Yet another Matlab course for control engineers

Table of Contents

How to follow this course

- browse the GitHub repository https://github.com/damianovar/TTK4225-2020
- download the TTK4225\trunk\Matlab\IntroductoryWeekCourse\Scripts\main.mlx file somewhere
- launch Matlab, and change the directory in the command window to where you downloaded `main.mlx` with the command `cd`. For example, it may be something like

```
cd C:\Users\damianov\GitHub\TTK4225\trunk\Matlab\IntroductoryWeekCourse\Scripts
```

- launch `main.mlx` from the command window, i.e.,

```
open main.mlx
```

General tips

"*Command Window*" vs "*Matlab .m scripts*" vs "*Matlab live scripts*"

Almost all code in Matlab can be written either in the Command Window (CW), in Matlab scripts (i.e., in own files with filename .m) or in Live scripts (i.e., in own files with filename .mlx). Executing the same code in different ways will lead to identical results, but there are different pros and cons in using the CW, the Matlab scripts, or the Live scripts. A summary is:

- **Command Window**: should be used only if you need to get help on some command or launch one line of code per time
- **.m scripts**: best when you want to do modular code (see the section on object oriented programming in Matlab)
- **live scripts**: best when you need to present some results to an audience

So: **do not use the Command Window for creating code.** If you want to make a small change at the beginning of the code, then in the CW you need to re-enter everything.

Getting help

Alternative 1: from the Command Window, simply enter "help xxx". E.g.,

```
% Get help on the "plot" command
help ode45;
```

Alternative 2 (will return more verbose information): again from the Command Window, enter "doc xxx". E.g.,

```
% Get the documentation on the "plot" command
doc plot;
```

Alternative 3 (if you do not want to pass through the Command Window): highlight some text and press F1.

Using the Matlab workspace

All the variables you create are added to the Matlab Workspace. If a variable is stored there, then it can be used wherever you wish, whether in a .m script, a .mlx script, or as a parameter in a Simulink chart. You can double-click on the different variables to look at their values.

You can also save / load workspaces using the corresponding "save" and "load" commands.

Using the path

Matlab Command Window always refer to some path. For example, now we are working here:

```
pwd
```

If we want we can move ourselves somewhere else with the "cd" command, e.g.:

```
cd 'C:\Users\damianov\Dropbox\CSS Outreach'
```

We can also add to the system path some pre-fixed paths, e.g.:

```
addpath 'C:\Users\damianov\MEGA\Software\+MatlabToTikZ'
```

In this way we can use all the functions that are present in that folder.

Publishing your .m scripts code and results

The command "publish" lets you publish your code and results to HTML, PDF, LaTeX, among others. This is useful when you want to document the code (but don't use this for course reports!). To use it, go to the editor, then to the "publish" tab, and then tap the arrow under the Publish icon. If you then press "Edit ..." you get a lot of choices. Here you can adjust various parameters for how it will be published, including whether you want to include the code or not. You can also change the format.

When you publish, Matlab will run your entire .m file, bringing with it all the plots that are being made. It uses cells (%%) to create headings. Example:

```
%% Test document

% This text will appear as plain text

%% Task 1

% Here we play with figures
figure(1);
clf(1);
plot(1:10);
```

```
%% Conclusions

% We conclude that Matlab can publish documents.
```

Publishing your .mlx scripts code and results

Publishing live scripts gives the possibility of keeping the interactivity elements of live scripts. The workflow is similar to publishing .m scripts. See https://se.mathworks.com/help/matlab/matlab_prog/publishing-matlab-code.html for more information.

Basic stuff

Control Flow Statements

For loops

```
% creates the vector and starts the loop
for iCounter = 10:-2:1
    %
    a = iCounter * 10;
    fprintf('my variable is now %.5f and %d\n', a, iCounter)
    % do something
    %
end % ends the loop
```

Remarks:

- no parentheses (as in C / C ++) to end the block

If-then-else

```
if iCounter == 20
    % do something
elseif iCounter >= 21
    % do something else
else%
    % do yet something else
end %
```

Remarks:

- "`if iCounter == 20`", NOT "`if iCounter = 20`"! Otherwise you are doing an assignment...

While

```
while iCounter == 2022
    % do something
end %
```

Switch

```
b = 5
c = 8
cUserChoice = 'a' % insert a character here
switch cUserChoice
   case 'a'
       d = b / c% do something
   case 'b'
       % do something else
   otherwise
       % do yet something else
end %
```

Remarks:

- 'switch' is VERY USEFUL in conjunction with varargin and nargin (i.e., when wanting to have a flexible number of function input arguments)

```
a = 'b';
b = 10;
c = 20;

switch a
 case 'x'
  fprintf('the user chose to do multiplications');
  d = b * c
 case '-'
  fprintf('the user chose to do subtractions');
  d = b - c
 otherwise
  fprintf('the user chose something not valid - I will do sums');
  d = b + c
end
```

Vectors

Create vectors using the colon operator ":"

Use a syntax like "`vector = first : spacing : last`". E.g.:

```
% create a vector from 1 to 10
afVectorA = 1:10

% create a vector containing 1 3 5 7 9
afVectorB = 1:2:10

% create a vector containing -1 -3 -5 -7 -9
afVectorC = (-1:-2:-10)'
```

Create vectors using the linspace and logspace methods

```
% create 30 samples linearly spaced between 1 and 100 (included
afVectorD = linspace( 1, 100, 30 )

% create 7 samples logarithmically spaced between 10^(-2) and 10^(+4)
afVectorE = logspace( -2, 4, 7 )
```

Create vectors using square brackets

Both with and without commas work the same (but be careful to be consistent!)

```
afVectorD = [ 1, 5,   9,  13];
afVectorE = [-1  0.5  80  4];
```

Extract values from a vector

```
afVectorA(1)     % extracts the first element
afVectorB(3)     % extracts the third element
mynewvector = afVectorC(1:2:4)   % extracts the second and third element
afVectorD([1,3]) % extracts the first, second and third element
afVectorE(2:end) % extracts from the second to the last element
```

Matrices

Create matrices using square brackets

```
aafMatrixA = [ 1  2  3;
               4  5  6;
               7  8  9 ]
```

Extract values from a matrix

Again, use the colon operator ":". For example:

```
B = aafMatrixA( 1:2, 2:3 )  % extracts the submatrix formed by the first two rows
                            % and the last two colums

aafMatrixA( :, 2:3 )    % extracts the submatrix formed by all the rows
                        % and the last two colums
```

Create multi-dimensional matrices

```
aaafMatrix = zeros( 3, 2, 4 );

for iCounterA = 1:size( aaafMatrix, 1 )
for iCounterB = 1:size( aaafMatrix, 2 )
for iCounterC = 1:size( aaafMatrix, 3 )
    %
    aaafMatrix( iCounterA, iCounterB, iCounterC ) = ...
        iCounterA * iCounterB - iCounterC;
```

```
    %
end %
end %
end %
```

Extract values from a multi-dimensional matrix

```
aaafMatrix( :, 2, : )          % get the 'second' slice
squeeze(aaafMatrix( :, 2, : )) % get the slice in a more convenient format
```

Using ";" to prevent display messages in the command window

```
afVectorA(1)     % shows the results in the command window
afVectorB(3);    % does not show the results in the command window
```

More advanced stuff

Making plots that don't make others cringe

If you want to do plots in Matlab, see the suggestions in http://folk.ntnu.no/damianov/Matlab/HowToMakePrettyFiguresWithMatlab.pdf and the templates in http://folk.ntnu.no/damianov/matlab.html.

If you want to do plots in LaTeX/TikZ, see the suggestions, templates and video-instructions in http://folk.ntnu.no/damianov/LaTeX.html.

Learn how to do plots in LaTeX/TikZ asap.

Importing data

- you just have a table of numbers? Then use `load()`
- you have a table of numbers with nonnumeric column and/or row headers? Then use importdata()
- you have a table of numeric/nonnumeric things, e.g., columns of characters or formatted dates or times? Then use `textscan()`
- you don't even have a table, i.e., each row has got its own number of columns? Then use `fgetl()`

Want more details? See the documentation and https://se.mathworks.com/learn/tutorials/matlab-onramp.html, lesson 11.

Good coding habits

In general, see http://folk.ntnu.no/damianov/Matlab/CodingRules.pdf

Important messages:

- people will judge you based on how you indent the code;
- program as a stereotypical granny drive;
- premature optimization is the root of ***most*** evil.

Use at least modular, at best object oriented programming paradigms

- **be modular**: if a piece of code does a specific logical operation, make it become a function
- **be object-oriented**: structure the logic of your program into objects that interact with one another, that have their own properties, and that have own methods to manage the interactions among the objects.

Pros for object-oriented: faster debugging, better maintainability, better portability, **you don't seem a fool** when you start working for serious teams / companies

Cons for object-oriented: need some little more time at the beginning of each project. As soon as the final code will be more than 300 lines of code probably better to do OOP.

For templates on how to create and manage classes in Matlab, see http://folk.ntnu.no/damianov/Matlab/@TemplateClass/.

Use nargin and varargin to increase modularity

Example:

```
function ExportLineplot(     ...
  strFileName,       ...
  aafSignals,       ...
  astrHeader       ) % optional

% flag if there exists the header
bHeaderIsPresent = ( nargin > 2 );

% if the header is not there, then create it
if( bHeaderIsPresent )
   astrHeader = % something
end;

% ... some other code ...

end % function
```

Use errors and warnings to find and prevent bugs

Always place warnings to follow up whether a user-choice has been made consistently with the available choices. Example (would have been more meaningful within a function, though):

```
% initialization
bUserChoiceIsWellFormed = false;

while( ~bUserChoiceIsWellFormed )
    %
    % tell the user her/his options
    fprintf( 'a = option 1, b = option 2' )
    %
    % get the user input
    cUserChoice = 'c';
    %
    % check
```

```
    if(     cUserChoice ~= 'a'  ...
        &&  cUserChoice ~= 'b'  )
        %
        warning('badly formed user choice!')
        %
    end % if
    %
end % while
```

Place errors to check if the code is in places where it should never enter. Example:

```
fMyPositiveParameter = -1;  % something strange happened before and created
                            % a situation that we should never encounter

if( fMyPositiveParameter < 0 )
    %
    error('something strange happened; Did a bug occur in method XXX?')
    %
end %
```

Adopt tricks to improve both the speed and the style

Check http://www-users.math.umn.edu/~lerman/math5467/matlab_adv.pdf, chapter 6.

Periodically profile your code

Matlab can profile (i.e., track the execution time) of your code, so that you can see what needs / is best to optimize. This is something you should do as soon as your simulations start taking tens of seconds. See Profile Your Code to Improve Performance for more information.

Use some serious version control system

If you never heard this term before, then: "Version control := strategy to manage changes to documents / software / whatever." A VCS is not only an efficient back-up system, but also a way of keeping a transparent history of changes, of who made them, and when.

The most famous ones nowadays are `github` and `gitlab`. Main concepts:

- the **repository**: the set of all the files and folders of a project + each file's revision history. This is stored in the web
- a **clone**: a local copy of the repository.

Main operations:

- **`git pull`**: get the modifications (if there is any) that have been made to the repository. Use this if a teammate has made some modifications on the remote repository and you would like to load those changes in your local copy
- **`git status`**: check if and how much the local copy and the remote repository differ
- **`git add`**: add some file / folder to the local copy

- **`git push`**: push the local modifications into the remote repository. Use this if you want the teammates to load in their local copies the modifications you made.

More information in https://guides.github.com/introduction/git-handbook/

Solving numerical optimization problems

I.e., finding the parameters that maximize or minimize a function that you can evaluate from quantitative perspectives. You will learn how this works in some details during your Masters degree!

Typical scenario: `goodness_of_my_simulation = function( a_lot_of_parameters )`. Then:

- may your parameters be whatever you want? Then use `fminunc()`
- do you have some constraints on your parameters? Then use `fmincon()`

Example:

```
% definition of the Rosenbrock's cost through an anonymous function
CostFunction = @(x) 100 * ( x(2) - x(1)^2 )^2 + ( 1 - x(1) )^2;

% definition of a constraint of the type A x >= b
A = [ 1, 2 ];
b = 1;

% definition of where we start the search
x0 = [ -1, 2 ];

% compute the optimal x
x = fmincon( CostFunction, x0, A, b )
```

PS: https://se.mathworks.com/help/matlab/matlab_prog/anonymous-functions.html. Anonymous functions help not having to edit and maintain files for functions that require only a **local** definition.

Using symbolic variables

Symbolic variables enable solving, plotting, and manipulating math equations using variables that should be considered as symbols, not just values.

Differentiation with symbolic variables

```
syms x;
f = sin(x)^2;
diff(f)
```

Differentiation without symbolic variables

Differentiation can always be done numerically; advice = use the in-built function `gradient()`

Integration with symbolic variables

```
syms x;
```

```
f = sin(x)^2;
int(f)
```

Partial derivatives with symbolic variables

First derivative with respect to y:

```
syms x y
diff(x*cos(x*y), y)
```

Second derivative with respect to y:

```
syms x y
diff(x*cos(x*y), y, 2) % note the number 2 here!
```

Second derivative with respect to `x` and `y`:

```
syms x y
diff(x*sin(x*y), x, y)
```

Mixed higher-order derivatives:

```
syms x y
diff(x*sin(x*y), x, x, x, y)
```

Ordinary differential equations

Disclaimer: this is a big topic!! See https://se.mathworks.com/help/matlab/ordinary-differential-equations.html for a more comprehensive treatment.

We here make a simple example: the Lotka-Volterra system, i.e.,

$$\begin{cases} \dot{y}_1 &= y_1 - \alpha y_1 y_2 \\ \dot{y}_2 &= -y_2 + \beta y_1 y_2 \end{cases}$$

and we want to simulate this dynamics.

The first thing is to define an opportune function capturing the dynamics of the system, i.e.,

```
function dy = dydt(t, y, a, b)

dy = [   y(1) - a * y(1) * y(2)  ; ...
       - y(2) + b * y(1) * y(2)  ];

end % function
```

Note that the functions like the above one should always have the two initial parameters as `t` and `y`, and always return column vectors. Then the code to solve the ODE can be something as follows:

```
% define the parameters of the model
a = 0.2;
b = 0.3;
```

```
% define the time interval for the analysis
tstart = 0;
tend   = 15;

% define the initial condition
y0 = [25; 20];

% solve the ODE
[t, y] = ode45(@(t, y) dydt(t, y, a, b), [tstart tend], y0);

% plot the time signals
plot(t,y)
title('Preys / Predators Populations Over Time')
xlabel('time [adim.]')
ylabel('population [adim.]')
legend('preys', 'predators', 'Location', 'North')

% plot the phase plane
plot(y(:,1), y(:,2))
title('preys / predators in the phase plane')
xlabel('preys [adim.]')
ylabel('predators [adim.]')
```

Interesting stuff for control engineers

Modelling

Analysis

PIDs

Root locus

Transfer functions

State space systems

Digital control

http://ctms.engin.umich.edu/CTMS/

References

Free courses, for generic audiences

1. http://www-users.math.umn.edu/~lerman/math5467/matlab_adv.pdf
2. https://se.mathworks.com/learn/tutorials/matlab-onramp.html

Free courses, but designed specifically for control engineers

1. http://ctms.engin.umich.edu/CTMS/

For profit courses

1. https://www.udemy.com/course/matlab-programming-fundamentals

Pedagogical material for instructors

1. Kirschner, Sweller, Clark. "Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching", Educational Psychologist, 41(2), 75-86, 2006