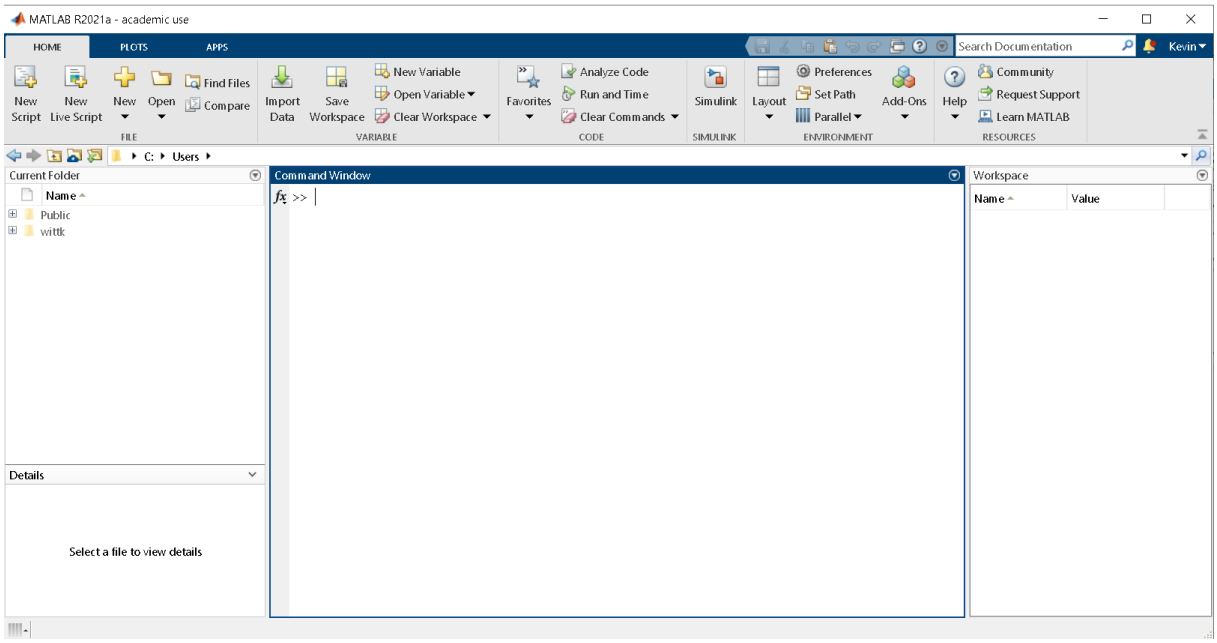
ME-474 MatLab Fundamentals

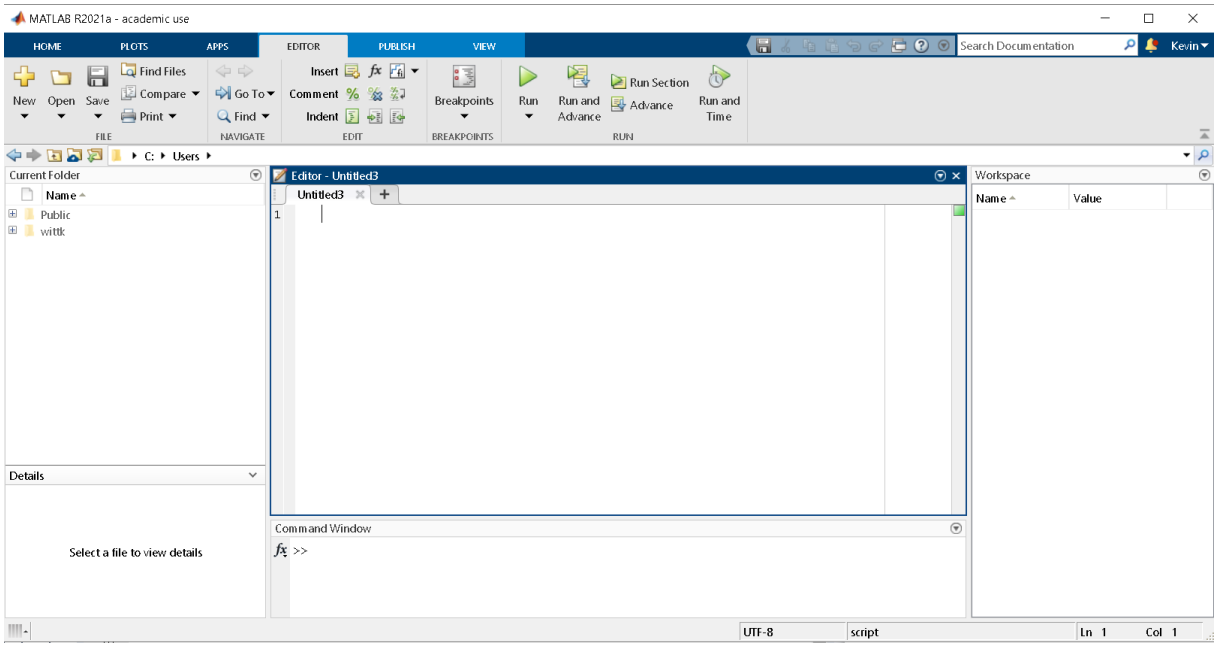
Kevin Wittkowski

September 27, 2021

MatLab is a software based on an interpreted language that allows you to perform matrix operations, image processing, calculations of ODE and PDE and much more. The interface you will see after opening it for the first time is



In the *command window* you can input commands and get directly the output they produce, however, all the operations will not be saved (i.e. if you close and reopen MatLab, they will not appear anymore). Regard it as the place where you read errors, warnings or as a simple hand calculator. The *current folder* window shows you all the files in the current folder (i.e. the one shown in the bar, $C : \backslash Users$, in this case). You can change the current folder using the buttons just above it or inserting the new location directly in the bar. When you open a *.m* script, the current folder will be set automatically to the folder in which the script is located. The *Workspace* contains all the variables you are using and shows you their type (e.g. "double" or "logical" or "char"... and dimension). Double click on them to see the elements of the variable. You will always work using scripts, so press on "New script" to generate your first script. Your window will be like this



Now what you write in the script ("Untitled3") can be saved and every time you run, it will be overwritten. To run your script, just press the "RUN" green button under the menu "editor". To avoid problems when working on multiple scripts, always start a new script with the line *clear all;close all;clc;* . This line will clear in sequence the variables in the workspace, close all figures (if there are) and clear the command line.

Hereafter you will find a list of the basic commands. For your convenience, keep Matlab open and try to insert the commands shown in this document in the sequence they are presented. Try to understand the effect each new line produces by printing in the command window the results line by line. This can be done easily by taking over the ";" at the end of each code line.

1 Array indexing and basic operations

Create a 4×4 matrix,

```
A = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16];
```

Note that "=" is used for assignments and "," divides elements in the same row while ";" divides elements in the same column. For example $A = [1, 2]$ is a row vector, while $A = [1; 2]$ is a column vector. Note that ";" is used to terminate the line/command. In case you missed it at the end of a line, MatLab will show you a warning in that line and you will see the corresponding operation being performed in the command line.
Now we substitute the element in the 3rd row and 4th column of A with a zero:

```
A(3,4) = 0;
```

and all the first column of A with ones:

```
A(:,1)=ones(size(A(:,1)));
```

and all the 4th row with zeros

```
A(4,:)=zeros(size(A(4,:)));
```

Now substitute rows from 1 to 3 and columns from 2 to 4 with "5":

```
A(1:3,2:4)=5*ones(size(A(1:3,2:4)));
```

Note that you declare an array using square brackets and you index it using curve brackets. Note also that to say "all the elements", MatLab uses ":". For example, $A(4,:)$ means "take the elements of A at row 4 and ANY column number". Similarly, the notation $A(1:3,1)$ will mean "take the elements of A with row number from 1 to 3 and column number 1. To see how the functions "zeros" and "ones" work, please refer to the documentation (Go to "HELP" section or google "matlab zeros" or "matlab ones"). The function *size* returns a vector containing the dimensions of a given array. It can be used also with vectors, but maybe the *length* function is what you need in that case.
To declare an equally-spaced vector of elements, we can use the notation

```
V=1:0.5:10;
```

that corresponds to a vector of elements from 1 to 10, with a spacing between the elements always equal to 0.5. If the spacing is unknown but you know how many divisions between the first and the last element you want, use

```
V=linspace(1,10,135);
```

This will divide the range between 1 and 10 in 135 values.
The operations that can be performed with vectors in MatLab are:

- **sum/difference**: given 2 arrays A of $m \times n$ elements and B, of the same dimension, $C = A + B$ is the array whose i-th,j-th element is the sum of the element i-th,j-th of A and of the element i-th,j-th of B. Similarly for $C = A - B$;
- **product**: $C = A * B$ is the matrix product of the matrices $A^{m \times n}$ and $B^{n \times q}$, so C will have m rows and q columns. For scalars, this operation reducess to the classic product between scalars;
- **element-wise product and division**: $C = A. * B$ is the product of each element in A for the corresponding element in B: $c_{ij} = a_{ij} b_{ij}$. A and B in this case must have the same size. Element-wise product is commutative. Similarly, $C = A./B$ will perform $c_{ij} = a_{ij}/b_{ij}$. For scalars, $C = A/B$ is the ratio between the scalar A and the scalar B;
- **power**: $C = A.\hat{}2$ will perform $c_{ij} = a_{ij}^2$, while $C = A\hat{}2$ will perform $C = A * A$, which is possible only for square matrices;
- **natural logarithm**: $C = log(A)$ will perform $c_{ij} = log(a_{ij})$
- **exponential**: $C = exp(A)$ will perform $c_{ij} = e^{a_{ij}}$;
- **linear system inversion**: given a linear system $A * x = B$, the solution is given in MatLab by $x = A \backslash B$. See the command *mldivide* and *inv* for further details;
- **concatenation of arrays**: $C = [A, B]$ is the operation that concatenates the arrays A and B so that the last column of A is adjacent to the first column of B. Similarly,

$C = [A; B]$ will concatenate A and B so that the last row of A will be adjacent to the first row of B. Matrix dimensions must be compatible.

Other operations are possible with arrays in MatLab, but these are the fundamental ones

2 Logical operations and loops

2.1 FOR and WHILE

Loops repeat operations for a certain number of times (that is predetermined) or until a certain condition is met (disregarding the number of times the operation is repeated). In the first case, a FOR loop is used, while, in the second, a WHILE loop is used.
Suppose we want to compute the sum of all numbers present into a given array. We know the length of the array, so we will use a FOR loop:

```
a=1:1:10; %will generate a vector [1,2,...,10]
s=0; %variable in which we store the partial sum
for i=1:length(a)
    s=s+a(i);
end
```

The results will be $s = 55$. Note that we have used % to write comments inside the code.
Note: to compute the sum of the elements in a vector, a more efficient way is to use the function $s = sum(a)$. See documentation for details.
Now suppose that A is a column vector of 10 random numbers uniformly distributed between 0 and 1:

```
A=rand(10,1);
```

We want to find at which element in A, the sum of all elements up to that element gives at least 2.5:

```
i=0;              % index
s=0;              % partial sum
while s<=2.5
    i=i+1;        % update index
    s=s+A(i);     % add current elem. of A
end
```

The loop sums all the elements of A as long as that sum is below or equal to 2.5. If the sum exceeds 2.5, the loop is stopped and the value of i will correspond to the element in A at which the condition was reached.
Note: the same result can be obtained more efficiently (i.e. without loops) with the command (see documentation for details):

```
i = find(cumsum(A)>=2.5, 1, 'first');
```

Note that index update is automatic in MatLab FOR loops but not in WHILE loops. Loops can be terminated also using the *break* command.

2.2 IF and SWITCH-CASE, conditions on array elements

Suppose we have the array

```
V=linspace(1,10,135);
```

and we want to find the values of this array that are in the range $[4, 6]$ or that are equal to 10. This can be done by looking at all elements of V and using a conditional statement:

```
j=1;
for i=1:length(V)
    if (V(i)>=4 && V(i)<=6) || V(i)==10
        W(j)=V(i);
        j=j+1;
    end
end
```

Now the vector W will contain the elements of V that are between 4 and 6 or that are equal to 10. The same operation can be performed more efficiently in MatLab using this notation:

```
Z = (V>4 & V<6) | V==10;
W = V(Z);
```

In this case, without using loops (loops are not computationally-efficient in MatLab), first you create a logical array Z, long 135 elements, containing "0" in the i-th position if the i-th element does not match the conditions, and "1" if it does. Then you pass Z to V and you get only the elements of V whose position corresponds to a "1" in Z.
Note: two valuable alternatives to the code just proposed are

```
i = find( (V>4 & V<6) | V==10 );
W = V(i);
```

and

```
W = V( (V>4 & V<6) | V==10 )
```

The loop just presented can be modified a bit to make things clearer:

```
j=1;
for i=1:length(V)
    if V(i)>=4 && V(i)<=6
        W(j)=V(i);
        j=j+1;
    elseif V(i)==10
        W(j)=V(i);
        j=j+1;
    else
    end
end
```

The output of this loop is exactly the same as before, but the commands "elseif" and "else" are used, instead of an "OR". In this case, the instruction reads: "for each element

in V, if that element is between 4 and 6, assign to the next element of W that value, otherwise, if the element is equal to 10, do the same thing, in any other case do nothing". The SWITCH is very similar to the IF/ELSEIF, but used when the number of conditions is discrete: suppose that, given a number, we want to compare it with just other 3 values, and do some operation as a consequence.

```
n = input('number:');

switch n
    case -1
        disp('NegativeOne')
    case 0
        disp('Zero')
    case 1
        disp('PositiveOne')
    otherwise
        disp('Other')
end
```

In this example, when the code is executed, the user will be prompted to insert a number in the command window and press enter. The code will assign that number to the variable n and compare that number with -1, 0, 1: for example, if n=-1, the command window will display "NegativeOne", otherwise, if n=8.2, the command window will display "Other".

3 PLOT, SURF and other chart commands

There are a lot of different chart types available in MatLab and it would be impossible to cover them all in a few pages. For this reason, we will describe briefly only the most useful for the purpose of this course.

3.1 PLOT

Suppose that we have a dataset

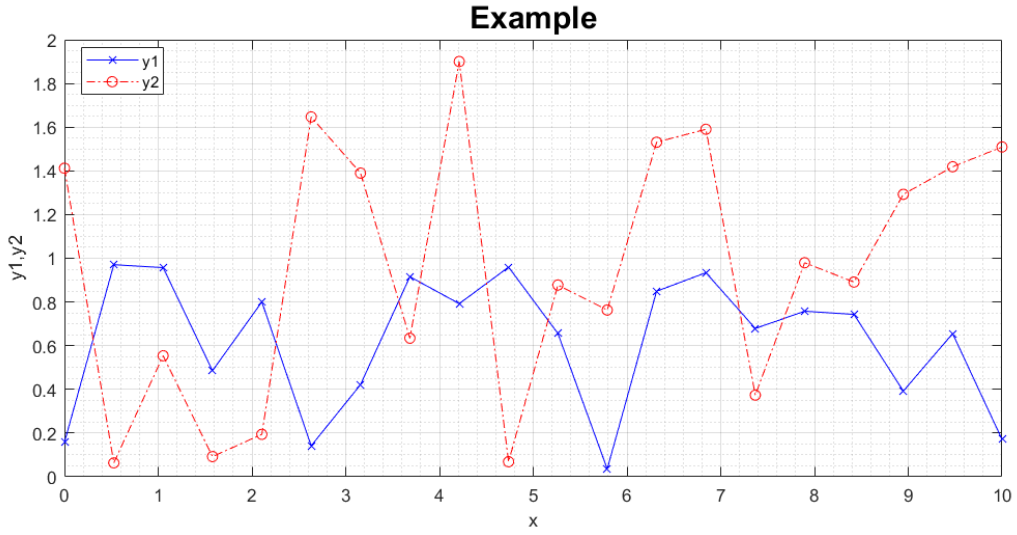
```
x=linspace(0,10,20);
y1=rand(size(x));
y2=2*rand(size(x));
```

and that we want to plot y1 and y2 as a function of x. A complete way of performing the task would be

```
figure(1)    % open a new figure window
plot(x,y1,'b-x','DisplayName','y1');hold on;
plot(x,y2,'r-.o','DisplayName','y2');
grid on;grid minor; % enable background grid coarse and fine
xlabel('x'); % axes names
ylabel('y1,y2');
title('Example','FontSize',18); % add title to the figure
legend('Location','best'); %add legend to the figure
```

```
hold off;
```

You will get something like this



A brief explanation about the plot interface:

- the option $'b-x'$ defines a color line blue (b), with continuous line (-) and x markers (x);
- the option $'DisplayName','curvename'$ displays the name of the curve (i.e. *curvename*) in the legend;
- *hold on* commands enable to overlap plots on the same figure window and *hold off* turns off that possibility.

I invite you to google "matlab plot" and read the documentation to find all the details. You can also write *doc plot* in the command window and press enter. Furthermore, you may find the command *subplot* useful.

3.2 PLOT3 and SURF/MESH

MatLab allows you to plot in 3D. This feature is useful if you want to see a performance map, for example. In all the examples, we will plot a certain "z" as a function of (x1,x2). If the data have a regular structure, SURF/MESH commands can be used, while if the data are given as a list of single unstructured data points, PLOT3 may be what you need. In case you want to visualize a surface from a list of sampling points, you may find the *griddata* command useful to get a proper grid on which you can use SURF/MESH. Suppose we have

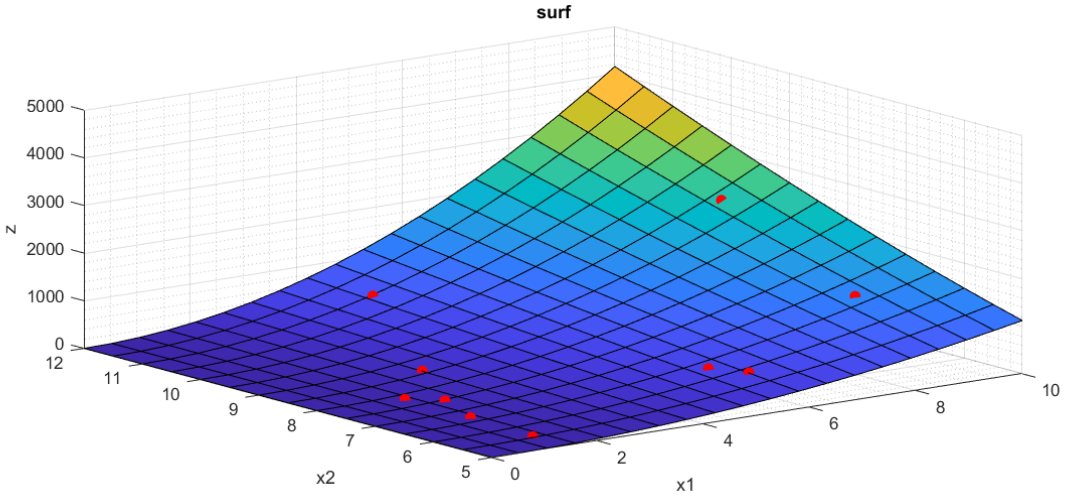
```
x1=linspace(0,10,20);
x2=5:0.5:12;

[xx1,xx2]=meshgrid(x1,x2);
z=xx1.^2.*xx2.^1.5;
```

```
figure(1)
surf(xx1,xx2,z);hold on;
xlabel('x1');ylabel('x2');zlabel('z');title('surf')
grid on;grid minor;
```

Now try to substitute the command *surf* with *mesh*: only the appearance of the plot will change, as the *surf* command fills the surface plot while the *mesh* command shows only the mesh-grid. Now we add some random points on the chart:

```
x1_r=10*rand(10,1);
x2_r=5+7*rand(10,1);

z_r=x1_r.^2.*x2_r.^1.5;
plot3(x1_r,x2_r,z_r,'r o','MarkerFaceColor','r');
```

You should obtain something like this



Remarks: the *meshgrid* command is used to generate a regular grid from the axis intervals $x1$ and $x2$. This grid is then used to compute the values of z in each $(x1, x2)$ point. Note: a valid alternative to *plot3* is *scatter3*. Please read the documentation.

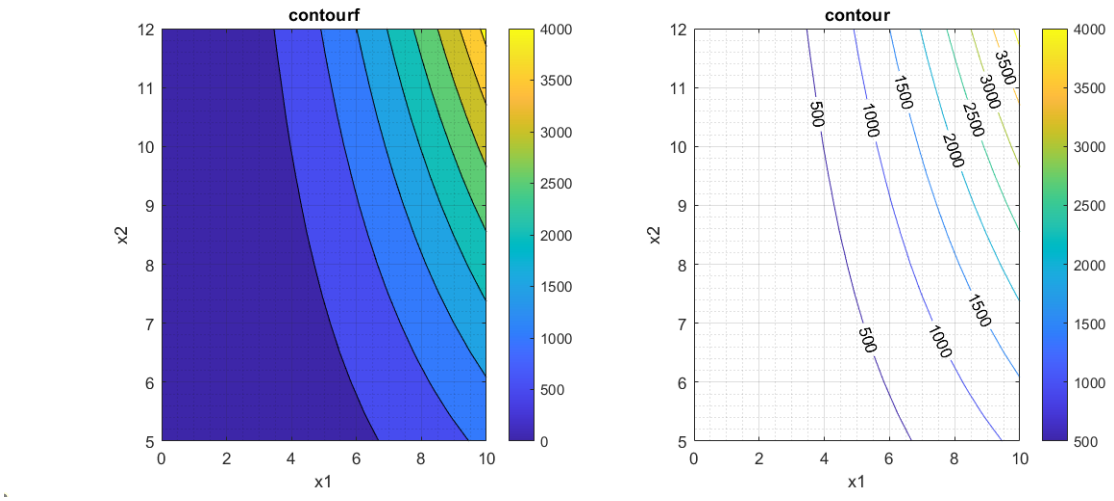
3.3 CONTOUR/CONTOURF

The idea behind the contour plots is to project a surface map like the one we have just plot onto its base plane, while keeping its coloring. This means that the highest regions in will keep the yellow coloring, the lowest will keep the blue coloring and so on. In this way we obtain the level curves of z:

```
x1=linspace(0,10,20);
x2=5:0.5:12;

[xx1,xx2]=meshgrid(x1,x2);
z=xx1.^2.*xx2.^1.5;
```

```
figure(1)

subplot(1,2,1)
contourf(xx1,xx2,z);hold on;
xlabel('x1');ylabel('x2');zlabel('z');title('contourf')
grid on;grid minor;
colorbar;

subplot(1,2,2)
contour(xx1,xx2,z,'ShowText','on');hold on;
xlabel('x1');ylabel('x2');zlabel('z');title('contour')
grid on;grid minor;
colorbar;
```

This is what you get



Some remarks about the present example:

- the *contourf* and *contour* commands perform a similar task, they only differ in the fact that in the first case the space between the iso-lines will be filled with color and in the second case it will be not;
- the use of *meshgrid* is the same as in the *surf* plot;
- the *subplot* command is used to plot two charts in the same figure. I invite you to read the documentation about this command;
- the option *'ShowText','on'* shows the values of z on the iso-z lines,
- the *colorbar* command shows the colorbar of the contour plot.

4 HELP and how to search what you need

There are two ways to get information about MatLab commands:

1. **if you know the name of the command**, you can use the documentation available in local on your PC. Suppose you want to use the command *eig* to compute eigenvalues and eigenvectors of a matrix, but you can't remember the syntax. Just write in the command line

   ```
   doc eig
   ```

2. **if you don't know the name of the command** -for example if you have to perform an operation that you've never seen before- just google what you need. The first search results will always come from MathWorks, which has a very good help, forum and file exchange.

   - The help is the same you get using *doc...*;
   - in the forum you will probably find out that other users had the same problem as you had, or you can get direct help by writing there. It is usually very fast to get answers there (if you describe well what you need!);
   - the file exchange may help you to get some particular files that perform very specific tasks (e.g. Kriging interpolation or plot arrangements...).

The documentation available on MathWorks is updated and full of examples, so it's the best source of information for MatLab users.