MATLAB PROJECT 0

This is an Individual Assignment which has to be completed in MATLAB and submitted as a PDF file through the Canvas page Assignments, Project 0. This Project is worth 10 points; 3 bonus points can be earned on this Project (see Page 19).

You will use Live Script and do all exercises within the Live Editor. Please see below a link to the video that explains how to use Live Editor and how to create a report (PDF file):
https://www.mathworks.com/videos/using-the-live-editor-117940.html?s_tid=srchtitle
The final script has to be generated by exporting the Live Script to PDF.

In this Project, you will learn how to use Live Editor and how to work with matrices in MATLAB. You will review some basic programming techniques. You will also learn how to create functions and work with them in MATLAB.
Please read the Instructions for Project 0 on the Canvas page "Assignments, Project 0" before you start working on this Project.

USING LIVE EDITOR

1) Open MATLAB application.
Note: If you would like to access MATLAB through UF Apps, please refer to the file "Accessing MATLAB off-campus" located on Canvas under Assignments, Project 0.

2) To begin working on Project 0 in MATLAB app, choose New Live Script under the Home tab or New →Live Script under the Live Editor tab in MATLAB. When the file opens up, choose the option "Text" under the Live Editor tab and type the title "Project 0". Then, type the heading "Exercise 1". Save the file with the name "MAS3114_Project0" – MATLAB saves it in the Current Folder with an extension .mlx.

You will do all your work on this Project within the Live Script except when creating a function – a function has to be created in a separate file, which will have an extension .m (please see Exercise 3 of the current project on a subject of creating a function in a file).

When working in the Live Script, **you may find it useful to insert Section Break after each exercise and, perhaps, after each part of an exercise if the exercise is long and contains multiple parts,** and run each Section separately by using the tab **Run Section** or **Run and Advance** instead of **Run** or **Run to End** (which would run the whole script) - it will allow you to run your script smoothly and avoid problems when exporting the Live Script to PDF.

3) If you would like to run a command and get an output, use the option "Code" in the Live Editor. If you do not want that output to be printed, type a semicolon after the command – MATLAB will run the command and store the output, but the output will not be displayed in the Live Script.

Important: The outputs that need to be displayed are usually specifies in the instructions. However, a general rule is that we always display the outputs listed in the name of the function and we usually suppress the intermediate outputs (in a "for loop" for example) unless it is specified otherwise in the Instructions. There may be some other outputs that need to be displayed.

4) If you type a long comment in your live Script, it is recommended to switch to the "Text" option – the text will be wrapped automatically. For typing a short comment or when marking parts of an exercise, such as (a), (b), (c) and etc., you can stay under the "Code" option, but type the sign % at the beginning of the line – when you run that section, the line that begins with % will not be executed. If you stay under the "Code" option for a long comment, please make sure that you will wrap the text manually to fit the page.

5) Editing the Live Script is comparatively easy: all you need to do is to make changes and re-run the Section. If you make a change in the Live Script, you will need to re-save the file.

NOTE on the Format of Project Instructions:
(1) The sign ** at the beginning of an Instructional sentence in a Project indicates that this task has to be perform in MATLAB by an appropriate command or a sequence of commands. This sign is not a MATLAB command – it is used in the Instructions to emphasize that the task has to be performed in MATLAB.
(2) The sign **%** at the beginning of an instructional sentence indicates that you need to type a comment in the Live Script.

Important Notes: 1) All exercises in this project have to be completed in the Live Script using the Live Editor, and the file has to be saved with the name "MAS3114_Project0".
2) Each exercise should begin with the line: **Exercise #.**
3) You should also mark down the parts of an exercise if it contains multiple parts, such as, Part1, Part2, (a), (b), and etc. - this will make grading of the Project easier.

Below is a link that you can use to access the resources available on MATLAB website:
https://www.mathworks.com/help/matlab/index.html

EXERCISE 1: WORKING WITH MATRICES IN MATLAB

Grading: Completed: 3 points; Not Completed/Partially Completed: 0 points

DESCRIPTION: In this Exercise, you will learn the basics on working with matrices in MATLAB. You have already created the Live Script "MAS3114_Project0". You will continue working on Exercises 1 – 4 within this file using Live Editor.

Part 1: Creating Matrices
A matrix is a rectangular array, and, in linear algebra the entries will usually be the numbers. The most direct way of creating a matrix is to type the numbers between the brackets using a space or comma to separate different entries in a row and using semicolon or **[Enter]** to create row breaks. For example, if you type

```
A=[-1 4 7 10; -2 5 8 11; -3 6 9 12]
```

and run the script, it will create a variable A in MATLAB and output it as a 3 by 4 matrix A.

Note: when we typed matrix A, we did not put a semicolon at the end of the line, which resulted in the output matrix A to be displayed. Remember, an output will be suppressed if you type a semicolon after the command. **Do not suppress any outputs in Exercise 1:** if some or all outputs **are not printed** in the submitted PDF file, Exercise 1 will be considered **Partially Completed**. Also notice that MATLAB is case sensitive.

**Type the following matrices (variables) in the Live Script and Run the Section:

$$A = \begin{bmatrix} -1 & 4 & 7 & 10 \\ -2 & 5 & 8 & 11 \\ -3 & 6 & 9 & 12 \end{bmatrix}, \quad B = \begin{bmatrix} -1 & 2 \\ -3 & 4 \\ -5 & 6 \end{bmatrix}, \quad X = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix}, \quad x = [2 \;\; 4 \;\; 6 \;\; 8], \quad y = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \end{bmatrix}.$$

You can save the variables in a file by typing, for example, *save* Project0. The variables can be reloaded in a future session by typing *load* Project0. As an alternative, you can re-load the variables in a new session by re-running the Sections of the Live Script where the variables were created.
Please also check out the small icons in the upper right corner of the Live Editor window: if you choose an option "Output inline", the part of the project that have been completed up to this point will look similar to the Example below; if you choose "Output on right", it will change the display accordingly. It may be easier to do the work in the Live Script when the outputs are shown "on the right", but, after the Live Script is exported to PDF, the outputs will be automatically displayed "inline".
Below is an Example how your Live Script may look after you typed the titles, headings, and the matrices and pressed Run Section (an option "Output inline" was chosen here).

Project 0

Exercise 1

Part 1. Creating matrices

```
clear
A=[-1 4 7 10; -2 5 8 11; -3 6 9 12]
```

```
A = 3×43×4
    -1     4     7    10
    -2     5     8    11
    -3     6     9    12
```

```
B=[-1 2; -3 4;-5 6]
```

```
B = 3×23×2
    -1     2
    -3     4
    -5     6
```

```
X=[1;3;5]
```

```
X = 3×13×1
     1
     3
     5
```

```
x=[2 4 6 8]
```

```
   x = 1×41×4
       2     4     6     8
```

```
y=[0;1;2;3]
```

```
   y = 4×14×1
       0
       1
       2
       3
```

Note: the command `clear` at the beginning of the script will clear out the saved variables when you re-run the script and it can be used to prevent overwriting the variables.

When A is a matrix, the command `size(A)` returns a vector whose entries are the numbers of rows and columns in A, respectively.
Output the sizes of the matrices A, B, **X, **x**, **y** in MATLAB.
% Are the matrices **x** and **y** of the same size? Explain please.

**Run the commands
```
size(A,1)
size(A,2)
```
%Comment on the outputs for these two commands.
Note: If you need a help with MATLAB command which name you know, you can type in the Command Window in MATLAB `help` followed by the name of the command and press [Enter]. You will be able to erase the reviewed information by typing `clc` and pressing [Enter].

Part 2. Accessing particular matrix entries and changing entries
**Execute each of the lines below.
% Briefly describe the results after executing each of the lines.
Note: A new matrix F is created here and it has been modified over execution of some of the commands.
```
A, A(1,3), A(:,3), A(2,:), A(:, 1:3)
A, A([1  2], [2  4])
F(:,4)=[-1  1  -4  3], F([1  3], [2  3]) = [1 -3; 2 -5]
A, F, F([2  3], :) = A([1  3], :)
F, F(:, [1 2])=F(:, [2 1])
F, F(2:4,:), F(end,:)
```

**Compose the following commands, run them, and display the outputs.
Note: the matrix F will be modified after executing parts (I) and (II).
(I) Display **y**, F, and display the output after replacing row 2 of F with the vector **y**.
(II) Interchange rows 1 and 3 in F.
(III) Output a new matrix F1 formed by the first three columns of F.
(IV) Output the last column of F and assign it to the vector **b**.
Hint: do not insert Section Break between Parts 1 and 2: the command `clear` at the beginning of Part 1 will prevent from “unexpected” over-writing of the matrix F when re-running Part 2.

Part 3. Pasting blocks together (concatenation)
** By using matrices **A, B, X,** and **x**, output the following block matrices in MATLAB and display them:

```
A,B,[A  B],[B  A]
A,X,[A  X]
A,x,[A ; x]
blkdiag(B,A)
```

%Write a comment how the matrices were created.
Hint: You may run in the Command Window the following commands (do not include it in the Live Script):

```
help horzcat
help vertcat
help blkdiag
```

Part 4. Some special functions for creating matrices:
**Execute each of the lines below:

```
eye(5), zeros(3,4), zeros(2), ones(3,2), ones(5)
diag([1 2 5 6 7]), diag([1 2 5 6 7],1), diag([1 2 5 6 7],-2)
B, diag(B), diag(diag(B))
A, triu(A), tril(A)
A,triu(A,1), triu(A,-1)
A,tril(A,-1),tril(A,2)
```

%Write a brief comment on each of the created matrices.

**Run the following commands in the Live Script:

```
magic(5), help magic
hilb(5), help hilb
pascal(4), help pascal
```

%Describe briefly how the matrices `magic`, `hilb`, and `pascal` are generated.

**Use appropriate MATLAB commands to output and display matrices C, D, and E:

$$C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad D = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 5 & 0 \end{bmatrix} \qquad E = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Part 5. Using the colon (vectorized statement) to create a vector with evenly spaced entries
** Execute the following commands in MATLAB:

```
V1=1:7, V2=2:0.5:6.5, V3=3:-1:-5
```

** Type and run the commands using a vectorized statement that would create the following vectors:
V4=[-5 - 4 - 3 -2 -1 0 1] , **V5**=[10 7 4 1 -2] , **V6**=[5 4.5 4 3.5 3 2.5 2],
V7: the numbers from 0 to -3 spread by 0.5 apart.

Part 6. Format commands
** Type the following commands in MATLAB and execute them:

```
%(a)
r=sqrt(2)
```

```
sym(r)
format long, r
format rat,r
format short,r
format
r
disp(r)

%(b)
P=0.0000632178
format long, P
format short, P
```
% Analyze the outputs and write a short comment on the ways MATLAB formats the numbers.

Note: By default, we use `format short` (or `format`) to print the numbers unless it is specified differently in the instructions. In order to restore the default format mode at any time, we type

```
format
```
To suppress the extra line feeds, we can type the command

```
format compact
```

Part 7. Operations on Matrices

The operations of matrix addition and scalar multiplication are defined by + and * signs, respectively.

Note: some variables in the commands below were input earlier. If you don't have them saved, you can simply re-run your Live Script from the beginning to have them re-loaded.

** Execute the following commands in MATLAB:
```
A, A+A
A, 3*A
T=eye(4), U=magic(4), T+U
x, y, x+y
```
%Write a comment how the matrix `x+y` has been created.

Note: the MATLAB operation `x+y` is not consistent with the operation of matrix addition in Algebra: matrices x and y are not of the same size and in Algebra they cannot be added.

**Execute the commands:
```
A, A', transpose(A)
```
% Write brief comments on the outputs.

Part 8. Matrix multiplication

Matrix operation **P*Q** is the usual matrix multiplication as it is introduced in Algebra course.

Note: the number of columns in matrix P has to be equal to the number of rows in Q (the inner dimensions have to match).

**Run the following in the Live Script.
```
x, y, x*y, y*x
```
%Comment in the Live Scrip how the entries of the matrices `x*y and y*x` were calculated.

Hint: read section Matrix Multiplication in the Lecture notes (Module 11) or in the textbook.

The operation **.*** is entry-by-entry multiplication (vs. usual matrix multiplication *).
** Input the matrices and execute the commands:

```
P=[1 2 3 4; 1 1 1 1], Q=transpose(P)
P*Q, P.*P, P.^2
```

%Explain how the entries of the matrices `P*Q, P.*P, P.^2` were calculated.
Try to run a command `P*P`.** See why you are receiving an error message. Delete the command and re-run the Section.

**Input a matrix $G = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

and execute the commands **G^2** and **G*G**. Verify that the outputs of these commands match by running two logical comparison statements:

```
G*G==G^2, isequal(G*G,G^2)
```

Part 9. Creating matrices with random entries
The command **rand** creates numbers between 0 and 1 that look very random. The command **randi** creates array of random integers.
**Run the commands:

```
 rand(4),   rand(3,4),  randi(100,2),  randi(10,2,4),  randi([10 40],2,4)
```

% Describe the dimensions of each matrix and the distribution set of random numbers for each of the commands.
Execute the commands: 5*rand(3)** that creates 3 by 3 matrix with random values between 0 and 5; -3+5***rand(3)** shifts the range of values that becomes between -3 and 2.
**Write and execute commands that display matrices of sizes 2 by 4 with the specified random values in the following intervals:
random numbers between -1 and 5;
random integer numbers in the range from -20 to 20.

EXERCISE 2: PROGRAMMING TECHNIQUES
Grading: out of 2 points

**Input and display the following matrices that will be used in this Exercise:

```
M=3*ones(3), N=[0 0 3; 0 3 0; 0 0 3], L=3*ones(3,2)
```

Part 1. Logical Operations
**Run the following commands:

```
isequal(M,L), M==L
```

%For which of the two commands MATLAB returns an error message? Explain why. Delete that command and re-run the section.
**Run the following.

```
N==0, N==3
```

%Have you received error messages? Explain why not.
**Run the following commands that compare matrices of the same size:

```
M==N, isequal(M,N), ~isequal(M,N)
```

%Write a comment on the ways how each of the commands compares matrices.

**Run the following commands:

```
M==N, M~=N
```

%Write a short comment on the outputs.

For more information on logical operations, refer to the following link:
https://www.mathworks.com/help/matlab/logical-operations.html

Part 2. Conditional Statements & Logical Commands

**Run the following conditional statement in the Live Editor:

```
N=[0 0 3; 0 3 0; 0 0 3]
if all(N,'all')
    disp('all entries of N are nonzero')
elseif ~any(N,'all')
    disp('all entries of N are zero')
elseif any(N) & any(N,2)
    disp('all columns and all rows of N are nonzero')
elseif any(N)
    disp('all columns are nonzero but there are zero rows')
elseif any(N,2)
    disp('all rows are nonzero but there are zero columns')
else
    disp('N has both zero and non-zero columns and rows')
end
```

Note: a row/column is called nonzero if there is at least one nonzero entry in that row/column.

%Write a short comment in the Live Script on each of the logical commands that is used in the statement above. You may need to run in the Command Window (not in the Live Script)

```
help all
help any
```

to obtain more information on these commands. (You can clear out the Command Window by typing clc and pressing "Enter").
Please notice that there are some differences between the ways the logical commands above work on the matrices vs. vectors. It is also important to remember that, if the output of a logical command is an array, then MATLAB evaluates the statement as "True" only on condition that all entries in the array are non-zero, otherwise, it evaluates the statement as "False".

**Run the following in the Live Script:

```
Q=[1 0 0; 2 3 0; 0 0 0]
%(a)
if any(Q,"all")
    disp('Q has nonzero entries')
else
    disp('All entries of Q are zero')
end
%(b)
if any(Q)
    disp('Q has nonzero entries')
else
    disp('All entries of Q are zero')
```

```
end
%(c)
if Q~=0
    disp('Q has nonzero entries')
else
    disp('All entries of Q are zero')
end
%(d)
if any(any(Q))
    disp('Q has nonzero entries')
else
    disp('All entries of Q are zero')
end
%(e)
if ~all(Q==0,"all")
    disp('Q has nonzero entries')
else
    disp('All entries of Q are zero')
end
%(f)
if ~all(Q==0)
    disp('Q has nonzero entries')
else
    disp('All entries of Q are zero')
end
```

% Write a comment which of the conditional statements (a)-(f) are incorrect for determining if a matrix has non-zero entries and explain why.
%What if Q were a vector? Determine which of the conditional statements would be incorrect?

Part 3. <u>"For" Loops and Vectorized Statements</u>

**Run in the Live Script the code given below.

```
format
A=zeros(3,4);
for i=1:3
    for j=1:4
        A(i,j)=i+j-1;
    end
end
A
```

Notice that the matrix A, first, was pre-allocated as `A=zeros(3,4);` and, then, its entries were re-calculated using a double "for" loop. The intermediate outputs in the loop are suppressed. The resulting matrix A is displayed after the "for" loop terminates.

%Write a comment that would give more details on how A was generated by using the "for" loop.

**Run the two codes given below. Both generate the same matrix whose 5 columns are the elementwise powers of the column vector z in the descending order from 4 to 0 according to the degree. The code in (a) uses a "for" loop and the code in (b) – a vectorized statement.

```
z=(1:4)';
m=size(z,1);
n=5;
%(a)
ZF=zeros(m,n);
for i=1:n
    ZF(:,i)=z.^(n-i);
end
ZF
%(b)
ZV=zeros(m,n);
n=size(ZV,2);
i=n-1:-1:0;
ZV=z.^i
```

% Write a comment how the matrices ZF and ZV were generated and what is the difference between the codes in (a) and (b) for creating the same matrix.
Hint: you may need temporarily display the intermediate outputs in the "for" loop by removing the semicolon after the command `ZF(:,i)=z.^(n-i);` and analyze the way it runs. Please do not forget to put the semicolon back in after you are done and re-run the script to have the intermediate outputs suppressed.

**Input in the Live Script
`format rat`
and generate and display the matrix `hilb(5)` in three ways:
(a) by running `H=hilb(5)`
(b) by using a double "for" loop – output and display the output as a matrix HF;
(c) by using a vectorized statement – output and display the output as a matrix HV.
Hint: for help with part (c), see Exercise 1 (**Part 5** and the command **x + y** in **Part 7)**. You can also run (in the Command Window) `help hilb` to see how the Hilbert matrices are generated (do not include this in your Live Script).
**After you have generated the three matrices, run a conditional statement given below to check if the matrices match:

```
if isequal(H,HF,HV)
    disp('Everything is fine!')
else
    disp('Oh no! There is a bug in my code!')
end
```

Write a set of commands in the Live Script to generate a Pascal matrix of size 5 x 5.
**Begin with the commands:

```
format
P=ones(5);
```

and then use a "for" loop to re-calculate the entries of P in order to output a Pascal matrix of the size 5 x 5. Display the output matrix P.
Hint: For help with this part, you may need to run (do not include it in the Live Script):
`help pascal`
and you may also need to review the topic Pascal Triangle.

**Run a conditional statement given below to verify that your output for P matches the output of MATLAB function `pascal(5)` (please make sure that you will receive the message):

```
if P==pascal(5)
    disp('the pascal matrix is constructed correctly')
end
```

Write a set of commands in the Live Script that will scale a given matrix B, whose entries are positive numbers, to a matrix C whose columns are probability vectors, that is, the sum down each column of C has to be 1.
**Begin with the lines below which input a matrix B, output the row vector S of the sums down each column of B, and pre-allocate a matrix C:

```
format
B=magic(4)
S=sum(B);
C=ones(4);
```

**Continue with composing a single "for" loop that will output matrix C by scaling each column of B by the reciprocal of the corresponding entry of S. Display the matrix C.
**Verify that C is constructed correctly: output the vector S1 of sums down each column of C and compose a conditional statement that will check if each entry of S1 is 1. If it is the case, program an output message:

```
disp('the columns of C are probability vectors')
```

Please be sure that you will receive this message when running your code fragment!

Use the following link for more information on the loops and conditional statements:
https://www.mathworks.com/help/matlab/control-flow.html
You may also find it useful to visit this link on MATLAB website:
https://www.mathworks.com/help/matlab/operators-and-elementary-operations.html

EXERCISE 3: CREATE AND CALL A FUNCTION IN MATLAB

Grading: out of 2 points

In this exercise, you will learn how to create a function in a file in MATLAB. The first function that you will create has a name `closetozeroroundoff`. We will be working with this function on a regular basis and you should have it saved in your Current Folder in MATLAB. This function is needed because of the round-off errors in MATLAB calculations. The rule that we employ here is that we consider an entry of an array equal to zero if the absolute value of that entry is less than 10^(-p), where the integer number p is a parameter that defines a margin for an error which will be specified for each particular case. After you have this function created and saved in your Current Folder in MATLAB, you can run it in your Live Script in the same way as you run a MATLAB function, such as, *size*, *transpose*, and etc.

**To create a function `closetozeroroundoff` in a file, click on New → Function in the Live Editor tab. It will open up a file "Untitled" in your Live Editor window. Delete the preset lines and copy and paste the code given below:

```
function B=closetozeroroundoff(A,p)
A(abs(A)<10^-p)=0;
B=A;
end
```

Save the file with the name of the function `closetozeroroundoff`
The file will have an extension .m
Note: you will have a separate file for each function that you create.

**Run the command
```
type closetozeroroundoff
```
or
```
type closetozeroroundoff.m
```
Either of these commands will print the function `closetozeroroundoff` in your Live Scrip.

Note: in order to run a function, which we created in a file and saved in the Current Folder in MATLAB, we do not need to have it "typed" previously – we use the command `type` to print the code in the Live Script.

**Next, type in the Live Script (or copy and paste) and run the code fragment given below:
```
format
p=1
H=hilb(8);
i=1:8;
G=(-1).^(i+i').*H
X=closetozeroroundoff(G,p)
```
%Write a comment on the way the function `closetozeroroundoff` worked on the matrix G to produce X. Describe which entries of G were replaced with zeros.

**Create another function in a file which is called `span`. The code is below:
```
%{
For an m x n matrix A, the function determines if a vector b is in the
span of the columns of A, that is, if b can be represented as a linear
combination of the columns of A. This is possible if and only if b is
in R^m and the system Ax=b has a solution; moreover, the entries of
a solution x are the weights on the columns of A to produce vector b.
%}
function x=span(A,b)
format
[m,n]=size(A);      %outputting the size of A
fprintf('matrix A has %i rows\n',m)
x=[];               %pre-allocating x
if length(b)~=m     %checking if the vector b is in R^m
    sprintf('the dimensions mismatch: vector b is not in R^%i',m)
    return          %terminating the code (the dimensions mismatch)
end
fprintf('the dimensions match: vector b is in R^%i\n',m)
%outputing variables:
rankA=rank(A);
aug=[A b];
%testing if b is in the span of the columns of A:
if rankA==m            %there is a pivot position in every row
    fprintf('the columns of A span the whole R^%i\n',m )
    disp('thus the vector b is in the span of the columns of A')
elseif rank(aug)==rankA
    fprintf('the columns of A do not span the whole R^%i\n',m)
    disp('but the vector b is in the span of the columns of A')
```

```
else
    disp('the vector b is not in the span of the columns of A')
    return      %terminating the code (the system Ax=b is inconsistent)
end
%outputting a solution using different approaches depending on the rankA
if rankA==m   %there is pivot position in every row
    x=A\b;    %outputting a solution
else          %rank of A is less than the number of rows
    R=rref(aug);          %outputting the reduced echelon form of matrix aug
    R=R(1:rankA,:);       %deleting the zero rows on the bottom
    x=R(:,1:n)\R(:,end); %outputting a solution
end
%checking if x is not a solution of Ax=b (within the given margin):
if any(closetozeroroundoff(A*x-b,7))
    disp('check the code!')
    x=[]     %re-assigning to x an empty output and displaying it
    return   %terminating the code (x is not calculated correctly)
end
%making the entries of x that are in the range of 10^-7 from 0 show as 0:
x=closetozeroroundoff(x,12);
%checking if the solution is unique and diplaying it:
if rankA==n    %there is a pivot position in every column (unique solution)
    disp('the unique vector of weights on columns of A to produce b is')
    disp(x)    %displaying the solution
else           %there are infinitely many solutions
    disp('a vector of weights on columns of A to produce b is')
    disp(x)    %displaying a solution
end
end
```
Note on the Format of the Function: The comment lines in the body of the function, which begin with the sign %, are optional – the main reason for placing them here was to explain the meanings of the lines of the code. You may have some similar comments or omit them entirely in your future functions. However, the messages that are output by the commands `disp`, `fprintf`, and etc., have to be present in the code in the way it is specified in the Instructions. It is also important to keep the correct alignments of the statements in the body of the function – it helps to verify the code if there is an error and it makes the code easy to follow.

**Print the function `span` in your Live Script using the command:

```
type span
```

**Then, type in the Live Script (or copy and paste) and Run Section:

```
%(a)
A=magic(3), b=rand(3,1)
x=span(A,b);
```

Note: Here we placed a semicolon at the end of the command that runs the function in order to suppress extra outputs for the variables located in the name of the function since the outputs for those variables have been already coded within the function to be displayed - without a semicolon, MATLAB will automatically display them one more time at the end. Placing or omitting a semicolon at the end of a command that runs the function gives a control over displaying the outputs for the variables that are located in the name of the function.

Important Note: The outputs for the variables located in the name of the function have to be always assigned and displayed. There may be some other outputs that are required by the Instructions to be displayed. However, all required outputs have to be displayed only once. **Having extra outputs displayed or missing required inputs/outputs will result in a loss of points on a Project**.
When empty matrices are assigned as outputs for the variables, those outputs are normally suppressed unless it is stated otherwise in the Instructions.

Next, run the logical commands below – they are intended to check if the output x from part (a) is a solution of Ax**=**b** (you need to copy and paste the lines below into the Live Script and press **Run Section** several times):

```
A*x==b
closetozeroroundoff(A*x-b,7)==0
```

%Compare the outputs of the two commands. Which one, in general, does not confirm that x is a solution of A**x**=**b**. Explain a reason why we need to use the function `closetozeroroundoff` when comparing matrices; also notice that the function `closetozeroroundoff` is run on the matrix which is the difference between the two matrices that we compare.

**Run the function `span` on the variables given below. You can just copy the text below, paste it into your Live Script, and press Run Section.

```
%(b)
A=magic(4),b=ones(3,1)
x=span(A,b);
%(c)
A=[magic(3);ones(1,3)],b=rand(4,1)
x=span(A,b);
%(d)
A=[magic(3);ones(1,3)],b=sum(A,2)
x=span(A,b);
%(e)
A=[magic(3),ones(3,1)],b=ones(3,1)
x=span(A,b);
%(f)
A=magic(4),b=ones(4,1)
x=span(A,b);
```

EXERCISE 4: WORKING WITH A FUNCTION IN LIVE EDITOR

Grading: out of 3 points

In this exercise, you will create your own function in a file and run it on the given sets of variables.
Theory: We will work here with geometrical interpretation of the solution set of a system of two equations in two variables:

$$\begin{aligned} a_1x_1 + b_1x_2 &= c_1 \\ a_2x_1 + b_2x_2 &= c_2 \end{aligned} \qquad (1)$$

The system in (1) can be written in a matrix form as

$$A\mathbf{x} = \mathbf{c}, \qquad (2)$$

where

$$A = \begin{bmatrix} a_1 & b_1 \\ a_2 & b_1 \end{bmatrix},\ \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \text{ and } \mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}.$$

Under the condition that not all coefficients in each equation in (1) are zero, that is, the coefficient matrix A does not have zero rows, each equation in (1) represents a line in $\mathbf{R}^2$. From Calculus we also know that the rows of the coefficient matrix A are normal vectors to the corresponding lines.
There are three possibilities of mutual location of two lines in $\mathbf{R}^2$: they may intersect at one point, they may be coincident, or they may have no points in common. With respect to the solution set of the system (1), the system may have only one solution, infinitely many solutions, or no solutions.

If a linear system has one solution or infinitely many solutions, it is called consistent. For a consistent system (1), geometrical form of the solution set depends on whether there is a free variable in the system or not. In the case if there is a free variable, the solution set represents a line (please refer to the Lecture for Module 5 where we discuss a parametric representation of the solution set); moreover, one of the equations of the system is a scalar multiple of the other and it can be discarded when looking for the solution set since it has the same solutions as the other equation. If there is no free variable, a consistent system has a unique solution and the lines are intersecting at one point. In order to find a solution, we can employ MATLAB backslash operator, \ , which outputs a solution of a consistent system of the form $B\mathbf{x} = \mathbf{d}$ as $\mathbf{x} = \mathrm{B}\backslash\mathbf{d}$. We can also compute the angle between two intersecting lines by a formula from Calculus.
If the system (1) is inconsistent, that is, there are no solutions, the two lines have no points in common, that is, they are parallel. For an inconsistent system, we will not be looking for any "type" of a solution, however, we can compute the distance between the parallel lines.

**Create a function in a file (see Exercise 3 of this Project) that begins with:
```
function [solution,x] = lines(A,c)
format
solution=[];
x=[];
rankA=rank(A);
```
The inputs are a 2 x 2 matrix A and a 2 x 1 vector c. The outputs that are listed in the name of the function will either stay empty if the system (2) is inconsistent, or, for a consistent system, the output `solution` will give a geometrical description of the solution set and the output `x` will be just one point (or the only one point) that belongs to both lines. We have also output the rank of A, `rankA`, which is an important characteristic of a matrix that will be employed later.

First, your function has to check if the input A is geometrically valid, that is, if each of the two equations of the system (1) represents a line (please refer to the **Theory** above). You can use here a conditional statement with a logical command `any(A,2)`.
**If an input A is geometrically valid, output the message:
```
disp('each equation of the system represents a line)
```
Otherwise, output the message:
```
disp('the input A is geometrically invalid')
```
and terminate the code.
Note: to terminate a code, you can use the MATLAB command `return`

For a valid input A, your function will continue.
**Output a new matrix `aug`, which is the augmented matrix of the system (2), as below:

```
aug=[A c];
```

Note: if there is a semicolon after a suggested command, the output is meant to be suppressed.

Continue with setting up a conditional **If … Else** statement that will check on the two general cases: whether the system (2) is **consistent** or whether it is **inconsistent.**
Hint: use here the Rouche-Capelli Theorem (see Lecture for Module 2) and employ the variable `rankA` and the command `rank(aug)` to set the condition.

If (the system is consistent),
output the message below:

```
disp('the system is consistent')
```

and continue with the tasks as follows:
First, code two possible cases for the solution set:
I. the solution set has only one point;
II. the solution set is a line.

Use here the variable `rankA` to set up a "**nested" conditional statement**. Remember, the rank of a matrix A is the number of pivot positions (or pivot columns) in A. A system (1) does not have a free variable if and only if all columns of A are pivot columns, or equivalently, `rankA` is equal to the number of columns of A.
****If there is no free variable**, output:

```
disp('the lines have only one point in common')
solution='point'
```

and continue your code with calculating the angel between intersection lines as below:
**First, output the normal vectors of the lines as below:

```
n1=A(1,:);
n2=A(2,:);
```

**Next, output (and display) the angle (in degrees) between the lines by using the commands:

```
cos_theta=abs(dot(n1,n2))/(norm(n1)*norm(n2));
if cos_theta >= 1
   theta=0;
else
   theta=acosd(cos_theta);
end
fprintf('angle between the intersecting lines is %i°\n',theta)
```

Note: The formula for `cos_theta` is a one from Calculus. The function `abs()` is used here because the angle `theta` between two intersecting lines has to be in the range from 0 to 90 degrees. The conditional statement above outputs the value of the angle `theta` (in degrees) counting a possibility of a round off error that may take `cos_theta` out of the range of the function cosine (in real analysis).
Else, if there is a free variable (the solution set represents a line), output:

```
disp('the lines are coincident')
solution='line'
aug=aug(1,:);
```

The last line of the code fragment above has discarded the second equation of the system (2) since it is a scalar multiple of the first one by outputting the new augmented matrix `aug`.
This is the end of the "nested" conditional statement.

For both cases **I** and **II**, continue your code with outputting a solution (or the only one solution) by using MATLAB backslash operator, \ . Proceed as below:

**Include the following line into your code:
```
x=aug(:,1:2)\aug(:,end);
```
**Then, verify that the output x is, indeed, a solution of the system (2), or equivalently, that x is a point that belongs to both lines. Proceed with composing a conditional statement using the function `closetozeroroundoff(A*x-c,7)`. If this function outputs the zero vector, display:
```
disp('point x is common to the two lines')
x
```
Otherwise, code a message similar to the one below and assign (and display) the empty output for x:
```
disp('x is not common to the lines? That cannot be true!')
x=[]
```
<u>Important</u>: receiving the last message and the empty output for x after running the function `lines` on the choices (a)-(g) below should prompt you to make corrections in your code.

****Else** (the system is inconsistent)

output a message:
```
disp('system is inconsistent and the lines have no points in common')
```
and continue your code with calculating <u>the distance</u> between two parallel lines.

To proceed with this task, we need to scale the matrix `aug` to have the normal vector to each line to be a unit vector. Proceed as follows:
**Place the following set of commands into your function:
```
N=ones(2,1);
for i=1:2
    N(i)=norm(A(i,:));
end
aug=aug./N;
```
Here, we first pre-allocated a column vector N and, then, re-assigned to its two entries the 2-norms of the corresponding rows of the matrix A by using a "for" loop. Next, we scaled rows of the matrix `aug` by using the command `aug=aug./N`. This resulted in each row from column 1 to column 2 of the matrix `aug` to become a unit vector.

**Next, output the normal (unit) vectors of the two parallel lines as follows:
```
n1=aug(1,1:2);
n2=aug(2,1:2);
```

<u>Note</u>: the unit vectors `n1` and `n2` are either the same or they are a (-1) multiple of each other. Specifically, if `dot(n1,n2)` outputs the value 1, the unit normal vectors are the same, otherwise, if the value is a (-1), they are opposite to each other.

Continue with outputting the distance, d, between two parallel lines by a known formula from Calculus adjusted to our normal vectors:

**Place the following commands into your function:

```
s=sign(dot(n1,n2));
d=abs(aug(1,end)-s*aug(2,end));
```

Note: the function `sign()` is a MATLAB function which, when applied to a non-zero number, outputs 1 if the number is positive and outputs (-1) if the number is negative.

**Finally, display the distance d with a message:

```
fprintf('the distance between the parallel lines is %i\n',d)
```

This is the end of the If … Else statements and the end of the function `lines`.

**Print the two functions in your Live Script (the function `closetozeroroundoff` was created in Exercise 3 of this Project) by using commands:

```
type lines
type closetozeroroundoff
```

**Run the function on the choices (a)-(g) as indicated below:

```
%(a)
A=[1 2;0 0],c=[5;-2]
[solution,x] = lines(A,c);
%(b)
A=[0 2;0 5],c=[5;1]
[solution,x] = lines(A,c);
%(c)
A=[0 2;0 5],c=[2;5]
[solution,x] = lines(A,c);
%(d)
A=pascal(2),c=[-1;3]
[solution,x] = lines(A,c);
%(e)
A=[1 2; 2 4],c=ones(2,1)
[solution,x] = lines(A,c);
%(f)
A=[1 2; 2 4],c=sum(A,2)
[solution,x] = lines(A,c);
%(g)
A=hilb(2),c=randi(5,2,1)
[solution,x] = lines(A,c);
```

This is the END of Exercise 4

PLEASE SEE NEXT PAGE FOR IMPORTANT INFORMATION ON PROJECT 0

Some Additional Important Comments on Project 0

Bonus! (3 points)

A student can earn a **3-point bonus** for completing Project 0 perfectly. This includes:
1) All exercises are completed as required.
2) The file is submitted for grading before the due date.
3) No missing inputs/outputs.
4) All required inputs, outputs, and messages are displayed.
5) All required functions are printed.
6) The command lines in the body of the created functions are aligned properly.
7) No extra conditions or commands.
8) No extra inputs/outputs.
9) No repetitions in the code.
10) All necessary comments are present.
11) The formatting is as required.

Publishing file

After you are finished working on the file **MAS3114_Project0** in the Live Editor, you need to export it to PDF file. In order to create a PDF file, click on Export under the Live Editor tab and choose the option Export to PDF. Save the PDF file in MATLAB with the same name.

Important: Next, you need to download the PDF file MAS3114_Project0 onto your computer and submit it for grading prior to the deadline through the Canvas page "Assignments → Project 0". If you are using MATLAB through UF Apps and need help on working with the files and saving them onto your computer, please refer to the file **Frequently Asked Questions on UF Apps (PDF)** which is located under the Resources on Canvas page "Assignments → Project 0". If you need more help with these tasks, **please call the UF Computing Help desk**.

Very Important: Before submitting your PDF file for grading, it is very important to verify that you are submitting the correct file and that all required inputs/outputs in your file are present and displayed properly.
The file will be graded as submitted!
"Late" submission is accepted one day past the deadline for a 1-point penalty.
Project 0 cannot be submitted or resubmitted after the final deadline.
For more details on submission and grading, visit Canvas page "Assignments → Project 0".