A Short Introduction to MATLAB

September 2, 2020

This short document is meant to present some of the functionalities of MATLAB that we judge useful for the 'Modelling and Simulation' course. Please note that this is by no means meant to be a complete and comprehensive tutorial of MATLAB. There are many great such tutorials openly-available on the internet if you want a more complete documentation. Here, we have gathered some basic MATLAB concepts and commands in the first three sections (MATLAB interface, matrices, arrays, loops and so on) while the fourth section is somewhat more advanced as it introduces the symbolic toolbox of MATLAB. This last part is especially important for the course assignments, so make sure you are comfortable with it!

The goal of this tutorial is that you experiment with the material introduced in your own MATLAB workspace as much as possible. We encourage you to copy and paste the code from the snippets to your own shell in order to get a sense of what it does. Please note that individual snippets are not always designed to be ran as a single block, e.g. you may need some variables that were defined in the snippet above. Some lines also purposely return an error, so keep an eye out. Once you're done with one section, we encourage you to try to solve the exercises at the end. Their solution will be uploaded to Canvas in a separate MATLAB file after the first supervision session.

1 Graphical Interface

MATLAB's graphical interface is quite intuitive and it is pretty fast to learn what everything does. In particular, you have two large regions: the editor where you write you code as a script or a function and the prompt where you execute it (anything you type in the prompt gets executed directly, you don't have to write it in the editor first). Please also notice that variables get saved in the workshop after something has been executed in the prompt. This can be very handy to reuse the variables, or to investigate them.

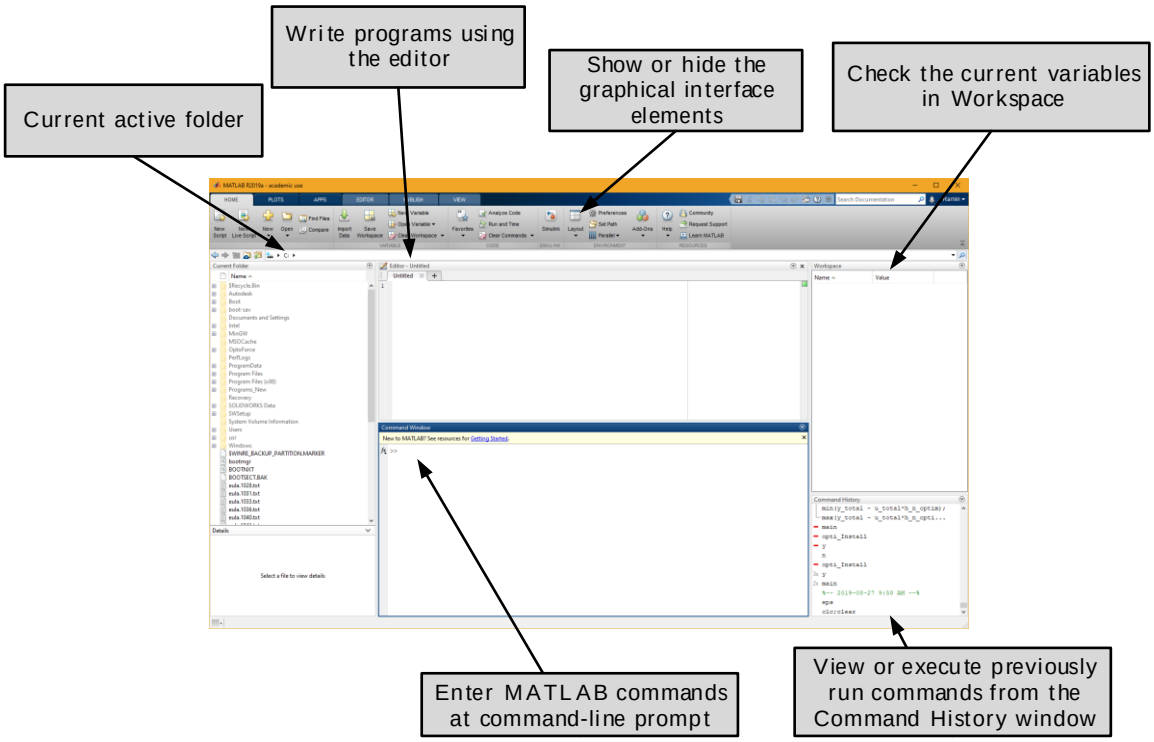

Figure 1.1: MATLAB's graphical interface

A few useful things to know about MATLAB before you start coding:

- End each line of code with a semicolon ';'. Otherwise what it does will be printed in your shell. This can become annoying very fast.
- A note for Python/C++ users: MATLAB array indexation starts at 1, not 0 (you'll get used to it).
- You can write comments using '%'. The commands 'CTRL + R' and 'CTRL + T' are quite useful since they respectively allow you the comment/uncomment the selected block of code.

2 Basic Commands

This section introduces some basic things about working with MATLAB and focuses on how to handle matrices. If you are already familiar with the concepts and functions presented here, good for you, you may only want to read through it quickly, or directly go to the small exercise at the end of this section. Otherwise, please feel free to try out some of the code snippets in your own workspace to get used to MATLAB.

2.a Vectors and matrices

MATLAB has been created to be very optimized when it comes to matrices operations, and for linear algebra in general. It is therefore very important that you are familiar with the creation and manipulation of matrices (and vectors) since you will have to work with some most of the time that you use MATLAB for this course.

In what follows, we try to only use small letters for vectors and capital letters for matrices.

- The **help** function from MATLAB can be fairly useful sometimes. **You should also get used to looking up on the internet when you have any coding-related questions, the answer often lies there.**

```
% Use the help function whenever you want to have more info on what
% something does
help arctan

% You may alternatively use the doc function
doc arctan

% It is easy to print anything with disp. It works with most variable types
disp('hello, world')
```

- Below are simple ways to create vectors and matrices:

```
% Define vectors and matrices by hand
a = [42, 10, 7.5, 88, 3.14];                          % line vector
b = [42 10 7.5 88 3.14];                              % equivalent to a
c = [42; 10; 7.5; 88; 3.14];                          % column vector
A = [1, 2, 3 ; 4, 5, 6];                              % 2*3 matrix
B = [0.42, 0.42; 0.28, 0.28; 0.1, 0.1; 0.31, 0.29]; % 4*2 matrix

% Check the dimensions of a vector/matrix
size(a)
length(a) % only use length for vectors !
size(B)
length(B) % see?

% Generate vectors/matrices containing only zeros or ones (or any scalar)
a = zeros(10,1);
b = ones(10,1);
A = zeros(3,3);
B = 41*ones(3,3);
I = eye(10);      % generate identity matrix
```

- Once you have generated some matrices, you can try to realize the following simple operations:

```
% + and - operations between vectors and matrices are written in the same
% way as with scalars
b = b - 2;
c = a - b + 2;
B = B + 1;
C = 2*B - (B + B);
C = A + B;         % matrices being added should have the same dimension !

% You can access single matrix elements A(i,j) and vector elements a(i)
% just as you would imagine
A = [1, 2, 3; 4, 5, 6];
a = 1:10;
A(1,3)
a(5)

% You can even select only some parts of a matrix/vector
B = A(:,1);   % select first column of A
B = A(2,:);   % select second row of A
B = A(1,2:3); % elements on first row of A and belonging to columns 2 or 3
b = a(2:5);
b = a(4:end); % all elements from element 4 to the end of a
```

```
% Transposition works like this
trans_A = A';
trans_a = a';
```

2.b Linear algebra

- The multiplication of matrices is slightly more complicated than addition/subtraction as one has to pay attention to the dimensions.

```
% The multiplication of matrices is done with the operator '*'
A = ones(2,3);
B = rand(3,3);    % generate a matrix with random elements between 0 and 1
C = A*B;
D = B*A;          % mind the dimensions when using matrix multiplication !
E = pi*B;         % but no problem for multiplying with a scalar

% Please note that the previous is different from term-by-term
% multiplication, which can be done with operator '.*'
I = eye(3);
C = B*I;
D = B.*I; % notice the difference with C
E = I./B; % also works for term-by-term division
```

- Here are some useful functions to investigate some properties of matrices:

```
% Use 'det' to get the determinant of any (square) matrix
A = [1, 3; 4, 2];
d = det(A);

% If (AND ONLY IF) the determinant is not 0 (or very close to 0) may you
% use 'inv' to get the inverse of the matrix
inv_A = inv(A);

% You can even play with the eigenvalues of the matrix if you want
lambda = eig(A);
```

- You might sometimes encounter linear systems of the form $Ax = b$ that you have to solve. Once you know that A is non-singular, you can use its inverse A^{-1} to compute the solution: $x = A^{-1}b$. However, it is quicker to use the \ operator in MATLAB rather than explicitly computing the inverse matrix, so it is usually the way to go.

```
% Use operator '\' to solve linear systems of the form Ax = b. But first
% make sure that A is singular. Don't be that student that tries to
% invert singular matrices ;)
A = [1 4 5; 2 3 5; 5 9 8];
b = [1; 1; 1];
x = A\b;
```

2.c Built-in functions

- In addition, there are a lot of classical mathematical functions which can be directly called in your scripts without needing to redefine them:

```
% Trigonometric and exponential functions are built-in just as you would
% expect, no surprises here. Same for pi and i (but not e)
cos(pi);
sin(pi/2);
tan(pi/4);
```

```
exp(1);
log(exp(1));
sqrt(2);
2^8;          % x to the power n is written x^n
```

- Here are some more useful functions for vectors and scalars:

```
% Some other useful functions whose names give them away
a = [16; -23; 10; 42; -37; 22; -30];
sum(a);
[x,i] = min(a); % x is the minimum value and i is its index in a
[x,i] = max(a);
a = abs(a);     % works with matrices, vectors and scalars
round(3.66);    % round to nearest integer
sign(-2);
```

- You can find more of such functions there:

```
help elfun   % get a list of elementary functions
help specfun % get a list of special functions
```

2.d Exercises

1. Solve the linear system:

$$2x_1 + x_2 - 2 = 0 \tag{2.1}$$
$$x_2 - 1 - 3x_1 = 0 \tag{2.2}$$

2. Solve:

$$4.2x + 1.3z = \pi \tag{2.3}$$
$$x + 3.5y + \frac{\pi}{2}z = 1 \tag{2.4}$$
$$2.2y + 3z = 2 \tag{2.5}$$

3. Try to solve:

$$2x + \frac{1}{2}y + \frac{5}{2}z = 3 \tag{2.6}$$
$$3x + 3z = \frac{3}{2} \tag{2.7}$$
$$x + 2y + 3z = 2 \tag{2.8}$$

3 Programming

Scripts and functions are written in m-files. A m-file is a plain text file containing MATLAB commands and saved with the filename extension.m. m-files can be edited in any text-editor. However, Matlab has an in-built editor for editing m-files (Fig. 3.1). There is no need to compile either type of M-file. Simply type in the name of the file (without the extension) in order to run it.

3.a Control flow

- MATLAB has six relational operators and three logical operators:

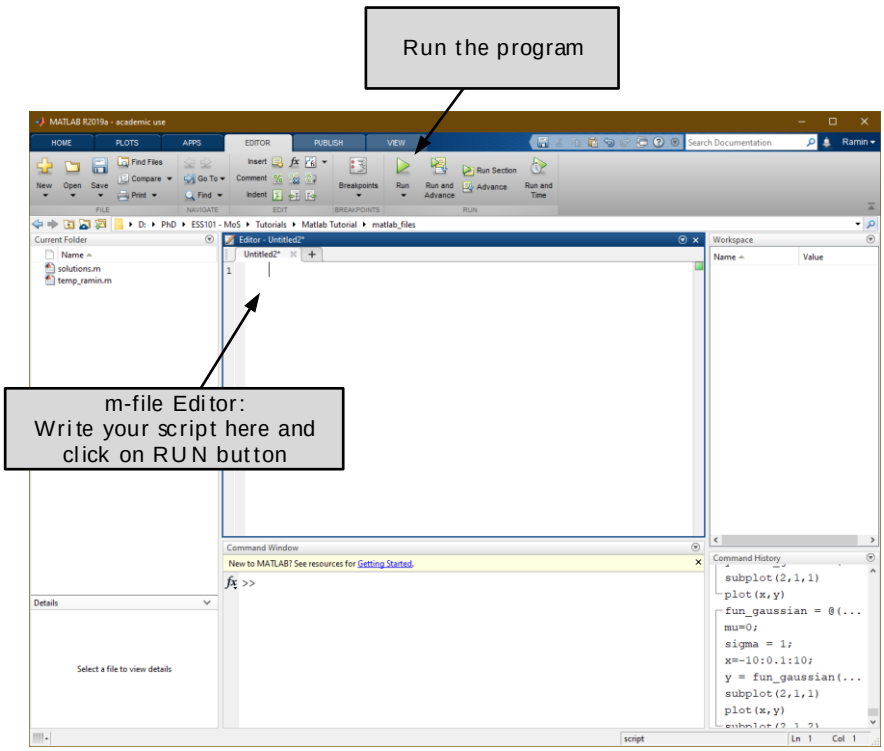

Figure 3.1: mFile editor

<	Less Than
<=	Less Than or Equal
>	Greater Than
>=	Greater Than or Equal
==	Equal To
~=	Not Equal To

~	not
&	and
|	or

- MATLAB has five flow control statements: *if* and *switch* for creating conditional execution of scripts and *for*, *while*, and *break* for creating loops.

```
% ------------------------------------------
% if:    ->     to have conditional running of part of the code
a = rand(1,1);
if a>0.5
    a=1;
else
    a=0;
end
% ------------------------------------------
% switch
a = 3;
switch a
    case 1
        b = 1;
    case 2
        b = 10;
    case 3
        b = 100;
    otherwise
        b = 0;
end
% switch, example 2
a = 'simulate';
switch a
    case 'simulate'
```

```
            disp('simulation is running ...');
        case 'predict'
            disp('prediction is running ...');
        otherwise
            disp('error!');
    end
    %--------------------------------------
    % for  -->   to create loops
    x = rand(1,10);     % create a vector containing rand numbers
    y = zeros(1,10);
    for i = 1:length(x)
        if x(i)>0.5
            y(i) = 1;
        else
            y(i) = -1;
        end
    end
    %--------------------------------------
    % while -> to create loops
    time = 0;
    simulationRun = true;
    while simulationRun
        time = time + 0.1;
        if time >= 10
            break
        end
    end
```

3.b Data import and export

Data can be loaded or saved using the `load` and `save` commands.

```
x=1;y=2;
save filename           % Saves all workspace variables to 'filename.mat'
save filename x,y       % Saves variables 'x' and 'y' in 'filename.mat'
%---------------------------------------
load filename           % Loads all variables from the file 'filename'
load x                  % Loads only the variable 'x' from the file
```

3.c 2D plotting

- Plotting the data in Matlab can be done with very few commands. The plot for the following code is shown in Fig. 3.2.

```
x=-pi:0.1:pi;
y1=sin(x);
plot(x,y1)
title(['Example of plot function'])
xlabel('xlabel text')      % to put label on x-axis
ylabel('ylabel text')      % to put label on y-axis
hold on;                   % to keep the previous plot on the figure while plotting new data
grid
y2 = cos(x);
plot(x,y2,'r--','LineWidth',2)
legend('sin(x)','cos(x)')
```

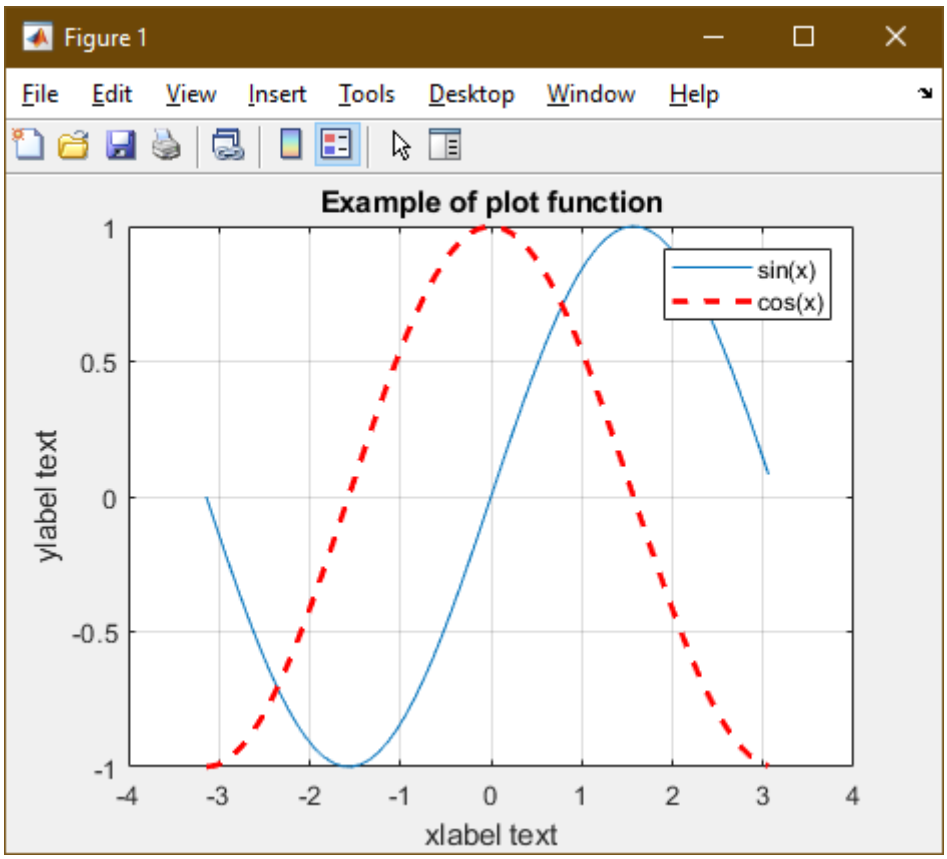

Figure 3.2: Plotting example

3.d Function definition

There are three approaches for defining new functions:

1. Functions can be defined inline, in your script:

```
functionName = @(a1,a2)(a1(1)+a1(2)–a2);
functionName([1;1],1);
```

2. Functions can be defined in a separate mFile:

```
function [c1,c2] = functionName(a1,a2)
    c1 = a1+a2;
    c2 = a1–a2;
end
```

3. Functions can be defined using the *matlabFunction* command. Please refer to the section for more details.

Note that `@functionName` is called a *function handle* and is used to refer to a function.

3.e Debugging

- A very usefull tool to debug the codes, especially in case of using functions, is breakpoint. Breakpoints can be set by clicking on the dash beside the number in m-file editor. Fig. 3.3 shows an example breakpoint. breakpoints can be set before the **run** step of the program. Then after running the program, Matlab would pause the script at specified location.

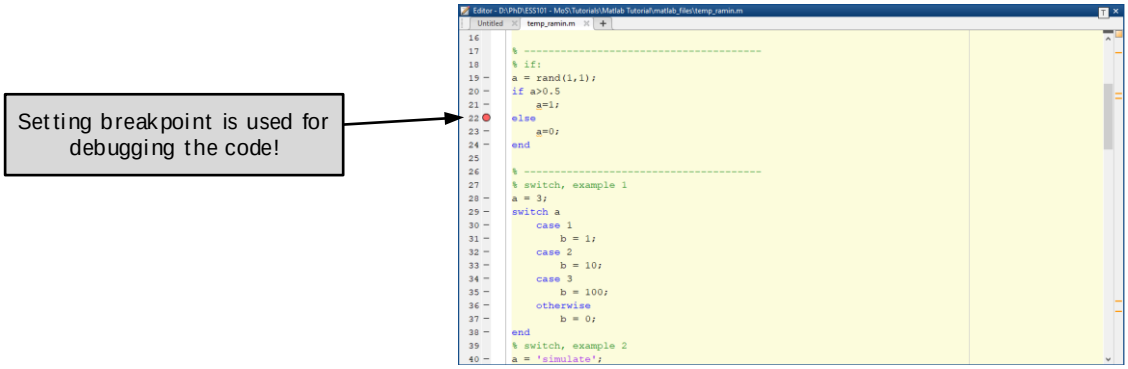

Figure 3.3: Create breakpoint for debugging the code

3.f Exercise

1. Write a function to print all prime factors of a given integer.

   (Hint: you are allowed to just use the trial division algorithm (ask your favorite search engine if you've never heard of it)).

2. Define the normal distribution function as an inline function and plot it for the given mean and variance. Please assign label to x-axis and y-axis and also enable the legend for the plotted data with given μ and σ^2 information.

$$f(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

   - $\mu = 0$, $\sigma^2 = 0.2$
   - $\mu = 0$, $\sigma^2 = 1$
   - $\mu = 2$, $\sigma^2 = 0.5$

4 Symbolic Toolbox

This section introduces the MATLAB symbolic toolbox, which enables you to easily write and use equations and functions of symbolic variables. In other words, you can use these tools to input functions of unknown variables of the form $f(x)$ directly into MATLAB. You can then compute their derivatives, integrate them, simplify them, solve equations with them or evaluate them for given values of x. It is a nice complement to the numerical computing abilities of MATLAB, and you will see that it is often quite useful when dealing with the equations of complex mechanical systems or the numerical integration methods of this course.

Therefore, we strongly recommend that you carefully read through this section if you are not already familiar with the symbolic toolbox. It might save you a lot of time when doing the assignments later !

4.a Defining symbolic variables

```
% Definition of symbolic vector size [n x 1]
n = 5;
x = sym('x',[n,1]);

% Definition of the same vector with assumption that it is real or positive
x = sym('x',[n,1],'real');
x = sym('x',[n,1],'positive');

% Definition of the same vector with formatted name
```

```
x = sym('x_%d',[n,1],'real');

% Definition of symbolic matrix [n x n]
M = sym('m',[n n],'real');

% Definition of symbolic matrix with formatted name
M = sym('m_%d%d',[n n],'real');

% Alternative way of defining symbols.
% Format: symx var1 var2 var3 assumption
syms p g k positive
```

4.b Symbolic manipulations

```
x = sym('x',[3,1],'real');
syms p k g real
P = [p;k;g];

% You can use matlab functions to build your symbolic expression to the
% desired level of complexity.
f = flipud(x).'*tanh(x);

% you can increase complexity in stages
f = sqrt(f*x(3)-k*x(2)*(cos(x(1))^2+sin(x(1))^2))^k;

% you can simplify expressions
f = simplify(f);

% you can make substitutions
fsubs = subs(f,k,x(3));         % individual substitution of k with x(3)
fsubs = subs(f,{p,g},{k,x(1)}); % substituting p for k, and g for x(1)
fsubs = subs(f,x,P);            % substitution of vector x with vector P

% you can differentiate w.r.t a specified variable
f_jacobian = jacobian(f,x);
f_gradient = f_jacobian';   % <---- so you don't forget :)

f_hessian  = hessian(f,x);
```

4.c Function generation

```
% You can generate functions that can be used in matlab from sybolic
% expressions. These can be either anonymous functions or generate matlab
% function files.

% Example:
n = 2;
x = sym('x_%d',[n,1],'real');
x0 = sym('x0_%d',[n,1],'real');
f = (x-x0).^2;

% Anonymous function: gives a handle to f(x,x0).
% NOTE: If yo do not put a vector argument within {}, the function will
% treat the individual vector elements as individual arguments.
f_num = matlabFunction(f,'vars',{x,x0});

x0n = rand(2,1);
xn  = rand(2,1);
```

```
fn = f_num(xn,x0n);

% You can also write function files (look in the same directory as this
% file for the result)
matlabFunction(f,'file','f_numFile','vars',{x,x0});

% you can call the function like this
fn = f_numFile(xn,x0n);

% You can specify in which folder you want the function to go like this
if not(isdir('functions'))
    mkdir functions % this makes the directory
end
matlabFunction(f,'file','functions/f_numFile','vars',{x,x0});

% you can have multiple outputs to one file. This can be useufull if you,
% for instance, want to get the derivative of a function with the function
% it self
Df  =  jacobian(f,x);
matlabFunction(f,Df,'file','f_numFileWithDerivative','vars',{x,x0});

[fn, Dfn] = f_numFileWithDerivative(xn,x0n);



% OBSERVE1: All symbolic variables need to be accounted for when the
% function is generated. That is, the following throws an error that x0_i
% must be included.
try
f_num = matlabFunction(f,'vars',x);
catch e
    warning(e.message)
end



% OBSERVE2: In general, the generate functions are much faster than the
% anonymous functions and the way to go!

% OBSERVE3: Using symbolic calculations and code generation removes two of
% the most common implementation errors: Manipulation errors (wrong signs
% etc.) and coding errors (simply writing the wrong thing).
% Dont be the person that keep on doing these errors after looking through
% this file.
```

4.d Exercises

1. Code the linearized version of the evolution function of the following non-linear state-space model (cf notes from lecture 2):

$$\dot{x} = f(x,u), \qquad f(x,u) = \begin{bmatrix} -x_1^2 - x_2 x_3 \tanh(u_1) \\ -x_1 x_2 u_1 u_2 \\ u_1 \cos(x_1) \sin(u_2) \end{bmatrix} \tag{4.1}$$

   around the point:

$$x_0 = [1,1,1]^T, \qquad u_0 = [1,1]^T. \tag{4.2}$$

   You can observe (by e.g. plugging in values of x, u close to x_0, u_0, and comparing f with its linear approximation) that the linear approximation of f performs well in a small region around x_0, u_0, but that it quickly deteriorates when moving away. If your approximation is vastly different from f around x_0, u_0, you should check your code !

2. Let f be a function of a scalar variable x. If f is 'smooth' enough, it can be approximated locally around a point x_0 by the first terms of its Taylor series:

$$f(x) \simeq f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2!}f''(x_0)(x - x_0)^2 + ... + \frac{1}{k!}f^{(k)}(x_0)(x - x_0)^k.$$

This formulation is quite powerful in the sense that you only need some information about the function and its derivatives at one point to know how it behaves in a neighbourhood of this point.

Try to plot a Taylor approximation (you can go up to order 3 or 4) of the function:

$$f(x) = \sin(2x) + \cos(x)e^{2x}$$

around the point $x_0 = 0$ and compare how different orders perform.

4.e Examples

These two examples use concepts that will be introduced later in the course (namely Newton's method and Runge-Kutta method) so you may want to skip them for now. But you might find them quite useful to look at for the assignments that will come later.

- Rootfinding with Newton's method

```
%% Rootfinding with newton.
syms x k real
% we want to find the root of
f = -atan(x)+atan(2*x) +3*x-pi;
Df = jacobian(f,x);
matlabFunction(f,Df,'file','f_newton','vars',{x});

x = linspace(-5,5);
fx = f_newton(x);

figure(1)
clf
hold on
grid on
plot(x,fx)


xk = 0; % initial guess
it = 0;
fprintf('#it\t|f|\t\t|dx|\n')
while 1
    it = it +1;
    [f,df] = f_newton(xk);
    plot(xk,f,'r*')

    xk = xk - f/df;
    fprintf('%2i\t%1.2e\t%1.2e\n',it,norm(f),norm(f/df))
    if norm(f)<1e-8
        fprintf('Solution is %2.16f\n',xk)
        break
    elseif it > 20
        break
    end
end
```

- Runge-Kutta method and the limitations of the symbolic toolbox

```
% While it is very convenient, Matlabs symbolic features can become VERY
% slow for functions of high symbolic complexity.
return
N  = 3;
dt = 1;

% System
x = sym('x',[3,1]);
f = [10*(x(2)-x(1)); x(1)*(28-x(3))-x(2); x(1)*x(2)-8/3*x(3)];
matlabFunction(f,'file','f_lorentz','vars',{x});

% ERK4 integrator
syms h real

k1 = h*f_lorentz(x);
k2 = h*f_lorentz(x+k1/2);
k3 = h*f_lorentz(x+k2/2);
k4 = h*f_lorentz(x+k3);
xn = x + 1/6*(k1 +2*k2 +2*k3 +k4);


matlabFunction(xn,'file','RK4_lorentz','vars',{x,h});

% N RK4 steps

hnum = dt/N;
xk = x;

% this will take some time to evaluate....
for i = 1:N
    xk = RK4_lorentz(xk,hnum);
end

% try to printout xk(x) if you dare... :)

% Taking the derivative of xk w.r.t x will take some time
Dxk = jacobian(xk,x);

% Writing the functions will take a looong time
matlabFunction(xk,Dxk,'file','N_RK4_lorentz','vars',{x});


% There are other tools that do this in an much more efficient way.
% A notable mention is CasADi, which is free of charge and available for
% MATLAB (and several other platforms).


% Good luck coding!
```