MATLAB Programming

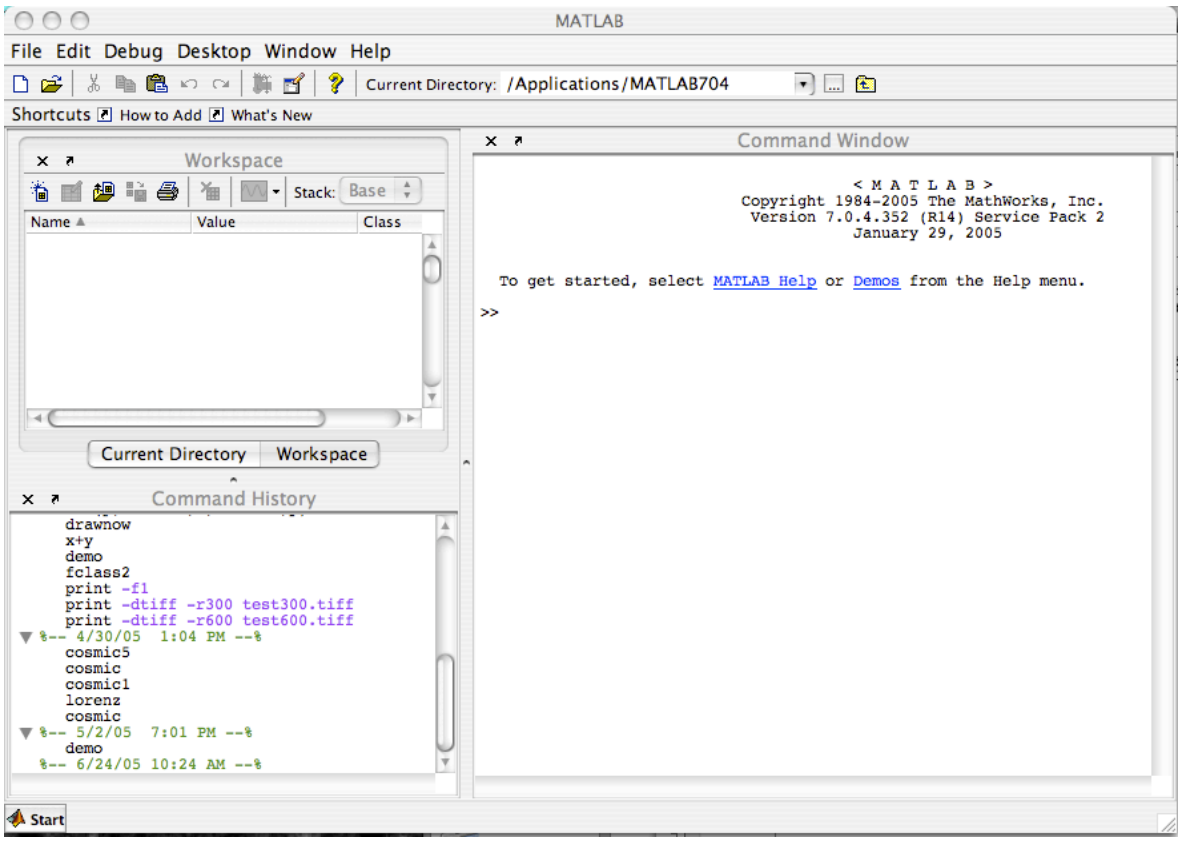
- MATLAB is an interactive system for doing numerical computations. MATLAB makes use of highly respected algorithms and hence you can be confident about your results.

- Powerful operations can be performed using just one or two commands. You can also build your own set of functions. Excellent graphics facilities are included.

0. Getting Started

MATLAB is available for student use on all Physics and Astronomy department laboratory computers and also on the computers in the department common room.

Under Mac OSX we initiate a MATLAB session by clicking on the MATLAB icon on the dock. The window shown below appears after a short time:



This window is the default layout of the MATLAB desktop. It is a set of tools for managing files, variables, and applications associated with MATLAB.

The **Command Window** is used to enter MATLAB commands at the command line prompt >>.

The **Command History Window** is used to view or execute previously run functions.

The **Current Directory/Workspace Window** lists the folders/files in the Current Directory (where you are working) or the values and attributes of the variables you have defined.

The **START button** at the lower left gives you quick access to tools and more.

The **Current Directory** line at the top tells you where MATLAB thinks your files are located. This should always point to the folder that you are working in so that your files are saved in your own directory. An examples would be to enter the pathname/ **Users/Physics50/MATLAB/yourname** or **use the ... button to browse for a folder.** This should always be done at the start of a new session.

When you open a MATLAB document, it opens in the associated tool. If the tool is not already open, it opens when you open the document and appears in the position it occupied when last used. Figures open undocked, regardless of the last position occupied.

How to open a document depends on the document type:

M-file: Select File -> Open and select the M-file. It opens in the Editor/Debugger.

Workspace variable: In the Workspace browser, double-click the variable. It opens in the Array Editor.

At startup, MATLAB automatically executes the master M-file MATLABrc.m and, if it exists, startup.m. The file MATLABrc.m, which is in the local directory, is reserved for use by The MathWorks, and by the system manager on multiuser systems.

The file startup.m is for you to specify startup options. For example, you can modify the default search path, predefine

variables in your workspace, or define Handle Graphics® defaults. Creating a startup.m file with the lines

```
addpath /Users/Physics50/MATLAB

cd /Users/Physics50/MATLAB
```

adds /Users/Physics50/MATLAB to your default search path and makes MATLAB the current directory upon startup.

Location of startup.m. Place the startup.m file in the

```
/Users/Physics50/MATLAB
```

directory, which is where MATLAB will look for it.

The departmental computers are already set up with an appropriate startup.m file.

Using MATLAB as a calculator

The basic arithmetic operators are + - * / ^ and these are used in conjunction with brackets ( ) . The symbol ^ is used to get exponents (powers): 2^4 = 16.

Example:

```
>> 2+3/4*5

ans = 5.7500
```

Note that in this calculation the result was 2+(3/4)*5 and not 2+3/(4*5) because MATLAB works according to the priorities

1. quantities in brackets
2. powers
3. * / working left to right
4. + - working left to right

Numbers and Formats

MATLAB recognizes several different kinds of numbers

Type	Examples
Integer	1362, -217897
Real	1.234, -10.76
Complex	3,21-4.3i (i = √-1)
Inf	Infinity(result of dividing by 0)
NaN	Not a number, 0/0

The "e" notation is used for very large or very small numbers:

$-1.3412e+03 = -1.3412x10^{3} = -1341.2$

$-1.3412e-01 = -1.3412x10^{-1} = -0.13412$

All computations in MATLAB are done in in double precision, which means 15 significant figures. The format - how MATLAB prints numbers - is controlled by the "format" command:

```
>> a=pi                          (pi is a built-in constant)

a = 3.1416
>> format short e

>> a

a = 3.1416e+00
>> format long e

>> a

a = 3.141592653589793e+00

>> format long

>> a

a = 3.14159265358979

>> format short                  {this is the default format}

>> a

a = 3.1416
```

Variable Names

Legal names consist of any combination of letters and digits, starting with a letter.

Allowed: NetCost, Left2Pay, x3, X3, z25c5

Not allowed: Net-Cost, 2pay, %x, @sign

Avoid these names:

eps = 2.2204e-16 = 2^{-54} = largest number such that 1 + eps is indistinguishable from 1

pi = 3.14159... = π

Suppressing Output

One often does not want to see the result of intermediate calculations. This can be accomplished by terminating the MATLAB statement with a semi-colon

```
>> x=-13; y=5+x, z=x^2+y

y = -8

z = 161
```

Note that several statements can be placed on one line, separated by commas or semi-colons.

Built-In Functions

MATLAB has many built-in elementary functions, for example,

sin, cos, tan, asin, acos, atan, atan2

sinh, cosh, tanh, asinh, acosh, atanh,

sqrt, exp, log, log10, abs, sign,

conj, imag, real, angle

round, floor, fix, ceil, rem

abs	absolute value
sqrt	square root

sign	signum
conj	complex conjugate
imag	imaginary part
real	real part
angle	phase angle of a complex number
cos	cosine
sin	sine
tan	tangent
exp	exponential
log	natural logarithm
log10	logarithm base 10
cosh	hyperbolic cosine
sinh	hyperbolic sine
tanh	hyperbolic tangent
acos	inverse cosine
acosh	inverse hyperbolic cosine
asin	inverse sine
asinh	inverse hyperbolic sine
atan	inverse tangent
atan2	two-argument form of inverse tangent - atan2(x,y)
atanh	inverse hyperbolic tangent
round	round to nearest integer
floor	round towards minus infinity
fix	round towards zero
ceil	round towards plus infinity

rem	remainder after division - rem(x,y)

1. Beginning to Use MATLAB

MATLAB works with essentially only one kind of object

a rectangular, numerical array of numbers,
possibly complex, called a matrix

In some situations, 1-by-1 matrices are interpreted as scalars and matrices with only one row or one column are interpreted as vectors.

Matrices can be introduced into MATLAB in several different ways:

- Entered by an explicit list of elements.
- Generated by built-in statements and functions.
- Created in M-files (MATLAB scripts).
- Loaded from external data files.

MATLAB contains no size or type declarations for variables. MATLAB allocates storage automatically, up to available memory.

Vectors

Vectors come in two flavors - row vectors and column vectors. In either case they are lists of numbers separated by either commas or spaces. The number of entries is known as the "length" of the vector and the entries are called "elements" or "components" of the vector. The entries must be enclosed by square brackets.

```
>> v = [1 3, sqrt(5)]

v = 1.0000    3.0000    2.2361

>> length(v)

ans = 3

>> v2 = [3+ 4 5]

v2 = 7     5

>> v3 = [3 +4 5]          {spaces can be very important}

v3 = 3     4     5
```

```
>> v+v3 {arithmetic can be done with vectors of the same length}

ans = 4.0000    7.0000    7.2361

>> v4 = 3*v

v4 = 3.0000    9.0000    6.7082

>> v5 = 2*v -3*v3

v5 = 7.0000   -6.0000  -10.5279

>> v+v2
??? Error using ==> plus
Matrix dimensions must agree.
```

In all vector arithmetic with vectors of equal length, the operations are carried out element-wise.

```
>> w = [1 2 3], z = [8 9] {build vectors from existing vectors}

w = 1     2     3

z = 8     9

>> cd=[2*z,-w], sort(cd)       {sort is a MATLAB command}

cd = 16    18    -1    -2    -3

ans = -3    -2    -1    16    18       {in ascending order}

>> w(2) = -2, w(3)     {change or look at particular entries}

w = 1    -2     3

ans = 3
```

The Colon Operator

This a shortcut for producing row vectors.

```
>> 1:4

ans = 1     2     3     4
```

```
>> 3:7

ans = 3     4     5     6     7

>> 1:-1

ans = Empty matrix: 1-by-0
```

More generally a:b:c produces a vector of entries starting with value a, incrementing by value b until it gets to c (it will not produce a value beyond c). The default increment is 1.

```
>> 0.32:0.1:0.6

ans = 0.3200    0.4200    0.5200

>> -1.4:-0.3:-2

ans = -1.4000   -1.7000   -2.0000
```

Extracting Parts of a Vector

```
>> r5 = [1:2:6,-1:-2:-7]

r5 = 1     3     5    -1    -3    -5    -7

>> r5(3:6)          {get 3rd to 6th entries}

ans = 5    -1    -3    -5

>> r5(1:2:7)        {get alternate entries}

ans = 1     5    -3    -7

>> r5(6:-2:1)

ans = -5    -1     3   {reverse order}
```

Column Vectors

```
>> c = [1; 3; sqrt(5)]
        {use semi-colons or returns instead of commas or spaces}

c = 1.0000
    3.0000
    2.2361
```

```
>> c2 = [3
         4
         5]

c2 = 3
     4
     5

>> c3 = 2*c-3*c2        {arithmetic OK if same length}

c3 = -7.0000
     -6.0000
    -10.5279
```

Transposing

Transposing is the process of converting between row and column vectors. It is denoted by '.

```
>> w = [1 -2 3]

w = 1    -2     3

>> w'

ans = 1
     -2
      3

>> c=[1; 2; 3]

c = 1
    2
    3

>> c'

ans =1     2     3

>> x=[1+3i,2-2i]

x = 1.0000 + 3.0000i   2.0000 - 2.0000i

>> x'               {give complex conjugate}
```

```
ans = 1.0000 - 3.0000i
      2.0000 + 2.0000i

>> x.'              {give transpose for complex vector}

ans = 1.0000 + 3.0000i
      2.0000 - 2.0000i
```

For a general matrix, this operation gives the complex conjugate transpose.

Plotting Elementary Functions

Suppose that we want to plot a graph of $y = \sin 3\pi x$ for $0 \le x \le 1$. We do this by sampling the function at a sufficiently large number of points and joining up the points (x,y) by straight lines. Suppose that we take N+1 points equally spaced a distance h apart:

```
>> N=10; h=1/N; x=0:h:1;
```

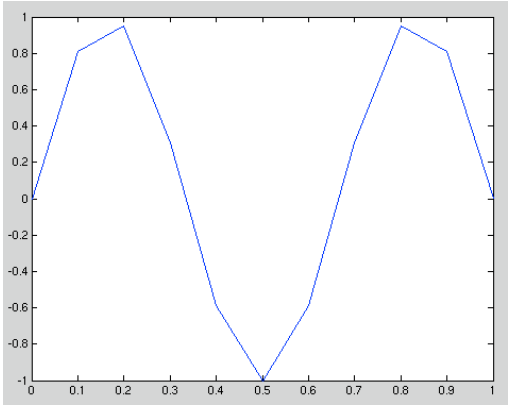
defines the set of points x=0,h,2h,....,1-h,1. The corresponding y-values are computed by

```
>> y=sin(3*pi*x);
```

and finally, we plot the points with

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

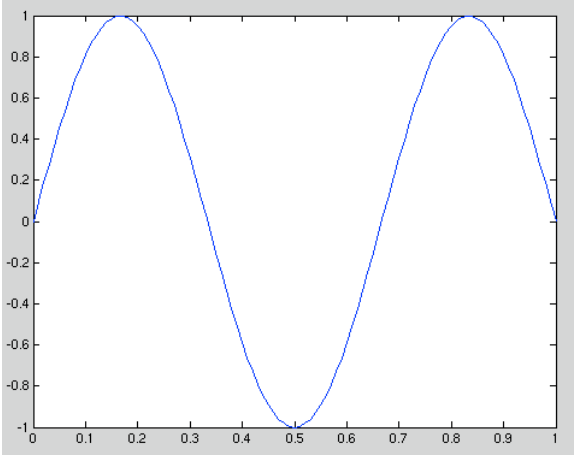
The result is shown in the figure below where it is clear that the value of N chosen above is too small.



On changing the value of N to 100:

```
>> N=100; h=1/N; x=0:h:1;
>> y=sin(3*pi*x);
>> plot(x,y)
```

we get the picture shown below:



To put a **title** and **label the axes**, we use

```
>> title('Graph of y = sin(3*pi*x)')
>> xlabel('x-axis')
>> ylabel('y-axis')
```

The strings enclosed in single quotes can be anything of our choosing.

A dotted grid may be added by

```
>> grid
```

The default is to plot solid lines. A solid black line is produced by

```
>> plot(x,y,'k-')
```

The third argument is a string whose first character specifies the color (optional) and the second character is the line style.

The options for colors and styles are:

Colors	Line Styles
y yellow	. point
m magenta	o circle
c cyan	x x-mark
r red	+ plus
g green	- solid
b blue	* star
w white	: dotted
k black	-. dashdot
	-- dashed

There are more plot symbols available. Use **help plot** or **help shapes** to find out more.

Multi-plots or several graphs on the same figure can be drawn using

```
>> plot(x,y,'r-',x,cos(3*pi*x),'g--')
```

A descriptive legend may be included with

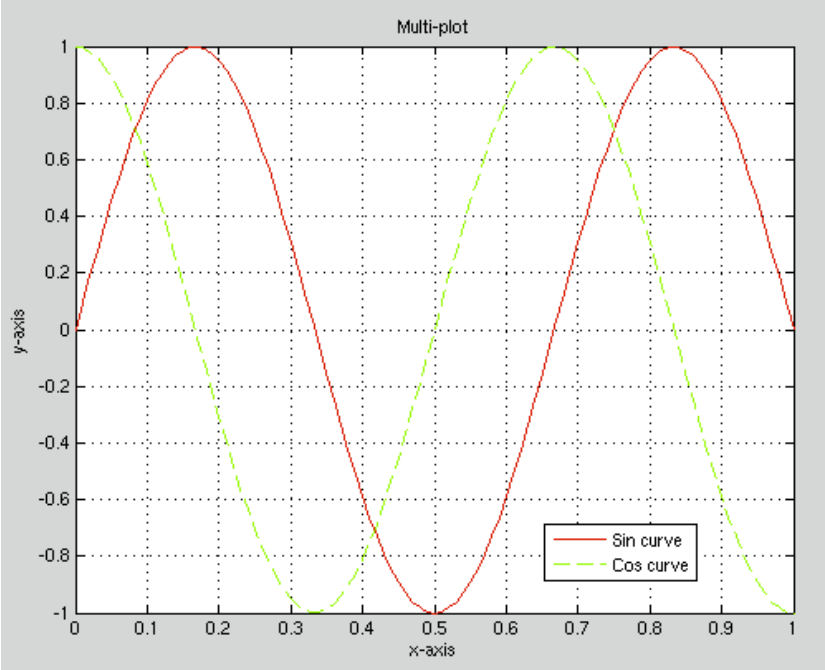
```
>> legend('Sin curve','Cos curve')
```

which will give a list of line-styles, as they appeared in the plot command followed by a brief description. MATLAB places the legend on the graph. If the position is not convenient it can be moved around with the mouse.

If we add

```
>> title('Multi-plot')
>> xlabel('x-axis')
>> ylabel('y-axis')
>> grid
```

the final plot looks like



A call to plot clears the graphics window before plotting the current graph. This is not convenient if we want to add further graphics to the figure at some later stage. To stop the window being cleared:

```
>> plot(x,y,'r-'); hold    %Current plot held
>> plot(x,0.5*y,'gx'); hold off
```

"**hold on**" holds the current picture; "**hold off**" releases it (but does not clear the window, which can be done with the command **clf**). "**hold**" on its own toggles the hold state.

To obtain a printed copy of the figure use the command:

```
>> print -f1                          {f1 = figure 1}
```

Alternatively, you can save the figure to a file for later printing (using another application) or editing. For example the following commands save the window as TIFF files with resolutions 300 DPI and 600 DPI respectively.

```
>> print -dtiff -r300 test300.tiff
>> print -dtiff -r600 test600.tiff
```

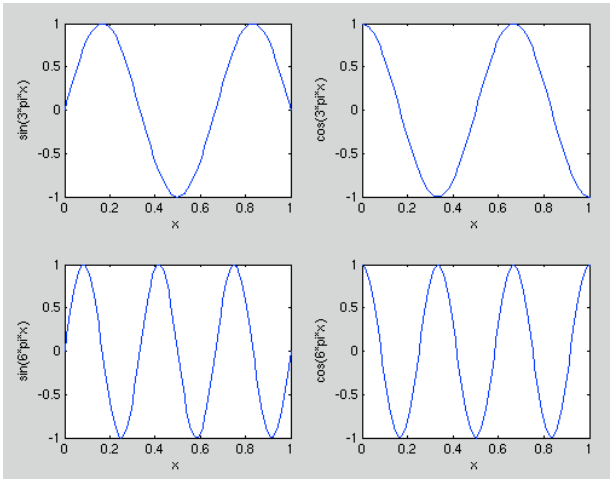
Other output file-type options are:

```
    -dps       % PostScript for black and white printers
    -dpsc      % PostScript for color printers
    -dps2      % Level 2 PostScript for black and white printers
    -dpsc2     % Level 2 PostScript for color printers
    -deps      % Encapsulated PostScript
    -depsc     % Encapsulated Color PostScript
    -deps2     % Encapsulated Level 2 PostScript
    -depsc2    % Encapsulated Level 2 Color PostScript
    -djpeg<nn> % JPEG image, quality level of nn (figures only)
                 E.g., -djpeg90 gives a quality level of 90.
                 Quality level defaults to 75 if nn is omitted.
    -dtiff     % TIFF with lossless run-length compression
    -dtiffnocompression % TIFF without compression
    -dpng      % Portable Network Graphic 24-bit truecolor image
```

The graphics window may be split into an m x n array of smaller window into which we may plot one or more graphs. The windows are counted 1 to mn row-wise, starting from the top left. Both **hold** and **grid** work on the current subplot.

```
>> subplot(221), plot(x,y)
>> xlabel('x'),ylabel('sin(3*pi*x)')
>> subplot(222), plot(x,cos(3*pi*x))
>> xlabel('x'),ylabel('cos(3*pi*x)')
>> subplot(223), plot(x,sin(6*pi*x))
>> xlabel('x'),ylabel('sin(6*pi*x)')
>> subplot(224), plot(x,cos(6*pi*x))
>> xlabel('x'),ylabel('cos(6*pi*x)')
```

subplot(221)(or subplot(2,2,1)) specifies that the window should be split into a 2 x 2 array and we select the first subwindow. The final plot looks like:



We often need to "zoom in" on some portion of a plot in order to see more detail. This is easily achieved using the command

```
>> zoom
```

Pointing the mouse to the relevant position on the plot and clicking the mouse will zoom in by a factor of two. Clicking on the mouse while holding the command key down give a menu of options allowing restoration of the original figure or zooming out. **zoom off** turns off the zoom capability.

The command **clf** clears the current figure window. The command **close 1** closes the window labeled "Figure 1". To open a new figure window type **figure** or to get a window labeled "Figure 9", for example, type **figure (9).** If "Figure 9" already exists then this command will bring the window to the foreground and all subsequent plotting commands will be drawn in it.

Once a plot has been created in the graphics window you might want to change the range of x and y values shown on the picture.

```
>> clf, N = 100; h = 1/N; x = 0:h:1;
>> y = sin(3*pi*x); plot(x,y)
>> axis([-0.5 1.5 -1.2 1.2]), grid
```

The **axis** command has four parameters, the first two are the minimum and maximum values of x to use on the axis and the last two are the minimum and maximum values of y to use on the axis. Note the square brackets.

Script Files

Script files are normal ASCII (text) files that contain MATLAB commands. It is essential that such files have names having an extension .m (e.g., scriptname.m) and, for this reason, they are commonly known as m-files.

The commands in this file may then be executed using

```
>> scriptname
```

Note: the command does not include the file name extension .m.

It is only the output from the commands (and not the commands themselves) that are displayed on the screen.

Script files are created with your favorite text editor such as BBEdit or TextWrangler. Type in your commands and then save (to a file with a .m extension.

To see the commands in the command window prior to their execution:

```
>> echo on
```

and **echo off** will turn echoing off.

Any text that follows % on a line is ignored. The main purpose of this facility is to enable comments to be included in the file to describe its purpose.

In MATLAB, there are two types of script files, namely, programs and functions. We will discussion both of these types later.

Products, Division and Powers of Vectors

Scalar Product (*)

We shall describe two ways in which a meaning may be attributed to the product of two vectors. In both cases the vectors concerned must have the same length. The first product is the standard scalar product. Suppose that u and v are two vectors of length n, u being a row vector and v a column vector:

$$u = [u_1, u_2, ..., u_n] \quad , \quad v = \begin{bmatrix} v_1 \\ v_2 \\ . \\ . \\ . \\ v_n \end{bmatrix}$$

The scalar product is defined by multiplying the corresponding elements together and adding the results to give a single number (scalar).

$$u \cdot v = \sum_{i=1}^{n} u_i v_i$$

We can perform this product in MATLAB by

```
>> u=[10, -11, 12], v=[20; -21; -22]
```

```
u = 10   -11    12

v = 20
   -21
   -22

>> prod=u*v    % row times column vector

prod = 167
```

Suppose we also define a row vector w and a column vector z by

```
>> w=[2,1,3], z=[7;6;5]

w = 2     1     3

z = 7
    6
    5
```

and we wish to form the scalar products of u with w and v with z.

```
>> u*w
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

An error results because w is not a column vector. Recall from earlier that transposing (with ') turns column vectors into row vectors and vice versa.

So, to form the scalar product of two row vectors or two column vectors,

```
>> u*w'     % u and w are row vectors

ans = 45

>> u*u'     % u is a row vector

ans = 365

>> v'*z     % v and z are column vectors

ans = -96
```

We shall refer to the Euclidean length of a vector as the norm of a vector; it is denoted by the symbol $\|u\|$ and defined by

$$\|u\| = \sqrt{\sum_{i=1}^{n} |u_i|^2}$$

where n is its dimension. This can be computed in MATLAB in one of two ways:

```
>> [sqrt(u*u'),norm(u)]

ans = 19.1050   19.1050
```

where **norm** is a built{in MATLAB function that accepts a vector as input and delivers a scalar as output.

Dot Product (.*)

The second way of forming the product of two vectors of the same length is known as the Hadamard product. It is not often used in Mathematics but is an invaluable MATLAB feature. It involves vectors of the same type. If u and v are two vectors of the same type (both row vectors or both column vectors), the mathematical definition of this product, which we shall call the dot product, is the vector having the components

$$uv = \left[u_1 v_1, u_2 v_2, ..., u_n v_n\right]$$

The result is a vector of the same length and type as u and v. Thus, we simply multiply the corresponding elements of two vectors.

In MATLAB, the product is computed with the operator .* and, using the vectors w, z defined earlier

```
>> w.*w

ans = 4     1     9

>> w.*z'

ans = 14     6    15
```

Dot Division of Arrays (./)

There is no mathematical definition for the division of one vector by another. However, in MATLAB, the operator ./ is defined to give element by element division|it is therefore only defined for vectors of the same size and type.

```
>> a = 1:5, b=6:10, a./b

a = 1     2     3     4     5

b = 6     7     8     9    10

ans = 0.1667    0.2857    0.3750    0.4444    0.5000

>> a./a

ans = 1     1     1     1     1

>> c = -2:2, a./c

c = -2    -1     0     1     2

Warning: Divide by zero.

ans = -0.5000   -2.0000       Inf    4.0000    2.5000
```

The previous calculation required division by 0 - notice the **Inf**, denoting infinity, in the answer.

```
>> a.*b - 24, ans./c

ans = -18   -10     0    12    26

Warning: Divide by zero.

ans = 9    10   NaN    12    13
```

Here we are warned about 0/0 - giving a NaN (Not a Number).

Dot Power of Arrays (.^)

To square each of the elements of a vector we could, for example, do u.*u. However, a neater way is to use the .^ operator:

```
>> u

u = 10   -11    12

>> u.^2

ans = 100   121   144

>> u.*u

ans = 100   121   144

>> u.^4

ans = 10000       14641       20736

>> v

v = 20
   -21
   -22

>> w

w = 2     1     3

>> v.^2


ans = 400
      441
      484

>> u.*w.^(-2)

ans = 2.5000  -11.0000    1.3333
```

Recall that powers (.^ in this case) are done first, before any other arithmetic operation.

Creating Matrices

For example, either of the statements

```
>> A = [1 2 3; 4 5 6; 7 8 9]
```

and

```
>> A = [1  2  3
        4  5  6
        7  8  9]
```

creates a 3-by-3 matrix and assigns it to a variable A.

In these cases the shape of the matrix is defined either by the semicolons (;) or the RETURN statements.

MATLAB responds to your command by printing:

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

MATLAB always prints out the variable in the executed line (which can be very awkward and time consuming for large arrays) unless you end the line with a semicolan (;). It is good practice to use the semicolan at the end of the line. You can always ask MATLAB to print the variable later by entering a command consisting of the variable name.

```
>> A

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

The elements within a row of a matrix may be separated by commas as well as a blank(as above).

When listing a number in exponential (powers of 10) form (e.g. 2.34e-9), blank spaces must be avoided.

Alternatively, one can create a file containing the array, name it with a suffix ending with a **.m** .

On MacOSX you would use BBEdit or TextWrangler to create a text file as above.

Suppose we created a text file named gena.m and it looks like

```
A=[1 2 3
   4 5 6
```

```
   7 8 10]
```

It looks the same as the command that we typed before!

We generate the array A(within your MATLAB workspace) by entering the command

```
>> gena       (the filename without the  .m  suffix).
```

MATLAB searches your current working directory for the file gena.m and, then reads the file and generates the array A. This is especially useful for very large files, so that errors can easily be corrected.

In either case, MATLAB responds

```
>> gena

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

The built-in functions **rand**, **magic**, and **hilb**, for example, provide an easy way to create matrices with which to experiment. The command **rand(n)** will create an **n x n** matrix with randomly generated entries distributed uniformly between 0 and 1

```
>> rand(5)

ans = 0.2190    0.3835    0.5297    0.4175    0.5269
      0.0470    0.5194    0.6711    0.6868    0.0920
      0.6789    0.8310    0.0077    0.5890    0.6539
      0.6793    0.0346    0.3834    0.9304    0.4160
      0.9347    0.0535    0.0668    0.8462    0.7012
```

while **rand(m,n)** will create an **m x n** array.

```
>> rand(3,4)

ans = 0.9103    0.0475    0.6326    0.3653
      0.7622    0.7361    0.7564    0.2470
      0.2625    0.3282    0.9910    0.9826
```

magic(n) will create an integral **n x n** matrix which is a magic square (rows and columns have common sum)

```
>> magic(4)

ans = 16     2     3    13
       5    11    10     8
       9     7     6    12
       4    14    15     1
```

hilb(n) will create the **n x n** Hilbert matrix (m and n denote, of course, positive integers).

```
>> hilb(4)

ans = 1.0000    0.5000    0.3333    0.2500
      0.5000    0.3333    0.2500    0.2000
      0.3333    0.2500    0.2000    0.1667
      0.2500    0.2000    0.1667    0.1429
```

We note that if no variable is assigned(as above), MATLAB automatically assigns the result to the variable named ans.

If we use, instead, the command(remember the semicolon suppresses screen printouts)

```
>> zz = hilb(4);
```

then entering the variable name causes a printout.

```
>> zz = 1.0000    0.5000    0.3333    0.2500
        0.5000    0.3333    0.2500    0.2000
        0.3333    0.2500    0.2000    0.1667
        0.2500    0.2000    0.1667    0.1429
```

We can get the size (dimensions) of a matrix with the command **size**

```
zz = 0.9218    0.4057    0.4103    0.3529
     0.7382    0.9355    0.8936    0.8132
     0.1763    0.9169    0.0579    0.0099

>> size(zz)

ans = 3     4

>> size(ans)
```

```
ans = 1     2
```

So zz is a 3 x 4 matrix and ans is 1 x 2 matrix (actually a row vector).

Transposing a vector changes it from a row to a column and vice versa. The extension of this idea to matrices is that transposing interchanges rows with the corresponding columns

```
>> zz

zz = 0.9218    0.4057    0.4103    0.3529
     0.7382    0.9355    0.8936    0.8132
     0.1763    0.9169    0.0579    0.0099

>> zz'  % actually complex conjugate transpose

ans = 0.9218    0.7382    0.1763
      0.4057    0.9355    0.9169
      0.4103    0.8936    0.0579
      0.3529    0.8132    0.0099

>> size(zz),size(zz')

ans = 3     4

ans = 4     3
```

Very useful matrix-generating commands are:

```
>> ones(2,3)

ans = 1     1     1
      1     1     1

>> zeros(2,3)

ans = 0     0     0
      0     0     0

>> ones(size(zz))

ans = 1     1     1     1
      1     1     1     1
      1     1     1     1
```

```
>> eye(3)          % identity matrix

ans = 1      0      0
      0      1      0
      0      0      1

>> d=[-3 4 2], D=diag(d)

d = -3     4     2


D = -3     0     0
     0     4     0
     0     0     2

>> aa = rand(3,3), diag(aa)

aa = 0.1389    0.6038    0.0153
     0.2028    0.2722    0.7468
     0.1987    0.1988    0.4451


ans = 0.1389
      0.2722
      0.4451
```

We can build larger matrices from smaller ones:

```
>> c=[0 1;3 -2;4 2], x=[8;-4;1]

c =  0     1
     3    -2
     4     2


x =  8
    -4
     1

>> g=[c,x]

g =  0     1     8
     3    -2    -4
     4     2     1
```

```
>> a=[5 7 9;1 -3 -7]

a =  5     7     9
     1    -3    -7

>> b=[-1 2 5;9 0 5]

b = -1     2     5
     9     0     5

>> h = [a;b]

h = 5     7     9
    1    -3    -7
   -1     2     5
    9     0     5
```

so we have added an extra column (x) to c in order to form g and have stacked a and b on top of each other to form h.

```
>> j=[1:4;5:8;20 0 5 4]

j =
     1     2     3     4
     5     6     7     8
    20     0     5     4
```

Now for a real tour de force.....

```
>> j=[1:4;5:8;9:12;20 0 5 4]

j =  1     2     3     4
     5     6     7     8
     9    10    11    12
    20     0     5     4

>> k=[diag(1:4) j;j' zeros(4,4)]

k =  1     0     0     0     1     2     3     4
     0     2     0     0     5     6     7     8
     0     0     3     0     9    10    11    12
     0     0     0     4    20     0     5     4
     1     5     9    20     0     0     0     0
     2     6    10     0     0     0     0     0
     3     7    11     5     0     0     0     0
     4     8    12     4     0     0     0     0
```

We note that the command **spy(k)** will produce a graphical display of the location of the nonzero entries in k (nz = number of nonzero entries is also generated).

We can tabulate functions (produce table formats) as follows:

```
>> x=0:0.1:0.5;y=4*sin(3*x); u=3*sin(4*x);
>> [x' y' u']

ans =    0         0         0
    0.1000    1.1821    1.1683
    0.2000    2.2586    2.1521
    0.3000    3.1333    2.7961
    0.4000    3.7282    2.9987
    0.5000    3.9900    2.7279
```

Note the use of the transpose to get column vectors. Even more directly we have

```
>> x=(0:0.1:0.5)'; [x 4*sin(3*x) 3*sin(4*x)]

ans =    0         0         0
    0.1000    1.1821    1.1683
    0.2000    2.2586    2.1521
    0.3000    3.1333    2.7961
    0.4000    3.7282    2.9987
    0.5000    3.9900    2.7279
```

Individual matrix and vector entries can be referenced or extracted with indices inside parentheses. The notation is

```
arrayname(row-number,column-number)
```

For example, A(2,3) denotes the entry in the second row, third column of matrix A and x(3) denotes the third coordinate of vector x. A matrix or a vector will only accept positive integers as indices.

```
>> A

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

```
>> A(2,3)

ans = 6

>> x=[-1.3 sqrt(3) (1+2+3)*4/5]
```

(note here how MATLAB does not care how you construct the elements)

```
x = -1.3000    1.7321    4.8000

>> x(2)

ans = 1.7321
```

The colon operator introduced earlier is very powerful in MATLAB. (Always remember powerful = dangerous)

```
>> 1:6

ans = 1   2    3    4    5    6
```

(generates a vector of regularly spaced (default=1) elements)

```
>> 1:.1:2

ans = 1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9  2.0
```

(generates a vector of regularly spaced (spacing=0.1) elements)

```
>> A(1,1:2)

ans = 1     2
```

or row 1 and columns 1 & 2

```
>> A(1:2,2)

ans = 2
      5
```

or rows 1 & 2 and column 2.

```
>> j=[1:4;5:8;9:12;20 0 5 4]
```

```
j =  1     2     3     4
     5     6     7     8
     9    10    11    12
    20     0     5     4

>> j(4,1)=j(1,1)+6

j =  1     2     3     4
     5     6     7     8
     9    10    11    12
     7     0     5     4

>> j(1,1)=j(1,1)-3*j(1,2)

j = -5     2     3     4
     5     6     7     8
     9    10    11    12
     7     0     5     4
```

In the following examples we extract (1) the 3rd column, (2) the 2nd and 3rd columns, (3) the 4th row, and (4) the "central" 2x2 matrix.

```
>> j(:,3)

ans = 3
      7
     11
      5

>> j(:,2:3)

ans = 2     3
      6     7
     10    11
      0     5

>> j(4,:)

ans = 7     0     5     4

>> j(2:3,2:3)

ans = 6     7
     10    11
```

Thus, : on its own refers to the entire column or row depending on whether it is the first or last index.

Dot product of matrices (.*)

The dot product works as for vectors: corresponding elements are multiplied together - so the matrices involved must have the same size.

```
>> a=[5 7 9;1 -3 -7],b=[-1 2 5;9 0 5]

a =  5     7     9
     1    -3    -7


b = -1     2     5
     9     0     5

>> a.*b

ans = -5    14    45
       9     0   -35

>> c=[0 1;3 -2;4 2]

c =  0     1
     3    -2
     4     2

>> a.*c
??? Error using ==> times
Matrix dimensions must agree.

>> a.*c'

ans = 0    21    36
      1     6   -14
```

Matrix - vector products

We turn next to the definition of the product of a matrix with a vector. This product is only defined for column vectors that have the same number of entries as the matrix has columns. So, if A is an m x n matrix and x is a column vector of length n, then the matrix-vector product Ax is legal.

An m x n matrix times an n x 1 matrix ⇒ a m x 1 matrix.

We visualize A as being made up of m row vectors stacked on top of each other, then the product corresponds to taking the scalar product of each row of A with the vector x: The result is a column vector with m entries.

$$Ax = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} A_{11}x_1 + A_{12}x_2 + A_{13}x_3 \\ A_{21}x_1 + A_{22}x_2 + A_{23}x_3 \end{pmatrix}$$

$$or \quad (Ax)_i = \sum_{k=1}^{3} A_{ik}x_k$$

or for

```
>> x=[8;-4;1]

x =  8
    -4
     1
```

$$\begin{pmatrix} 5 & 7 & 9 \\ 1 & -3 & -7 \end{pmatrix} \begin{pmatrix} 8 \\ -4 \\ 1 \end{pmatrix} = \begin{pmatrix} 40 - 28 + 9 \\ 8 + 12 - 7 \end{pmatrix} = \begin{pmatrix} 21 \\ 13 \end{pmatrix}$$

It is somewhat easier in MATLAB:

```
>> a*x

ans = 21
      13
```

Unlike multiplication in arithmetic, A*x is not the same as x*A.

Matrix-Matrix Products

To form the product of an m x n matrix A and a n x p matrix B, written as AB, we visualize the first matrix (A) as being composed of m row vectors of length n stacked on top of each other while the second (B) is visualized as being made up of p column vectors of length n.

The entry in the i^{th} row and j^{th} column of the product is then the scalar-product of the i^{th} row of A with the j^{th} column of B. The product is an m x p matrix:

Check that you understand what is meant by this definition by thinking about the following examples.

```
>> a=[5 7 9;1 -3 -7],b=[0 1;3 -2;4 2]

a =  5     7     9
     1    -3    -7

b =  0     1
     3    -2
     4     2

>> c=a*b

c = 57     9
   -37    -7

>> d=b*a

d =  1    -3    -7
    13    27    41
    22    22    22

>> e=b'*a'

e = 57   -37
     9    -7
```

We see that e = c' suggesting that (a*b)' = b'*a' Why is b*a a 3 x 3 matrix while a*b is 2 x 2?

Systems of Linear Equations

Mathematical formulations of engineering problems often lead to sets of simultaneous linear equations. A general system of linear equations can be expressed in terms of a coefficient matrix A, a right-hand-side (column) vector b and an unknown (column) vector x as

$$Ax = b$$

or component-wise, as

$$A_{11}x_1 + A_{12}x_2 + \ldots\ldots + A_{1n}x_n = b_1$$
$$A_{21}x_1 + A_{22}x_2 + \ldots\ldots + A_{2n}x_n = b_2$$
.....
.....
$$A_{n1}x_1 + A_{n2}x_2 + \ldots\ldots + A_{nn}x_n = b_n$$

When A is non-singular(has an inverse) and square (n x n), meaning that the number of independent equations is equal to the number of unknowns, the system has a unique solution given by

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

where A^{-1} is the inverse of A. Thus, the solution vector x can, in principle, be calculated by taking the inverse of the coefficient matrix A and multiplying it on the right with the right-hand-side vector b.

This approach based on the matrix inverse, though formally correct, is at best inefficient for practical applications (where the number of equations may be extremely large) but may also give rise to large numerical errors unless appropriate techniques are used.

Various stable and efficient solution techniques have been developed for solving linear equations and the most appropriate in any situation will depend on the properties of the coefficient matrix A. For instance, on whether or not it is symmetric, or positive definite or if it has a particular structure. MATLAB is equipped with many of these special techniques in its routine library and they are invoked automatically.

The standard MATLAB routine for solving systems of linear equations is invoked by calling the matrix left-division routine,

```
>> x = A\b
```

where "\" is the matrix left-division operator known as "backslash".

Example: Given

```
>> a = [2 -1 0;1 -2 1;0 -1 2],b=[1;0;1]
```

```
a =  2    -1     0
     1    -2     1
     0    -1     2

b =  1
     0
     1
```

we solve the corresponding equations using two methods (1) $x = A^{-1}b$ and (2) x = A\b.

```
>> x = inv(a)*b

x = 1.0000
    1.0000
    1.0000

>> x = a\b

x = 1.0000
    1.0000
    1.0000
```

Example: Given

```
>> a = [2+2i -1 0;-1 2-2i -1;0 -1 2],b=[1+i;0;1-i]

a = 2.0000 + 2.0000i  -1.0000                  0
   -1.0000             2.0000 - 2.0000i  -1.0000
         0            -1.0000             2.0000

b = 1.0000 + 1.0000i
         0
    1.0000 - 1.0000i

>> x = a\b

x = 0.6757 - 0.0541i
    0.4595 + 0.2432i
    0.7297 - 0.3784i
```

Characters, Strings and Text

The ability to process text in numerical processing is useful for the input and output of data to the screen or to disk-files. In order to manage text, a new datatype of "character" is

introduced. A piece of text is then simply a string (vector) or array of characters.

The assignment,

```
>> t1='A'

t1 = A
```

assigns the value A to the 1 x 1 character array t1.

The assignment,

```
>> t2='BCDE'

t2 = BCDE
```

assigns the value BCDE to the 1 x 4 character array t2.

Strings can be combined by using the operations for array manipulations.

The assignment,

```
>> t3=[t1,t2]

t3 = ABCDE
```

assigns a value ABCDE to the 1 x 5 character array t3.

The assignment,

```
>> t4=[t3,' are the first 5      ';...
'characters in the alphabet.']

t4 = ABCDE are the first 5
     characters in the alphabet.
```

assigns the value

```
'ABCDE are the first 5      '
'characters in the alphabet.'
```

to the 2 x 27 character array t4. It is essential that the number of characters in both rows of the array t4 is the same, otherwise an error will result.

The **three dots ...** signify that the command is **continued** on the following line.

Sometimes it is necessary to convert a character to the corresponding number, or vice versa. These conversions are accomplished by the commands '**str2num**' - which converts a string to the corresponding number, and two functions, '**int2str**' and '**num2str**', which convert, respectively, an integer and a real number to the corresponding character string. These commands are useful for producing titles and strings, such as 'The value of pi is 3.1416'. This can be generated by the command

['The value of pi is ',num2str(pi)]

```
>> N=5;h=1/N;
>> ['The value of N is ',num2str(N),...
', h = ',num2str(h)]

ans = The value of N is 5, h = 0.2
```

Loops

There are occasions that we want to repeat a segment of code a number of different times (such occasions are less frequent than other programming languages because of the : notation).

Example: Draw graphs of $\sin(n\pi x)$ on the interval $-1 \le x \le 1$ for n = 1,2,....,8.

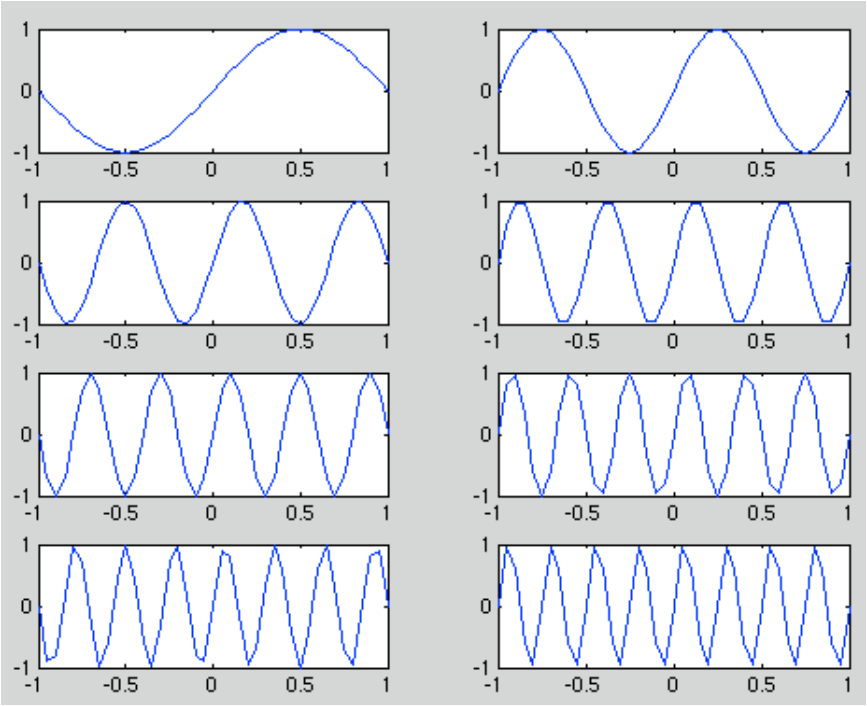
We could do this by giving 8 separate plot commands but it is much easier to use a loop. The simplest form would be

```
>> x=-1:0.05:1;
>> for n=1:8;subplot(4,2,n),plot(x,sin(n*pi*x));end
```

In programs or scripts this is written in form

```
for n=1:8
 subplot(4,2,n),plot(x,sin(n*pi*x))
end
```

The result looks like:



All the commands between the lines starting "**for**" and "**end**" are repeated with n being given the value 1 the first time through, 2 the second time, and so forth, until n = 8. The subplot constructs a 4 x 2 array of subwindows and, on the n^{th} time through the loop, a picture is drawn in the n^{th} subwindow.

We may use any legal variable name as the "loop counter" (n in the above examples) and it can be made to run through all of the values in a given vector (1:8 in the above example).

We may also use for loops of the type

```
>> for counter = [23 11 19 5.4 6]
     .......
   end
```

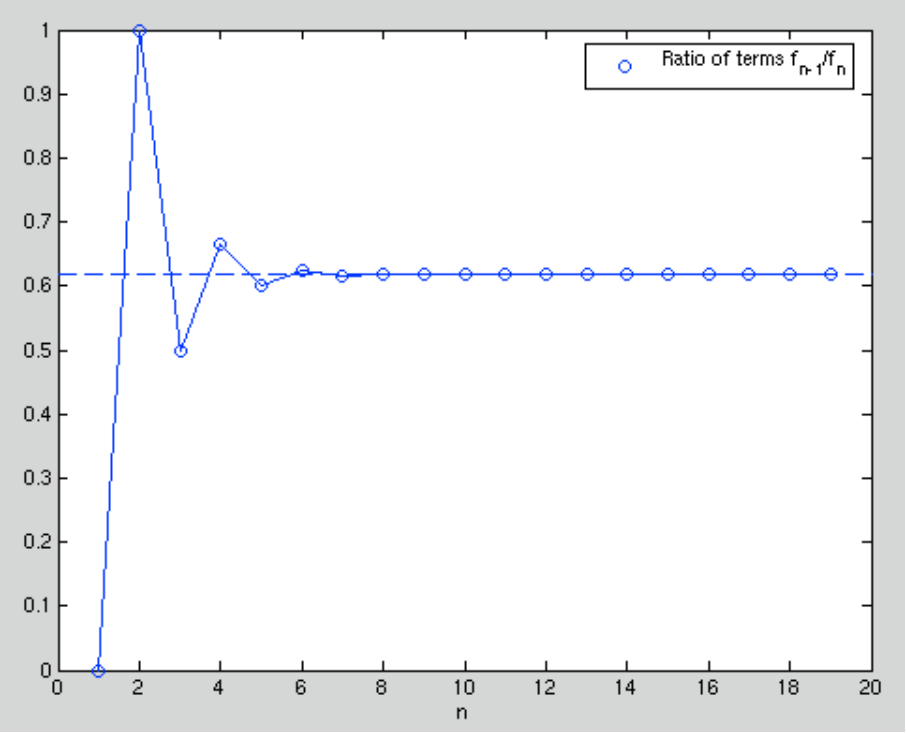
which repeats the code as far as the end with counter=23 the first time, counter=11 the second time, and so forth.
Example: The Fibonnaci sequence starts off with the numbers 0 and 1, then succeeding terms are the sum of its two immediate predecessors. Mathematically, $f_1 = 0, f_2 = 1$ and

$$f_n = f_{n-1} + f_{n-2} \quad , \quad n = 3,4,5,.....$$

Test the assertion that the ratio f_{n-1}/f_n of two successive values approaches the golden ratio $\left(\sqrt{5}-1\right)/2 = 0.6180....$.

```
>> hold off
>> F(1) = 0; F(2) = 1;
>> for i = 3:20;F(i) = F(i-1) + F(i-2);end
>> plot(1:19, F(1:19)./F(2:20),'o' )
>> hold on, xlabel('n')
>> plot(1:19, F(1:19)./F(2:20),'-' )
>> legend('Ratio of terms f_{n-1}/f_n')
>> plot([0 20], (sqrt(5)-1)/2*[1,1],'--')
>> hold off
```

The last of these commands produces the dashed horizontal line.



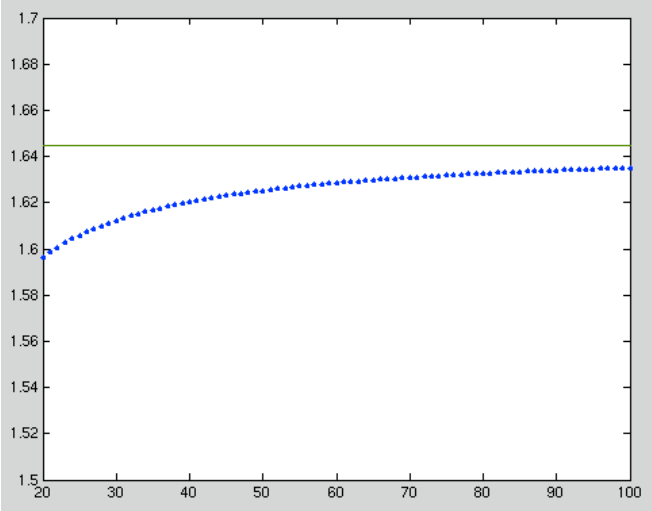
Example: Produce a list of the values of the sums

$$S_{20} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + ... + \frac{1}{20^2}$$
$$S_{21} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + ... + \frac{1}{21^2}$$
.....
$$S_{100} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + ... + \frac{1}{20^2} + \frac{1}{21^2} + ... + \frac{1}{100^2}$$

There are a total of 81 sums. The first can be computed using sum(1./(1:20).^2) (The function **sum** with a vector argument sums its components). A suitable piece of MATLAB code might be

```
>> S=zeros(100,1);
>> S(20)=sum(1./(1:20).^2);
>> for n=21:100;S(n)=S(n-1)+1/n^2;end
>> clf; plot(20:100,S(20:100),'.',[20 100],[1 1]*pi^2/6,'-')
>> axis([20 100 1.5 1.7])
>> [(98:100)' S(98:100)]

ans = 98.0000    1.6348
      99.0000    1.6349
     100.0000    1.6350
```

where a column vector S was created to hold the answers. The first sum was computed directly using the sum command then each succeeding sum was found by adding $1/n^2$ to its predecessor. The little table at the end shows the values of the last three sums - it appears that they are approaching a limit (the value of the limit is $\pi^2/6 = 1.64493.....$). Plot is shown below:



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
element-wise:

>> x=[-2 pi 5;-1 0 1]

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
    -1.0000         0    1.0000

>>  x > 3 & x < 4

ans = 0     1     0
      0     0     0

>> x > 3 | x == 0

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.

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
    -1.0000         0    1.0000

L =  0     1     1
     0     1     1

>> pos = x.*L

pos =  0    3.1416    5.0000
       0         0    1.0000
```

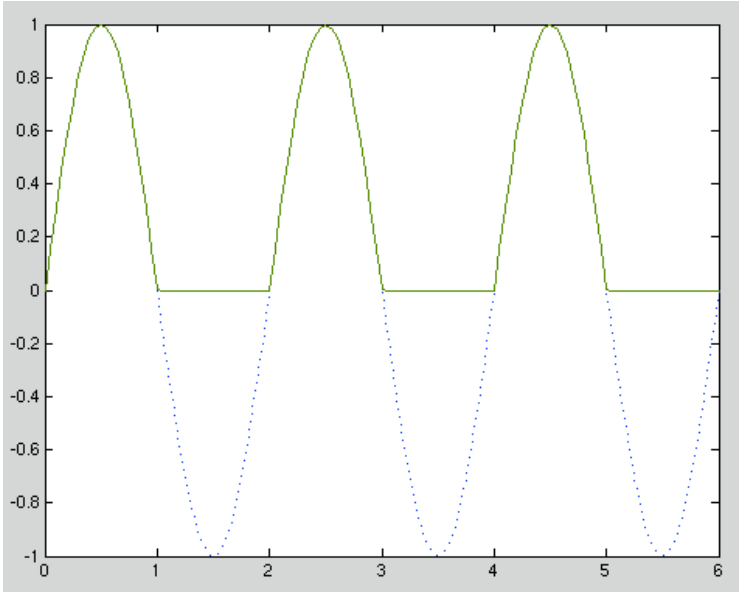
so the matrix pos contains just those elements of x that are non-negative.

Now consider

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

```
>> plot(x,y,':',x,Y,'-')
```

which results in



While Loops

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.

```
>> 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 $S+(n+1)\text{^}2<100$ is true.

Example: Find the approximate root of the equation $x=\cos x$.

We can do this by making a guess $x_1=\pi/4$,say, then computing the sequence of values

$$x_n=\cos x_{n-1} \quad , \quad n=2,3,4,.....$$

and continuing until the difference between two successive values $|x_n-x_{n-1}|$ is small enough.

Method 1:

```
>> 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 6
    0.7854    0.7071    0.7602    0.7247    0.7487   0.7326
    Columns 7 through 12
    0.7435    0.7361   0.7411    0.7377    0.7400    0.7385
    Columns 13 through 18
    0.7395    0.7388         0         0         0         0
    Columns 19 through 20
         0         0
```

There are a number of deficiencies with this program. The vector x stores the results of each iteration but we don't know in advance how many there may be. In any event, we are rarely interested in the intermediate values of x, only the last one. Another problem is that we may never satisfy the condition $d \le 0.001$, in which case the program will run forever - we should place a limit on the maximum number of iterations.

Incorporating these improvements leads to

Method 2:

```
>> xold = pi/4; n = 1; d = 1;
>> while d>0.001&n<20;n=n+1;xnew=cos(xold);d=abs(xnew-
xold);xold=xnew;end
>> [n,xnew,d]

ans = 14.0000    0.7388    0.0007
```

We continue around the loop so long as d > 0.001 and n < 20. For greater precision we could use the condition d > 0.0001, and this gives

```
>> [n,xnew,d]

ans = 19.0000    0.7391    0.0001
```

from which we may judge that the root required is x = 0.739 to 3 decimal places.

The general form of **while** statement is

```
while a logical test
     commands to be executed
     when the condition is true
end
```

if...then...else...end

This allows us to execute different commands depending on the truth or falsity of some logical tests. To test whether or not π^e is greater than, or equal to, e^π:

```
>> a = pi^exp(1); c = exp(pi);
>> if a >= c;b = sqrt(a^2-c^2);end
```

so that b is assigned a value only if a ≥ c. There is no output so we deduce that a = π^e < c = e^π. A more common situation is

```
>> if a >= c;b = sqrt(a^2-c^2);else;b = 0;end

b = 0
```

which ensures that b is always assigned a value and confirming a < c in this case.

A more extended form is

```
>> if a >= c;b = sqrt(a^2 - c^2);elseif a^c > c^a;b = c^a/
a^c;else;b = a^c/c^a;end

b = 0.2347
```

The general form of the if statement is

```
if logical test 1
     commands to be ececuted
     if test 1 is true
elseif logical test 2
     commands to be executed if test 2
     is true but test 1 is false
.........
.........
end
```

Function m-files

These are a combination of the ideas of script m-files discussed earlier and mathematical functions.

Example: The area, A, of a triangle with sides of length a, b and c is given by

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

where $s = (a+b+c)/2$. Write a MATLAB function that will accept the values a, b and c as inputs and return the value of A as output.

The main steps to follow when defining a MATLAB function are:

1. Decide on a name for the function, making sure that it does not conflict with a name that is already used by MATLAB. In this example the name of the function is to be **area**, so its definition will be saved in a file called area.m

2. The first line of the file must have the format:

   function [*list of outputs*] = *function name*(*list of inputs*)

   For our example, the output (A) is a function of the three variables (inputs) a, b and c so the first line should read

   ```
   function [A] = area(a,b,c)
   ```

3. Document the function. That is, describe briefly the purpose of the function and how it can be used. These lines should be preceded by % which signify that they are comment lines that will be ignored when the function is evaluated.

4. Finally include the code that defines the function. This should be interspersed with sufficient comments to enable another user to understand the processes involved.

The complete file might look like:

```
function [A] = area(a,b,c)
% Compute the area of a triangle whose
% sides have length a, b and c.
% Inputs:
% a,b,c: Lengths of sides
% Output:
```

```
% A: area of triangle
% Usage:
% Area = area(2,3,4);
% Written by jrb, June 29, 2005.
s = (a+b+c)/2;
A = sqrt(s*(s-a)*(s-b)*(s-c));
%%%%%%%%% end of area %%%%%%%%%%
```

The command

```
>> help area
```

will produce the leading comments from the file:

```
  Compute the area of a triangle whose
  sides have length a, b and c.
  Inputs:
  a,b,c: Lengths of sides
  Output:
  A: area of triangle
  Usage:
  Area = area(2,3,4);
  Written by jrb, June 29, 2005.

    Reference page in Help browser
       doc area
```

To evaluate the area of a triangle with sides of length 10, 15, 20:

```
>> Area = area(10,15,20)

Area = 72.6184
```

where the result of the computation is assigned to the variable **Area. Functions must assign values to a variable.** The variable s used in the definition of the function above is a "local variable": its value is local to the function and cannot be used outside:

```
>> s
??? Undefined function or variable 's'.
```

If we were to be interested in the value of s as well as A, then the first line of the file should be changed to

```
function [A,s] = area(a,b,c)
```

where there are two output variables.

This function can be called in several different ways:

1. No outputs assigned

```
>> area(10,15,20)

ans = 72.6184
```

gives only the area (first of the output variables from the file) assigned to **ans;** the second output is ignored.

2. One output assigned

```
>> Area = area(10,15,20)

Area = 72.6184
```

again the second output is ignored.

3. Two outputs assigned

```
>> [Area, hlen] = area(10,15,20)

Area = 72.6184

hlen = 22.5000
```

Examples of functions

We revisit the problem of computing the Fibonnaci sequence defined by $f_1 = 0, f_2 = 1$ and

$$f_n = f_{n-1} + f_{n-2} \quad , \quad n = 3,4,5,.....$$

We want to construct a function that will return the nth number in the Fibonnaci sequence f_n.

Input: Integer n
Output: f_n

We shall describe four possible functions and try to assess

which provides the best solution.

Method 1:

```
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);
```

This code resembles that given in an earlier example. We have simply enclosed it in a function m-file and given it the appropriate header.

Method 2:

The first version was rather wasteful of memory - it saved all the entries in the sequence even though we only required the last one for output. The second version removes the need to use a vector.

```
function f = Fib2(n)
% Returns the nth number in the
% Fibonacci sequence.
if n==1
  f = 0;
elseif n==2
  f = 1;
else
  f1 = 0; f2 = 1;
  for i = 2:n-1
    f = f1 + f2;
    f1=f2; f2 = f;
  end
end
```

Method 3:

This version makes use of an idea called "recursive programming" - the function makes calls to itself.

```
function f = Fib3(n)
```

```
% Returns the nth number in the
% Fibonacci sequence.
if n==1
f = 0;
elseif n==2
f = 1;
else
f = Fib3(n-1) + Fib3(n-2);
end
```

Method 4:

The final version uses matrix powers. The vector y has two components,

$$y = \begin{bmatrix} f_n \\ f_{n+1} \end{bmatrix}$$

```
function f = Fib4(n)
% Returns the nth number in the
% Fibonacci sequence.
A = [0 1;1 1];
y = A^n*[1;0];
f=y(1);
```

Assessment:

One may think that, on grounds of style, the 3rd is best (it avoids the use of loops) followed by the second (it avoids the use of a vector). The situation is much different when it cames to speed of execution. When n = 20 the time taken by each of the methods is (in seconds)

Method	Time
1	0.000246
2	0.000205
3	0.531082
4	0.000298

It is impractical to use Method 3 for any value of n much larger than 10 since the time taken by method 3 almost doubles whenever n is increased by just 1. When n = 150

Method	Time
1	0.000316
2	0.000245
3	4.176754 (n=24)
4	0.000313

Clearly the 2nd method is much the fastest.

We will see how to time such operations shortly.

Further Built-in Functions

Rounding Numbers
There are a variety of ways of rounding and chopping real numbers to give integers. Use the definitions given earlier in order to understand the output given below:

```
>> x = pi*(-1:3), round(x)

x = -3.1416         0    3.1416    6.2832    9.4248

ans = -3     0     3     6     9

>> fix(x)

ans = -3     0     3     6     9

>> floor(x)

ans = -4     0     3     6     9

>> ceil(x)

ans = -3     0     4     7    10

>> sign(x), rem(x,3)

ans = -1     0     1     1     1

ans = -0.1416         0    0.1416    0.2832    0.4248
```

The sum Function

The "**sum**" applied to a vector adds up its components (as in sum(1:10)) while, for a matrix, it adds up the components in **each column** and returns a row vector.

sum(sum(A)) then sums all the entries of A.

```
>> A = [1:3;4:6;7:9]

A =  1     2     3
     4     5     6
     7     8     9

>> s = sum(A), ss = sum(sum(A))
s = 12    15    18

ss = 45

>> x = pi/4*(1:3)'

x = 0.7854
    1.5708
    2.3562

>> A = [sin(x), sin(2*x), sin(3*x)]/sqrt(2)

A = 0.5000    0.7071    0.5000
    0.7071    0.0000   -0.7071
    0.5000   -0.7071    0.5000

>> s1 = sum(A.^2), s2 = sum(sum(A.^2))

s1 = 1.0000    1.0000    1.0000

s2 = 3.0000
```

The sums of squares of the entries in each column of A are equal to 1 and the sum of squares of all the entries is equal to 3.

```
>> A*A'

ans = 1.0000         0   -0.0000
           0    1.0000    0.0000
     -0.0000    0.0000    1.0000

>> A'*A
```

```
ans = 1.0000         0   -0.0000
           0    1.0000    0.0000
     -0.0000    0.0000    1.0000
```

It appears that the products AA' and A'A are both equal to the identity:

```
>> A*A'- eye(3)

ans = 1.0e-15 *
   -0.1110         0   -0.0278
         0   -0.2220    0.0622
   -0.0278    0.0622   -0.2220

>> A'*A - eye(3)

ans = 1.0e-15 *
   -0.1110         0   -0.0278
         0   -0.2220    0.0622
   -0.0278    0.0622   -0.2220
```

This is confirmed since the differences are at round-off error levels (less than 10^{-15}). A matrix with this property is called an *orthogonal* matrix.

max & min

These functions act in a similar way to **sum.** If x is a vector, then **max(x)** returns the largest element in x

```
>> x = [1.3 -2.4 0 2.3], max(x), max(abs(x))

x = 1.3000   -2.4000         0    2.3000

ans = 2.3000

ans = 2.4000
```

When we ask for two outputs, the first gives us the maximum entry and the second the index of the maximum element.

```
>> [m, j] =max(x)

m = 2.3000
```

```
j = 4
```

For a matrix, A, **max(A)** returns a row vector containing the maximum element from each column. Thus to find the largest element in A we have to use **max(max(A)).**

Random Numbers

The function **rand(m,n)** produces an m x n matrix of random numbers, each of which is in the range 0 to 1. **rand** on its own produces a single random number.

```
>> y = rand, Y = rand(2,3)

y = 0.9501

Y = 0.2311    0.4860    0.7621
    0.6068    0.8913    0.4565
```

Repeating these commands will lead to different answers.

Example: 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)

Recall that floor takes the largest integer that is smaller than a given real number.

```
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.
%
% Usage: T = dice(3)
d = floor(1 + 6*rand(n,2));
%% end of dice

>> dice(3)
```

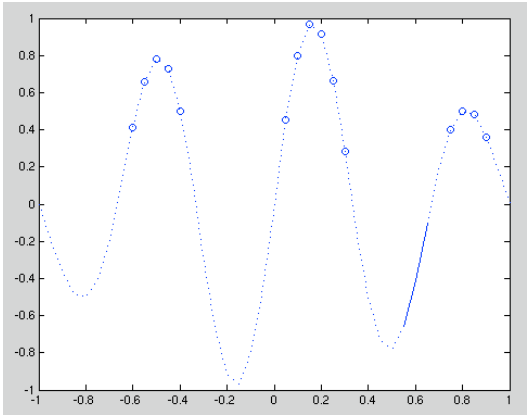
```
ans = 1     4
      5     5
      3     6

>> sum(dice(100))/100

ans = 3.6400    3.2500
```

The last command gives the average value over 100 throws (it should have the value 3.5).

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 8
     9    10    11    12    13    22    23    24
    Columns 9 through 15
    25    26    27    36    37    38    39

>> plot(x,y,':')
>> hold on, plot(x(k),y(k),'o')
>> km = find(x > 0.5 & y < 0)

km = 32    33    34

>> plot(x(km),y(km),'-')
```



find for matrices

The **find**-function operates in much the same way for matrices:

```
>> A =[-2 3 4 5; 0 5 -1 6; 6 8 0 1]

A = -2     3     4     5
     0     5    -1     6
     6     8     0     1

>> k = find(A == 0)

k = 2
    9
```

Thus, we find that A has elements equal to 0 in positions 2 and 9. To interpret this result we have to recognize that "**find**" first reshapes A into a column vector - this is equivalent to numbering the elements of A by columns as in

```
                1     4     7    10
                2     5     8    11
                3     6     9    12

>> n = find(A <= 0)

n =  1
     2
     8
     9

>> A(n)

ans = -2
       0
      -1
       0
```

Thus, n gives a list of the locations of the entries in A that are ≤ 0 and then A(n) gives us the values of the elements selected.

```
>> m = find(A' == 0)

m =  5
    11
```

Since we are dealing with A', the entries are numbered by rows.

Plotting Surfaces

A surface is defined mathematically by a function f(x,y) - corresponding to each value of (x,y) we compute the height of the function by

$$z = f(x,y)$$

In order to plot this we have to decide on the ranges of x and y - suppose $2 \le x \le 4$ and $1 \le y \le 3$. This gives us a square in the (x,y)-plane. Next, we need to choose a grid on this domain, that is, number of points or interval between points in each coordinate interval. For example, suppose we choose a grid with intervals 0.5 in each direction. The x- and y-coordinates of the grid lines are

```
>> x = 2:0.5:4; y = 1:0.5:3;
```

Finally, we have to evaluate the function at each point of the grid and "**plot**" it.

We construct the grid with the **meshgrid** command:

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

X = 2.0000    2.5000    3.0000    3.5000    4.0000
    2.0000    2.5000    3.0000    3.5000    4.0000
    2.0000    2.5000    3.0000    3.5000    4.0000
    2.0000    2.5000    3.0000    3.5000    4.0000
    2.0000    2.5000    3.0000    3.5000    4.0000

Y = 1.0000    1.0000    1.0000    1.0000    1.0000
    1.5000    1.5000    1.5000    1.5000    1.5000
    2.0000    2.0000    2.0000    2.0000    2.0000
    2.5000    2.5000    2.5000    2.5000    2.5000
    3.0000    3.0000    3.0000    3.0000    3.0000
```

If we think of the i^{th} point along from the left and the j^{th} point up from the bottom of the grid) as corresponding to the $(i,j)^{th}$ entry in a matrix, then (X(i,j),Y(i,j)) are the coordinates of the point.

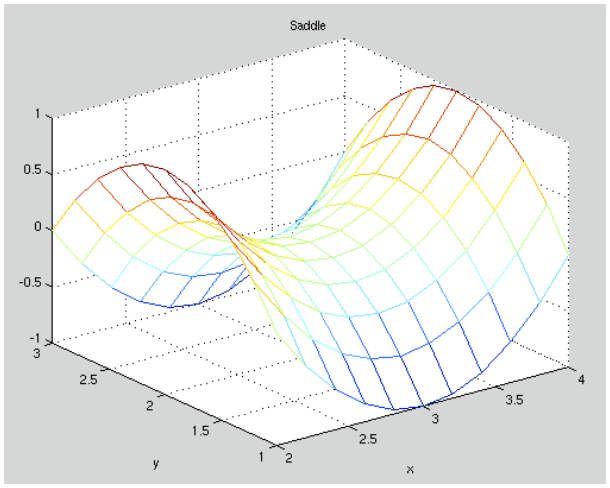
We then need to evaluate the function f using X and Y in place of x and y, respectively.

Example: Plot the surface defined by the function

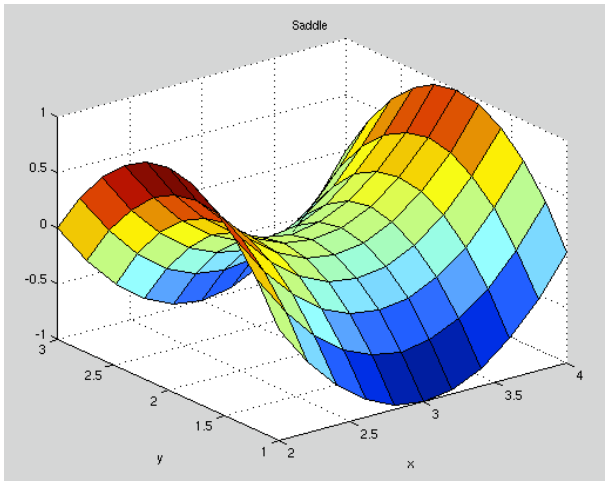
$$f(x,y) = (x-3)^2 - (y-2)^2$$

on the domain $-2 \le x \le 4$ and $1 \le y \le 3$.

```
>> [X,Y] = meshgrid(2:0.2:4,1:0.2:3);
>> Z = (X-3).^2 - (Y-2).^2;
>> mesh(X,Y,Z)
>> title('Saddle'),xlabel('x'),ylabel('y')
```



We now repeat the previous example replacing **mesh** by **surf.**

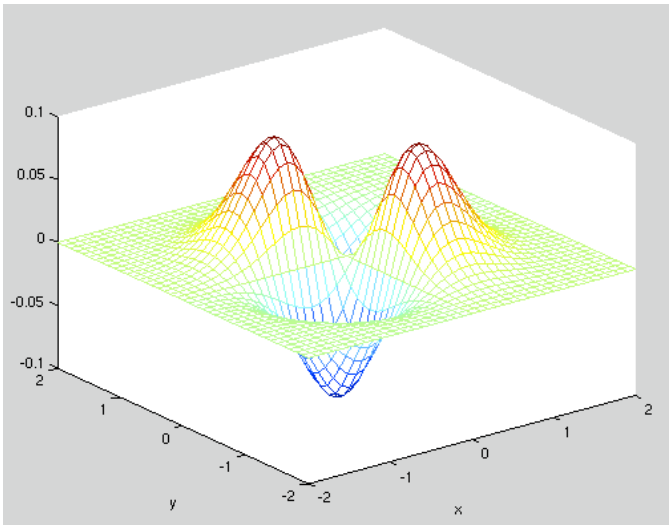


Example: Plot the surface defined by the function

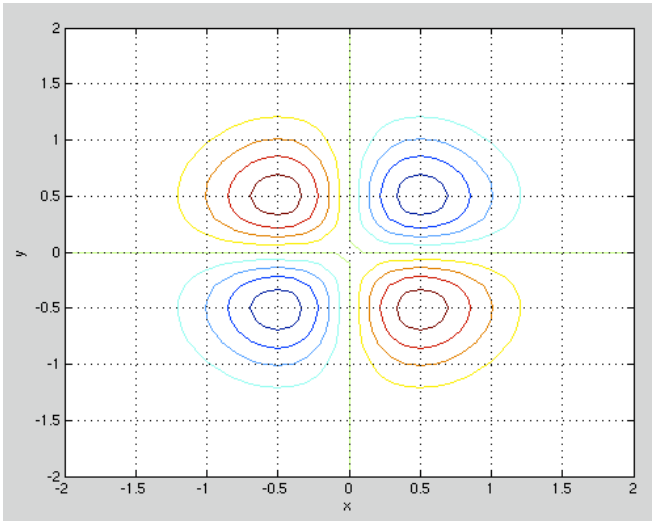
$$f(x,y) = -xye^{-2(x^2+y^2)}$$

on the domain -2 ≤ x ≤ 2 and -2 ≤ y ≤ 2. Find the values and locations of the maxima and minima of the function.

```
>> [X,Y] = meshgrid(-2:0.1:2,-2:0.1:2);
>> f = -X.*Y.*exp(-2*(X.^2+Y.^2));
>> figure(1)
>> mesh(X,Y,f),xlabel('x'),ylabel('y'),grid
```

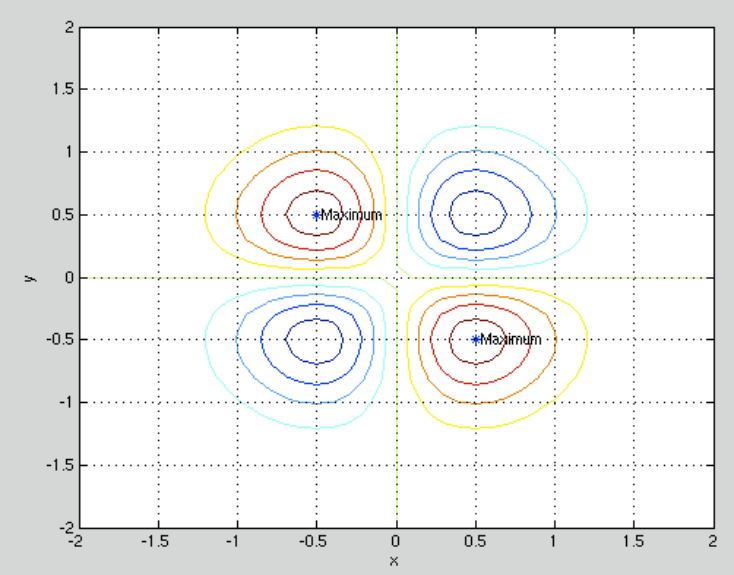



```
>> figure(2)
>> contour(X,Y,f)
>> xlabel('x'),ylabel('y'), grid, hold on
```



To locate the maxima of the "**f**" values on the grid:

```
>> fmax = max(max(f))

fmax = 0.0920

>> kmax = find(f == fmax)

kmax =   641
        1041

>> Pos = [X(kmax), Y(kmax)]

Pos = -0.5000    0.5000
       0.5000   -0.5000

>> plot(X(kmax), Y(kmax),'*')
>> text(X(kmax), Y(kmax), ' Maximum')
>> hold off
```



Additional Graphics Commands

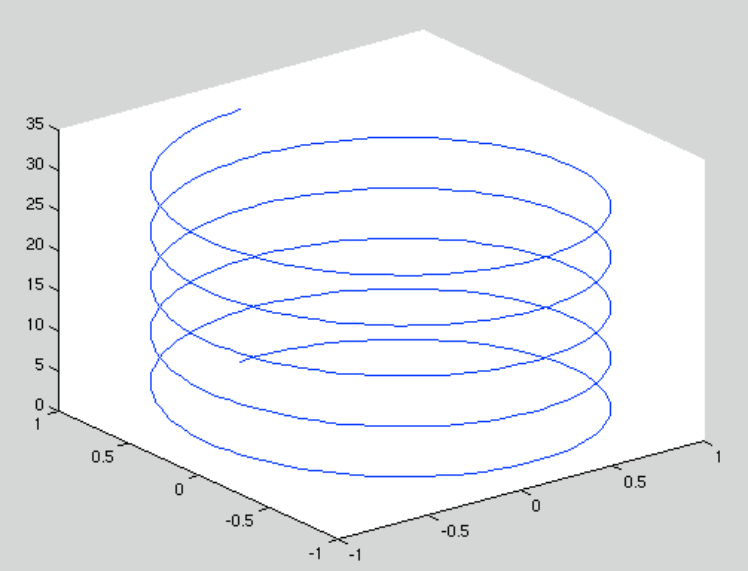
Line Plots

The 3-dimensional analog of the **plot** command is the **plot3** command. If x, y, and z are three vectors of the same length, then the command plot3(x,y,z) generates a line in 3-space

through the points whose coordinates are the elements of x, y, and z. For example,

```
>> t=0:pi/50:10*pi;
>> plot3(sin(t),cos(t),t)
```

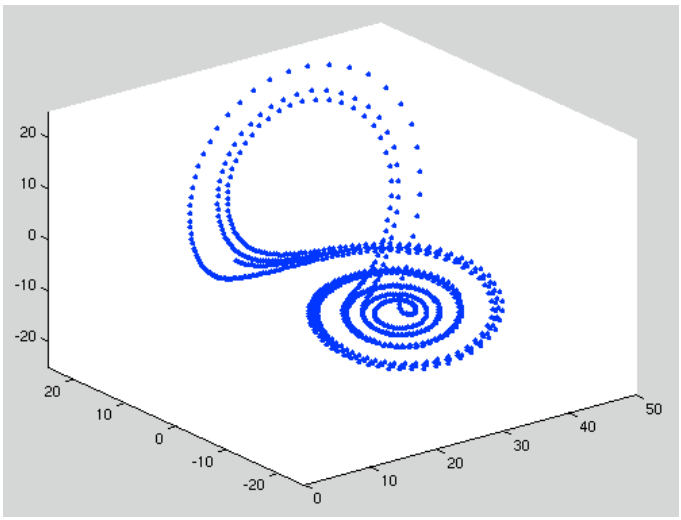
produces the helix shown below.



Another example that we will get to run much faster later on is

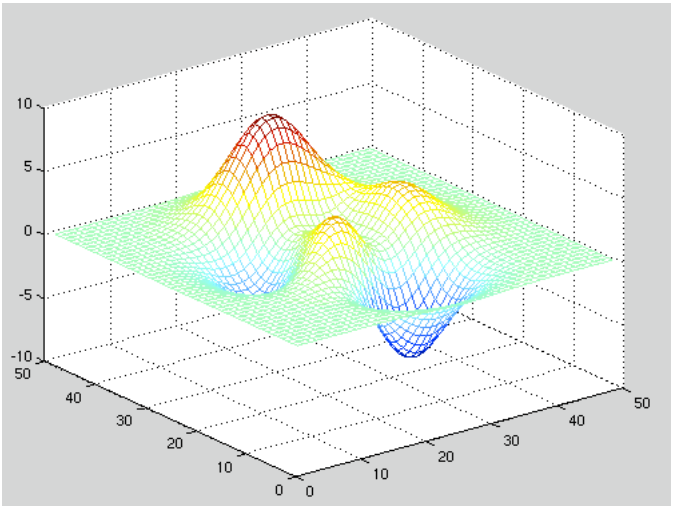
```
>> a = [-8/3 0 0; 0 -10 10; 0 28 -1];
>> y = [35 -10 -7]';
>> h = 0.01;
>> plot3(y(1),y(2),y(3),'.')
>> axis([0 50 -25 25 -25 25])
>> hold on
>> n = 0;
>> while n < 1000
     n = n+1;
     a(1,3) = y(2);
     a(3,1) = -y(2);
     ydot = a*y;
     y = y + h*ydot;
     plot3(y(1),y(2),y(3),'.')
     drawnow
   end
```

which produces the chaotic Lorentz strange attractor plot (butterfly effect) shown below.

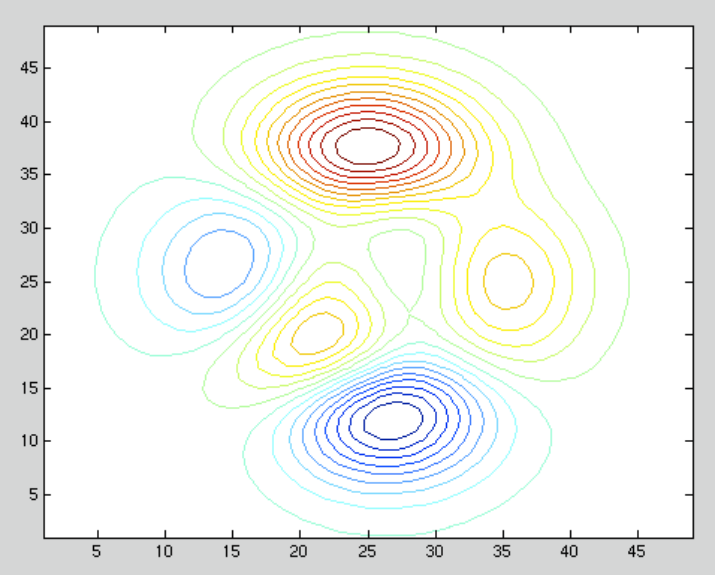


The peaks command generates an array of z-values from a function f(x,y) that has three local maxima and three local minima. The general form of the peaks command is peaks(n) which produces an n x n array. If no argument is used, then it defaults to n = 49. It is a useful test data set.

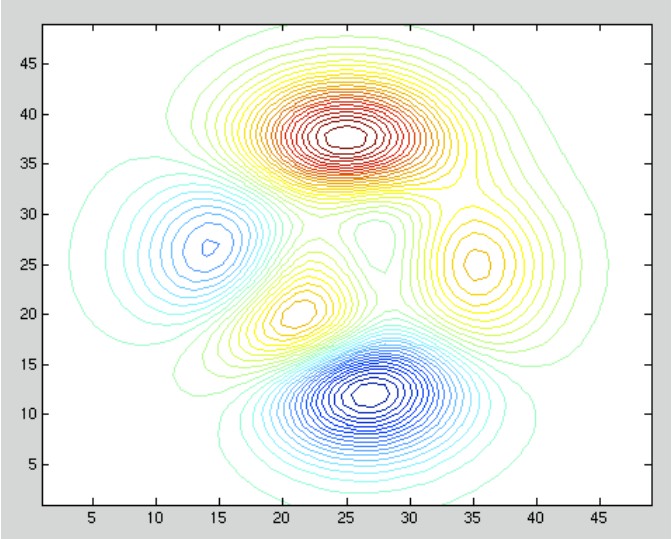
```
>> mesh(peaks)
```




```
>> contour(peaks,20) % 2nd argument specifies number of contours
```




```
>> contour(peaks,40)
```



Images

MATLAB displays an image by mapping each element in a matrix to an entry in the current color map, which is not necessarily related to the image in any way.

Normally images have their own colormap, which only includes the actual colors in the image. This is referred to as its color palette.

There are two distinct functions for displaying images, namely **pcolor** and **image.** They are similar in their operations. Both produce 2-D pictures with brightness or color values proportional to the elements of a given matrix. The most important differences are:

If A is an m x n array, then

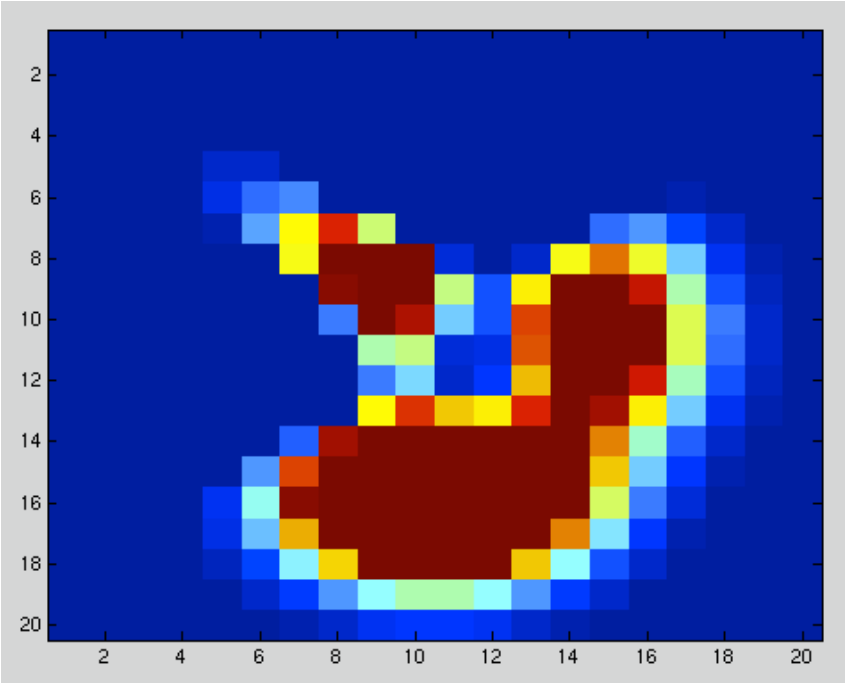
[1] **image**(A) produces an m x n array of cells while **pcolor**(A) produces and m x n grid of lines and hence only an (m-1_x(n-1) array of cells.

[2] **image**(A) always uses flat shading(each cell has constant color) while **pcolor**(A) has several shading modes(faceted, flat, interp).

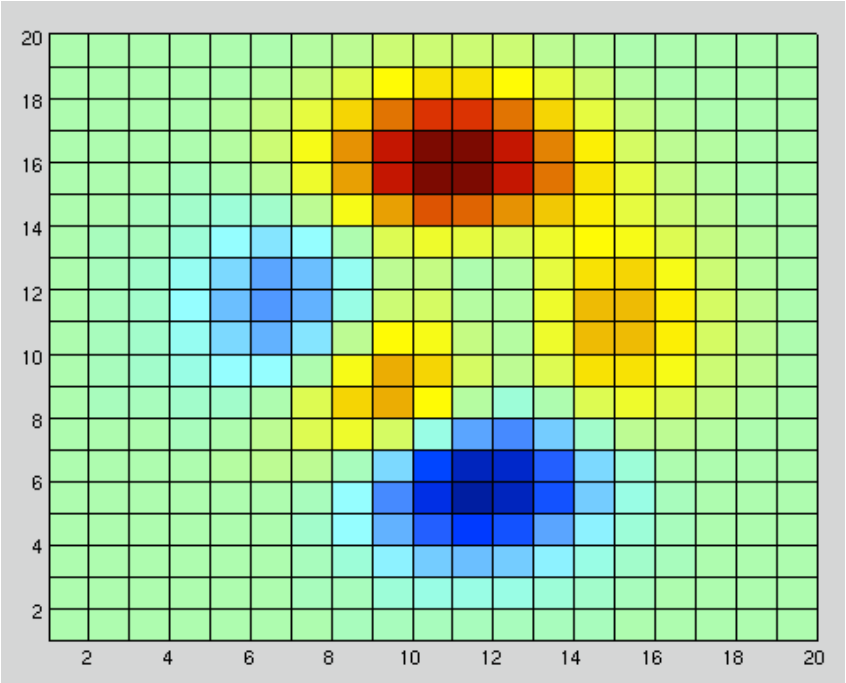
[3] **image** uses the elements in A to lookup color values directly in the **colormap** (the **colormap** has values only for integers in the range 0 to length_of_**colormap**(default=64; max=256)). **pcolor's** input matrix(A) is scaled by the color axis limits (set by **caxis** command).

For example, consider the array 30*peaks(20) as shown by both **image** and **pcolor**

```
>> image(30*peaks(20))
```

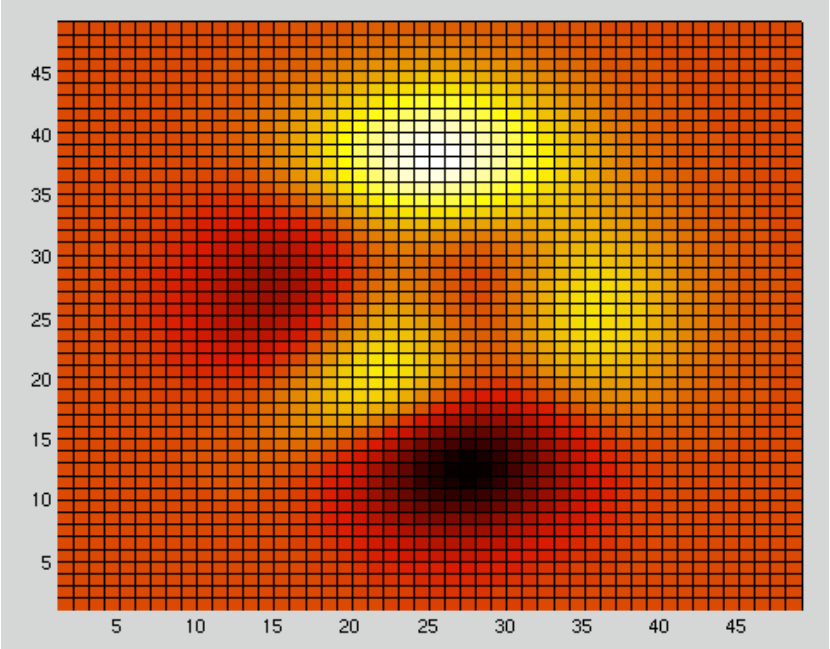


```
>> pcolor(30*peaks(20))
```

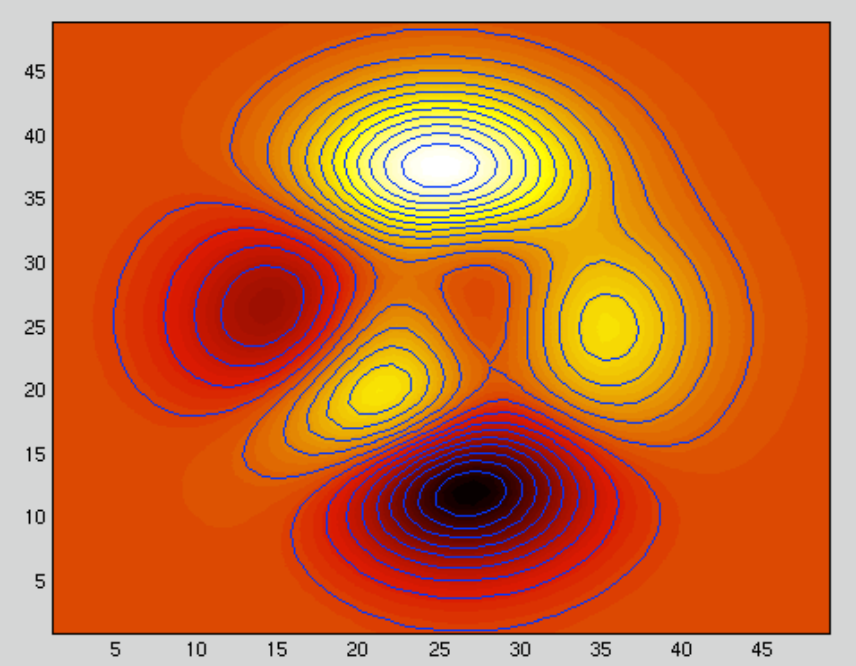


Some further examples are:

```
>> pcolor(peaks)
>> colormap(hot)
```



```
>> pcolor(peaks)
>> colormap(hot)
>> shading flat
>> shading interp
>> hold on
>> contour(peaks,20,'b')
>> hold off
```



Timing

MATLAB allows the timing of sections of code by providing the functions **tic** and **toc**. **tic** switches on a stopwatch while **toc** stops it and returns the CPU time (Central Processor Unit) in seconds. The timings will vary depending on the model of computer being used and its current load.

```
>> tic,for j=1:1000,x=pi^3;end,toc
Elapsed time is 0.004118 seconds.
>> tic,for j=1:1000,x=pi^3+sin(cos(pi/4));end,toc
Elapsed time is 0.008207 seconds.
>> tic,for j=1:10000,x=pi^3+sin(cos(pi/4));end,toc
Elapsed time is 0.155270 seconds.
```

Reading and Writing Data Files

Direct input of data from keyboard becomes impractical

when

- the amount of data is large and
- the same data is analyzed repeatedly

In these situations input and output is preferably accomplished via data files.

The simplest way to accomplish this task is to use the commands **save** and **load** that, respectively, write and read the values of variables to disk files.

```
a = 0.9501    0.8913    0.8214    0.9218    0.9355
    0.2311    0.7621    0.4447    0.7382    0.9169
    0.6068    0.4565    0.6154    0.1763    0.4103
    0.4860    0.0185    0.7919    0.4057    0.8936

>> save a
```

Look in your current directory and you will now find a file named a.mat. It contains the variable a. We now change a in the workspace.

```
>> a = 789

a = 789
```

We can recover the old value of a by using the **load** command.

```
>> load a
>> a

a = 0.9501    0.8913    0.8214    0.9218    0.9355
    0.2311    0.7621    0.4447    0.7382    0.9169
    0.6068    0.4565    0.6154    0.1763    0.4103
    0.4860    0.0185    0.7919    0.4057    0.8936
```

Suppose that we have a data file containing ascii text and it is named weather.dat and it looks like:

```
30 4.0
31 3.7
38 4.1
49 3.7
59 3.5
68 2.9
74 2.7
```

```
72 3.7
65 3.4
55 3.4
45 4.2
34 4.9
```

The command

```
>> load weather.dat
```

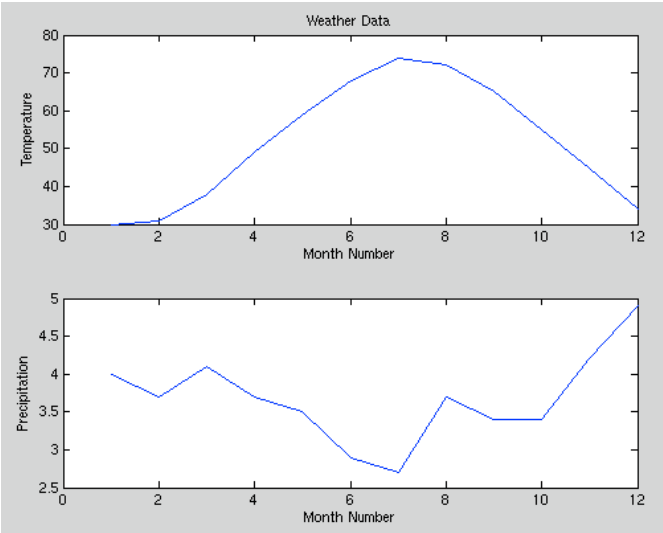
produces a new variable named weather which contains a 12 x 2 matrix.

```
>> weather

weather = 30.0000    4.0000
          31.0000    3.7000
          38.0000    4.1000
          49.0000    3.7000
          59.0000    3.5000
          68.0000    2.9000
          74.0000    2.7000
          72.0000    3.7000
          65.0000    3.4000
          55.0000    3.4000
          45.0000    4.2000
          34.0000    4.9000
```

We can then use this data as follows:

```
>> temp = weather(:,1);
>> precip = weather(:,2);
>> subplot(2,2,1)
>> subplot(2,1,1)
>> plot(temp)
>> title('Weather Data')
>> xlabel('Month Number')
>> ylabel('Temperature')
>> subplot(2,1,2)
>> plot(precip)
>> xlabel('Month Number')
>> ylabel('Precipitation')
```

to obtain the plot



In most cases of scientific work, however, when data are written to or read from a file it is crucially important that a correct data format is used. The data format is the key to interpreting the contents of a file and must be known in order to correctly interpret the data in an input file. There a two types of data files: formatted and unformatted. Formatted data files use format strings to define exactly how and in what positions of a data-record the data is stored. Unformatted storage, on the other hand, only specifies the number format.

Example: Suppose the numeric data is stored in a file 'table.dat' in the form of a table, as shown below.

```
100 2256
200 4564
300 3653
400 6798
500 6432
```

The three commands,

```
>> fid = fopen('table.dat','r');
>> a = fscanf(fid,'%3d%4d');
>> fclose(fid)
```

respectively

1. open a file for reading (this is designated by the string 'r'). The variable **fid** is assigned a unique integer which

identifies the file used (a file identifier). We use this number in all subsequent references to the file.

2. read pairs of numbers from the file with file identifier **fid**, one with 3 digits and one with 4 digits, and

3. close the file with file identifier **fid.**

This produces a **column vector** a with elements, 100 2256 200 4564 ...500 6432. This vector can be converted to a 5 x 2 matrix by the command

```
>> A = reshape(a,2,5)'

A =      100        2256
         200        4564
         300        3653
         400        6798
         500        6432
```

Formatted Files

Some computer codes and measurement instruments produce results in formatted data files. In order to read these results into MATLAB for further analysis the data format of the files must be known. Formatted files in ASCII format are written to and read from with the commands **fprintf** and **fscanf.**

> **fprintf(fid, 'format', variables)** writes variables in a format specified in string 'format' to the file with identifier fid
>
> **a = fscanf(fid, 'format',size)** assigns to variable a data read from file with identifier fid under format 'format'.

Some sample formats:

```
%6.2f  %12.8f
```

This format control string specifies the format for each line of data - a fixed-point value of six characters with two decimal places + two spaces + a fixed-point value of twelve characters with eight decimal places, for example,

```
1234.12  1234.12345678
```

```
Valid conversion specifications are:

%s - Sequence of characters
%d - Base 10 integers
%e, %f - Floating-point numbers

Some examples illustrate formats

>> fprintf('pi^3 is %12.8f\n',pi^3)
pi^3 is  31.00627668
```

The example prints pi^3 with 8 digits past the decimal point in a space of 12 characters. Note that the last character is '\n', which is newline. If this were excluded, the next line of output would start at the end of this line (sometimes you want that!).

```
>> fprintf('pi^3 is %12.8e\n',pi^3)
pi^3 is 3.10062767e+01
```

The example prints pi^3 with 8 digits past the decimal point in a space of 12 characters using exponential (powers of ten) notation.

```
>> fprintf('%5d %5d %5d %5d\n',round(100*rand(1,4)))
   84     2    68    38
```

This example prints the output as integers each allotted 5 spaces.

Example: Suppose a sound pressure measurement system produces a record with 512 time - pressure readings stored in a file 'sound.dat'. Each reading is listed on a separate line according to a data format specified by the string, '%8.6f %8.6f'.

A set of commands reading time - sound pressure data from 'sound.dat' is,

Step 1: Assign a namestring to a file identifier.

```
>> fid1 = fopen('sound.dat','r');
```

The string 'r' indicates that data is to be read (not written) from the file.

Step 2: Read the data to a vector named 'data' and close the file,

```
>> data = fscanf(fid1, '%f %f');
>> fclose(fid1);
```

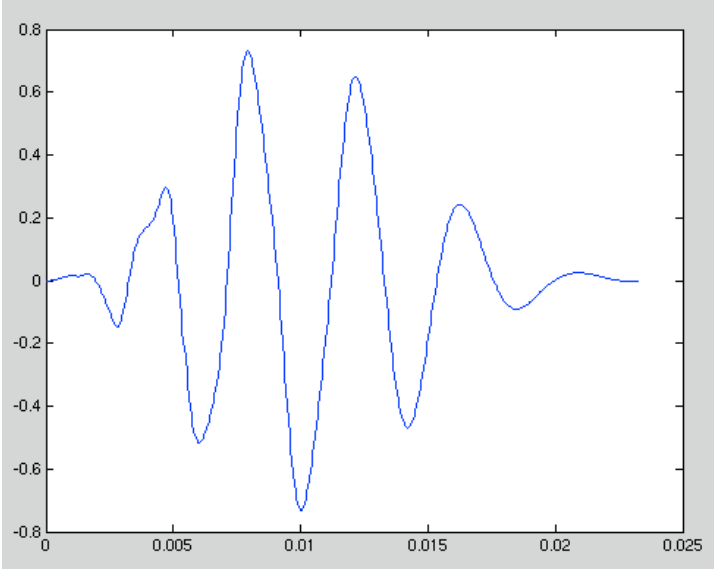
Step 3: Partition the data in separate time and sound pressure vectors,

```
>> t = data(1:2:length(data));
>> press = data(2:2:length(data));
```

The pressure signal can be plotted in a diagram,

```
>> plot(t, press);
```

The result is shown below.



In this laboratory we will be able to do all file read/writes using load/save and ascii files

If we want to write an ascii file with a specific format(say for use by someone using a different programming language), then we cannot use the **save** command.

Example: Create a ascii file called gxp.dat containing a short table of the exponential function.

```
>> x = 0:.1:1;
>> y = [x; exp(x)];
>> fid = fopen('gxp.txt', 'wt');
>> fprintf(fid, '%6.2f %12.8f\n', y);
```

```
>> fclose(fid)
Now examine the contents of gxp.dat:

>> type gxp.dat

  0.00   1.00000000
  0.10   1.10517092
  0.20   1.22140276
  0.30   1.34985881
  0.40   1.49182470
  0.50   1.64872127
  0.60   1.82211880
  0.70   2.01375271
  0.80   2.22554093
  0.90   2.45960311
  1.00   2.71828183

This file can be read back using

>> load gxp.dat

exp =      0    1.0000
      0.1000    1.1052
      0.2000    1.2214
      0.3000    1.3499
      0.4000    1.4918
      0.5000    1.6487
      0.6000    1.8221
      0.7000    2.0138
      0.8000    2.2255
      0.9000    2.4596
      1.0000    2.7183
```

Programming in MATLAB

M-files

MATLAB can execute a sequence of statements stored in files. Such files are called M-files because they must have the file type suffix of .m as the last part of their filename. Much of your work with MATLAB will be in creating and refining M-files. There are two types of M-files: script files and function files .

Script or Program files

A script file consists of a sequence of normal MATLAB statements. If the file has the filename, say, rotate.m , then the MATLAB command **rotate** will cause the statements in the file to be executed. Variables in a script file are global and will change the value of variables of the same name in the environment(workspace) of the current MATLAB session.

An M-file can reference other M-files, including referencing itself recursively. We will see m-file examples later in these notes and you will write many m-files.

Function files (discussed earlier)

Function files provide extensibility to MATLAB. You can create new functions specific to your problem which will then have the same status as other MATLAB functions. Variables in a function file are by default local.

A script file can call function files.

We first illustrate with a simple example of a function file.

```
function y = fofx(x)
% function to calculate y given x
y=cos(tan(pi*x));
```

This should be placed in a diskfile with filename **fofx.m** (corresponding to the function name). The first line declares the function name, input arguments, and output arguments; without this line the file would be a script file. Then a MATLAB statement **z = fofx(4)**, for example, will cause the variable z to be assigned the value cos(tan(4*pi)). Since variables in a function file are local their names are independent of those in the current MATLAB environment. The % symbol indicates that the rest of the line is a comment; MATLAB will ignore the rest of the line. However, the first few comment lines, which document the M-file, are available to the on-line help facility. Such documentation should always be included in a function file. Some of MATLAB's functions are built-in while others are distributed as M-files.

```
>> zz=fofx(5)

zz = 1
```

Examples of Script or Program m-files

```
% trapazoid rule for integral of x^2 between 0 and 1
% exact answer is 1/3
h = input('mesh size h = ');
x = (0:h:1);
lenx = length(x);
y = x.^2;
int = (h/2)*(y(1)+2*sum(y(2:(lenx-1)))+y(lenx))
```

Save it as **trapint.m** .

Now enter the command

```
>> trapint
```

You will be prompted for a mesh size h. Enter a value.
The printed result is the value of the integral.

```
>> trapint
mesh size h = 0.1

int = 0.3350

>> trapint
mesh size h = 0.01

int = 0.3333
```

```
% generating noisy data
x = 0:0.025:2;
y = sin(pi*x);
yn = y + 0.25*(rand(size(x)) - 0.5);
plot(x,y,'--',x,yn,'-')
title(['dashed line: sin(pi*x)', ...
 '  solid line: noisy data'])
xlabel('noise is random from [-1/8,1/8]')
```

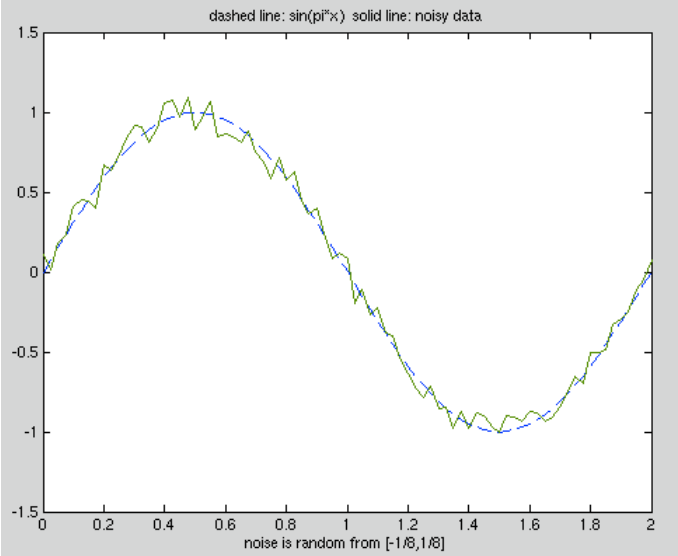
Save it as **noisedat.m** .

Now enter the command

```
>> noisedat
```

The result will be a plot of the noisy data.

```
>> noisedat
```



The next task is to plot sin(j*pi*t) for j = 1 ... 5. Use t = 0:0.025:1 . We will illustrate three approach with three m-files.

Approach #1 uses a single plot command listing each graph.

```
t = 0:0.025:1;
plot(t,sin(pi*t), t,sin(2*pi*t), t,sin(3*pi*t), ...
t,sin(4*pi*t), t,sin(5*pi*t))
```

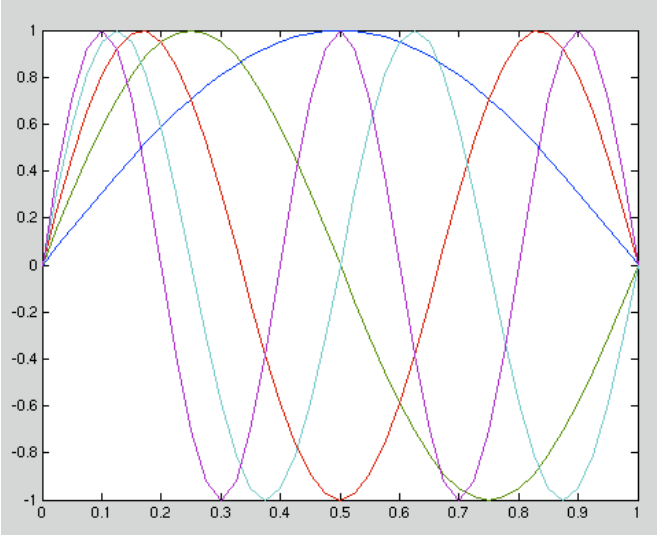
Approach #2 uses a loop and the command **hold on.**

```
t = 0:0.025:1;
plot(t,sin(pi*t))
hold on
for j = 2:5
  plot(t,sin(j*pi*t))
end
hold off
```

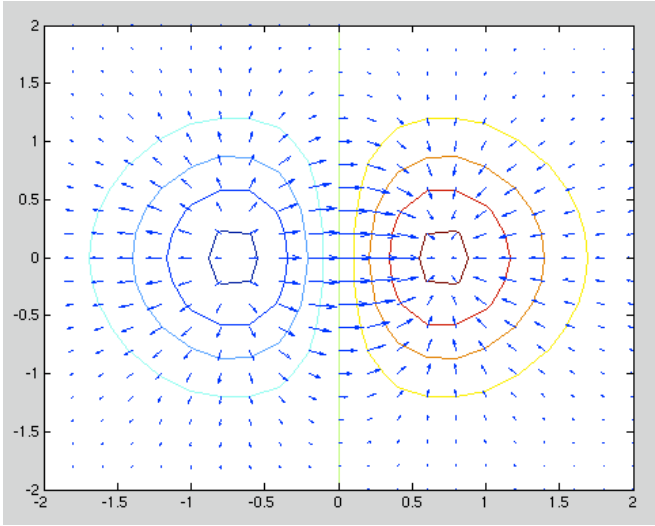
Approach #3 builds a matrix.

```
t = 0:0.025:1;
lt = length(t);
A = (1:5)'*pi*t;
plot(t,sin(A))
```

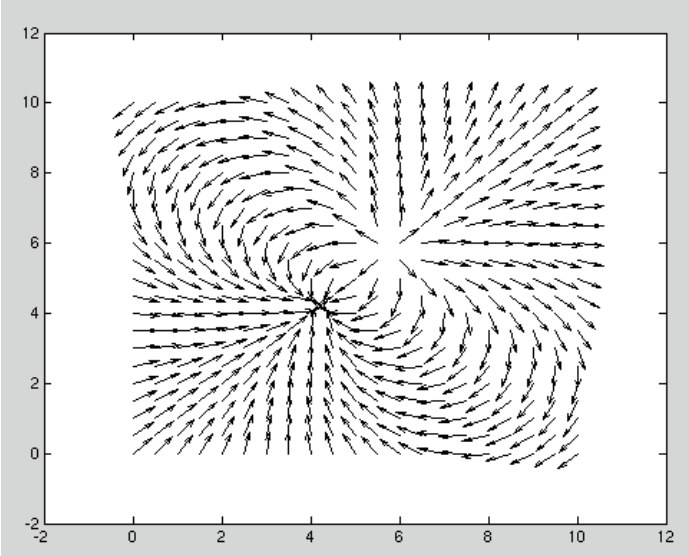
```
>> approach3
```



```
% quiverex.m
% potential = x*exp(-x^2-y^2)
% field components using gradient function
% plot of potential contours and field vectors
xord = -2:.2:2;
yord = -2:.2:2;
[x,y] = meshgrid(xord,yord);
z = x .* exp(-x.^2 - y.^2);
[px,py] = gradient(z,.2,.2);
contour(x,y,z)
hold on
quiver(x,y,px,py)
hold off
```

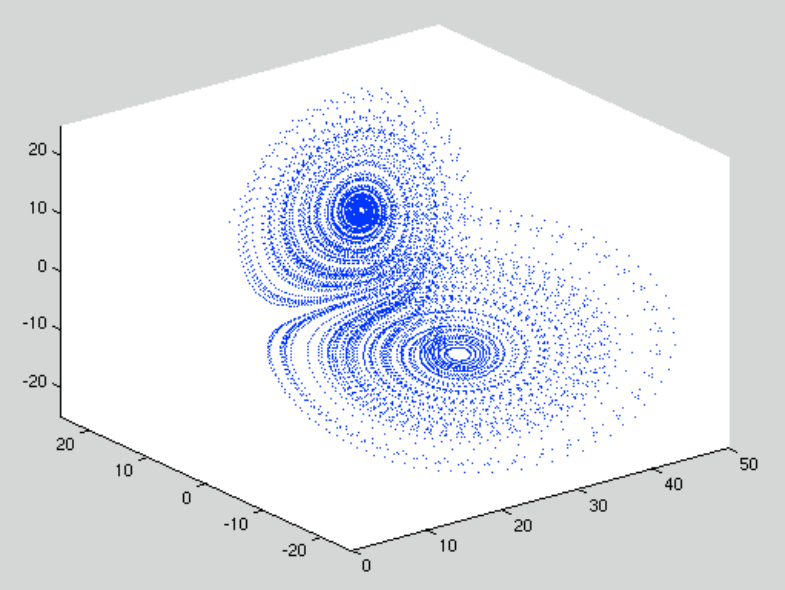


```
% elec3.m
% vector field plot for 2 charges
x=0:.5:10;
y=0:.5:10;
[X,Y]=meshgrid(x,y);
Z=0.1./sqrt((X-5.75).^2+(Y-5.75).^2);
Z=Z-0.1./sqrt((X-4.25).^2+(Y-4.25).^2);
[px,py]=gradient(Z,.1,.1);
norm=sqrt(px.^2 + py.^2);
px=-px./norm;
py=-py./norm;
quiver(X,Y,px,py,'k');
```



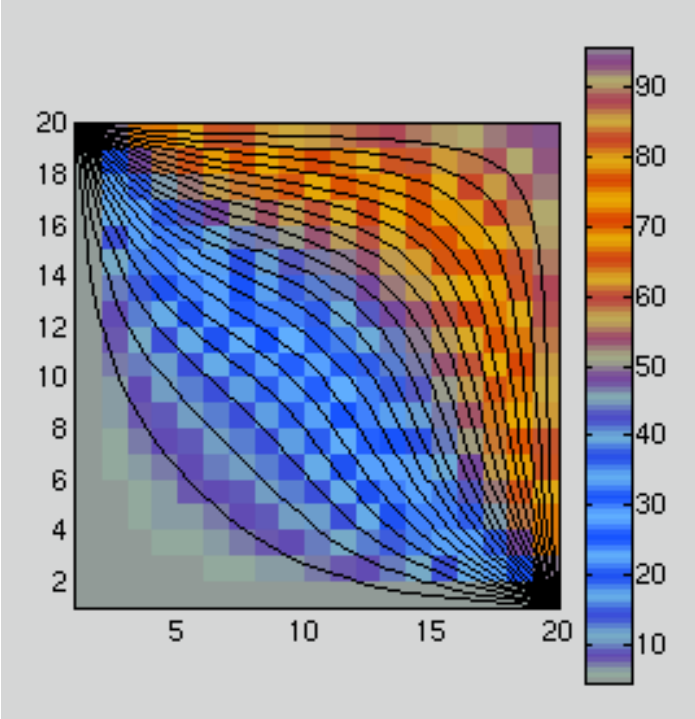

```
% lorenz1.m
% Plot of Lorentz attractor done earlier
% now using Handle Graphics commands (much faster)
y=input('Enter starting point - (0<x<50, -25<y<25, -25<z<25) in [] : ' );
y=y';
a=[-8/3 0 0; 0 -10 10; 0 28 -1];
%y=[35 -10 -7]';
h=.01;
p=plot3(y(1),y(2),y(3),'.', ...
        'EraseMode','none','MarkerSize',2)
axis([0 50 -25 25 -25 25])
hold on
n=0;
while (n < 1000)
   n=n+1;
```

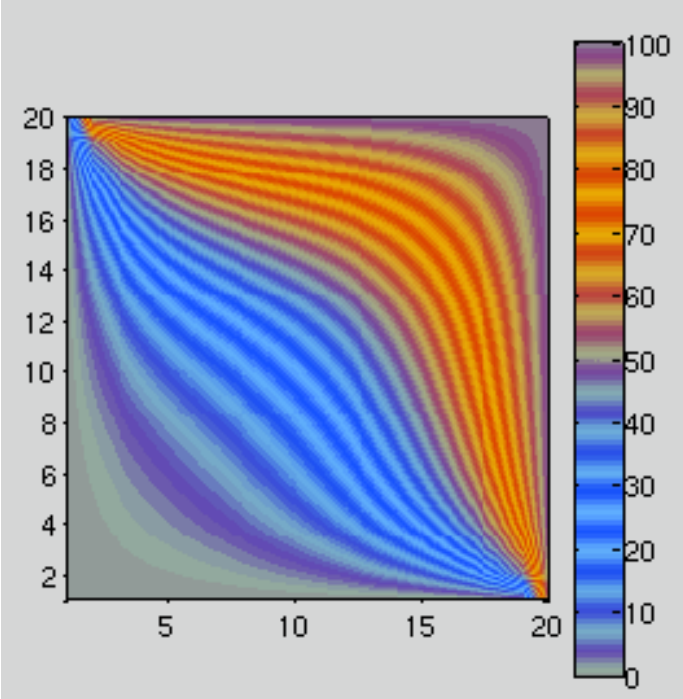
```
    a(1,3)=y(2);
    a(3,1)=-y(2);
    ydot=a*y;
    y=y+h*ydot;
    set(p,'XData',y(1),'YData',y(2),'ZData',y(3))
    drawnow
end
```



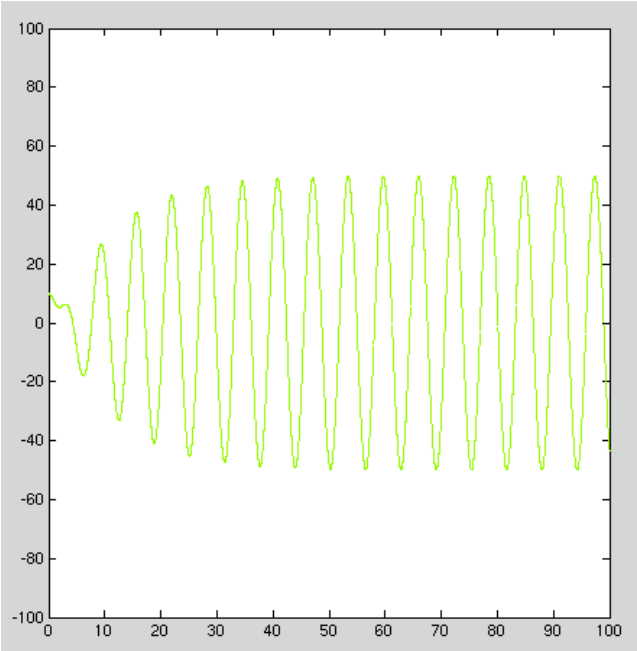

```
% MATLAB code for simple relaxation method
% initial potential values
% or boundary values
p=zeros(20,20);
% set boundary values
p(1,1:20)=zeros(1,20);
p(20,1:20)=100*ones(1,20);
p(1:20,1)=zeros(20,1);
p(1:20,20)=100*ones(20,1);
% iteration
for i=1:100
  i
  p=0.25*(p(:,[20,[1:19]])+p(:,[[2:20],1])+ ....
                              p([20,[1:19]],:)+p([[2:20],
1],:));
% reset changed boundaries
% set boundary values
p(1,1:20)=zeros(1,20);
p(20,1:20)=100*ones(1,20);
p(1:20,1)=zeros(20,1);
```

```
p(1:20,20)=100*ones(20,1);
end
x=[1:20];
y=[1:20];
% interpolate 20x20 --> 200x200
tx=1:.1:20;
ty=1:.1:20;
[X,Y] = meshgrid(tx,ty);
pot=interp2(x,y,p,X,Y);
figure('Position',[200 100 300 300])
pcolor(tx,ty,pot)
hold on
colormap(waves)
shading flat
axis('square')
colorbar
hold off
figure('Position',[100 10 300 300])
pcolor(x,y,p)
hold on
colormap(waves)
shading flat
axis('square')
contour(x,y,p,20,'k')
colorbar
hold off

% function waves.m
function map = waves(m)
m=128;
R = zeros(m,3);
j=1:128;
r=0.5*(1-sin(2*pi*j/128));
b=0.5*(1+sin(2*pi*j/128));
g=0.2*(2+sin(2*pi*j/8));
R(:,1)=r';
R(:,2)=g';
R(:,3)=b';
map=R;
```


20
18
16
14
12
10
8
6
4
2
5
10
15
20
90
80
70
60
50
40
30
20
10


20
18
16
14
12
10
8
6
4
2
5
10
15
20
100
90
80
70
60
50
40
30
20
10
0

```
% motion1d.m
% one-dimensional motion of a damped-driven oscillator
% Solves ODE using Runge-Kutta method (will learn later)
% uses acceleration function acellx.m
clear
% set initial conditions
x=10;
vx=0;
% set time step
h=.01;
% open window
figure('Position',[50 50 500 500]);
% set plot handle and parameters
p=plot(0,x,'.g','EraseMode','none','MarkerSize',2)
axis(100*[0 1 -1 1]);
count=-1;
% Runka-Kutta solution of DEQ
while (count < 10000)
  count=count+1;
  fx1=vx;
  gx1=accel1x(x,vx,count*h);
  fx2=vx+h*gx1/2;
  gx2=accel1x(x+h*fx1/2,vx+h*gx1/2,(count+0.5)*h);
  fx3=vx+h*gx2/2;
  gx3=accel1x(x+h*fx2/2,vx+h*gx2/2,(count+0.5)*h);
  fx4=vx+h*gx3;
  gx4=accel1x(x+h*fx3,vx+h*gx3,(count+1)*h);
  x=x+h*(fx1+2*fx2+2*fx3+fx4)/6;
  vx=vx+h*(gx1+2*gx2+2*gx3+gx4)/6;
  set(p,'XData',count*h,'YData',x)
  drawnow
end

% function accellx.m
function a=accel1x(x,vx,t)
a=-x - .2*vx +10*sin(t);
```


100
80
60
40
20
0
-20
-40
-60
-80
-100
0
10
20
30
40
50
60
70
80
90
100