MATLAB PROJECT 1

Please read the **Instructions** located on the Assignments page prior to working on the Project.

BEGIN with creating Live Script **Project1.**

Note: All exercises in this project will be completed in the Live Script using the Live Editor. Please refer to the MATLAB video that explains how to use the Live Script:
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.

Each exercise has to begin with the line
Exercise#
You should also mark down the parts such as (a), (b), (c), and etc. This makes grading easier.
Important: we use the default `format short` **for the numbers in all exercises unless it is specified otherwise**. We do not employ `format rat` since it may cause problems with running the codes and displaying matrices in Live Script. If `format long` has been used, please make sure to return to the default `format` in the next exercise.
.
EXERCISE1 (4 points) **Difficulty: Easy**
Jordan Block is a square $n \times n$ matrix ($n \geq 2$) with a scalar r on the main diagonal and 1's on the diagonal right above it. All other entries are zero.
**Create a function in a file that begins with

```
function J=jord(n,r)
```

which, when possible, produces an $n \times n$ Jordan matrix with a scalar r on its main diagonal.

**First, the function will verify whether $n > 1$ and n is an integer number. You can use a built-in MATLAB function such as `mod` or `fix` .
**If n does not satisfies the above conditions, the code has to assign an empty matrix to J and display a message that `'Jordan Block cannot be built'` - you can use `disp, sprintf,` or `fprintf` commands.
**If n is an integer number and $n > 1$, you will proceed with constructing a Jordan Block matrix J as described at the beginning of this exercise.

After the function `jord` is created and saved in a file, we return to the Live Script.

**First, we set the format mode by typing

```
format
format compact
```

Then, we print in the Live Script the created function **jord by typing

```
type jord
```

**Next, we input

```
r=rand(1)
```

and run the function `J=jord(n,r)` for each of the following n to get either a non-empty output J or an empty matrix and a message that Jordan Block cannot be built:

(a) `n=0;`
(b) `n=-2;`
(c) `n=3.5;`
(d) `n=-2.5;`
(e) `n=4;`

EXERCISE2 (5 points) **Difficulty: Moderate**

DESCRIPTION: In this part of the project, you will create a function **added** in a file. You will code the sum of two matrices A and B, when they are of the same size, by using any of these two definitions: (1) the sum of A and B is the matrix C whose columns are the sums of the corresponding columns of A and B or (2) the sum of A and B is the matrix C whose entries are the sums of the corresponding entries of the matrices A and B. Then, you will verify if your output matrix C matches the output of a MATLAB built-in function for the sum, A+B.

**Create the function which begins with:

```
function C = added(A,B)
```

**First, your function has to verify whether the input matrices A and B have the same size. If not, output a message `'the matrices are not of the same size and cannot be added'`, and assign an empty matrix to C. After that, the program terminates.

**If the matrices can be added, you will continue with calculating the sum C of A and B using one of the two definitions of the sum of two matrices (see above) – to code it, you can employ "for" loop or vectorized statement. Output and display the matrix C.

**Next, you will use a logical "if" statement to verify whether the calculated matrix C matches the output for a built-in MATLAB function A+B. If the outputs C and A+B do not match, program an output message `'check the code!'` – use `disp` or `fprintf` commands – and, if you receive this message, you will need to correct the code and re-run Section.
This will be the end of your function `added`**.** Save it in your Current folder in MATLAB.

The rest of the work has to be done in the Live Script.

**Print your function in the Live Script:

```
type added
```

**Run the function `added` on the following sets of variables:

(a) `A=magic(3), B=ones(4)`
(b) `A=ones(3,4), B=ones(3,3)`
(c) `A=randi(100,3,4), B=randi(100,3,4)`

**You should receive a non-empty output matrix C in part (c), and you will continue working in the Live Script with the matrices A, B, and C from part (c). You will "demonstrate" by means of your function `added` and logical statements that the Commutative and Distributive properties of the matrix addition hold (see (1)-(2) below):

(1) You will verify that the sum of A and B is the same as the sum of B and A. If it is the case, the program should output a message that `'commutative property holds for the given A and B'`.
(2) You will verify that the product of a scalar k by the matrix C, kC, is the same as the sum of kA and kB. If this is the case, your program will output a message that `'distributive property holds for the given A and B'`.
For this part you will need to input a scalar:

```
k=fix(10*rand(1))+5
```

Note: Make sure that you will receive the desired output messages for (1) and (2) above.

EXERCISE3 (5 points) **Difficulty: Moderate**

Theory: The transformation defined by the matrix $A=\begin{bmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{bmatrix}=\begin{bmatrix}c & -s\\ s & c\end{bmatrix}$ performs a rotation in x_1x_2 – plane through the angle θ radians in the counterclockwise direction for a positive angle θ. The same type of a "plane" rotation in x_ix_j - plane can be performed in $\mathbb{R}^n$ $(n\geq 2,\ 1\leq i<j\leq n)$, and it is called a Givens rotation.
A Givens rotation is represented by a matrix G which is defined by the following parameters: n $(n\geq 2)$, $theta=\theta$ (the angle of rotation), and i, j $(1\leq i<j\leq n)$. A matrix G can be created by starting with the $n\times n$ identity matrix, that is `G=eye(n),` and, then, assigning the entries of the matrix A to the entries of `G` at the intersections of rows i, j with columns i, j $(i<j)$ as follows:

```
G(i,i)=c; G(i,j)=-s; G(j,i)=s; G(j,j)=c;
```

Geometrical Meaning: The matrix-vector product of G and a column vector x in $\mathbb{R}^n$ performs a rotation of x in the x_ix_j -plane through θ radians, affecting, possibly, only rows (=entries) i and j in x.

**Write a MATLAB function which begins with the line

```
function G=givensrot(n,i,j,theta)
```

and produces an $n\times n$ Givens matrix G under the conditions that $1\leq i<j\leq n$ and $n\geq 2$ - these conditions have to be verified in your code.
**If at least one of the conditions does not hold, the program outputs an empty matrix

```
G=[];
```

and terminates with an output message that Givens rotation matrix cannot be constructed.
**If all conditions hold, output and display the matrix G, as it specified in the Theory above.

**Print the function `givensrot` in your Live Script.

```
type givensrot
```

**Run the function `G=givensrot(n,i,j,theta)` on each set of the variables in (1) - (5). (Type *pi* for π in MATLAB.)

(1) `n=1;i=1;j=2;theta=pi`
(2) `n=4;i=3;j=2;theta=pi/2`
(3) `n=5;i=2;j=4;theta=pi/4`
(4) `n=2;i=1;j=2;theta=-pi/2`
(5) `n=3;i=1;j=2;theta=pi`

The rest of the work on this exercise has to be done in the Live Script using the output matrix G from part (5):

**Based on the Geometrical Meaning of the product of G and a vector, predict what would be the matrix (denote it GI) whose columns are the images of the vectors e_1, e_2, e_3, respectively, under the Givens rotation defined in part (5); here, e_1, e_2, e_3 are the columns of the 3 x 3 identity matrix `I=eye(3)`, that is,

$$\mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{e}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

Type and display GI in the Live Script.

**Then, calculate the actual products of G and each of the vectors e_1, e_2, e_3 (in the order indicated) – these products will form the columns of the matrix G*I (output and display G*I).

**Run a logical statement that would verify that your predicted matrix GI and the calculated matrix G*I match within the margin 10^(-7). If the matrices match, output a message that your prediction is correct – please make sure that you will receive this message.
Important: to compare two matrices, you need to use the function `closetozeroroundoff` with p=7 (this function was created in Project 0).
Print **closetozeroroundoff in your Live Script.

```
type closetozeroroundoff
```

**Next, complete one more task. Input a vector:

```
x=ones(3,1);
```

Output and display the image of this vector under the transformation defined by the matrix G.

EXERCISE4 (5 points) **Difficulty: Easy**

We can define a *Toeplitz matrix* as an $m \times n$ matrix in which each descending diagonal from left to right is constant. For example,

$$A = \begin{bmatrix} 4 & 3 & 2 & 1 \\ 5 & 4 & 3 & 2 \\ 6 & 5 & 4 & 3 \end{bmatrix}$$

is a 3 by 4 Toeplitz matrix.

Part 1. In this part, we will produce an $m \times n$ Toeplitz matrix using a vector **a** with $(m+n-1)$ entries by assigning

$$A(i, j) = \mathbf{a}(n+i-j) \qquad i = 1{:}\,m,\ j = 1{:}\,n.$$

In the example above, a 3×4 matrix A has been generated by the vector $\mathbf{a} = [1, 2, 3, 4, 5, 6]$.

In this part of the exercise, you will write a code that creates an $m \times n$ Toeplitz matrix, when possible. Then, you will run the function on the sets of variables *m*, *n*, and **a** to get the corresponding Toeplitz matrices.

**Create a function in the file that begins with
```
function A=toeplitze(m,n,a)
```

Here, the inputs *m* and *n* will define the size of the Toeplitz matrix.
First, your function has to check whether the number of entries in the vector **a is $(m+n-1)$. If it is not the case, output a message that the dimensions mismatch and assign an empty matrix to A. If vector **a** has exactly $(m+n-1)$ entries, output and display the Toeplitz matrix according to the formula given above.
This is the end of the function `toeplitze`.

****Print** the function `toeplitze` in your Live Script.
```
type toeplitze
```

****Run** the function `A=toeplitze(m,n,a)`
on the following sets of variables (display the vector **a** for each of the below):
(a) `m=4; n=2; a=1:5`
(b) `m=4; n=3; a=1:5`
(c) `m=4; n=3; a=1:7`
(d) `m=3; n=4; a=randi(10,1,6)`
(e) `m=4; n=4; a=[zeros(1,3), 1:4]`

**Next, proceed in your Live Script with the following tasks:
(1) Type the vector **a** which defines a 6 by 6 upper triangular Toeplitz matrix with random integer entries in the range between 0 and 100. Display the vector **a**. Run the function `A=toeplitze(m,n,a)`on your **a** and the variables `m,n` indicated in this part.
(2) Output a 5 by 5 identity matrix by running the function `A=toeplitze(m,n,a)`on the corresponding set of variables. Display the vector **a** (which you need to construct first).

Part 2. MATLAB has a built-in function, called `toeplitz`**,** which takes a different approach to constructing a Toeplitz matrix. The specifications are on this page:
https://www.mathworks.com/help/matlab/ref/toeplitz.html?s_tid=doc_ta

**(a) Output a (symmetric) Toeplitz matrix T by running a MATLAB built-in function
```
T=toeplitz(r)
```
with
```
r=1:5;
```
**Run a logical statement in the Live Editor to verify that T is, indeed, a symmetric matrix. Output the corresponding message if it is the case.

**(b) By running the function `T=toeplitz(c,r),` output a 6 by 5 Toeplitz matrix with the same *r* and your choice of the vector *c* which should allow you to avoid the MATLAB "warning" about "conflicting" entries.

More on Matrices; Application

EXERCISE5 (6 points) **Difficulty: Hard**

Theory: A vector with nonnegative entries is called a **probability vector** if the sum of its entries is 1. A square matrix is called **right stochastic** if its rows are probability vectors; a square matrix is called **left stochastic** if its columns are probability vectors; and a square matrix is called **doubly stochastic** if both, the rows and the columns, are probability vectors.

**Write a MATLAB function that begins with

```
function [S1,S2,L,R]=stochastic(A)
L=[];
R=[];
```

It accepts as the input a square matrix A with nonnegative entries. Outputs L and R will be the left stochastic and the right stochastic matrices generated, when possible, according to the instructions given below. You will also calculate and display the vectors `S1=sum(A,1)` and `S2=sum(A,2)` with the corresponding messages:

```
fprintf('the vector of sums down each column is\n')
S1=sum(A,1)
fprintf('the vector of sums across each row is\n')
S2=sum(A,2)
```

Then, you will use a conditional statement to proceed with the tasks outlined below. You may also find it helpful to employ a logical command **all**.

**First, your function has to check whether A contains both a zero column and a zero row. If yes, output a message "A is neither left nor right stochastic and cannot be scaled to either of them". The outputs L and R will be empty matrices (as assigned previously) – the empty outputs should be suppressed.

**Then, the function checks whether A is: (1) doubly stochastic (assign: `L=A;R=A);` or (2) only left stochastic (assign: `L=A;` R will stay empty), or (3) only right stochastic (assign `R=A;` L will stay empty). In each of these cases, also output a message that comments on the type of the matrix A.

**Finally, we consider a possibility that A is neither left nor right stochastic, but can be scaled to the left stochastic and/or to the right stochastic. You will output a message "A is neither left nor right stochastic but can be scaled to a stochastic matrix" and proceed with the scaling in the ways outlined below:

(1) If neither S1 nor S2 has a zero entry, we are **scaling*** A to the left stochastic matrix L and the right stochastic matrix R. You will also check if it is the case that the matrices L and R are equal – in this case, A has been scaled to a doubly stochastic matrix.

(2) If S1 does not have a zero entry but S2 does, we scale A only to the left stochastic matrix L (R stays empty).

(3) And, if S2 does not have a zero entry but S1 does, we scale A only to the right stochastic matrix R (L stays empty).

Notes: In each of the cases, the non-empty outputs have to be displayed with the corresponding messages. For the case of a doubly stochastic matrix, output the corresponding message and display either L or R – to determine if L=R, you will need to use the function *closetozeroroundoff* with p=7. (This function was created in Project 0.)

***Scaling**: To scale A to the left stochastic matrix L, we use vector S1 and multiply each column of A by the reciprocal of the corresponding entry of S1. To scale A to the right stochastic matrix R, we use the vector S2 and multiply each row of A by the reciprocal of the corresponding entry of S2.

**Print the functions `stochastic`, `jord`, and `closetozeroroundoff` in your Live Script. (The function `jord` was created in Exercise 1 of this project.)
**Run the function `[S1,S2,L,R]=stochastic(A)` on each of the matrices below (display the input matrices in your Live Script):

```
(a) A=[0.5,0,0.5,0; 0,0,1,0;0.5,0,0.5,0;0,0,0,1]
(b)A = transpose(A)
(c)A=[0.5,  0,  0.5; 0,  0,  1; 0,  0,  0.5]
(d)A=transpose(A)
(e)A=[0.5,  0,  0.5; 0,  0.5,  0.5; 0.5,  0.5,  0]
(f)A=magic(3)
(g)B=[1 2;3 4;5 6]; A=B*B'
(h)A=jord(5,4)
(k)A=randi(10,5,5);A(:,1)=0;A(1,:)=0
```

NOTE: Please make sure you will verify that all your outputs and messages match the corresponding definitions of the stochastic matrices. If not, make corrections in your code!

EXERCISE6 (5 points) **Difficulty: Moderate**

In this exercise, you will use **Newton's method** to approximate a real zero of a given function.

Theory: A number x is a zero of a function f if $f(x)=0$. A real zero is an x-intercept of the function. Also, the **zeros** of a function f are the **roots** of the equation $f(x)=0$.

To approximate a real zero of the function $f(x)$ by **Newton's method**, we, first, choose an initial approximation x_0 of the specified zero. The consecutive N iterations $x_1, x_2, ..., x_N$ are defined by the formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n = 0:N.$$

Note: if $f'(x)$ is close to zero at an initial point x_0, the process will converge very slowly. It makes Newton's method sensitive to the choice of an initial approximation.

In this exercise, you will work with two functions:
$F(x) = \arctan(x) + x - 1$ and $G(x) = x^3 - x - 1$.

First, you will create a **function handle. Please refer to the documentation in MATLAB: https://www.mathworks.com/help/matlab/matlab_prog/creating-a-function-handle.html?searchHighlight=handle&s_tid=doc_srchtitle#buvu9u8-1
We will create handles to anonymous functions and calculate derivatives of the function handles as it follows.

**Begin with typing in the Live Script:

```
format
format compact

syms x
F = @(x) atan(x) + x - 1
F1 = eval(['@(x)' char(diff(F(x)))])

G=@(x) x.^3-x-1
G1=eval(['@(x)' char(diff(G(x)))])
```

Notes: (1) a MATLAB command `syms x` defines a symbolic variable x;
(2) F and G are the function handles and F1 and G1 are the function handles for the first derivatives of F and G, respectively.

**Next, we create and output 2-D plots of the functions F, G, and y=0 on the interval [-2, 2] in order to visualize the x-intercepts and choose initial approximations. An initial value x_0 should be chosen close to the x-intercept which we are approximating.

**Type in the Live Script the code given below – it will output the graphs of F and G together with the function y=0 after you Run the section.

```
yzero=@(x) 0.*x.^(0)
x=linspace(-2,2);
plot(x,F(x),x,yzero(x));
plot(x,G(x),x,yzero(x));
```

Note: here we have created another symbolic function `yzero`.

It is obvious from the properties of the function `F(x)` that it has only one real zero, which we will approximate. Concerning the function `G`, which is a polynomial of the third degree (we will denote it p), we are going to verify that it has only one real zero. In order to do that, we find all zeros of the polynomial p using MATLAB built-in functions `sym2poly` and `roots`: the function `sym2poly(p)` outputs the vector of the coefficients of the polynomial p (in descending order according to the degree), and the composition of two functions `roots(sym2poly(p))`outputs all zeros of the polynomial p.
**Type in the Live Script:

```
syms x
p=x^3-x-1;
roots(sym2poly(p))
```

After you Run Section, this part of the code will output the three zeros of the polynomial p – two of them are complex conjugate numbers and one is a real zero that we will approximate.

Next, we proceed with constructing a function in the file that approximates a real zero.
**Create a function called `newtons`. It begins with:

```
function root=newtons(fun,dfun,x0)
format long
```

The inputs `fun` and `dfun` are the function and its first derivative, respectively, and `x0` is the initial approximation. The output `root` will be our approximation of the real zero of a function. We will program consecutive iterations according to the Newton's method, and we will assign to `root` the iteration which will be the first one falling within a margin of 10^(-12) from the MATLAB approximation `x` of that zero – the last is delivered by a built-in MATLAB function `fzero`. The details are below:

**Type the line

```
x=fzero(fun,x0)
```

in your function `newtons` and output `x` with a message that it is a MATLAB approximation of the real zero of the function.

**Then, your function `newtons` will calculate consecutive iterations x_n, $n = 0 : N$, using Newton's Method (see Theory above). You can employ a "while loop" here. The loop will terminate when, for the first time, `abs(` x_N `-x)<10^(-12)` for some consecutive iteration x_N. Output with a corresponding message the number of iterations N, and assign the last iteration, x_N, to the output `root`.

This will be the end of your function `newtons`.

**Print the function `newtons` in your Live Script.

**Next, proceed with the following tasks in the Live Script:

Part (a)

You will work with the function F in this part. Using the graph of the function F, choose three different values of the initial approximation `x0` of the zero of F.

**Then, input the function handles:

```
fun=F;
dfun=F1;
```

and run the function `root=newtons(fun,dfun,x0)` for each of your three choices of `x0` - one at a time.

Part (b)

You will work with the function G in this part.

**First, input the corresponding function handles.

**Then, run `root=newtons(fun,dfun,x0)` for each of the initial approximations (1)-(8) of the real zero of `G`:

(1) `x0=1.3;`
(2) `x0=1;`
(3) `x0=0.6;`
(4) `x0=0.577351;`
(5) Pick the initial value `x0` to be the positive zero of the derivative of `G(x)`. Type `x0=1/sqrt(3)`
(display `x0`) and run the function `root=newtons(fun,dfun,x0)`.
(6) `x0=0.577;`
(7) `x0=0.4;`
(8) `x0=0.1;`

See next page for the BONUS on this Exercise.

Analyze the patterns of the consecutive iterations after you run your function `newtons` on the choices of the initial approximations (1)-(8) in Part (b).

BONUS: 1 point!
% Describe the general pattern for the numbers of iterations which you observed for the choices (1)-(8), excluding choices (4)-(6) out of this sequence.

BONUS: 1 point!
% What did you notice when you run your function on choices (4)-(6)? How would you explain that "strange" pattern for the consecutive iterations?

Hint: to get the bonus points, you could temporarily display all intermediate iteration that come from the "loop". You may also need to review the Newton's Method theory in more detail.

THIS IS THE END OF PROJECT 1