = 2\pi/2\pi = 1s$.
- Phase $\theta = 0$.

A time-shifted version of this signal, having the same amplitude and frequency is

$$y(t) = 2\cos[2\pi(t - 0.125)]$$

where the time shift $\tau = 0.125s$ and the corresponding phase is $\theta = 2\pi(-0.125) = -\pi/4$.

A third signal is the related sine signal having the same amplitude and frequency

$$z(t) = 2\sin 2\pi t$$

Preparing a script to compute and plot $x(t)$, $y(t)$, and $z(t)$ over two periods, from $t = -1s$ to $t = 1s$:

```
% sinusoidal representation and plotting
%
t = linspace(-1,1,101);
x = 2*cos(2*pi*t);
y = 2*cos(2*pi*(t-0.125));
z = 2*sin(2*pi*t);
plot(t,x,t,y,t,z),...
  axis([-1,1,-3,3]),...
  title('Sinusoidal Signals'),...
  ylabel('Amplitude'),...
  xlabel('Time (s)'),...
  text(-0.13,1.75,'x'),...
  text(-0.07,1.25,'y'),...
  text(0.01,0.80,'z'),grid
```

Thus, `linspace` has been used to compute the time row vector `t` having 101 samples or elements with values from -1 to 1. The three signals are computed as row vectors and plotted, with axis control, labels and annotation. The resulting plot is shown in Figure 6.2. Observe that $x(t)$ reaches its peak value of 2 at time $t = 0$, which we can verify as being correct since we know $\cos 0 = 1$. The signal $y(t)$ is $x(t)$ delayed by $\tau = 0.125s$. The signal $z(t)$ is 0 for $t = 0$, since $\sin 0 = 0$. It reaches its first peak at $t = T_0/4 = 0.25s$, as $\sin(2\pi \cdot 0.25) = \sin(\pi/2) = 1$.

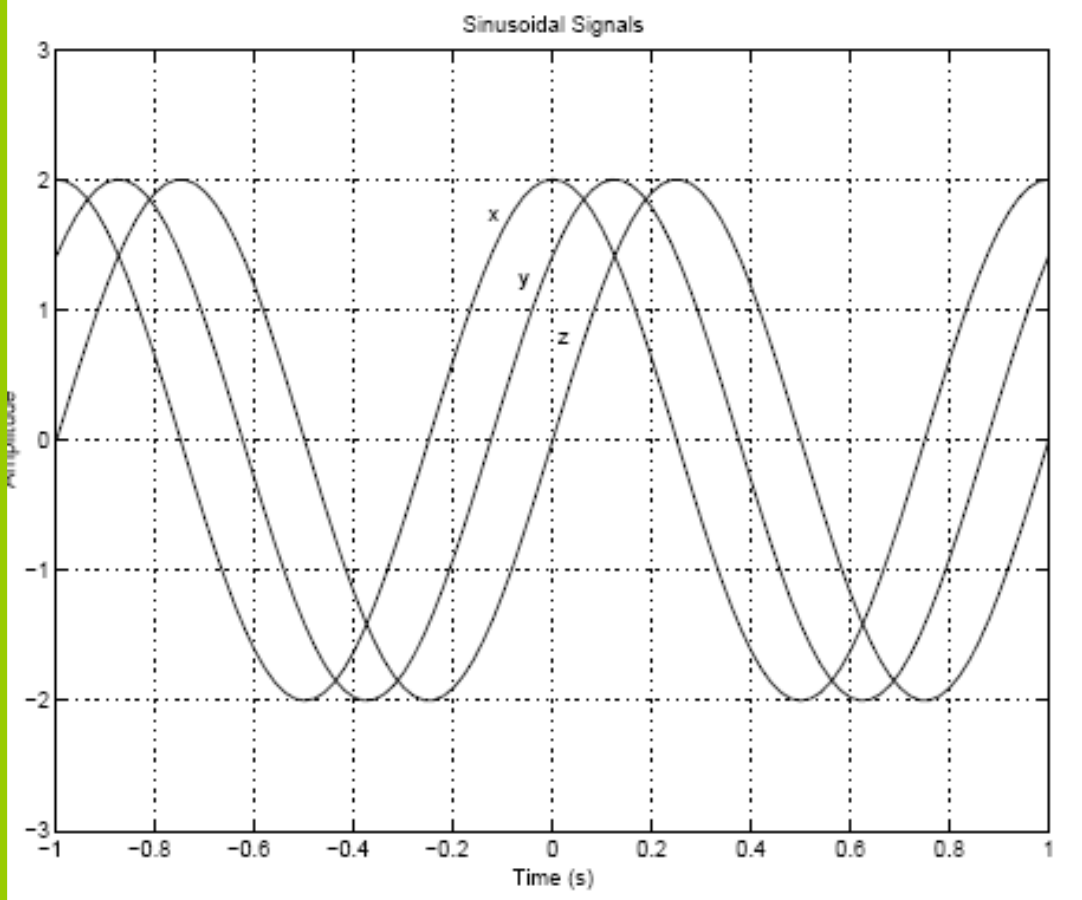


Figure 6.2: Sinusoidal signals

Practical # 26
Well Depth Problem

Finding the depth of a well using roots of a polynomial

Consider the problem of finding the depth of a well by dropping a stone and measuring the time t to hear a splash. This time is composed of the time t_1 of free fall from release to reaching the water and the time t_2 that the sound takes to travel from the water surface to the ear of the person dropping the stone. Let g denote the acceleration of gravity, d the well depth (approximately equal to the distance between the hand or the ear of the person and the water surface), and c the speed of sound in air. The depth d, the distance traveled by the stone during time t_1 is

$$d = \frac{g}{2} \cdot t_1^2$$

or

$$t_1 = \sqrt{2d/g}$$

The same distance traveled by the sound during t_2 is

$$d = c \cdot t_2$$

or

$$t_2 = d/c$$

The total time is

$$t = t_1 + t_2 = \sqrt{2d/g} + d/c$$

Squaring the equation above and rearranging the terms

$$d^2 - 2(tc + c^2/g)d + c^2t^2 = 0$$

The depth d is the solution (roots) of the quadratic polynomial equation above. If the measured time t was 2.5s and the speed of sound c in air at atmospheric pressure and 20° Celsius is 343m/s, the following MATLAB script can be used to calculate the depth d.

```
% Well depth problem
%
% Define input values
t = 2.5;          % time to hear splash (s)
g = 9.81;         % acceleration of gravity (m/s^2)
c = 343;          % speed of sound in air (m/s)

% Calculate polynomial coefficients
a(1) = 1;
a(2) = -2*(t*c + c^2/g);
```

```
a(3) = (c*t)^2;

% Find roots corresponding to depth
depth = roots(a)
```

The displayed results from this script:

```
depth =
  1.0e+004 *
    2.5672
    0.0029
```

As is the case in many problems involving roots of a quadratic equation, one of the solutions is not physically reasonable. In this problem, the first root gives an impossibly large well depth, while the second root gives a reasonable depth. Investigating this solution with MATLAB:

```
>> d = depth(2)
d =
   28.6425
>> t1 = sqrt(2*d/g)
t1 =
    2.4165
>> t2 = d/c
t2 =
    0.0835
>> t = t1 + t2
t =
    2.5000
```

Thus, the depth is 28.6m, confirming the time of 2.5s.

Practical # 27
3D Plots

Three-dimensional plots

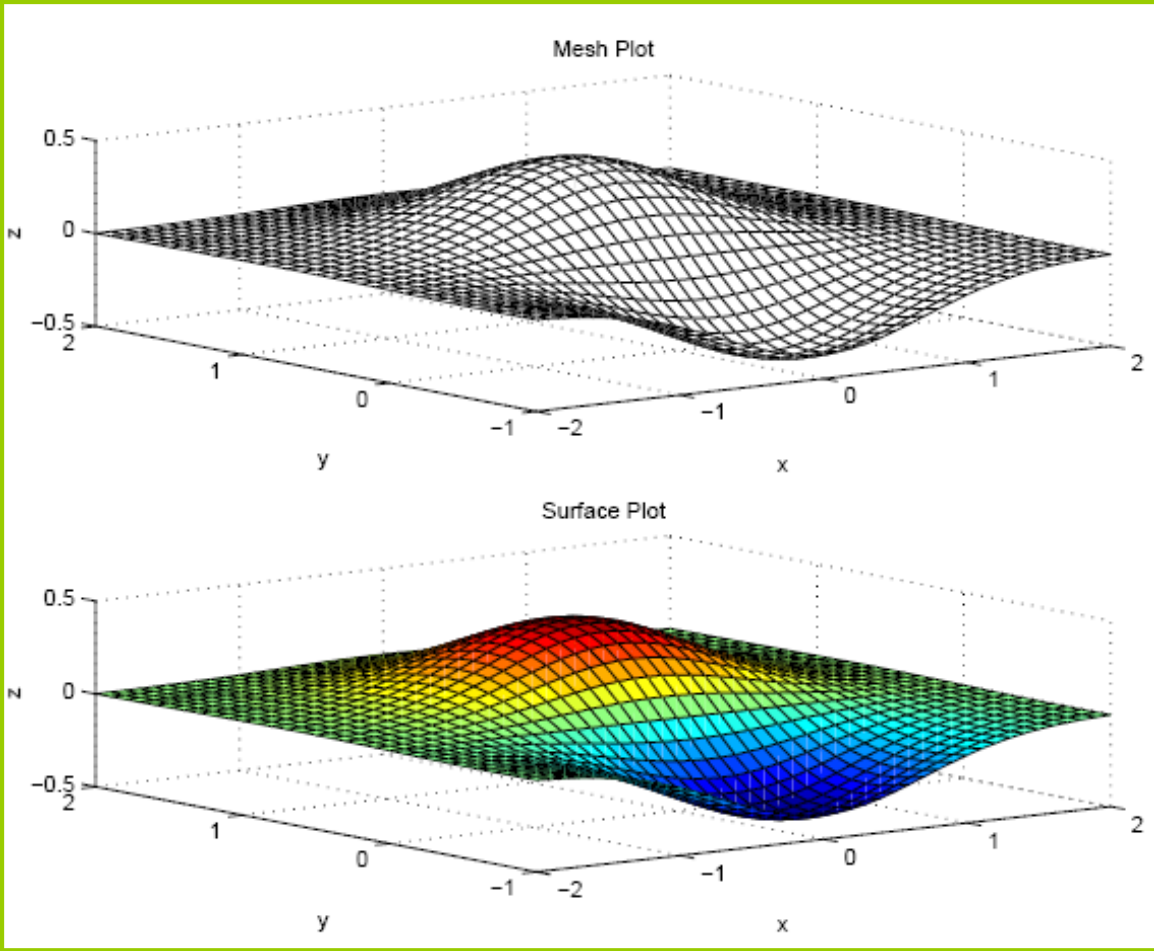
Among several ways to plot a three-dimensional (3D) surface in MATLAB, a **mesh plot** and a **surface plot** will be described, followed by a description of a **contour plot**. For further information on 3D plots, type `help graph3d`.

`mesh(x_pts,y_pts,Z)`	Generates an open mesh plot of the surface defined by matrix `Z`. The arguments `x_pts` and `y_pts` can be vectors defining the ranges of the values of the x- and y-coordinates, or they can be matrices defining the underlying grid of x- and y-coordinates.
`surf(x_pts,y_pts,Z)`	Generates a shaded mesh plot of the surface defined by matrix `Z`. The arguments `x_pts` and `y_pts` can be vectors defining the ranges of the values of the x- and y-coordinates, or they can be matrices defining the underlying grid of x- and y-coordinates.

For example, to plot the function $f(x, y)$ in the example above, the following commands are executed

```
% Mesh and surface plots of a function of two variables
%
x = -2:0.1:2;
y = -1:0.1:2;
[X Y] = meshgrid(x,y);
Z = Y.*exp(-(X.^2 + Y.^2));
subplot(2,1,1),mesh(X,Y,Z),...
  title('Mesh Plot'),xlabel('x'),...
  ylabel('y'),zlabel('z'),...
subplot(2,1,2),surf(X,Y,Z),...
  title('Surface Plot'),xlabel('x'),...
  ylabel('y'),zlabel('z')
```

Note that the xy grid has been made finer by incrementing both x and y by 0.1. Also note that the arguments `X` and `Y` could have been replaced by `x` and `y` in both the `mesh` and `surf` commands. The resulting plots are shown in Figure 6.11. Note that you need to know something about the


Mesh Plot
0.5
0
−0.5
z
2
1
0
−1
y
−2
−1
0
1
2
x
Surface Plot
0.5
0
−0.5
z
2
1
0
−1
y
−2
−1
0
1
2
x

Practical # 28(A)
Minimizing a Function of One Variable

Minimizing a Function of One Variable

To find the minimum of a function of a single variable:

`x = fminbnd('F',x1,x2)`	Returns a value of `x` that is the local minimizer of `F(x)` in the interval [`x1`, `x2`], where `F` is a string containing the name of the function.
`[x,fval] = fminbnd(...)`	Also returns the value of the objective function, `fval`, computed in `F`, at `x`.

For example:

```
>> fminbnd('cos',0,4)
ans =
    3.1416
```

To use this command to find the minimum of more complicated functions, it is convenient to define the function in a function M-file. For example, consider the polynomial function

$$y = 0.025x^5 - 0.0625x^4 - 0.333x^3 + x^2$$

Defining the function file `fp5.m`:

```
function y = fp5(x)
% FP5, fifth degree polynomial function
y = 0.025*x.^5 - 0.0625*x.^4 - 0.333*x.^3 + x.^2
```

Observe in Figure 6.5 that this function has two minima in the interval $-1 \leq x \leq 4$. The minimum near $x = 3$ is called a *relative* or *local* minimum because it forms a valley whose lowest point is higher than the minimum at $x = 0$. The minimum at $x = 0$ is the true minimum and is also called the *global* minimum. Searching for the minimum over the interval $-1 \leq x \leq 4$:

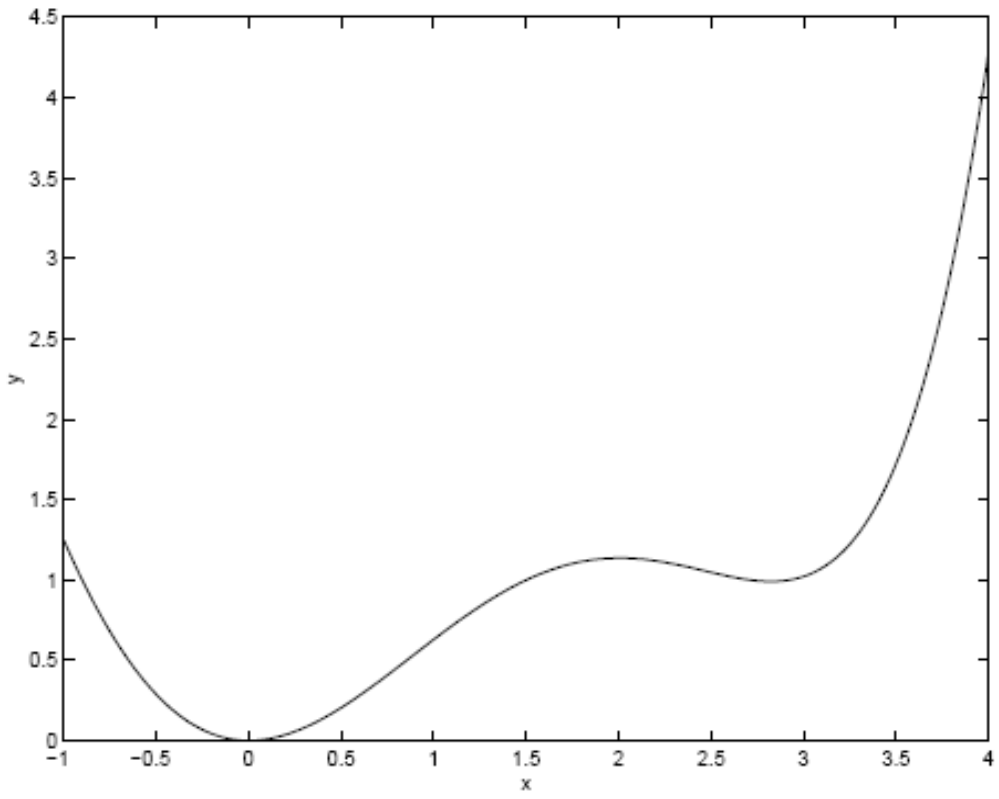
```
>> xmin = fminbnd('fp5',-1,4)
xmin =
  2.0438e-006
```

The resulting value for `xmin` is essentially 0, the true minimum point. Searching for the minimum over the interval $0.1 \leq x \leq 2.5$:

```
>> xmin = fminbnd('fp5',0.1,2.5)
xmin =
    0.1001
```

This misses the true minimum point, as it is not included in the specified interval. Also, `fminbnd` can give incorrect answers. If the interval is $1 \leq x \leq 4$:

```
>> xmin = fminbnd('fp5',1,4)
xmin =
    2.8236
```

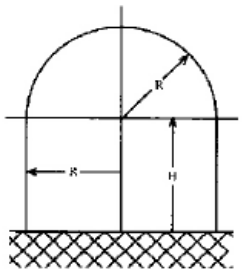


Plot of the function $y = 0.025x^5 - 0.0625x^4 - 0.333x^3 + x^2$

Practical # 28(B)
Tank Design Problem

Minimum cost tank design

A cylindrical tank with a hemispherical top, as shown in Figure 7.2, is to be constructed that will hold 5.00×10^5L when filled. Determine the tank radius R and height H to minimize the tank cost if the cylindrical portion costs $\$300/m^2$ of surface area and the hemispherical portion costs $\$400/m^2$.



Tank configuration

Mathematical model:

Cylinder volume: $V_c = \pi R^2 H$
Hemisphere volume: $V_h = \frac{2}{3}\pi R^3$
Cylinder surface area: $A_c = 2\pi RH$
Hemisphere surface area: $A_h = 2\pi R^2$

Assumptions:

Tank contains no dead air space.
Concrete slab with hermetic seal is provided for the base.
Cost of the base does not change appreciably with tank dimensions.

Computational method:

Express total volume in meters cubed (note: $1m^3 = 1000L$) as a function of height and radius

$$V_{tank} = V_c + V_h$$

For $V_{tank} = 5 \times 10^5 L = 500m^3$:

$$500 = \pi R^2 H + \frac{2}{3}\pi R^3$$

Solving for H:

$$H = \frac{500}{\pi R^2} - \frac{2R}{3}$$

Express cost in dollars as a function of height and radius

$$C = 300A_c + 400A_h = 300(2\pi RH) + 400(2\pi R^2)$$

Method: compute H and then C for a range of values of R, then find the minimum value of C and the corresponding values of R and H.

To determine the range of R to investigate, make an approximation by assuming that $H = R$. Then from the tank volume:

$$V_{tank} = 500 = \pi R^3 + \frac{2}{3}\pi R^3 = \frac{5}{3}\pi R^3$$

Solving for R:

$$R = \left(\frac{300}{\pi}\right)^{\frac{1}{3}}$$

From MATLAB:

```
>> Rest = (300/pi)^(1/3)
Rest =
    4.5708
```

We will investigate R in the range 3.0 to 7.0 meters.

Computational implementation:

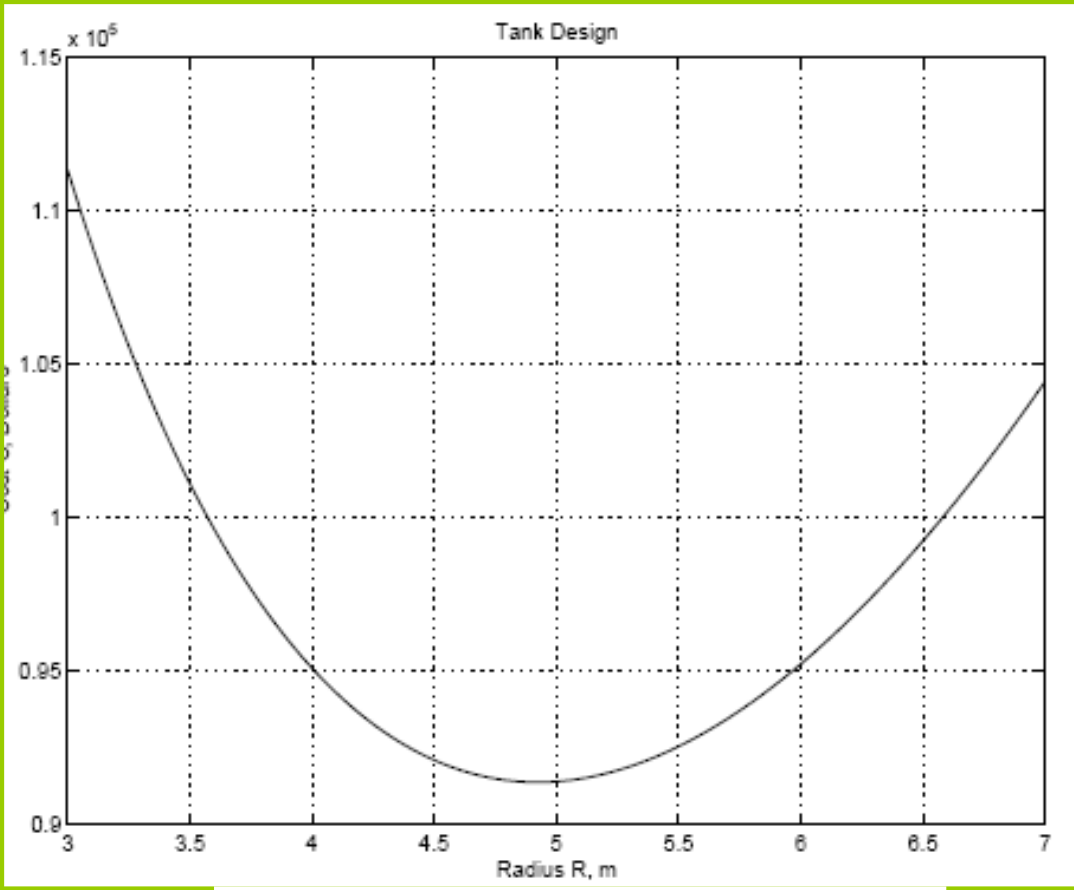
MATLAB script to determine the minimum cost design:

```
% Tank design problem
%
% Compute H & C as functions of R
R = 3:0.001:7.0;                        % Generate trial radius values R
H = 500./(pi*R.^2) - 2*R/3;             % Height H
C = 300*2*pi*R.*H + 400*2*pi*R.^2;      % Cost

% Plot cost vs radius
plot(R,C),title('Tank Design'),...
  xlabel('Radius R, m'),...
  ylabel('Cost C, Dollars'),grid

% Compute and display minimum cost, corresponding H & R
[Cmin kmin] = min(C);
disp('Minimum cost (dollars):')
disp(Cmin)
disp('Radius R for minimum cost (m):')
disp(R(kmin))
disp('Height H for minimum cost (m):')
disp(H(kmin))
```

Running the script:

```
Minimum cost (dollars):
  9.1394e+004
Radius R for minimum cost (m):
    4.9240
Height H for minimum cost (m):
    3.2816
```

Note that the radius corresponding to minimum cost (`Rmin = 4.9240` is close to the approximate value, `Rest = 4.5708` that we computed to assist in the selection of a range of R to investigate.



Tank design problem: cost versus radius

Practical # 29
Factorial

Factorial

The **factorial** of n is expressed and defined as

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 1$$

This can be computed as `prod(1:n)`, as in the following examples:

```
>> nfact = prod(1:4)
nfact =
    24
>> nfact = prod(1:70)
nfact =
  1.1979e+100
```

Practical # 30
Coin Flipping

Random Number Generation

Many engineering problems require the use of **random numbers** in the development of a solution. In some cases, the random numbers are used to develop a simulation of a complex problem. The simulation can be tested over and over to analyze the results, with each test representing a repetition of the experiment. Random numbers also used to represent noise sequences, such as those heard on a radio.

Uniform Random Numbers

Random numbers are characterized by their frequency distributions. **Uniform random numbers** have a constant or uniform distribution over their range between minimum and maximum values.

The `rand` function in MATLAB generates uniform random numbers distributed over the interval [0,1]. A *state vector* is used to generate a random sequence of values. A given state vector always generates the same random sequence. Thus, the sequences are known more formally as **pseudorandom sequences**. This generator can generate all the floating point numbers in the closed interval $[2^{-53}, 1-2^{-53}]$. Theoretically, it can generate over 2^{1492} values before repeating itself.

Command	Description
`rand(n)`	Returns an `n`×`n` matrix `x` of random numbers distributed uniformly in the interval [0,1].
`rand(m,n)`	Returns an `m`×`n` matrix `x` of random numbers distributed uniformly in the interval [0,1].
`rand`	Returns a scalar whose value changes each time it is referenced. `rand(size(A))` is the same size as A.
`s = rand('state')`	Returns a 35-element vector `s` containing the current state of the uniform generator.
`rand('state',s)`	Resets the state to `s`.
`rand('state',0)`	Resets the generator to its initial state.
`rand('state',J)`	Resets the generator to its J-th state for integer `J`.
`rand('state',sum(100*clock))`	Resets the generator to a different state each time.

The following commands generate and display two sets of ten random numbers uniformly distributed between 0 and 1; the difference between the two sets is caused by the different states.

```
rand('state',0)
```

```
set1 = rand(10,1);
rand('state',123)
set2 = rand(10,1);
[set1 set2]
```

The results displayed by these commands are the following, where the first column gives values of `set1` and the second column gives values of `set2`:

```
    0.9501    0.0697
    0.2311    0.2332
    0.6068    0.7374
    0.4860    0.7585
    0.8913    0.6368
    0.7621    0.6129
    0.4565    0.3081
    0.0185    0.2856
    0.8214    0.0781
    0.4447    0.9532
```

Note that as desired, the two sequences are different.

To generate sequences having values in the interval $[x_{min}, x_{max}]$, first generate a random number r in the interval $[0, 1]$, multiply by $(x_{max} - x_{min})$, producing a number in the interval $[0, (x_{max} - x_{min}]$, then add x_{min}. To generate a row vector of 500 elements, the command needed is

```
x = (xmax - xmin)*rand(1,500) + xmin
```

The sequence `data1`, plotted in Figure 7.1, was generated with the command:

```
data1 = 2*rand(1,500) + 2;
```

Thus, $x_{max} - x_{min} = 2$ and $x_{min} = 2$, so $x_{max} = 4$ and the range of the data sequence is (2,4),

Flipping a coin

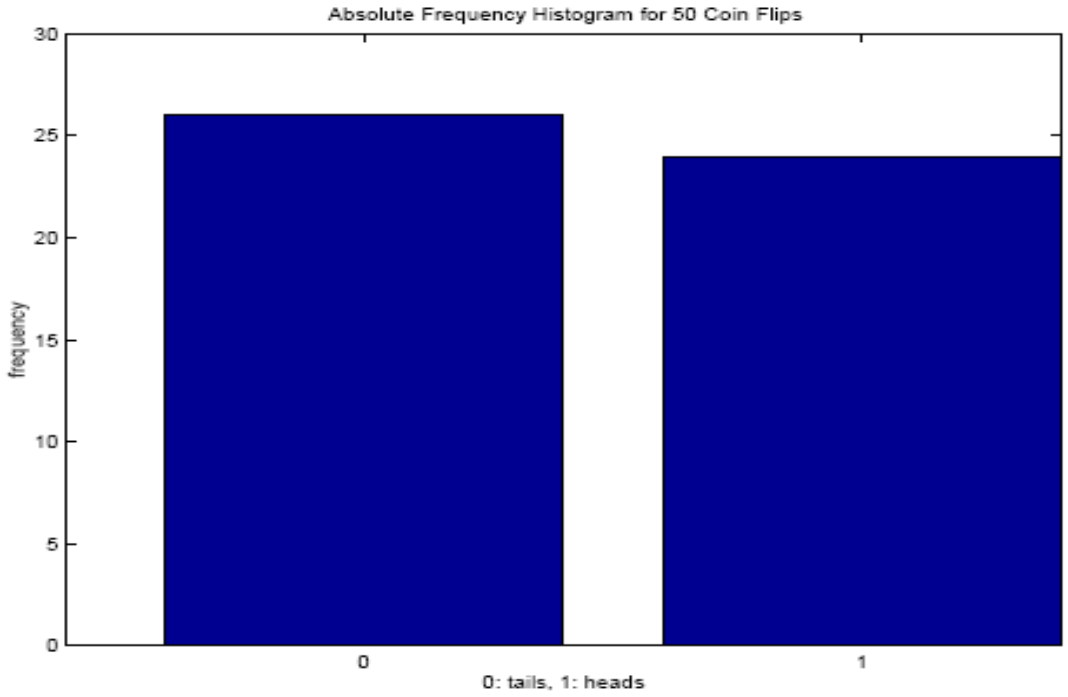
When a fair coin is flipped, the probability of getting heads or tails is 0.5 (50%). If we want to represent tails by 0 and heads by 1, we can first generate a uniform random number in the range [0,2). The probability of a result in the range [0,1) is 0.5 (50%), which can be mapped to 0 by truncation of this result. Similarly, the probability of a result in the range [1,2) is 0.5 (50%), which can be mapped to 1 by truncation. The trunction function in MATLAB is `floor`.

A script to simulate an experiment to flip a coin 50 times and display the results is the following:

```
% Coin flipping simulation
%
% Coin flip:
n = 50;                      % number of flips
coin = floor(2*rand(1,n))    % vector of n flips: 0: tails, 1:heads

% Histogram computation and display:
xc = [0 1];                  % histogram centers
y = hist(coin,xc);           % absolute frequency
bar(xc,y), ylabel('frequency'),...
   xlabel('0: tails, 1: heads'),...
   title('Absolute Frequency Histogram for 50 Coin Flips')
```

The output displayed by the script:

```
coin =
  Columns 1 through 12
     1     1     0     0     1     0     1     1     1     0     1     1
  Columns 13 through 24
     1     0     0     0     0     1     1     0     1     0     1     1
  Columns 25 through 36
     0     1     1     1     0     0     0     0     0     0     1     0
  Columns 37 through 48
     0     1     0     1     0     1     1     0     1     1     0     0
  Columns 49 through 50
     0     0
```



Histogram of 50 Coin Flips

Practical #31
Generating Discontinuous Signals

Generating discontinuous signals

The logical operators allow the generation of arrays representing signals with discontinuities or signals that are composed os segments of other signals. The basic idea is to multiply those values in a array that are to be retained with ones, and multiply all other values with zeros.

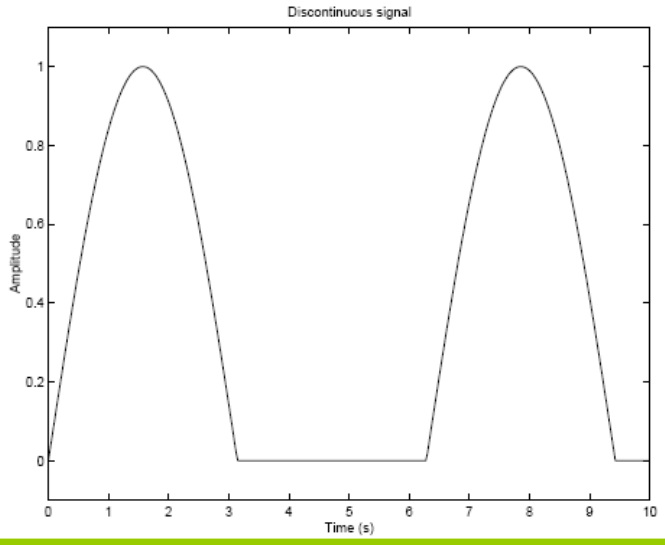
Consider the discontinuous signal:

$$x(t) = \begin{cases} \sin(t) & \sin(t) > 0 \\ 0 & \sin(t) < 0 \end{cases}$$

The following script computes and plots this signal over the range t = [0,10] s.

```
t = linspace(0,10,100);      % create x vector
x = sin(t);                  % compute sine vector
x = x.*(x>0);                % set negative values of sin(t) to zero
%
% Plot x and label
plot(t,x),xlabel('Time (s)'),ylabel('Amplitude'),...
     title('Discontinuous signal'), axis([0 10 -0.1 1.1])
```

The plot that is generated is shown in Figure 8.1.



Practical # 32
Height and Speed of a Projectile

Height and speed of a projectile

The height and speed of a projectile (such as a thrown ball) launched with a speed of v_0 at an angle θ to the horizontal are given by

$$h(t) = v_0 t \sin\theta - 0.5gt^2$$

$$v(t) = \sqrt{v_0^2 - 2v_0 gt\sin\theta + g^2t^2}$$

where g is the acceleration due to gravity. The projectile will strike the ground when $h(t) = 0$, which gives the time to return to the ground, $t_g = 2(v_0/g)\sin\theta$. For $\theta = 40°$, $v_0 = 20$ m/s, and $g = 9.81$ m/s^2, determine the times when the height is no less than 6m and the speed is simultaneously no greater than 16 m/s.

This problem can be solved by using the `find` command to determine the times at which the logical expression `(h >= 6 & (v <= 16)` is ture.

```
% Set the values for initial speed, gravity, and angle
v0 = 20; g = 9.81; theta = 40*pi/180;
% Compute the time to return to the ground
t_g = 2*v0*sin(theta)/g;
% Compute the arrays containing time, height, and speed.
t = [0:t_g/200:t_g];
h = v0*t*sin(theta) - 0.5*g*t.^2;
v = sqrt(v0^2 - 2*v0*g*sin(theta)*t + g^2*t.^2);
% Determine when the height is no less than 6,
% and the speed is no greater than 16.
u = find(h>6 & v <= 16);
% Compute the corresponding times
t_1 = t(u(1))
t_2 = t(u(end))
```

The beginning of the time interval, t_1 is the value of `t` indexed by the first element of `u` and the end of the time interval, t_2 is the value of `t` indexed by the last element of `u`.

The results are:

```
t_1 =
    0.8518
t_2 =
    1.7691
```

This problem could have been solved by plotting $h(t)$ and $v(t)$, but the accuracy of the results would be limited by our ability to pick points off the graph. In addition, the graphical approach is more time-consuming.

Practical # 33
Soda Can Cooling Problem

Update Processes

Many problems in science and engineering involve modelling a process where the main variable is updated over a period of time. In many situations, the updated value is a function of the current value. A comprehensive study of update processes is beyond the scope of this class, but a simple example can be considered.

A can of soda at temperature 25° C is placed in a refrigerator, where the ambient temperature F is 10° C. We want to determine how the temperature of the soda changes over a period of time. A standard way of approaching this type of problem is to subdivide the time interval into a number of small steps, each of duration Δt. If T_i is the temperature at the *beginning* of step i, the following model can be used to determine T_{i+1}:

$$T_{i+1} = T_i + K\Delta t(F - T_i)$$

where K is the *conduction coefficient*, a parameter that depends on the insulating properties of the can and the thermal properties of the soda. Assume that units are chosen so that time is in minutes and that an interval $\Delta t = 1$ minute provides sufficient accuracy. A script to compute, display, and plot this update process for $K = 0.05$:

```
% Define input values
K = 0.05;                           % Conduction coefficient
F = 10;                             % Refrigerator temperature (degrees C)

% Define vector variables
t = 0:100;                          % Time variable (min)
T = zeros(1,101);                   % Preallocate temperature vector
T(1) = 25;                          % Initial soda temperature (degrees C)

% Update to compute T
for i = 1:100;                      % Time in minutes
  T(i+1) = T(i) + K * (F - T(i));   % Compute T
end

% Display results every 10 minutes, plot every minute
disp([ t(1:10:101)' T(1:10:101)' ])
plot(t,T),grid,xlabel('Time (min)'),ylabel('Temperature (degrees C)'),...
  title('Cooling curve')
```

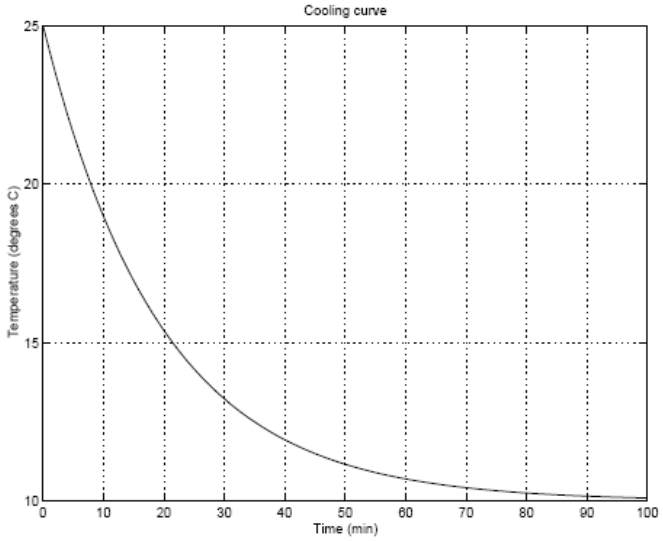
The statement `T = zeros(1,101)` preallocates a vector for the temperature of the soda. The script would work without this statement, but it would be much slower, as `T` would have to be *redimensioned* during each repeat of the `for` loop, in order to make space for a new element each time.

The `for` loop computes the values for `T(2),...,T(101)`, ensuring that temperature `T(i)` corresponds to `t(i)`.

The displayed output of the script is:

```
         0   25.0000
   10.0000   18.9811
   20.0000   15.3773
   30.0000   13.2196
   40.0000   11.9277
   50.0000   11.1542
   60.0000   10.6910
   70.0000   10.4138
   80.0000   10.2477
   90.0000   10.1483
  100.0000   10.0888
```

The resulting graph is shown in Figure 8.2. Note that the initial slope of the temperature is steep and that temperature of the soda changes rapidly. As the difference between the soda temperature and the refrigerator temperature becomes smaller, that change in soda temperature slows.



Practical # 34
Speech Recognition

Consider the design of a system to recognize the spoken words for the ten digits: "zero," "one," "two," ..., "nine." A first step in the design is to analyze data sequences collected with a microphone to see if there are some statistical measurements that would allow recognition of the digits.

The statistical measurements should include the mean, variance, average magnitude, average power and number of zero crossings. If the data sequence is $x(n)$, $n = 1, \ldots, N$, these measurements are defined as follows:

$$\text{mean} = \mu = \frac{1}{N}\sum_{n=1}^{N} x(n)$$

$$\text{standard deviation} = \sigma = \left[\frac{1}{N}\sum_{n=1}^{N}(x(n)-\mu)^2\right]^{\frac{1}{2}}$$

$$\text{average magnitude} = \frac{1}{N}\sum_{n=1}^{N}|x(n)|$$

$$\text{average power} = \frac{1}{N}\sum_{n=1}^{N}(x(n))^2$$

The number of zero crossings is the number of times that $x(k)$ and $x(k+1)$ differ in sign, or the number of times that $x(k)\cdot x(k+1) < 0$.

MATLAB functions for reading and writing Microsoft wave (".wav") sound files include:

Function	Description
`x = wavread(wavefile)`	Reads a wave sound file specified by the string `wavefile`, returning the sampled data in `x`. The ".wav" extension is appended if no extension is given. Amplitude values are in the range [-1,+1].
`[x,fs,bits]=wavread(wavefile)`	Returns the sample rate (`fs`) in Hertz and the number of bits per sample (`bits`) used to encode the data in the file.
`wavwrite(x,wavefile)`	Writes a wave sound file specified by the string `wavfile`. The data should be arranged with one channel per column. Amplitude values outside the range [-1,+1] are clipped prior to writing.
`wavwrite(x,fs,wavefile)`	Specifies the sample rate `fs` of the data in Hertz.

The following is a script ("digit.m") to prompt the user to specify the name of a wave file to be read and then to compute and display the statistics and to plot the sound sequence and a histogram of the sequence. To compute the number of zero crossings, a vector `prod` is generated whose first

element is `x(1)*x(2)`, whose second element is `x(2)*x(3)`, and so on, with the last value equal to the product of the next-to-the-last element and the last element. The `find` function is used to determine the locations of `prod` that are negative, and `length` is used to count the number of these negative products. The histogram has 51 bins between −1.0 and 1.0, with the bin centers computed using `linspace`.

```
% Script to read a wave sound file named "digit.wav"
% and to compute several speech statistics
%
file = input('Enter name of wave file as a string: ');
[x,fs,bits] = wavread(file);
n = length(x);
%
fprintf('\n')
fprintf('Digit Statistics \n\n')
fprintf('samples: %.0f \n',n)
fprintf('sampling frequency: %.1f \n',fs)
fprintf('bits per sample: %.0f \n',bits)
fprintf('mean: %.4f \n',mean(x))
fprintf('standard deviation: %.4f \n', std(x))
fprintf('average magnitude: %.4f \n', mean(abs(x)))
fprintf('average power: %.4f \n', mean(x.^2))
prod = x(1:n-1).*x(2:n);
crossings = length(find(prod<0));
fprintf('zero crossings: %.0f \n', crossings)
subplot(2,1,1),plot(x),...
  axis([1 n -1.0 1.0]),...
  title('Data sequence of spoken digit'),...
  xlabel('Index'), grid,...
subplot(2,1,2),hist(x,linspace(-1,1,51)),...
  axis([-0.6,0.6,0,5000]),...
  title('Histogram of data sequence'),...
  xlabel('Sound amplitude'),...
  ylabel('Number of samples'),grid
```

For a hand example to be used to test the script to be developed for this problem, consider the sequence:

$$[0.25 \quad 0.82 \quad -0.11 \quad -0.02 \quad 0.15]$$

Using a calculator, we compute the following:

$$\text{mean} = \mu = \frac{1}{5}(0.25 + 0.82 - 0.11 - 0.02 + 0.15) = 0.218$$

$$\text{standard deviation} = \sigma \quad = \quad \left[\frac{1}{5}\left((0.25-\mu)^2 + (0.82-\mu)^2 + (-0.11-\mu)^2\right.\right.$$

$$+ (-0.02-\mu)^2 + (0.15-\mu)^2\Big)\Big]^{\frac{1}{2}} = 0.365$$

$$\text{average magnitude} = \frac{1}{5}(|0.25| + |0.82| + |-0.11| + |-0.02| + |0.15|) = 0.270$$

$$\text{average power} = \frac{1}{5}\left(0.25^2 + 0.82^2 + (-0.11)^2 + (-0.02)^2 + 0.15^2\right) = 0.154$$

$$\text{number of zero crossings} = 2$$

The MATLAB commands to compute and write a wave file `test.wav` containing the test sequence:

```
>> x = [0.25 0.82 -0.11 -0.02 0.15]
>> wavwrite(x,'test.wav')
```

Executing the script `digit.m` on the test data:

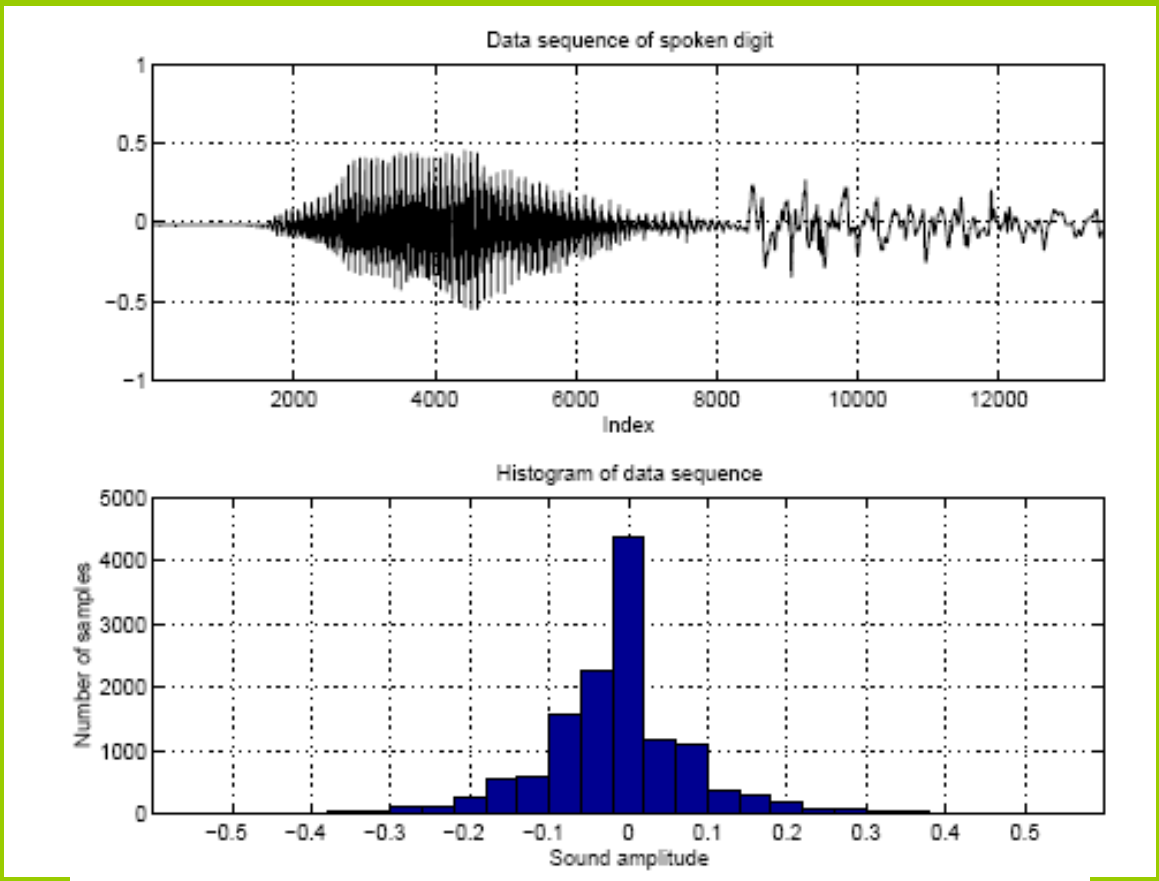
```
>> digit
Enter name of wave file as a string: 'test.wav'

Digit Statistics

samples: 5
sampling frequency: 8000.0
bits per sample: 16
mean: 0.2180
standard deviation: 0.3648
average magnitude: 0.2700
average power: 0.1540
zero crossings: 2
```

Observe that the results from the script agree with the hand-calculated values, giving us some confidence in the script.

Executing the script on a wave file `zero.wav` created using the "Sound Recorder" program on a PC to record the spoken word "zero" as a wave file:

```
>> digit
Enter name of wave file as a string: 'zero.wav'

Digit Statistics

samples: 13493
sampling frequency: 11025.0
bits per sample: 8
mean: -0.0155
standard deviation: 0.1000
average magnitude: 0.0690
average power: 0.0102
zero crossings: 271
```

The resulting plots are shown in Figure



Data sequence and histogram for the spoken word 'zero'

Creating a Sound File Using MatLab

You can also create wave files using MATLAB and then listen to the signals in these files using PC Sound Recorder. For example, to compute and write a wave file containing 16000 samples in 2 seconds of an 800Hz sinusoid:

```
>> t = linspace(0,2,16000);
>> y = cos(2*pi*800*t);
>> wavwrite(y,'cosine.wav')
```

Practical # 35
Ship Sailing Problem

Figure 9.1 shows a ship sailing on bearing 315° (NW) with a speed of 20 knots. The local current due to the tide is 2 knots in the direction 67.5° (ENE). The ship also drifts at 0.5 knots under wind in the direction 180°.

The following script is written to do the following:

- Represent the ship velocity as a vector `S`, the current velocity as a vector `C`, and the wind-drift velocity as a vector `W`. Calculate the true-velocity vector `T` (velocity over the ocean bottom). North represents the first component and East the second component of the vectors.

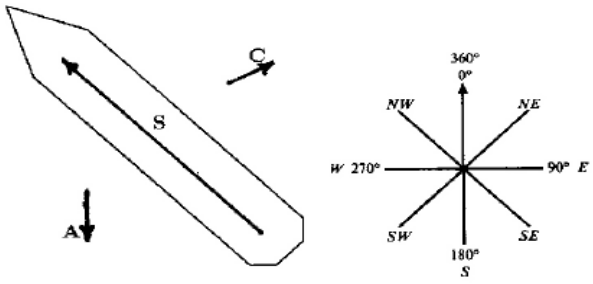


Figure 9.3: Ship course components

- Calculate the true ship speed, that is, the ship speed over the ocean bottom.
- Calculate the true ship course, that is, the direction of sailing over the ocean bottom, measured in degrees clockwise from north.

```
disp('Ship velocity vector:')
S = 20*[cos(315*pi/180) sin(315*pi/180)]    % ship vector
disp('Current velocity vector:')
C = 2*[cos(67.5*pi/180) sin(67.5*pi/180)]   % current vector
disp('Wind drift velocity vector:')
W = 0.5*[-1 0]                              % wind vector
disp('True velocity vector:')
T = S + C + W
disp('Ship speed (knots):')
speed = norm(T)
course = atan2(T(2),T(1))*180/pi;           % ship course, degrees
if course < 0
   course = 360 + course;                   % correct course if negative
end
disp('Ship course (degrees)')
course
```

The function `atan2` was used to compute the course heading, as it provides a four-quadrant result. However, it produces results in the range $-\pi$ to π, which have to be converted to degrees in the range 0 to 360.

The displayed results:

```
Ship velocity vector:
S =
   14.1421  -14.1421
Current velocity vector:
```

```
C =
    0.7654    1.8478
Wind drift velocity vector:
W =
   -0.5000         0
True velocity vector:
T =
   14.4075  -12.2944
Ship speed (knots):
speed =
   18.9401
Ship course (degrees)
course =
  319.5248
```