Matlab Tutorial

Overview

This is a simple tutorial to get you started with Matlab. Matlab makes it easy to perform scientific computations without having to learn a programming language such as Fortran, C, or C++. This tutorial is developed keeping assignment_0.pdf for AM205 in mind. It can be used for other classes or self-learning as well.

Topics covered:

- Matlab Basics: start up, simple commands, Matlab as calculator
- Matlab Variables: Vectors and Matrices
- Linear Equations
- Simple programming: control of flow, functions, m-files
- Plotting

References

You should make use of online search tools such as google to search for relevant matlab material. There is a lot of material on the web. Additional resources include:

- Numerical Computing with MATLAB by Cleve Moler

Using Matlab

Opening Matlab:

This depends on your operating system. On Mac, it is usually installed in /Applications and you should see a Matlab icon with a MATLAB_<version number>. Double clicking the icon should open it. On Windows, there should either be a shortcut on the desktop or it can be found under programs in the start-up.

Matlab Development Environment (Matlab Desktop):

This can change with new releases. The following video gives a quick introduction:


The key items are "Command Window" (where you type commands in this tutorial), "Current Folder", and the "Workspace".

Getting Help

Note: In Matlab, any line beginning with % is a comment (not executed by matlab). Lines without % are commands that are executed by matlab. You can copy and paste the matlab statements in this tutorial into matlab

Open matlab and in the command window, type:

```
doc
```

This will open a separate window with documentation and a search box. There are other ways! For example, if you want to find out more about a specific command, you can do:
%help <command>
%As an example, to find out more about print:
help print
% when in doubt, try help!

Preliminaries

For space reasons, the results are not shown in the code block unless necessary.

%Matlab can be used as a calculator (with +, -, *, / indicating addition, subtraction, multiplication and division)

3*2
(3+5)*15

%Exponentiation is done with ^

5^2

% Square root operation with 'sqrt'

sqrt(25)

%Matlab already knows the meaning of some variables, eg, pi

pi
sin(pi/2)
log(pi) %natural logarithm of pi 
log10(pi) %logarithm with base 10

%you can use the format command to change the output format for numbers
%for details on format, type

help format

%example of long format:

format long
pi % displays 3.141592653589793
format short
pi % displays 3.1416

Arrays

All matlab variables are arrays. Even single numbers are arrays of dimension 1. Below, we start experimenting with some arrays.
% The array x contains numbers ranging from 0 to 1.0 in increments of 0.25.
% Note that the ";" at the end suppresses the output. Experiment by leaving it out.
x=[0,0.25,0.50,0.75,1.00];

% typing x (without semi-colon) will output the array
x

%The same array can be created with a simpler command.
%The following syntax with "::" indicates that the array starts from 0
% and is incremented by 0.25 till we reach 1.0. This is much easier for larger
% arrays (such as the array t, which contains 100 numbers).
x=[0:0.25:1.0]
t=[0:0.01:1.0]

%The following (without brackets) also creates an array.
v=0:0.01:1.0

% The function 'linspace' can be used to create an array of 100 equally spaced elements.
v=linspace(0,1.0); note that the default is 100 equally spaced points between 0 and 1.

% To create an equally spaced array between 1 and 1.0 with 5 points
x=linspace(0,1.0,5); the third argument specifies the number of points.

% Sum of elements of an array
sum(x)

% Note we already have created an array t before! We create a second array:
x=2*pi*t; This multiplies each element of t by the scalar 2*pi.

% y will be an array consisting of sin of the elements of the array x
y=sin(x);

Element-wise operations on an array
Sometimes we need to carry out operations on individual elements of an array. Let us start with an array \( x \)

\[
x = [0, 0.25, 0.50, 0.75, 1.00];
\]

We would like to square each element of the array. In mathlab, this can be done with:

\[
x.^2  \quad \text{Notice the dot (.) in front of exponentiation (^).}
\]

This returns:

\[
\text{ans} = \\
\begin{array}{cccc}
0 & 0.0625 & 0.2500 & 0.5625 & 1.0000 \\
\end{array}
\]

Element-wise operations work on Matrices too!

Quick plotting

% We have already created arrays x and y. We can now plot y as a function of x.
plot(x, y)
xlabel('angle'); % will add x-label
ylabel('Amplitude'); % will add y-label
title('Plot of sin(\Theta)') % adds a title. Note \Theta is from the Latex typesetting program.

More on plotting later.

Vectors

v1 = [1.5, -2, 4, 10]; % A 1-dimensional array can be handled as a vector.

Example – Dot product, angle between vectors:
v2=[3.1,-1,2,2.5]; %Another 1-dimensional array

% The two arrays v1 and v2 are basically vectors with all the vector operations defined.

dot(v1,v2); % dot product of two vectors

% Angle between the vectors v1 and v2
% To compute the angle between two vectors, we need the magnitude of the two vectors
abs_v1=sqrt(sum(v1.^2)) % Notice we are performing element-wise exponentiation (i.e. .^2)
abs_v2=sqrt(sum(v2.^2))
costheta=dot(v1,v2)/(abs_v1*abs_v2); cosine of the angle between the vectors

% An easier way: instead of computing the magnitude yourself, you can use the norm function
costheta=dot(v1,v2)/(norm(v1)*norm(v2));

% To find out more about norm, do:
help norm % warning: the response is detailed and technical!
theta=acos(costheta); % Now compute the angle (in radians) between the vectors.

% convert radians to degrees
radtodeg(theta)

% convert degrees to radians
radtodeg(0.6678)

(compare)
use:
format long

Matrices

% A matrix consisting of two rows and 6 columns is defined
% like an array except that the two rows are separated by ";
% Notice how the semicolon symbol (;) can take on different meanings depending
% on the context. Below you see it used to define a matrix and to suppress the output.

x=[1.0,1.2,1.4,1.6,1.8,2.0;2.0,2.2,2.4,2.6,2.8,3.0];

x=
   1.0000    1.2000    1.4000    1.6000    1.8000    2.0000
   2.0000    2.2000    2.4000    2.6000    2.8000    3.0000

x=[1:0.2:2;2:0.2:3]
x =
% output:  1.0000  1.2000  1.4000  1.6000  1.8000  2.0000
%  2.0000  2.2000  2.4000  2.6000  2.8000  3.0000

% Transpose of a matrix is obtained by superscripting a matrix with the prime character, i.e. "'":

y=x' % transpose of x

% We extract the last four columns of the second row by "slicing":
% x(2,[3:6])

%ans =
% 2.4000  2.6000  2.8000  3.0000

% Random matrix, identity matrix, etc.:

a=rand(2)

%ans =
% 0.8147  0.1270
% 0.9058  0.9134
% Note: you won't get the same matrix! why? hint is in the name.

% A 3X3 Identity matrix:

b=eye(3)

%output:
% 1 0 0
% 0 1 0
% 0 0 1

% Determinant of a matrix obtained with det function (defined only for square matrices)

u=[5,11,-3;4,-5,0;7,8,-9]

det(u)

% the answer should be 420.0

% Inverse of a matrix obtained with inv function.

inv(u)

% produces

% 0.1071  0.1786 -0.0357
% 0.0857 -0.0571 -0.0286
% 0.1595  0.0881 -0.1643

% product of a matrix with its determinant should ideally be an identity matrix.
% In practice, it will be close but may not be exact.

u*inv(u)

% produces

% 1.0000  0  0.0000
% 0.0000  1.0000  -0.0000
Matrix example (Linear Equations)

We now solve the linear equation:

\[2x_1+4x_2+3x_3=4\]
\[x_1-2x_2-2x_3=0\]
\[-3x_1+3x_2+2x_3=-7\]

\[Ax=B\]

\[
A=\begin{bmatrix}
2 & 4 & 3 \\
1 & -2 & -2 \\
-3 & 3 & 2
\end{bmatrix};
\]
\[
B=\begin{bmatrix}
4 \\
0 \\
-7
\end{bmatrix};
\]

Solution \(x=\text{inv}(A)B\)

\(x=\text{inv}(A)*B\)

% Use of the backslash operator (\) is a more efficient way:

\(x=A\backslash B\)

For loops

For loop:
% Let us consider the Taylor series for the sin function
% The series is summation over
% n (from 1 to k) of (-1)^(n-1)x^(2n-1)/(2n-1)!
% For an arbitrary number of terms k, how do we evaluate sin(x) for a given
% value of x?
% The task is one of carrying out summation of k terms. This is
% accomplished via the "for" loop. The syntax is:
% for i=1:k
% ....
% (do some operations that depend on i)
% ......
% end
% For the sin series, here is one way to implement the summation:
% salfunc=0; % initialize the sin function
% for i=1:k
% km1=2*i-1;
% kp1=i-1;
% kcoef=(-1)^kp1;
% salfunc=salfunc+kcoef*x.^km1/factorial(km1); %This is the summation step. On the
% right hand side, The k'th term
% of the series is evaluated and added
% to the previously stored value
% of sin function. Then salfunc (left
% hand side) is replaced with the
% new value on the right hand side.
% end
% salfunc % salfunc contains the approximation of sin(x)
% err=sin(x)-val; % you can easily calculate the error

We will later consider the control of flow within a loop under functions.

More advanced topics

Plotting revisited

complex plots (plot annotations, axis labels, etc.)

x = -pi:.1:pi;
y = sin(x);
p = plot(x,y)

% gca returns the handle to the current axes for the current figure.
set(gca,'XTick',-pi:pi/2:pi)
set(gca,'XTickLabel',{-pi,'-pi/2','0','pi/2','pi'})

% \pi, \leq, \Theta are all from "latex" typesetting system.
xlabel('-\pi \leq \Theta \leq \pi')
ylabel('sin(\Theta)')
title('Plot of sin(\Theta)')

% \Theta appears as a Greek symbol (see String)
% Annotate the point (-pi/4, sin(-pi/4))
text(-pi/4,sin(-pi/4),'\leftarrow sin(-\pi/4)',... % Notice the line continuation
    'HorizontalAlignment','left')

% Change the line color to red and
% set the line width to 2 points
set(p,'Color','red','LineWidth',2)

Some 3D plotting

Meshgrid:
If we have two arrays that define two orthogonal axes, MATLAB makes it easy to produce a grid from the axes to form the basis for plotting. This is done with the `meshgrid` command. Let's define the two axes and see how the `meshgrid` command works.

```matlab
x=[-8:4:8]; % an array of points on the x-axis [-8 -4 0 4 8]
y=[-8:4:8]; % an array of points on the y-axis [-8 -4 0 4 8]
[X Y]=meshgrid(x,y) % returns two matrices X and Y:
```

% Note (X[j],Y[j]) define points in the 2D space defined by the x and y axes.
% Thus, if we have a function defined on x and y, i.e. f(x,y),
% evaluation of the function over all the ordered pairs of points
% in x and y can now use the whole matrices X and Y instead of looping
% over all the points. See below for an example of how this is done in the
% context of a 3D plot below.

---

**Sinc function** ([https://en.wikipedia.org/wiki/Sinc_function](https://en.wikipedia.org/wiki/Sinc_function))

```matlab
% Plot of 3D sinc function.
x=[-8:.5:8];
y=[-8:.5:8];
% Look up "meshgrid" in MATLAB document
% Returns rectangular grid
[X Y]=meshgrid(x,y);
R = sqrt(X.^2 + Y.^2); % Notice how element-wise operation is used.
Z = sin(R)./R; % Sinc function
surf(Z); % surface plot
mesh(Z); % mesh plot
contour(Z); % contour plot
```

---

**Another example:**

```matlab
[X,Y] = meshgrid(-2:.2:2, -2:.2:2);
2 = X .* exp(-X.^2 - Y.^2);
surf(X,Y,Z)
```

---

**m-FILES (SCRIPTS AND FUNCTIONS)**
Scripts:
A collection of matlab commands in a file named `<name>.m`.
Example: `example.m`

```
p=[0:0.01:100];
q=2*pi*p;
r=sin(q);
```

To run the script, save the script file (say `example.m`) in a directory accessible by matlab and just type the script name (without `.m`) in the command window. In this case it will be 'example'.

Functions:
Matlab functions are callable sub-procedures that are put in a file named `<function_name>.m`

Example function (stat.m from matlab documentation) is as follows:

```matlab
function [mean, stdev] = stat(x)
% x, which is an array of numbers, is the input
% mean and stdev are the return values (outputs)
% n = length(x);
mean = sum(x)/n;
stdev = sqrt(sum((x-mean).^2/n));
```

Put the above function in a file called stat.m in a directory which is in matlab's path. You can use this function as below:

```
x=[1.0,1.2,1.4,1.6,1.8,2.0]

[mean stdev]=stat(x)

%returns
%mean=1.50
%stdev=0.3416
```

More functions
The following are some functions relevant to `assignment_0.pdf` of AM205.

They are meant to get you started with the problems, but they are not “complete solutions”. They can be useful for learning functions in matlab for other classes as well.

Function (cheby.m):
function [m,val] = cheby(k,x)
% Chebyshev polynomial recurrent relation
% Relevant to question 2 in assignment_0
% Note: x and k are not vectors.
if k==0  % if ... elseif.. block controls the flow
% based on the value of k in this example.
    m=0;
    val=1.0;
elseif k==1
    m=1;
    val=x;
elseif k>=2
    tch(1)=1;
    tch(2)=x;
    for i=3:k+1
        tch(i)=2*x*tch(i-1)-tch(i-2);
    end
    m=i-1;
    val=tch(i);
end

Function: sinTaylorSeries.m

function [sinfunc,err] = sinTaylorSeries(k,x)
% The Taylor series for the sin function
% The series is summation over
% n (from 1 to k) of (-1)^(n-1)x^(2n-1)/(2n-1)!
% Relevant to question 5 of assignment_0.
% Note x is a vector.
sinfunc=0;
for i=1:k
    km1=2*i-1;
    kp1=i-1;
    kcoef=(-1)^kp1;
    sinfunc=sinfunc+kcoef*x.^km1/factorial(km1);
end
val=sinfunc;
err=sin(x)-val;

Function: tanderiv.m

function [deriv,act,err] = tanderiv(x,h)
% Derivative of tan: (tan(x+h)-tan(x-h))/2h
% relevant to question 4 in assignment_0.
% Note x and h are not vectors.
y1=tan(x+h);
y2=tan(x-h);
deriv=(y1-y2)/(2*h);
act=sec(x)*sec(x);
err=abs(deriv-act)/act;

Image Processing (Quick Introduction)
Here are the images we will be using: baboon.png and Brain-MRI.jpg

%%The primary functions for loading images is 'imread' and for displaying images is 'imshow'.

im=imread('baboon.png');
imshow(im);

%% Images are basically two dimensional array of pixels (picture elements) with a third dimension
%% that shows the intensity of pixel. If the image is a gray scale image,, the intensity is a single number
%% at each pixel. If it is a color image, it is stored as a "third dimension" with 3 value
%% each corresponding to the intensity of r (red), g (green) and b (blue) colors. Thus if we do:

size(im)

%% The result is 512 512 3, which indicates it is a two dimensional array of 512 x 512 pixels with three colors.
%% To get the 'red' intensity, you can do:

r=im(:, :, 1);

%% This is now a two dimensional array. The color intensity can vary anywhere from 8 bits (256 levels)
%% for grayscale to 24 bits (8 bits for each color) for color images. There are even 30-, 36-, or 48- bit
%% images. As far as Matlab is concerned,. the image once loaded is a three dimensional array with the last
%% dimension taking on only 3 values.

max(r(:)),min(r(:))

%% will show 255 and 0 (the red color is stored in 8 bits).

imshow(r)

%% will show the image in grayscale with intensity corresponding to "red" color intensity in the
%% original image. You can convert the color image to a grayscale image:

img=rgb2gray(im);
imshow(img)

This is now a two dimensional image with a grayscale intensity of 8 bits. We have compressed the image at the expense of color!

Grayscale Images:
Many medical images (such as MRI, CT, ultrasound, etc.) are likely to be grayscale images. They can be 10 or 12 bit images and it is convenient to represent them using 16 bits as computers can 16 bits efficiently.

```matlab
A = imread('Brain-MRI.jpg');
size(A)
imshow(A)
```

%% We can convert this to an grayscale image with:
```
B = rgb2gray(A)
```

%% This will be a 2D array of pixels with gray shades represented by 8 bits (256 shades).
%% We can plot a histogram of color intensity (x-axis intensity, y axis -- number of pixels) with:
```
imhist(B)
```
%% We can reduce the variation in the intensity among the pixels (and thereby visually enhance the darker regions) by using the matlab function histeq.
```
histeq(B)
```
%% Will display the image with formerly darker regions more bright and brighter regions darker.

**Additional tutorial material:** fwdmatlabtutorialsetc.zip