

MATLAB Notes

Fall 2007

Physics 002A and 026

Professor John Boccio
Physics and Astronomy Department
Swarthmore College

Let us now learn to use MATLAB.

It is a Calculator

```
>> 1+1
ans =
     2
>> 2*3
ans =
     6

>> 5/6
ans =
    0.8333
>> exp(-3)
ans =
    0.0498
>> atan2(-1,2)
ans =
   -0.4636
>> sin(5)
ans =
   -0.9589
>> ans                                % is comment symbol
ans =                                   % and is last calculated result
   -0.9589
>> 1.23e15
ans =                                   % a number like 1.23 x 1015
   1.2300e+15
```

The up-arrow key will display previous command. The left- and right-arrow keys and the delete key can then be used to edit an old command. Hitting RETURN then executes it. Try it!

Making Script Files

MATLAB "programs" are called scripts and are text files containing MATLAB commands

```
% this is a comment line
% program test.m
clear
g=0.1
t=2
f=1-exp(-g*t) % compute the decay fraction
```

Running Script Files

```
>> test

g =
    0.1000
t =
     2
f =
    0.1813

% this is a comment line
% program test.m
clear;
g=0.1;          % semicolon stops output
t=2;
f=1-exp(-g*t) % compute the decay fraction

>> test

f =
    0.1813
```

If a line of code gets too long, just break it anywhere and add the continuation symbol `....` as shown below

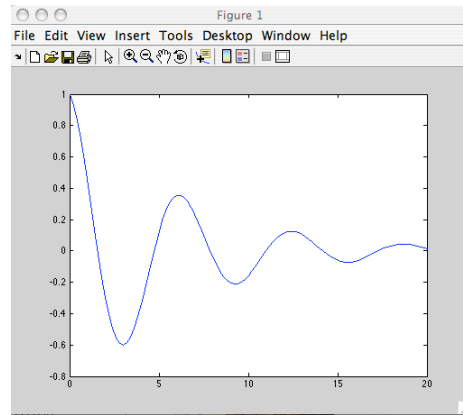
```
a=sin(x)*exp(-y)- cos(x)*exp(y) +tan(x)*log(z) + sqrt(b)* ...
    sqrt(x*y*z);
```

A Sample Script (just to see how it all works)

Create the script `sample1.m` shown below:

```
% this is a comment line
% program sample1.m
clear; % clear all variables from memory
close all; % close all open figure windows
h=input(' Enter the step-size h - ');
x=0:h:20; % build an array of points [0,h,2h,....,20]
f=exp(-x/6).*cos(x); % build the array [f(0),f(h),....,f(20)]
plot(x,f)
fprintf(' Plot completed, h = %g \n',h)

>> sample1
Enter the step-size h - 0.2
Plot completed, h = 0.2
```



Variables

Variables are case-sensitive. You could write

```
Force_of_1_on_2 = G*Mass_1*Mass_2/Distance_between_1_and_2^2
```

which is readable and informative, but you will develop repetitive stress injury. Instead this is good enough.

```
F = G*m1*m2/r12^2
```

Numerical Accuracy

All numbers have 15 digits of accuracy; only 5 displayed normally.

```
>> 355/113
ans =
    3.1416
>> format long
>> 355/113
ans =
    3.141592920353983
>> format short
>> 355/113
ans =
    3.1416
>> exp(20)
ans =
    4.8517e+08
>> format long
>> pi
ans =
    3.141592653589793
```

Assigning Values to Variables

```
>> a=20;
>> a
a =
    20
```

Matrices

```
>> n=2;
>> size(n)
ans =
     1     1
>> a=[1,2,3,4,5];
>> a
a =
     1     2     3     4     5
>> size(a)
ans =
     1     5           % #rows #columns
>> c=[1,2,3;4,5,6;7,8,9];
>> c
c =
     1     2     3
     4     5     6
     7     8     9
>> size(c)
ans =
     3     3
% carriage returns at the end of each line
>> a=[1 2 3 4
     5 6 7 8
     9 10 11 12
    13 14 15 16];
>> a
a =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
>> size(a)
ans =
     4     4
```

Accessing Elements

```
>> a(3,2)
ans =
    10
>> a(3,2:4)
ans =
    10    11    12

>> a(2:3,2:3)
ans =
     6     7
    10    11
>> a(4,:)
ans =
    13    14    15    16
```

Strings

```
>> s='This is a string';
>> s
s =
This is a string
>> size(s)
ans =
     1    16
>> s(1:7)
ans =
This is

>> ss=['Inserting a number into a string.', ' bb = ', num2str
(bb)];
>> ss
ss =
Inserting a number into a string. bb = 23
```

Input, Calculating and Output**Input**

```
>> N=input(' Enter a value for N - ');
Enter a value for N - 76
>> sin(N)
ans =
    0.566107636898180
```

Suppose I have a data file (text) of the following form:

```
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
```

If its name is weather.dat, then the command

```
>> load weather.dat
>> weather
weather =
  30.000000000000000    4.000000000000000
  31.000000000000000    3.700000000000000
  38.000000000000000    4.100000000000000
  49.000000000000000    3.700000000000000
  59.000000000000000    3.500000000000000
  68.000000000000000    2.900000000000000
  74.000000000000000    2.700000000000000
  72.000000000000000    3.700000000000000
  65.000000000000000    3.400000000000000
  55.000000000000000    3.400000000000000
  45.000000000000000    4.200000000000000
  34.000000000000000    4.900000000000000
>> size(weather)
ans =
    12     2
```

i.e., we produce an array variable called weather.

Calculating

Addition and Multiplication

```
>> a=[1,2,3;4,5,6;7,8,9];
>> b=[3,2,1;6,4,5;8,7,9];
>> a+b
ans =
     4     4     4
    10     9    11
    15    15    18
>> a-b
ans =
```

```
    -2    0    2
    -2    1    1
    -1    1    0
>> a*b      % matrix multiplication
ans =
    39    31    38
    90    70    83
   141   109   128
>> a*[1;2;3]
ans =
    14
    32
    50
>> a*[1 2 3]'      % transpose
ans =
    14
    32
    50
>> a^3      % matrix multiplication
ans =
    468    576    684
   1062   1305   1548
   1656   2034   2412
>> a=2*weather-1;
>> a
a =
  1.0e+02 *
  0.5900000000000000    0.0700000000000000
  0.6100000000000000    0.0640000000000000
  0.7500000000000000    0.0720000000000000
  0.9700000000000000    0.0640000000000000
  1.1700000000000000    0.0600000000000000
  1.3500000000000000    0.0480000000000000
  1.4700000000000000    0.0440000000000000
  1.4300000000000000    0.0640000000000000
  1.2900000000000000    0.0580000000000000
  1.0900000000000000    0.0580000000000000
  0.8900000000000000    0.0740000000000000
  0.6700000000000000    0.0880000000000000
>> b=weather.*a; % element by element
>> b
b =
  1.0e+04 *
  0.1770000000000000    0.0028000000000000
  0.1891000000000000    0.0023680000000000
  0.2850000000000000    0.0029520000000000
```

```

0.4753000000000000    0.0023680000000000
0.6903000000000000    0.0021000000000000
0.9180000000000000    0.0013920000000000
1.0878000000000000    0.0011880000000000
1.0296000000000000    0.0023680000000000
0.8385000000000000    0.0019720000000000
0.5995000000000000    0.0019720000000000
0.4005000000000000    0.0031080000000000
0.2278000000000000    0.0043120000000000
>> [1,2,3].*[3,2,1]
ans =
     3     4     3

>> [1,2,3]./[3,2,1]
ans =
    0.3333333333333333    1.0000000000000000    3.0000000000000000
>> [1,2,3].^2
ans =
     1     4     9

```

Complex Arithmetic

```

>> i
ans =
           0 + 1.0000000000000000i

>> z1=1+2i
z1 =
    1.0000000000000000 + 2.0000000000000000i
>> z1=1+2*i
z1 =
    1.0000000000000000 + 2.0000000000000000i
>> z2=2-3i
z2 =
    2.0000000000000000 - 3.0000000000000000i
>> z1-z2
ans =
   -1.0000000000000000 + 5.0000000000000000i
>> z1*z2
ans =
    8.0000000000000000 + 1.0000000000000000i
>> z1/z2
ans =
  -0.307692307692308 + 0.538461538461538i
>> real(z1)
ans =
     1
>> imag(z1)
ans =

```

```

      2
>> conj(z1)
ans =
    1.0000000000000000 - 2.0000000000000000i
>> abs(z1)
ans =
    2.236067977499790
>> angle(z1)
ans =
    1.107148717794090
>> exp(i*pi/4)
ans =
    0.707106781186548 + 0.707106781186547i
>> x=pi*(0:2)/5;
>> x
x =
         0         0.6283         1.2566
>> sin(x)
ans =
         0         0.5878         0.9511

```

Mathematical Functions

```

cos(x)    sin(x)    tan(x)    sec(x)    csc(x)    cot(x)
acos(x)   asin(x)   atan(x)   atan2(y,x) exp(x)    log(x)
log10(x)  log2(x)   sqrt(x)   cosh(x)   sinh(x)   tanh(x)
sech(x)   csch(x)   coth(x)   acosh(x)  asinh(x)  atanh(x)
factorial(x) sign(x) airy(n,x) besselh(n,x) besseli(n,x)
besselj(n,x) besselk(n,x) bessely(n,x) beta(x,y) betainc(x,y,z)
betaln(x,y) ellipj(x,m) ellipke(x) erf(x) erfc(x) erfcx(x)
erfinv(x) gamma(x) gammainc(x,a) gammaln(x) expint(x)
legendre(n,x)

```

Housekeeping Functions

```

abs(x)          absolute value of a number(real or complex)
clc             clears the command window
ceil(x)         nearest integer looking towards +infinity
clear           clear all assigned variables
close all       close all figure windows
close 3         close window number 3
fix(x)          nearest integer looking towards zero
fliplr(A)       flip matrix A, left for right
flipud(A)       flip matrix A, up for down
floor(x)        nearest integer looking towards -infinity
length(a)       number of element in a vector

```

<code>mod(x,y)</code>	integer remainder of x/y
<code>rem(x,y)</code>	integer remainder of x/y
<code>rot90(A)</code>	rotate a matrix by 90 degrees
<code>round(x)</code>	nearest integer
<code>sign(x)</code>	sign of x; 0 if x=0
<code>size(c)</code>	dimensions of matrix

All functions operate on arrays.

```
>> floor([1.5,2.7,-1.5])
ans =
     1     2    -2
```

Output

```
>> fprintf('  N =%g \n',500)
  N =500
>> fprintf('  x =%1.12g \n',pi)
  x =3.14159265359
>> fprintf('  x=%1.10e \n',pi)
  x=3.1415926536e+00
>> fprintf('  x=%6.2f \n',pi)
  x=  3.14
>> fprintf('  x =%12.8f  y =%12.8f \n',5,exp(5))
  x =  5.00000000  y =148.41315910
```

Arrays and x-y Plotting

```
>> clear; close all;  %close the figure windows
>> x=0:01:10;
```

In this case the array (vector) `x` is a cell-edge grid where the values are at the edge of the intervals.

```
x=0.0,0.01,0.02,0.03,.....,1.0
```

A cell-center grid is one that has `N` subintervals, but the data points are at the center of the intervals, i.e.,

```
>> dx=0.01;
>> x=0.5*dx:dx:10-0.5*dx;
      x=0.005,0.015,0.025,.....,0.995
```

If middle number left out, then interval assumed to be 1:

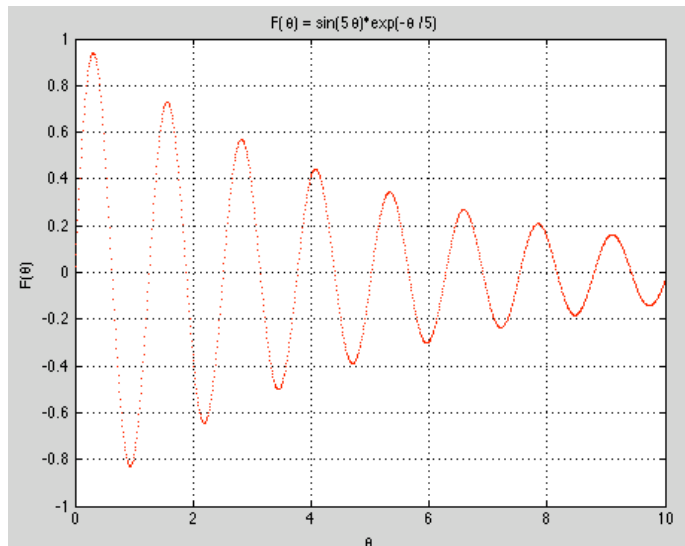
```
>> 1:5
ans =
     1     2     3     4     5
>> -2:3
ans =
    -2    -1     0     1     2     3
>> 2:.1:2.5
```

```
ans =
    2.0000    2.1000    2.2000    2.3000    2.4000    2.5000
>> 3:-.2:2
ans =
    3.0000    2.8000    2.6000    2.4000    2.2000    2.0000
```

Now try these commands;

```
>> y=sin(5*x).*exp(-x/5);
>> plot(x,y,'r-')
>> plot(x,y,'ro')
>> plot(x,y,'r.')
>> plot(x,y,'r.','MarkerSize',2)
>> xlabel('\theta')
>> ylabel('F(\theta)')
>> title('F(\theta) = sin(5\theta)*exp(-\theta /5)')
>> grid;
```

You should see:



If you do not like the axis limits, they can be changed:

```
>> axis([-1,11,-2,2]);
```

Generating Multiple Plots

```
>> x=0:0.01:20;
>> f1=sin(x);
>> f2=cos(x)./(1+x.^2);
>> figure('Position',[100,200,400,300]);
>> plot(x,f1)
>> figure('Position',[500,200,400,300]);
>> plot(x,f2)
```

Overlaying Plots

Create and run the script below:

```

clear; close all;
% Method #1: ask for multiple plots on same plot line
x=0:0.01:20;
y=sin(x);
y2=cos(x);
figure('Position',[100,200,400,300]);
plot(x,y,'r-',x,y2,'b-')
title('Method #1')
% Method #2: use HOLD ON
figure('Position',[600,200,400,300]);
plot(x,y,'r-')
hold on;
plot(x,y2,'b-')
title('Method #2')
hold off;

```

xyz Plots: Curves in 3D Space

Create and run the script below. It generates a spiral on the surface of a sphere using spherical coordinates.

```

clear; close all;
dphi = pi/100; % set spacing of azimuthal angle
N=30; % set number of azimuthal trips
phi=0:dphi:N*2*pi;
theta = phi/N/2; % go from North to South once
r = 1; % sphere of radius 1
% convert spherical to Cartesian
x=r*sin(theta).*cos(phi);
y=r*sin(theta).*sin(phi);
z=r*cos(theta);
% plot the spiral
plot3(x,y,z,'b-');
axis equal; % so that sphere looks like a sphere
In the Tools menu, choose Rotate3D and use the cursor to rotate
the drawing. Very cool!!!!

```

Loops and Logic

For Loops

Let us do the sum

$$\sum_{n=1}^N \frac{1}{n^2}$$

for N a large integer. Create and run this script:

```

clear; close all;
s=0; % set a variable to zero - will eventually be answer

```

```

N=10000; % set the upper limit of the sum
for n=1:N % start the loop
    % add 1/n^2 to s each time and put answer back in s
    s=s+1/n^2;
end % end the loop
fprintf(' Sum = %g \n',s) % print the answer

```

Another way, which takes advantage of MATLAB functions and does not use a loop is

```

>> n=1:10000;
>> sum(1./n.^2)

```

MATLAB has a factorial(n) or gamma(x) function. Let us write a simple one using a for loop to create the appropriate product. Create and run this script:

```

p=1; % set first term in product
N=20 % set upper limit of product
for n=2:N % start loop at n=2 since p=1 already loaded
    p=p*n ; % multiply by n each time and put answer in p
end
fprintf(' N! = %g \n',p)

```

Also run factorial(20) and gamma(21).

Suppose we were solving a differential equation by power series substitution of the form

$$f(x) = \sum_{n=1}^{\infty} a_n x^n$$

and that we found the coefficient a_n satisfied the recursion relation

$$a_1 = 1 ; a_{n+1} = \frac{2n-1}{2n+1} a_n$$

Consider the following script:

```

clear;
a(1)=1; % put first element into array
N=19; % load 19 more
for n=1:N
    a(n+1)=(2*n-1)/(2*n+1)*a(n); % the recursion relation
end
disp(a') % display array of values

```

The basic **logic elements** for use in **logical conditions** are:

Equal	==
Less than	<
Greater than	>
Less than or equal	<=
Greater than or equal	>=
Not equal	~=
And	&
Or	
Not	~

If Command

Create and run this script with several examples:

```
clear;
a=1;b=3;
% if a number a is positive set c to 1
% if a is 0 or negative set c to 0
if a>0
    c=1;
else
    c=0;
end
[a,c]
% if either a or b is non-negative, add them to obtain c
% otherwise multiply a and b to obtain c
if a>=0 | b>=0 % either non-negative
    c=a+b;
else
    c=a*b; % otherwise
end
[a,b,c]
```

While Command

Create and run this script:

```
clear;
% while command is used when number of times loop needed is not
known
% stop running loop when a condition is met
% evaluate terms in a sum until the added term is less than
1x10^-10
term=1; % load first term in sum 1/1^2=1
s =term; % sum variable
n=1; %set counter to 1
while term > 1e-10 % loop until term gets too small
    n=n+1; % add 1 to n (the counter)
    term=1/n^2; % calculate next term
```

```

s=s+term; %add terms until condition is met
end
fprintf(' Sum = %g   Number of terms = %g \n',s,n)

```

In the remaining sections we raise the sophistication level.

Ordinary Differential Equations(ODEs)

Decay of a Radioactive Sample

N unstable atoms with exponential decay rate γ obey 1st-order ODE

$$\frac{dN}{dt} = -\gamma N$$

The solution is

$$N(t) = N(0)e^{-\gamma t}$$

Simple Harmonic Oscillator(SHO)

Equation of motion of a mass m bouncing in a weightless environment on a spring with spring constant k is the 2nd-order ODE:

$$\frac{d^2x}{dt^2} = -\omega_0^2 x \quad \text{where} \quad \omega_0 = \sqrt{\frac{k}{m}}$$

It has two fundamental solutions:

$$x_1(t) = \cos \omega_0 t \quad \text{and} \quad x_2(t) = \sin \omega_0 t$$

MATLAB commands use 1st-order ODEs only. We can convert higher order ODEs to system of coupled 1st-order ODEs as follows:

$$\begin{aligned} \frac{dx}{dt} &= v \\ \frac{dv}{dt} &= -\omega_0^2 x \end{aligned}$$

So we can always assume that we have a 1st-order set of ODEs. Mathematica, as we will see can give us an analytic solution in many cases, but not all. MATLAB can only give us an array which is a numerical approximation. But MATLAB will find the approximation very quickly and you have control over how it does it, so it is often more efficient to use MATLAB for difficult ODEs than to use Mathematica.

We will discuss two methods. The first is simple, intuitive, and inaccurate. The second is a little more complicated, not terribly intuitive, but very accurate. Both are crude solution

techniques. MATLAB has its own solvers which are better and we will look at later. The crude methods allow us to learn about the ideas they are based on.

Euler's Method

Although inaccurate, it is the basis for all better methods. We seek the solution of the ODE on a grid of time values (we stop thinking of time as continuous). We assume specific times t_n separated by small time steps τ . hence, instead of $x(t)$ we will find $x_n=x(t_n)$. If τ made smaller and smaller, we come closer and closer to the true solution. If we have a single 1st-order ODE as in the radioactive decay above, then the equation we are solving is (finite difference method):

$$\frac{N_{n+1} - N_n}{\tau} = -\gamma N_n \Rightarrow N_{n+1} = (1 - \gamma\tau)N_n$$

or

$$\begin{aligned} N_1 &= (1 - \gamma\tau)N_0 = (1 - \gamma\tau)N(0) \\ N_2 &= (1 - \gamma\tau)N_1 = (1 - \gamma\tau)^2 N(0) \\ &\dots\dots\dots \\ N_{n+1} &= (1 - \gamma\tau)^n N(0) \end{aligned}$$

We can solve this with the script:

```
% radioactive decay
tau=0.1;
gamma=0.5;
factor=1-gamma*tau;
N0=10.0;
steps=100;
n=0:steps;
N=N0*(factor).^n;
t=n*tau;
plot(t,N,'ro')
hold on
plot(t,10.0*exp(-gamma*t),'b-');
hold off
```

or the script:

```
% radioactive decay
tau=0.1;
gamma=0.5;
factor=1-gamma*tau;
```

```
N0=10.0;
steps=100;
N=[];
N(1)=N0;
for n=2:steps+1
    N(n)=factor*N(n-1);
end
n=0:steps;
t=n*tau;
plot(t,N,'ro')
hold on
plot(t,10.0*exp(-gamma*t),'b-');
hold off
```

Make Your Own Functions: Inline and M-Files

Inline Functions

Create and run this script:

```
clear; close all;
% inline function definition with arguments
f=inline('sin(x.*y)./(x.^2+y.^2)','x','y');
x=-8:.2:8;y=x;
figure
plot(x,f(x,2))
[X,Y]=meshgrid(x,y);
figure
surf(X,Y,f(X,Y))
% use Tools:Rotate3D
```

M-File Functions

These are subprograms stored as `func_name.m` files. A function is different than a script in that the input parameters it needs are passed to it with argument lists.

In general function variables are **local** to the function, i.e., they are not remembered after function is executed.

It is possible to pass information in and out of functions by using the **global** command to declare certain variables to be visible in all MATLAB functions and scripts in which the same global command appears.

Thus, if we put the command
 `global a b c;`

both in a main script that calls a function M-file and in the function M-file then if a, b, and c are given values in the main script, they will also have these values in the function M-file