MATLAB PROJECT 2

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

BEGIN with creating Live Script **Project2.**
Note: All exercises in this project will be completed in the Live Script using the Live Editor.

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.

Part I. Elementary Row-Operations

Exercise 1 (6 points) **Difficulty: Moderate**
In this exercise you will write three functions that create the three types of elementary matrices and, then, perform elementary row-operations on a given matrix using the created functions.
Theory: To create an elementary $n \times n$ matrix, we apply a row operation, such as, row replacement, row interchanging, or scaling, to the $n \times n$ identity matrix. This will produce an elementary matrix of one of the three types, E1, E2, or E3, respectively. When an $n \times m$ matrix A is left-multiplied by an elementary matrix E1, E2, or E3, it performs the same operation on the matrix A as was the operation that created that elementary matrix from the identity matrix.

**Create the three functions in MATLAB – each function outputs one of the three types of elementary $n \times n$ matrices `E1`, `E2`, or `E3`:

```
function E1=ele1(n,r,i,j)
function E2=ele2(n,i,j)
function E3=ele3(n,j,k)
```

**To generate these matrices, start with the identity matrix `eye(n)`and proceed as follows:
For `ele1`: matrix `E1` is obtained from `eye(n)`by replacing (row j) with [(row j) + (row i)$\cdot r$].
For `ele2`: matrix `E2` is obtained from `eye(n)`by interchanging rows i and j.
For `ele3`: matrix `E3` is obtained from `eye(n)`by multiplying row j by k.
(Here, i, j are the indexes, each is in the range from 1 to n, and k is a scalar.)

**Print the created functions in your Live Script:

```
type ele1
type ele2
type ele3
```

**Next, type in the Live Script

```
format
format compact
```

and input the matrix $A=\begin{bmatrix} 0 & 1 & 3 & 1 \\ 2 & 4 & 6 & -2 \\ 3 & 1 & 4 & 2 \end{bmatrix}$.

First, reduce matrix A to the reduced echelon form **by hand on paper**. After you completed hand calculations, go over the elementary row operations applied to the matrix A and, on each step, identify the variables, such as *n*, *i*, *j*, and etc., which are the inputs in the corresponding function `ele1`, `ele2`, or `ele3`.

**Then, return to the Live Script and start reducing matrix A to the reduced echelon form using functions `ele1`, `ele2`, and `ele3`.
Here is an example how you can proceed with this task: the first possible operation could be interchanging rows 1 and 2 in A – this will place a non-zero number into the pivot position. We are using here function `ele2` on the variables `n=3`, `i=1,j=2` (where `n=3` is the number of rows in A, and rows `i=1` and `j=2` are interchanging). Then, we run the function `ele2` to output the matrix `E2`:

```
E2=ele2(3,1,2);
```

(we output `E2`, but do not display it in the Live Script).
Next, we are left-multiplying matrix A by `E2` – this will apply the row interchanging operation to the matrix A – the result has to be output (and displayed) as a new matrix `A1`:

```
A1=E2*A
```

After you are done with your first row operation, proceed with your second row operation (you will apply it to the new matrix `A1`): create the corresponding elementary matrix and left-multiply `A1` by the created elementary matrix to output (and display) a new matrix `A2`, which is the result of applying your second row operation.
You will continue this way and generate a sequence of the matrices `A1`, `A2`, … until you arrive at the reduced echelon form of A, matrix `AN` (`N` will be a specific number).
Note: make sure that you will output and display all matrices `A1`, `A2`, …, `AN`.

**Use a logical statement to check if your matrix `AN` matches the output of a built-in MATLAB function that produces the reduced echelon form of A, `rref(A)`. If this is the case, output a corresponding message. If the message is not displayed after you run your logical statement, consider making a revision of the previous steps.

Part II. Basic Operations

In this part of the project, you will create a function that utilizes the row reduction algorithm to calculate and output the inverse matrix of an invertible matrix A.

Exercise 2 (4 points) **Difficulty: Easy**

**Create the following function in MATLAB:

```
function F=inverses(A)
```

which takes as input an $n \times n$ matrix A. All of the below have to be included into the function.

First, the function has to determine whether A is invertible – use a built-in MATLAB function **rank.
If A is not invertible, the function returns an empty matrix
`F=[];`
and outputs a message "matrix A is not invertible". After that, the program terminates.

If A is invertible, the function continues with calculating the matrix F, which is the inverse of the matrix A.
To calculate `F`, we reduce matrix `[A eye(n)]` into the reduced echelon form and output (and display) the last $n \times n$ block of the reduced echelon form – it is the inverse matrix `F`. You will need to use a MATLAB built-in function **rref for this part.
Then, we calculate the inverse matrix using a built-in MATLAB function **inv:
`P=inv(A)`
Output (and display) the matrix `P` with a message that `P` is the inverse of A calculated using a MATLAB function.
**Finally, we use a logical statement to determine if the matrices `F` and `P` match. You will need to employ *closetozeroroundoff* function in your code with p=5. If `F` and `P` match, output a corresponding message. Make sure that you will receive this message, otherwise, consider making corrections in your code.
This is the end of your function `inverses.`

**Type the function `inverses` and `closetozeroroundoff` in your Live Script.
**Run the function `F=inverses(A)` on the matrices in parts (a)-(e). Notice that some inputs are suppressed.

(a) `A=[4 0 -7 -7;-6 1 11 9;7 -5 10 19;-1 2 3 -1]`
(b) `A=[1 -3 2 -4;-3 9 -1 5;2 -6 4 -3;-4 12 2 7]`
(c) `A=magic(3);`
(d) `A=hilb(5);`
(e) `A=magic(6);`

Part III. Solving Equations

In this part of the project, you will learn the basics on solving equations $A\mathbf{x} = \mathbf{b}$ when A is an invertible matrix. You should read the content carefully and perform the indicated tasks in MATLAB.
Solving Matrix Equations. Suppose A is an invertible $n \times n$ matrix and you are solving equation A**x** = **b**. By the invertible matrix Theorem, there exists a unique solution for every **b** in $\mathbb{R}^n$. We have several methods of finding the solution. Three of them are considered below.

(1) There is a special operator in MATLAB, `\` , called **backslash**, that usually gives an excellent result. To use it, store A and **b** and type A**b.** Backslash is the best of the methods. It works fast and minimizes a round-off error. It also checks the condition number on the coefficient matrix. If the condition number is large, it will warn you by printing message: "Matrix is close to singular or badly scaled. Results may be inaccurate."
For more information on condition number, please refer to MATLAB documentation:
https://www.mathworks.com/help/matlab/ref/cond.html?s_tid=doc_ta

(2) The second method is using a built-in MATLAB function **inv,** which also checks the condition number, but calculating **inv**(A) requires a lot more arithmetic than backslash.

(3) The third method employs a MATLAB built-in function **rref**. This function was written to help students to learn Linear Algebra, so its algorithm is not accurate, and it will not warn you if your system is one of those for which it is hard to get an accurate solution. This function should not be used to solve real world problems. However, for the matrices with integer entries, you could use the command `rref([A b])` to get the reduced echelon form of the augmented matrix and, then, output the last column as the solution.

Exercise 3 (6 points) **Difficulty: Moderate**

Part 1. Solving a system A**x** = **b**

**Create a function in MATLAB that begins with:

```
function [C,N]=solvesys(A,b)
[~,n]=size(A);
format long
```

We are using `format long` to display a number in exponent format with 15 digit mantissas. If, later on, you will need to switch back to the default format, type and run

```
format
```

The inputs are an $n \times n$ matrix A and an $n \times 1$ vector **b**. If A is invertible, the outputs are the matrix C, whose 3 columns **x1, x2, x3** are the solutions obtained by the three methods described above, respectively, and N is the vector whose 3 entries are the 2-norms of the vectors of the differences between each two distinct solutions.

**First, check if A is invertible - use a MATLAB built-in function *rank*.

If A is not invertible, output the message `'A is not invertible'` and assign empty outputs:

```
C=[];
N=[];
```

Also, in this case, you will need to identify which of the two possibilities holds for the system A**x** = **b** when A is not invertible**:** `'The system is inconsistent'` or `'The system is consistent, but the solution is not unique'`. Use the command *rank* to check the system on each of the two possible cases and program two possible output messages. After that, terminate the program.

If A is invertible, the function continues with solving equation A**x** = **b** by the three methods described above and outputs the solutions **x1, x2, x3** for each of the methods (1)-(3), respectively. These solutions have to form the columns of the matrix C. Thus, we will assign `C=[x1,x2,x3]` and display C with the message that `'Solutions obtained by different methods are the columns of'`

(display C)

**Next, the function has to return a column vector

```
N=[n1;n2;n3];
```

where

```
n1=norm(x1-x2);
n2=norm(x2-x3);
n3=norm(x3-x1);
```

which has to be displayed with a message `'Norms of differences between solutions are the entries of'`
(output N)

Note: The entries of the vector **N** are calculated by using a built-in MATLAB function `norm`, which, when applied to a vector, calculates the square root of the sum of squares of the entries of the vector – it is also called the 2-norm. The vector **N** gives an idea how "close" are the solutions obtained by various methods.

**Print the function `solvesys` in your Live Script.
Run the function `[C,N]=solvesys(A,b)` for the following choices of the matrix A and vector **b. Type the matrices and the vectors as they are displayed below – notice, some inputs are suppressed.

```
%(a)
A=magic(4);I=eye(size(A,1));b=I(:,end)
%(b)
A=magic(4),b=A(:,end)
%(c)
A=magic(5);b=fix(10*rand(size(A,1),1))
%(d)
A=eye(6);b=fix(10*rand(size(A,1),1))
%(e)
A=magic(7); b=fix(10*rand(size(A,1),1))
%(f)
A=hilb(7);
```

(in part (f), the vector b is the one that you stored for part (e)).

% Write a comment on the output for part (d) by comparing the solution with the vector **b**.
% Analyze the output vector N throughout the choices (c)-(f). For which of them the solutions obtained by different methods are not really "close" to each other?

Part 2. Condition numbers

**Calculate the condition numbers for the matrices `A=magic(7)` and `A=hilb(7)`.

```
c1=cond(magic(7))
c2=cond(hilb(7))
```

% Compare c1 and c2 with number 1 and comment in your Live Script why, on your opinion, the norms of the differences between the solutions for the coefficient matrix in part (f) are so large compared with the ones for the matrix in part (e).

**Explore sensitivity to perturbations of a badly conditioned matrix `hilb(7)`:
Input in the Live Script:

```
A=hilb(7);
```

Run the following code:

```
b=ones(7,1);
x=A\b;
b1=b+0.01;
y=A\b1;
norm(x-y)
c3=rcond(A)
```

**Re-run the code above for:

```
A=magic(7);
```

%Comment on sensitivity to perturbations of `magic(7)` compared with `hilb(7)` by analyzing the corresponding outputs for the `norm(x-y)` and `c3`.

More information on the reciprocal condition numbers can be found on the MATLAB page: https://www.mathworks.com/help/matlab/ref/rcond.html?s_tid=doc_ta

Part IV. Area, Volume, and Graphics in MATLAB

GOAL: In this part, you will work with applications of matrices and determinants to Geometry and, specifically, to calculating area and volume. You will also touch a basis of graphics in MATLAB, while working with transformations of a plane.

Exercise 4 (4 points) **Difficulty: Easy**

Theory: The area of a parallelogram in $\mathbb{R}^2$ built on non-parallel vectors $\mathbf{v}_1, \mathbf{v}_2$, and the volume of a parallelepiped in $\mathbb{R}^3$ built on vectors $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$, which are not parallel to the same plane, is $|\det A|$ (or abs(det A)), where $A = [\mathbf{v}_1 \ \ \mathbf{v}_2]$ and $A = [\mathbf{v}_1 \ \ \mathbf{v}_2 \ \ \mathbf{v}_3]$, respectively.
(See Section 3.3 of the textbook for details.)
In this exercise, you will be given 2 vectors in $\mathbb{R}^2$, $\mathbf{v}_1$ and $\mathbf{v}_2$, and 3 vectors in $\mathbb{R}^3$, $\mathbf{v}_1, \mathbf{v}_2$, and $\mathbf{v}_3$, on which a parallelogram and parallelepiped, respectively, may or may not be built.
**Create a function in MATLAB that begins with:

```
function D=areavol(A)
```

which takes as an input a matrix A, whose columns are the vectors on which a parallelogram or parallelepiped may possibly be built.
First, your function has to check whether the given vectors are linearly independent - use the MATLAB built-in function **rank to check if it is the case.
If the vectors are not linearly independent, a parallelogram in $\mathbb{R}^2$ or parallelepiped in $\mathbb{R}^3$ cannot be built. In this case, the function outputs a corresponding message, which has to be specific to whether it is a parallelogram or a parallelepiped that cannot be built. It also assigns an empty output to `D`. Then, the program terminates.
If the vectors are linearly independent, the function calculates the area and the volume, `D`, and outputs each with the corresponding message:
"The area of the parallelogram is"
(output `D`)
and
"The volume of the parallelepiped is"
(output `D`)
Hint: in order to display a correct message whether it is a parallelogram or parallelepiped, which can or cannot be built, and whether it is the area or the volume, you should keep a track on the number of rows (or columns) of A and use the conditional statement.

**Type in the Live Script
```
format
```
**Print the function `areavol`:
```
type areavol
```

**Run the function `D=areavol(A);` on each of the following matrices (display the input matrices A):

(a) `A=randi(10,2)`
(b) `A=fix(10*rand(3))`
(c) `A=magic(3)`
(d) `B=randi([-10,10],2,1); A = [B,3*B]`
(e) `X=randi([-10,10],3,1);Y=randi([-10,10],3,1);A=[X,Y,X-Y]`

Notice: the matrices in (d) and (e) are created by using vectors.

Exercise 5 (4 points) **Difficulty: Easy**

In this exercise, you will be plotting figures in x_1x_2-plane and consider their images under the transformations of a plane, such as, reflections and shear.

**First, create the standard matrices of the indicated transformations of the x_1x_2-plane (for help, please refer to the textbook, Section 1.9):

R1 (reflection across the x_1-axis)

R2 (reflection across the x_2-axis)

VS (vertical shear with $k = 2$)

Output these matrices and display them in your Live Script**.**

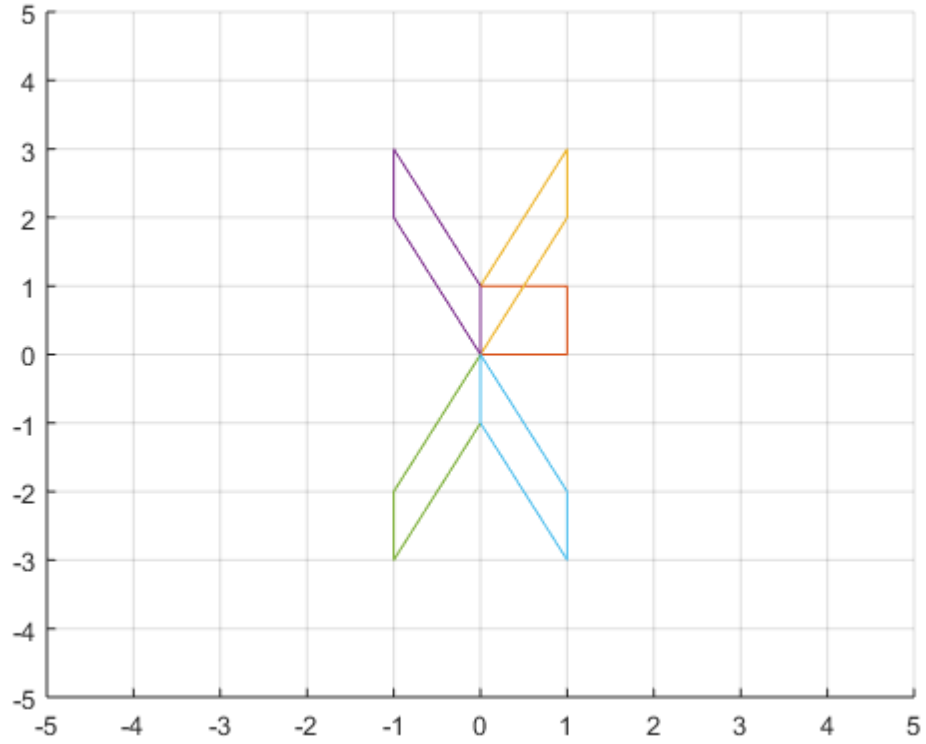
**Create the function `transf` in MATLAB – the code is below:
```
function C=transf(A,E)
C=A*E;
x=C(1,:);y=C(2,:);
plot(x,y)
v=[-5 5 -5 5];
axis(v)
end
```
%Analyze each line of the function `transf` and write comments in your Live Script.
**Print the function `transf` in your Live Script.
**Next, perform a sequence of the transformations starting with the unit square whose vertices are defined by the columns of the matrix E:

$$E = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Proceed as follows:
**Type the set of commands given below and Run Section.
```
E=[0 1 1 0 0;0 0 1 1 0];
A=eye(2);
hold on
grid on
E=transf(A,E)
```

Your first outputs will be the original matrix E and the plot, which is the unit square with the vertices defined by the columns of E.

**Then, you will perform step-by-step a sequence of transformations – on each step, you will be assigning one of the matrices VS, R1, or R2 to the matrix A and running the function `E=transf(A,E)` until you obtain the figure below:



For example, your next transformation will be the vertical shear, that is, you will type

```
A=VS;
E=transf(A,E)
```

After you Run Section, you will receive the second figure, while the first one is held. The code will also output and display the new matrix E.

Note: make sure that you will have displayed in your Live Script all output matrices E and the final plot.

Part V. Cofactors, Determinants, and Inverse Matrices

In this part of the project, you will calculate a matrix of cofactors of a given $n \times n$ matrix **a** and use it to compute the determinant and the inverse of an invertible matrix **a**.
Theory: 1) The determinant of a matrix **a** can be computed by using a cofactor expansion across any row or down any column of **a**. For the reference see Section 3.1 of the textbook and the lecture Module 13, Part II.
2) The inverse of an $n \times n$ invertible matrix **a** can be computed by using a method based on the Cramer's Rule. Specifically,

$$\mathbf{a}^{-1} = \frac{1}{\det \mathbf{a}} \operatorname{adj} \mathbf{a},$$

where adj **a** denotes the *adjugate* (or classical *adjoint*) of a matrix **a,** which is the transpose of the matrix whose entries are cofactors of **a** .
(For the reference see Section 3.3 of the textbook.)

Exercise 6 (6 points) **Difficulty: Very Hard**

Part 1

** Create a function that begins with

```
function C=cofactor(a)
format compact
```

First, the function has to generate $(n-1)\times(n-1)$ matrices A_{ij} $(i=1:n,\ j=1:n)$: each A_{ij} is obtained from **a** by deleting row *i* and column *j*.
Then, it outputs an $n\times n$ matrix

$$C = \left[C(i,j)\right](i,j=1:n),$$

whose entries are the cofactors calculated by the formulas $C(i,j) = (-1)^{i+j}\det(A_{ij})$.

Note: in this part, we use a MATLAB built-in function `det` on the matrices A_{ij} of the sizes $(n-1)\times(n-1)$.

**Output and display the matrix C with the message `'the matrix of cofactors is'` (display `C`).

Part 2

** Create a function that will compute the determinant `D` of a matrix **a**.
The function begins with

```
function D=determine(a,C)
D=[];
n=size(a,1);
```

First, the function has to check if the determinant of **a** is a zero - use command ***rank*** – you cannot use a built-in MATLAB function *det* for this part.
**If the determinant is a 0, the function outputs: `'the determinant of the matrix is'`

```
D=0
```

and the program terminates.

If the determinant is not a 0, you will calculate cofactor expansions across the rows and down the columns of the matrix **a.
First, store in your code an $n\times 2$ matrix `E`, for example,

```
E=zeros(n,2);
```

Then, recalculate the columns of the matrix E in the following way: the *n* entries of the first column of `E` are the cofactor expansions across *n* **rows** of **a**, respectively, and the *n* entries of the second column of `E` are the cofactor expansions down *n* **columns** of **a**, respectively.

**According to the theory, all entries of the matrix `E` have to be the same number, and we will verify it in our code. To do that, we will take the first entry `E(1,1)`as a reference, and check if

each entry of the matrix `E` matches the reference value. You should use the function `closetozeroroundoff` with p=7 on each of the differences `E(1,1) - E(i,j)`, where `i` runs from 1 to *n* and `j` runs from 1 to 2. If at least one of the entry `E(i,j)` does not match the reference value `E(1,1),` we output a message `'Something went wrong!` and terminate the program with an empty output for `D` (which has been assigned previously).

**If all entries of `E` match `E(1,1)` (within the given precision), your function will continue with one more task: it will calculate the determinant using a built-in MATLAB function
```
d=det(a);
```
and, then, it will verify that `d` matches `E(1,1)` – you should use here the function `closetozeroroundoff` with p=7.

**If `d` and `E(1,1)` match, output `D` with a message `'the determinant is'`
```
D=E(1,1)
```
If `d` and `E(1,1)` do not match, output a message `'Check the code!'` and, in this case, the empty output for `D` will stay.

Important: please make sure that you will not receive empty outputs after running the function `determine` on the variables given below. Notice that the non-empty value for `D` will be used in the function which you will create next.

Part 3

Create a function that computes the matrix $B = \mathbf{a}^{-1}$ for an invertible matrix **a**. It accepts as inputs: matrix `C`, which is the output of the function `cofactor,` and the value `D,` which is the non-empty output of the function `determine`.

**The function begins with:
```
function B=inverse(a,C,D)
B=[];
```

First, the function has to check if the matrix **a is invertible. You can either use the command ***rank*** to verify it directly or you can employ the output `D` of the function `determine`.
If **a** is not invertible, output the corresponding message and terminate the code – the empty output for `B` will stay.
If **a** is invertible, output the corresponding message, and proceed with calculating the inverse of the matrix **a**, matrix B, by means of the formula

$$B = \frac{1}{D} * \text{transpose}(C).$$

Here, `D` is a non-empty and non-zero output of the function `determine` and `C` is the matrix of cofactors, which is the output of the function `cofactor`.

Then, you will check whether your matrix `B` matches the output of the built-in MATLAB function **inv.**
**First, output (do not display) the matrix
```
F=inv(a);
```

**Next, use *closetozeroroundoff* function with the parameter p=7 to check if `B` and `F` match. If it is the case, output a message that `'the inverse is calculated correctly and it is the matrix'`
(display `B`).
If `B` and `F` do not match, output a message that can be something like `'What is wrong?!'`

Part 4
**Type the functions *closetozeroroundoff*, *cofactor*, *determine*, and *inverse* in the Live Script.

For each of the matrices **a in choices (a)-(e) below, run the functions in the following order:

```
C=cofactor(a)
D=determine(a,C)
B=inverse(a,C,D)
```

```
(a) a=diag([1,2,3,4])
(b) a=ones(4)
(c) a=magic(5)
(d) a=magic(6)
(e) a=hilb(4)
```

(Display the input matrices in your Live Script.)

****THIS IS THE END OF PROJECT 2**