Short Introduction to Matlab & First Experiment

Subject *Chaos & Fractals*

Kateřina Staňková

September 12, 2012

1 Basics

In Matlab, every variable is stored as a matrix. The following commands show you how to assign numbers, vectors, and matrices to variables. The Matlab prompt is >>. The commands below are typed in after the prompt, and concluded with a carriage return (enter). The response of Matlab appears on the next line. Please try typing these commands as you read this document.

```
>> a=5

a =

    5

>> v=[1;4;0]

v =

     1
     4
     0

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

A =

     1     2     3
     4     5     6
     7     8     9


>> v'
```

```
ans =

     1     4     0

>> A'

ans =

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

Notice that the rows of a matrix are separated by semicolons, while the entries on any given row are separated by commas (spaces may also be used). As mentioned above, each of variables `a`, `v`, and `A` is regarded as a matrix; the scalar a is a 1×1 matrix, the vector v is a 3×1 matrix, and A is a 3×3 matrix.

The size of a matrix can be found using the function `size(A)`, i.e.,

```
>> size(A)

ans =

     3     3

>> size(v)

ans =

     3     1

>> size(a)

ans =

     1     1
```

The length of a row or a column vector can be found using the function `length`. Thus:

```
>> length(v)

ans =

     3

>> length([1 2 3 4])

ans =
```

```
     4
```

You can also display elements of a matrix/vector:

```
>> A(2,3)

ans =

     6

>> v(1)

ans =

     1

>> v(2)

ans =

     4

>> v(3)

ans =

     0

>> a(1)

ans =

     5
```

Indices have to be positive integers. Thus:

```
>> a(0)
??? Subscript indices must either be real positive integers or logicals.
```

If you want to prevent Matlab from displaying what you entered, you use a semicolon at the end:

```
>> x=[1,2,3];
```

Matlab does not respond because you ended the line with semicolon, however it "remembers" your definition of `x`.

The following examples demonstrate how complex units are displayed in Matlab. They also show that the square root function is a built-in feature:

```
>> sqrt(-1)

ans =

        0 + 1.0000i
```

The variable `ans` contains the result of the most recent computation which can then be used as an ordinary variable in subsequent computations (notice also built-in functions `real` and `imag`):

```
>> 2+5-489

ans =

  -482

>> sqrt(ans)

ans =

        0 +21.9545i

>> real(ans)

ans =

     0

>> imag(ans)

ans =

     0

>> sqrt(-482)

ans =

        0 +21.9545i

>> imag(ans)

ans =

   21.9545
```

Another built-in variable that is often useful is π :

```
>> pi

ans =

    3.1416
```

Only a few digits of π are displayed (try typing `format long` and then typing `pi` to see more digits). More significantly, only a finite number of digits of π are known to Matlab. This is because Matlab only deals with approximate arithmetics of real numbers. In particular, numbers smaller than a certain size cannot be represented by Matlab. This minimum size is stored in a built-in Matlab variable called `eps`:

```
>> eps

ans =

  2.2204e-016
```

Besides the square root function, many other common functions are predefined. They include:

`abs` – absolute value

`angle` – phase angle of a complex number in radians (type `help angle` for details)

`real`, `imag` – real part, imaginary part of complex numbers

`conj` – complex conjugation

`round` – rounds to the nearest integer

`fix` – rounds to the nearest integer towards zero (thus `fix(3.4)` returns 3, while `fix(-3.4)` returns -3)

`floor` – rounds to the nearest integer towards $-\infty$

`ceil` – rounds to the nearest integer towards $+\infty$

`sign` – signum function

`rem` – remainder (needs two input variables, type `help rem` for more details)

`sin`, `cos`, `tan` – usual trigonometric functions

`asin`, `acos`, `atan` –usual inverse trigonometric functions

Some examples:

```
>> cos(2)^2+sin(2)^2

ans =
```

```
     1

>> exp(1)

ans =

    2.7183

>> log(ans)

ans =

     1
```

Matlab has a comprehensive online help system which includes a list of built-in special functions and routines, as well as a list of other commands on which help is available. To obtain the list just type `help`. To get help on a particular command, type `help` followed by the command. An equally useful way to get help is to use the Matlab Help window, which is accessed by going to the Help menu at the top of the screen. The Help window allows you to search the Matlab manuals for information on the topic of your choice. For example:

```
>> help exp
 EXP    Exponential.
    EXP(X) is the exponential of the elements of X, e to the X.
    For complex Z=X+i*Y, EXP(Z) = EXP(X)*(COS(Y)+i*SIN(Y)).

    See also expm1, log, log10, expm, expint.

    Overloaded methods:
       sym/exp
       zpk/exp
       tf/exp
       codistributed/exp

    Reference page in Help browser
       doc exp
```

Another command is `help help` which describes how to find help. To view a few demonstrations, try typing `demo` in the Command Window. The demonstrations can also be accessed from the `Help` window.

2 Matrix operations

Using Matlab we can perform standard arithmetic operations on matrices: addition, subtraction, and multiplication, as well as more advanced computations: finding row

echelon form, finding eigenvalues and eigenvectors of a matrix and much more. Some of these latter operations are useful in studying systems of differential equations.

2.1 Matrix arithmetics

If `A` and `B` are matrices, then Matlab can compute the sum, difference, and the product of these two matrices (when these operations are well-defined). To do this, it is enough to type `A+B`, `A-B`, and `A*B`, respectively. Recall that order is important in matrix multiplication:

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

A =

     1     2
     3     4

>> B=[0,1;2,0]

B =

     0     1
     2     0

>> A*B

ans =

     4     1
     8     3

>> B*A

ans =

     3     4
     2     4
```

If A is a square $(n \times n)$ matrix, then typing A^2 yields the matrix product $A \times A$. In general typing A^m gives the m-fold product $\underbrace{A \cdot A \cdot \ldots \cdot A}_{\text{m times}}$.

Generally, applying to a matrix A any of the built-in functions returns a matrix of the same dimensions containing the values of the function as if it had been applied on each of its element:

```
>> A=[pi, pi/2; pi*3, 3*pi/2]
```

```
A =

    3.1416    1.5708
    9.4248    4.7124

>> sin(A)

ans =

    0.0000    1.0000
    0.0000   -1.0000
```

We say that built-in Matlab functions are *vectorized*, indicating that they can be effortlessly applied to large vectors or matrices without the need to write complicated loops to cycle through the indices, as in some lower-level programming languages. If you want Matlab to multiply two matrices `A` and `B` that have the same dimension in a elementwise fashion (rather than the usual matrix multiplication) you should use the operator `.*` rather than `*`. Similarly, componentwise division of two matrices of the same dimensions can be accomplished by writing `A./B` which creates a matrix whose entries are `A(i,j)/B(i,j)`. Matlab also contains several commands to make it easy to input certain standard types of matrices. For example, the built-in function `zeros(m,n)` returns an $m \times n$ matrix of zeros. Similarly, `ones(m,n)` returns an $m \times n$ matrix of ones. If `v` is a vector of n components, then the function `diag(v)` returns a square $n \times n$ diagonal matrix whose entries are all zero with the exception of those running down the diagonal, which contains the corresponding entries of the vector `v`. The case when `v=ones(1,n)` is especially important because this gives a diagonal matrix with all ones down the diagonal; this is the $n \times n$ identity matrix. Matlab provides a special built-in function in this case: `eye(n)` ("eye" sounds like the letter I, which is usually used to denote the identity matrix).

In the special case of a matrix with only one row, we have a row vector instead of a matrix, and Matlab provides some specialized ways to input certain types of row vectors that occur frequently in practice. In Matlab, the notation `lo:step:hi` where `lo`, `step`, and `hi` are numbers, denotes a row vector whose first entry is `lo` and whose consecutive entries differ by `step`, and whose last entry does not exceed `hi`. For example,

```
>> x=0:1:10

x =

     0     1     2     3     4     5     6     7     8     9    10

>> y=0.1:0.4:2.1

y =

    0.1000    0.5000    0.9000    1.3000    1.7000    2.1000
```

If the parameter `step` is omitted, Matlab assumes that it is equal to one:

```
>> y=0.1:5.1

y =

    0.1000    1.1000    2.1000    3.1000    4.1000    5.1000

>> y=1:4

y =

     1     2     3     4

>> y=1:4.2

y =

     1     2     3     4
```

2.2 More advanced operations

Matlab can be used to do more advanced operations on matrices. For example, if A is a square matrix, then by typing `det(A)` Matlab computes its determinant.

In solving systems of linear equations, computing both the row echelon form of a matrix and finding the eigenvalues and eigenvectors of the matrix are very important tools. Let's see how to do this using Matlab. If A is a matrix then typing

```
>>rref(A)
```

leads to a row echelon form of A. The particular row echelon form Matlab finds is called the row-reduced echelon form (row canonical form). This is a matrix for which all the pivots are 1 and all entries above the pivots are zero. Usually, a row echelon form is not unique. For example, the matrix

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & -1 & 2 \end{pmatrix}$$

is in row echelon form, but not in row-reduced echelon form. However, this matrix is equivalent (by replacing the first row by the sum of the two rows) to

$$\begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & -2 \end{pmatrix}$$

which is in row-reduced form. Matlab would return the latter for any matrix row-equivalent to it. If `A` is a square $n \times n$ matrix, then typing

```
>> [R,S] = eig(A)
```

causes Matlab to compute a square $n \times n$ matrix `R` whose columns are eigenvectors of `A`. The other object returned by `eig` is a diagonal $n \times n$ matrix `S` whose diagonal entries are the eigenvalues of `A` corresponding to the eigenvectors returned in the columns of `R`. Technically speaking, all the columns of `R` are eigenvectors of `A` only if `A` is a diagonalizable matrix. If the eigenvalues are all distinct, then `A` is diagonalizable, and sometimes `A` is diagonalizable even if it has some repeated eigenvalues. To be more precise, let us consider an example:

```
>> A=[1,0;2,2];
>> [R,S]=eig(A)

R =

         0    0.4472
    1.0000   -0.8944


S =

     2     0
     0     1
```

Thus, we see by looking at the diagonal matrix `S` that the matrix `A` has two distinct eigenvalues, namely $\lambda = 2$ and $\lambda = 1$. From looking at the matrix `R` we then see that an eigenvector of `A` corresponding to the eigenvalue $\lambda = 2$ is

$$v = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

and an eigenvector of A corresponding to the eigenvalue $\lambda = 1$ is

$$v = \begin{pmatrix} 0.4472 \\ -0.8944 \end{pmatrix}.$$

Of course, eigenvectors are only defined up to multiplication of all entries by the same nonzero constant, so, for example,

$$v = \begin{pmatrix} 0 \\ -5 \end{pmatrix}$$

is also an eigenvector of A corresponding to the eigenvalue $\lambda = 2$. To find inverse matrix of A, type:

```
>> inv(A)

ans =

   1.000000000000000                   0
  -1.000000000000000   0.500000000000000
```

If the matrix is singular, its inverse clearly does not exist:

```
>> inv([-2 2; 0 0])
Warning: Matrix is singular to working precision.

ans =

   Inf   Inf
   Inf   Inf
```

3 Graphs

Among the many features of Matlab, the ability to create graphs is one of the most useful. Here, we will describe how to deal with the usual situations, including plots of points in the Cartesian plane and graphs of the built-in functions. To plot a number of ordered pairs (x,y) connected by straight lines, we just build row vectors x and y containing the x and y values and ask Matlab to make a plot:

```
>> x=1:5;
>> y=0:.1:.4;
>> plot(x,y)
```

Of course, since the vectors x and y contain two different coordinates of the same set of points in the plane, these vectors have to have the same length. If one wants to exhibit only the points (without connecting them with straight lines) the last command can be replaced by `plot(x,y,'o')`. Because the Matlab functions are vectorized, constructing the needed vectors to graph a built-in function is easy. We first construct a vector of the desired x values, then the corresponding y values come from applying the built-in function to the vector x containing the x values:

```
>> x=0:.2:2*pi;
>> y=cos(x);
>> plot(x,y)
```

This produces a plot of the cosine function over the interval $0 \leq x \leq 2\pi$. To create a plot of a function such as $y = 2x/(x+3)$, which involves multiplication and division, we need to use the vectorized versions of these operations, writing `.*` for multiplication and `./` for division:

```
>> x=-1:.1:1;
>> y=2.*x./(3+x);
>> plot(x,y)
```

Some useful commands in making more sophisticated plots are the following:

`xlabel('x axis label')`, `ylabel('y axis label')` labels the horizontal and vertical axes, respectively, in the current plot.

`title('plot title')` adds a title to the current plot.

`axis([a b c d])` changes the viewing window on the current graph to $a \leq x \leq b$, $c \leq y \leq d$.

`grid` adds a rectangular grid to the current plot or eliminate a grid if it is already present in the current plot.

`hold on` freezes the current plot so that the subsequent plots you tell Matlab to make will be displayed on the same axes with the current one.

`hold off` releases the current plot; the next plot will erase the current before displaying.

`subplot` puts multiple plots in one graphics window.

`legend` creates a small box inside the current plotting window that distinguishes and identifies multiple plots in the same window.

`num2str(N)` returns the value of N as a string, which is helpful in producing informative titles of graphs.

Here is an example of using some of these commands, related to Section 3.6 in your book. First, let's make a list of x-values:

```
>> x=-10:.01:10;
```

Now we make a plot of $F(x) = x^2 - 2$:

```
>> plot(x,x.^2-2)
```

Matlab opens a new window containing the plot. Suppose now we want to view the same plot zooming in on the portion with $-5 \leq x \leq 5$ and $-2 \leq y \leq 2$. This modification of the current plot is easily accomplished using the axis command:

```
>> axis([-5 5 -2 2])
```

You can issue repeated `axis` commands, in order to try to find the part of the plot that is the most interesting. To place a background grid on the current plot, which can be helpful in locating the coordinates of points on the plot, just use:

```
>> grid
```

You can plot several different graphs on the same axes. One way to do this is to plot the graphs separately and tell Matlab not to erase the current plot in between with the hold on command:

```
>> plot(x,x.^2-2,'r')
>> hold on;
>> c= -10;
>> plot(x,x.^2+c,'g')
>> hold off;
```

Here the `'r'` and `'g'` tell Matlab to make the plots in red and green color, respectively. Another way to get the same result is to use a single plot command:

```
>> plot(x,sin(x/10)+sin(.09*x),'r',x,cos(x/5),'g')
```

Regardless of how the two graphs were put onto the same set of axes, you can add a legend to the plot to indicate which curve is which as follows:

```
>> legend('sin(x/10)+sin(.09*x)','cos(x/5)')
```

Adding a title to the plot is also easy:

```
>> title('Two curves')
```

Finally, while colors are a good way to distinguish different curves on the screen, it is often easier to distinguish them on a black-and-white printout if they correspond to different kinds of lines. For example,

```
>> plot(x,sin(x/10)+sin(.09*x),'-',x,cos(x/5),'--')
```

plots the first graph with a solid curve, and the cosine graph with a dashed curve. It might make sense to change the dimensions of `x` here to see the graphs better, to for example `x=-100:0.01:100` These plot directives can be combined. For example, `'or'` means plot points only, and make them red, while `'--g'` means connect the points plotted with a green dashed curve.

4 Creating m-files

So far, everything that we did with Matlab was from the Command Window. The way that most people interact with Matlab is actually with m-files. These can either be scripts or functions, the difference of which we will discuss later. For now, let us simply create and save an m-file, and make sure we are working in the correct directory.

1. Opening and creating new m-file

   (a) From the main Matlab window, go up to *File* $\rightarrow$ *New* $\rightarrow$ *M-File* in older versions of Matlab, *File* $\rightarrow$ *New* $\rightarrow$ *Script* in the newer versions of Matlab. This should open the editor window and you should have an empty screen named `untitled`. You can also type `edit` at the Command Window. This will open the editor.

   (b) You'll want to save this file. This is a bit tricky in sometimes. For now, save this wherever you want and name the file `test.m`, by either going to *File*$\rightarrow$*Save As*. Note that Matlab scripts and functions should always end with .m.

2. Creating a content of the script: In the Matlab editor (the file you just named `test.m`) you can type any Matlab commands you would like. None of these commands will be run until you tell Matlab to run them. In other words, I can type up a whole assignment in one Matlab m-file, and simply run it once. Type the following lines in to the editor window:

```
a=2;
b=1;
c = 3;
c = a+b
d = c+sin(b)
e = 2*d
f = exp(-d)
```

Save the file again, by either going to *File→Save* or by pressing *Ctrl-S*.

3. We will now run this file and Matlab will execute each command in your file one after another. There are four ways to run this file:

   (a) Go to the top of the Matlab editor window and click on the icon that looks like a white box with a green arrow.

   (b) Hit F5.

   (c) Go to *Debug→ Run test.m*.

   (d) Type `test` at the Matlab Command Window. This will ONLY work if your file (`test.m`) is in the same directory that Matlab is currently accessing.

   The last method mentioned here brings up a very good point. If you created `test.m` in a directory that is NOT where Matlab is currently looking (at the top of the main Matlab window, it shows the Current Directory) then you need to do one of two things. If you try using method (a),(b), (c) then Matlab should prompt you to change your directory. Simply click on the Change Directory button and your file should run. If you use the last method, then you need to explicitly change the directory from the main Matlab window to the same place that you saved this file.

4. Choose one of the methods and run. What happens? Each of the commands that you did not end with a semicolon should be printed out. Type who. Note that all of your variables from the script file are defined.

4.1 Creating and running a function file

If you are interested in programming in general, or if you'd like to use Matlab for some other advanced subjects, then you will have to eventually have to learn how to create and use functions.

1. Create an m-file again: Start off the same as when you created a script. This time, let us save this file as `myfunction.m`. The big difference with function files is that we generally access them using the Command Window. So either save `myfunction.m` in the same directory as listed in the main Matlab window, or save it some place else and change the current directory to this location. In fact, the easiest way to make sure the directories are correct, is just to run this empty `myfunction.m` file as a script.

2. Let us first create a function file. The difference is in the use of the word function. Make the first line of your file `myfunction.m` the following:

   ```
   function myfunction(x)
   ```

   First, note that we have named this function `myfunction`. That is because we named the m-file containing this function `myfunction.m` . You should **always** make the function name and file name the same. Second, note that we have `(x)` after `myfunction`. What this says is that the function takes an argument, and you call that argument `x`.

3. Type `clear all` in the Command Window to clear all variables and `clc` to clear the screen. Now run your function by typing `myfunction(5)` in the Command Window. You can put any argument you would like in instead of 5, since this function doesn't do anything right now.

4. Now, let us make our function actually do something. Let us take whatever the value of x is that is coming into the function, and assign the variable `y` to be the value `mod(2*x,1)` . Do this by typing: `y=mod(2*x,1)` under the function statement. Moreover, change the first line of the file to `function y=myfunction(x)`. Resave you file and run your function three times, by typing

   ```
   x=0:0.01:1;
   subplot(3,1,1);
   y=myfunction(x)
   plot(x,y);
   subplot(3,1,2)
   z=myfunction(y);
   plot(x,z)
   subplot(3,1,3)
   w=myfunction(z);
   plot(x,w);
   ```

   in the Command window. Is there anything weird about this plot?

5. Type `who` in the Command Window. What variables are defined? Then type `clear all` and type `who` again. Please note a big difference between a script and a function. In a script, everything we run becomes part of the Matlab space. But in the function, the variables are only defined **WHILE** the function is running, and they are not defined afterwards.

6. This is how you create and run a file with multiple input/output arguments. Change the function file to be:

   ```
   function [y, z, w]=myfunction(x,number)
   %this is a comment
   y = mod(2*x,number);
   z = mod(2*y,number);
   w = mod(2*z,number);
   ```

Save the file, then run from the command window as:

```
clear all
close all
[a b c]=myfunction(0:0.001:1,1)
subplot(3,1,1)
plot(0:0.001:1,a)
subplot(3,1,2);
plot(0:0.001:1,b)
subplot(3,1,3);
plot(0:0.001:1,c)
```

Type `who` to see what variables are saved in the system.

In `myfunction.m` you now have `x` and `number` as input arguments and `y`, `z`, and `w` as output arguments.

Notice that there is a problem with discontinuity in the doubling function. We can also create the doubling function in a much more complex manner. Create file `doubling.m` with the text

```
function y=doubling(x)
N=length(x);
for ii=1:N
    if (0<=x(ii))&(x(ii)<1/2)
        y(ii)=2*x(ii);
    else
        y(ii)=2*x(ii)-1;
    end
end
```

This is another (and more complex) way how to create the doubling function. Now, type in the command window:

```
>> y=doubling(x);
>> plot(x,y);
>> y=doubling(x);
>> plot(x,y);
>> close all
>> x=0:0.001:1;
>> y=doubling(x);
>> plot(x,y);
>> figure(2)
>> x=0:0.01:1;
>> y=doubling(x);
>> plot(x,y)
>> figure(3)
```

```
>> x=0:0.1:1;
>> y=doubling(x);
>> plot(x,y)
```

Most probably the discontinuity still appears to be 'continuous'. Try now edit the text `doubling.m` as follows:

```
function y=doubling(x)
N=length(x);
for ii=1:N
    if (0<=x(ii))&(x(ii)<1/2)
        y(ii)=2*x(ii);
    elseif x(ii)==max(x)/2
        y(ii)=NaN;
    else
        y(ii)=2*x(ii)-1;
    end
end
```

Save the changes and run the following commands in the command window:

```
>>close all
>> subplot(3,1,1);x=0:0.1:1;y=doubling(x); plot(x,y)
>> subplot(3,1,2);x=0:0.01:1;y=doubling(x); plot(x,y)
>> subplot(3,1,3);x=0:0.001:1;y=doubling(x); plot(x,y)
```

Some questions:

- How did we resolve the issue with discontinuity? Can we do it in a more elegant manner? (1 point)
- How can we make n-fold composition of the doubling function by calling the m-function `doubling.m`?

5 Experiment: The computer may lie

Task

- Read the assignment of the experiment on page 25, including the Notes and Questions, the hints that I provide below, plus let me know if anything is unclear.
- Work on the Procedure using Matlab and write a short document (maximally two pages, plus figures, if you wish) about your findings. As you can use graphics, I am including some hints how to do that below.
- When selecting initial seeds for either function, please include also at least 1 or 2 points that are known to be important ((eventually) fixed or periodic points) and points close to them.

- Please comment on what is happening and why is it happening with the computer computations
- You can send me the document via email on address `k.stankova@maastrichtuniversity.nl` by September 16. You can receive 4 bonus points for complete assignment.

Hints

- Having the file `doubling.m`, you can create another m-function file ,experiment.m, with text:

```
function [A]=experiment(x0);
for ii=1:100
    x0=mod(2*x0,1);
    A(ii)=x0; %saves all points on the orbit into a vector
end
```

  Then `A` contains whole orbit of `x0`. You can call the function by typing

```
>> orbit=experiment(1/3)
```

  This way you can see orbit with initial seed `1/3`
- Type `format long` in order to see what happens with the orbit with higher accuracy.
- Use also the seeds mention at the top of page 25
- If drawing the points $(x_0, F(x_0))$, $(F(x_0), F^2(x_0)), \ldots$, use function `hold on` to keep them drawing into the same figure, together with the plot of the doubling function for all $x \in [0,1]$. However, it might be more useful to draw for each seed x_0 just values of $F^n(x_0)$ with respect to n.
- Even more hints regarding doubling function can be found in `http://unizar.es/galdeano/actas_pau/PDF/089.pdf`