# Contents

# 1 MATLAB Basics

## 1.1 Example: Plotting a function

Sample MATLAB code illustrating several Matlab features:
code to plot the graph of $y = \sin(2\pi x)$, $x \in [0, 1]$.

What is really going on when you use software to graph a function?

1. The function is sampled at a set of points $x_k$ to obtain $y_k = f(x_k)$.
2. The points $(x_k, y_k)$ are then plotted together with some interpolant of the data (piecewise linear or a smoother curve such as splines of Bezier curves).

In MATLAB you specify $x_k$, then compute $y_k$. The command `plot(x,y)` outputs a piecewise linear interpolant of the data.

```
% Set up gridpoints x(k)
x=[0.0, 0.1,0.2,0.3,0.4...
   0.5,0.6,0.7,0.8,0.9,1.0];
% Set up function values y(k)
n=length(x);
y=zeros(1,n);
for k=1:n
  y(k)=sin(2*pi*x(k));  %pi is predefined
end
% plot a piecewise linear interpolant to the points (x(k),y(k))
plot(x,y)
```

Notes:

(1) The % sign denotes the begining of a comment. Code is well commented!

(2) The continuation symbol is ...

(3) The semicolon prevents displaying intermediate results (try it, what happens if you omit it?)

(4) `length,zeros,sin,plot` are built in MATLAB functions. Later on we will write our own functions.

(5) In Matlab variables are defined when they are used. Reason for `y=zeros(1,n)`: allocate amount of memory for variable $y$; initialize. How much memory space is allocated to $y$ if that line is absent, as you step through the loop? Why is `zeros` not used before defining $x$?

(6) In Matlab all variables are matrices. Column vectors are nx1, row vectors are 1xn, scalars are 1x1 matrices. What is output of `size(x)`?

(7) All vectors/matrices are indexed starting with 1. What is `x(1), x(2), x(10), x(0)`?

(8) Square brackets are used to define vectors. Round brackets are used to access entries in vectors.

(9) Note syntax of `for` loop.

(10) What is the output of

```
plot(x,y,'*')
plot(x,y,'*-')
plot(x,y,'*-r')
```

See Tutorial for further plotting options, or `help plot`. Lets create a second vector $z = cos(2\pi x)$ and plot both $y$ vs x in red dashed curve with circle markers, and $z$ in green solid curve with crosses as markers. Then use default. But first:

## 1.2 Scripts

MATLAB Script m-file: A file (appended with .m) stored in a directory containing MATLAB code. From now on I will write all code in scripts so I can easily modify it. To write above code in script:

```
make a folder with your name
if appropriate, make a folder in that folder for current howework/topic
in MATLAB, go to box next to "current directory" and find directory
click on "File/New/M-file", edit, name, save
or click on existing file to edit
execute script by typing its name in MATLAB command window
lets write a script containing above code
```

Save your work on Floppy or USBport, or ftp to other machine. No guarantees that your folders will be saved.

Now I'd like to modify this code and use more points to obtain a better plot by changing the line defining x, using spacing of 1/100 instead of 1/10. But how to set up x with 101 entries??

## 1.3 Setting up vectors

### Row Vectors

- Explicit list

  ```
  x=[0 1 2 3 4 5];
  x=[0,1,2,3,4,5];
  ```

- Using a:increment:b. To set up a vector from x=a to x=b in increments of size h you can use

  ```
  x=a:h:b;
  ```

  Here you specify beginning and endpoints, and stepsize. Most natural to me. However, if (b-a) is not an integer multiple of stepsize, endpoint will be omitted. Note, if h omitted, default stepsize =1. What is?

  ```
  x=0:0.1:1;
  x=0:5;
  ```

  We already used this notation in the for loop!

- Using linspace

  ```
  x=linspace(a,b,n);
  ```

  Use linspace to set up `x=[0 1 2 3 4]`, `x=[0,0.1,0.2,...,1]`, `x=[0,0.5,1,1.5,2]`, `x=a:h:b`

- Using for loops

### Column Vectors

- Explicit list

  ```
  x=[0;1;2;3;4;5];
  ```

- Transpose row vector

```
        x=[0:.1:1]';
```

**Matrices**

```
A=[1 1 1; 2 0 -1; 3 -1 2; 0 1 -1];
A=zeros(1,4); B=zeros(5,2); C=eye(3); D=ones(2,4);   %special matrices
```

How to access entry in second row, first column? What is A(1,1), A(2,0)?

Now we have a better way to define $x$ using $h = 1/100$! Do it.

Try `x-y` where x row, y column!!

## 1.4 Vector and Matrix operations

We can replace the `for` loop in the example in §1.1 by

```
y=sin(2*pi*x);
```

The sine function is applied to a vector $x$ and applies the sine operation to each entry in the vector. This is the same as

```
y(1:n)=sin(2*pi*x(1:n));
```

and thereby the same as (!)

```
k=1:n
y(k)=sin(2*pi*x(k));
```

Note that this looks almost like the `for` loop in the example in §1.1 but it is not a loop. All entries of `y` are set at once. We can now give a short version of the MATLAB code to plot the graph of $y = \sin(2\pi x)$:

```
  x=0:.01:1;
  y=sin(2*pi*x);
  plot(x,y)
```

Since vectors are special cases of matrices, the above operation can also be applied to matrices. The statement `B=sin(A)` applies the sine function to every entry in A.

**Setting up matrices.** Before further addressing matrix operations, I want to mention another possibility to set up matrices. First note that the line `A=[1 1 1; 2 0 -1; 3 -1 2; 0 1 -1];` in §1.3 can be written as

```
A=[1 1 1
   2 0 -1
   3 -1 2
   0 1 -1
  ];
```

(the commas and semicolons are not necessary in this form). If you create, possibly using FORTRAN or C, a file that contains a matrix of numbers, and enter `A=[` and `];` before and after, you can read these numbers in as a script. For example, I created (in FORTRAN) a file called t22dat.m that contains the following lines

```
%    t        xcore     ycore     ylmb     ulmb     uf        xf       jctr  n
a=[
 0.00000  0.000000  0.924700  0.81650  0.16004  0.590545  0.000000    2  400
 0.02500  0.002659  0.924680  0.81650  0.16013  0.590451  0.014763   93  400
 0.05000  0.005320  0.924620  0.81650  0.16039  0.590168  0.029521   92  400
 0.07500  0.007981  0.924520  0.81650  0.16081  0.589698  0.044270   93  400
 0.10000  0.010643  0.924380  0.81650  0.16141  0.589041  0.059004   92  400
 0.12500  0.013301  0.924190  0.81650  0.16216  0.588201  0.073720   93  400

...

59.90000 12.920807  0.822010  0.81665  0.21526  0.216202 13.726140   73 3761
59.92500 12.926320  0.822030  0.81665  0.21526  0.216193 13.731545   72 3764
59.95000 12.931837  0.822050  0.81665  0.21526  0.216184 13.736949   73 3768
59.97500 12.937345  0.822080  0.81665  0.21526  0.216175 13.742354   72 3772
];
```

(where the dots denote the remaining 2390 lines). I can now read this matrix into MATLAB and extract the vectors `t` and `ylmb` for example as follows

```
t22dat          %this executes all 2402 lines in the script
t=a(:,1)        %extract vector t, note colon notation
ylmb=a(:,4)     %extract vector ylmb
plot(t,ylmb)    %plot one vs the other
```

Note: entries of vectors are accessed using round brackets. Vectors are defined using square brackets.

**More matrix operations.**

```
    A+B                  %A,B need to have same dimensions
    A*B                  %A,B need to have proper dimensions (number
                         %    of columns of A=number of rows of B)
```

If `x` is a vector, what is

```
        x*x
```

? Answer: error!, Inner matrix dimensions dont agree. If `x` and `y` are 1xn row vectors,

```
        x*y'
```

Answer: the **inner product** $\sum_{k=1}^{N} x_k y_k$. For example to compute the Euclidean norm of $x$, $\sqrt{\sum_{k=1}^{N} x_k^2}$ you can use the one-line code

```
        euclidnormx=sqrt(x*x');
```

What is

```
        y'*x
```

Answer: an nxn matrix called the **outer product**.

What if you instead of plotting $y = sin(x)$ you want to plot $y = x^2$? Given a vector x you want to create a vector y whose entries are the square of the entries of x. The following

```
        x=0:.01:1;
        y=x*x;
or
        y=x^2;     %the hat is MATLABs symbol for exponentiation
```

wont work. Instead, to perform componentwise operations you need to replace * by .*, $\hat{}$by .$\hat{}$, etc. for example:

```
y=x.^2;
y=A.^2;
y=1./x;
y=x.*z;    %where x,z are vectors/matrices of same length
```

Another useful built in MATLAB function is

```
s=sum(x);
```

Use the help command to figure out what it does.

## 1.5 Plotting

**Labelling plots.** In class we used:

```
plot(x,y,'r:x')        %options for color/line type/marker type
xlabel('this is xlabel')
ylabel('this is ylabel \alpha \pi')  %using greek alphabet
title('this is title')
text(0.1,0.3,'some text')        %places text at given coordinates
text(0.1,0.3,'some text','FontSize',20)    %optional override fontsize
set(gca,'FontSize',20)      %override default fontsize
axis([0,1,-2,2])          %sets axis
axis square               %square window
axis equal                %units on x- and y-axis have same length
figure(2)                 %creates new figure or accesses existing figure(2)
```

**Plotting several functions on one plot.** Suppose we created `x=0:.1:1; y1=sin(x); y2=cos(x);`. Two options. First one:

```
plot(x,y1,x,y2)                 %using default colors/lines/markers
plot(x,y1,'b--O',x,y2,'r-')   %customizing
legend('sin(x)','cos(x)',3)   %nice way to label, what does entry '3' do?
```

Second one: using hold command:

```
plot(x,y1)
hold on
plot(x,y2)
hold off
```

Type `help hold` to see what this does. If you dont want to save the previous plot you can view consecutive plots using pause command in your script (what happens if pause is missing from below?)

```
plot(x,y1)
pause
plot(x,y2)
```

**Creating several plots.**

```
subplot(3,2,1)    %creates and accesses 1st subplot of a 3x2 grid
figure(3)         %creates and accesses new window
```

## 1.6 Printing data

Printing tables:

```
disp(' j   x  sin(x)')
for k=1:n
   disp(sprintf('%4d %5.1f %10.4f',k,x(k),y(k)));
end
```

What does the format `%3.1f` specify? Type `help sprintf` to see how to format integers, character strings, etc.

Printing figure:

```
print              %prints current figure to printer
print -deps name   %creates name.eps file (encapsulated postscript)
                   % of current figure and stores in directory
print -depsc name  %creates name.eps file in color of current figure
print -dpsc name   %creates name.ps (postscript) file in color
```

## 1.7 For loops

```
% The command for repeats statements for a specific number of times.
% The general form of the while statement is
```

```
FOR variable=expr
   statements
END

% expr is often of the form i0:j0 or i0:l:j0.
% Negative steps l are allowed.
% Example : What does this code do?

n = 10;
for i=1:n
   for j=1:n
      a(i,j) = 1/(i+j-1);
   end
end
```

## 1.8 While loops

```
% The command while repeats statements an indefinite number of times,
% as long as a given expression is true.
% The general form of the while statement is

WHILE expression
   statement
END

% Example 1: What does this code do?
x = 4;
y = 1;
n = 1;
while n<= 10;
   y = y + x^n/factorial(n);
   n = n+1;
end

% Remember to initialize $n$ and update its value in the loop!
```

## 1.9 Timing code

```
tic   % starts stopwatch
  statements
toc   % reads stopwatch
```

Exercise: Compare the following runtimes. What do you deduce?

```
tic; clear, for j=1:10000, x(j)=sin(j); end, toc
tic; clear, j=1:10000; x(j)=sin(j); toc
tic; for j=1:10000, x(j)=sin(j); end, toc
```

Exercise: Compare the following runtimes. What do you deduce?

```
clear
tic; for j=1:10000, sin(1.e8); end, toc

format long, pi2=2*pi; 1.e8/pi2
clear, alf = 1.e8-1.5915494e7;
tic; for j=1:10000, sin(alf); end, toc

clear
tic; for j=1:10000, sin(0.1); end, toc
```

## 1.10 Functions

MATLAB Functions are similar to functions in Fortran or C. They enable us to write the code more efficiently, and in a more readable manner.

The code for a MATLAB function must be placed in a separate `.m` file having the same name as the function. The general structure for the function is

```
function <Output parameters>=<Name of Function><Input Parameters>
% Comments that completely specify the function

<function body>
```

When writing a function, the following rules must be followed:

- Somewhere in the function body the desired value must be assigned to the output variable!
- Comments that completely specify the function should be given immediately after the `function` statement. The specification should describe the output and detail all input value assumptions.
- The lead block of comments after the `function` statement is displayed when the function is probed using `help`.
- All variables inside the function are local and are not part of the MATLAB workspace

Exercise 1: Write a function with input parameters $x$ and $n$ that evaluates the $n$th order Taylor approximation of $e^x$. Write a script that calls the function for various values of $n$ and plots the error in the approximation.

Solution: The following code is written in a file called `ApproxExp.m`:

```
function y=ApproxExp(x,n);
% Output parameter: y (nth order Taylor approximation of $e^x$)
% Input parameters: x (scalar)
%                   n (integer)

sumo = 1;
for k=1:n
   sumo = sumo + x^k/factorial(k);
end
y = sumo;
```

(What does this code do? First, set k=1. Then k=2. Then k=3. etc. Write out the result after each time through loop.) A script that references the above function and plots approximation error is:

```
x=4;
for n=1:10
   z(n) =ApproxExp(x,n)
end
exact=exp(4)
plot(abs(exact-z))
```

Exercise 2: Do the same as Exercises 1, but let `x` and `y` be vectors.

Example: An example of a function that outputs more than one variable. The function computes the approximate derivative of function fname, the error in the approximation, and the estimated error. The following code is written in a file called `MyDeriv.m`:

```
function [d,err]=MyDeriv(f,fp,a,h);
% Output parameter: d (approximate derivative using
%                      finite difference (f(h+h)-f(a))/h)
%                   err (approximation error)
% Input parameters: f  (name of function)
%                   fp (name of derivative function)
%                   a (point at which derivative approx)
%                   h (stepsize)

d = ( f(a+h)-f(a) )/h;
err = abs(d-fp(a));
```

(Note: this works using MATLAB versions 7.0 but not 6.1. For an older MATLAB version you may have to use the feval command. See help, Daishu or me.)

A script that references the above function and plots the approximation error:

```
a=1;
h=logspace(-1,-16,16);
n=length(h);
for i=1:n
   [d(i),err(i)]=MyDeriv(@sin,@cos,a,h(i));
end
loglog(h,err)
```

Exercise: What happens if you call

```
      d=MyDeriv(@sin,@cos,a,h)
or simply
      MyDeriv(@sin,@cos,a,h)
```

You can replace `sin` and `cos` by user defined functions, for example `'f1'` and `'df1'`. Do it. That is, you need to write the function that evaluates f1 and df1 at x (in files f1.m and df1.m).

## 1.11 Anonymous Functions

An anonymous function is a function that is not stored in a program file, but is associated with a variable whose data type is function_handle. Anonymous functions can accept inputs and return outputs, just as standard functions do. However, they can contain only a single executable statement.

For example, create a handle to an anonymous function that finds the square of a number:

```
sqr = @(x) x.^2;
```

Variable sqr is a function handle. The @ operator creates the handle, and the parentheses () immediately after the @ operator include the function input arguments. This anonymous function accepts a single input x, and implicitly returns a single output, an array the same size as x that contains the squared values.

Find the square of a particular value (5) by passing the value to the function handle, just as you would pass an input argument to a standard function.

```
a = sqr(5)
```

Variables in the Expression

Function handles can store not only an expression, but also variables that the expression requires for evaluation.

For example, create a function handle to an anonymous function that requires coefficients a, b, and c.

```
a = 1.3;
b = .2;
c = 30;
parabola = @(x) a*x.^2 + b*x + c;
```

Because a, b, and c are available at the time you create parabola, the function handle includes those values. The values persist within the function handle even if you clear the variables:

```
clear a b c
x = 1;
y = parabola(x)
y =
   31.5000
```