MATLAB PROJECT 3

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

BEGIN with creating Live Script **Project3.**

Note: All exercises in this project have to 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 the Live Script. If `format long` had been used in an exercise, please make sure to return to the default `format` in the next exercise.

Part I. Subspaces & Bases

Exercise 1 (4 points) **Difficulty: Moderate**

In this exercise, you will create a function that works with the column spaces of two matrices A and B. First, it will determine whether Col A and Col B are subspaces of the same Euclidian vector space $\mathbb{R}^m$. If yes, your code has to determine whether it is the case that either Col A or Col B or both are the whole $\mathbb{R}^m$. If Col A and Col B are not the whole $\mathbb{R}^m$ but they have the same dimension, we will determine if Col A = Col B. Obviously, when two subspaces have the same dimension, it may not be true that they are the same set. For example, a line through the origin in $\mathbb{R}^3$ is a one-dimensional subspace of $\mathbb{R}^3$ but two lines might be different.
You will use a MATLAB built-in function `rank()` within your code. Remember, **the rank of a matrix** can be defined as **the dimension of the column space of the matrix**.
**Create a function in a file that begins with

```
function []=columnspaces(A,B)
format
m=size(A,1);
n=size(B,1);
```

First, the function has to check if Col A and Col B are subspaces of the same vector space $\mathbb{R}^m$.
**If Col A and Col B are subspaces of different Euclidean spaces, output the message

```
fprintf('Col A and Col B are subspaces of different spaces')
```

and terminate the program.
**If Col A and Col B are subspaces of the same vector space $\mathbb{R}^m$, output the message:

```
fprintf('Col A and Col B are subspaces of R^%i\n',m)
```

and continue your code with outputting and displaying the dimensions of Col A and Col B with corresponding messages.

Next, compose a **conditional statement** that would check on the following cases:
**If the dimensions of Col A and Col B are different, output a message

```
fprintf(['dimensions of Col A and Col B are different\n' ...
        'and Col A cannot be the same as Col B\n'])
```

and check if either Col A or Col B is the whole $\mathbb{R}^m$. Output a corresponding message for each of the two cases. An example of a message for Col A is below:

```
fprintf('Col A is the whole R^%i\n',m)
```

**If the dimensions of Col A and Col B are the same, first, check if both Col A and Col B are the whole $\mathbb{R}^m$. If it is the case, output a message:

```
fprintf('Col A = Col B = R^%i\n',m)
```

If it is not the case, check whether Col A and Col B are the same set. If it is the case, output a message:

```
disp('ColA = ColB')
```

Otherwise, output a message:

```
fprintf(['dimensions of Col A and Col B are the same ' ...
                    'but ColA ~= ColB\n'])
```

Hint: A simple way to verify that the subspaces of $\mathbb{R}^m$, Col A and Col B, which are not the whole $\mathbb{R}^m$ but have the same dimension, are equal to each other is by using a MATLAB function `rank()` and the fact that Col A = Col B if and only if the columns of B are in Col A and the columns of A are in Col B.
Note: You cannot use a MATLAB built-in function `colspace()` within the function `columnspaces`.
This is the end of the conditional statement and **the end of the function `columnspaces`.**

**Print the function `columnspaces` in your Live Script.
**Run the function on the choices (a)-(j) as indicated below:

```
%(a)
A=magic(3)
B=hilb(3)
columnspaces(A,B)
%(b)
A=magic(4)
B=hilb(4)
columnspaces(A,B)
%(c)
A=pascal(4)
B=magic(4)
columnspaces(A,B)
%(d)
A=magic(4)
B=eye(3)
columnspaces(A,B)
%(e)
A=magic(4)
B=rref(A)
columnspaces(A,B)
%(f)
A=magic(4);
B=rref(A);
```

```
A=A'
B=B'
columnspaces(A,B)
%(g)
A=[2 -4 -2 3;6 -9 -5 8;2 -7 -3 9;4 -2 -2 -1;-6 3 3 4]
B=rref(A)
columnspaces(A,B)
%(h)
A=[2 -4 -2 3;6 -9 -5 8;2 -7 -3 9;4 -2 -2 -1;-6 3 3 4];
B=([rref(A);zeros(5,4)]);
A=A'
B=B'
columnspaces(A,B)
%(i)
A=magic(4)
B=[eye(3);zeros(1,3)]
columnspaces(A,B)
%(j)
A=pascal(3)
B=[hilb(3),eye(3)]
columnspaces(A,B)
```

%Based on the outputs for the parts (f) and (h), comment in the Live Script what fact from the general theory concerning row operations and Row spaces has been demonstrated here.
%Based on the outputs for parts (e) and (g), write a comment in the Live Script on a possible effect of the elementary row operations on the Column space of a matrix.

BASES FOR COLUMN SPACE, NULL SPACE AND ROW SPACE

Exercise 2 (4 points) **Difficulty: Moderate**
In this exercise, you will create bases for the Col A, Nul A, and Row A in various ways by using built-in MATLAB functions, `colspace()` and `null( ,'r')`, as well as the functions that you will create yourself, `shrink` and `noll`. You will also investigate properties of the subspaces associated with a matrix A.

Part 1: Column Space
The column space of an $m \times n$ matrix A is a subspace of $\mathbb{R}^m$ spanned by the columns of A. It may happen that the set of the columns of A is not linearly independent, that is, not all columns of A will be in a basis. We can "shrink" matrix A into the matrix B of pivot columns of A, which form a basis for the Col A, by using the function `shrink` (the code is below):

```
function [pivot,B]=shrink(A)
[~,pivot]=rref(A);
B=A(:,pivot);
end
```

This function also outputs the vector `pivot` which lists the indexes of the pivot columns of A.

**Create the function `[pivot,B]=shrink(A)` in a file.

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

**Input the matrix (you will be using it in the tasks below):

```
A=magic(4)
```

Run the function

```
[pivot,colbasis1]=shrink(A)
```

Your function will output and display the indexes of the pivot columns of A and a matrix `colbasis1` whose columns form a basis for the Col A.

There is a MATLAB built-in function `colspace(sym(A))` that creates a symbolic matrix whose columns form a basis for the column space of a symbolic matrix A.

**Type and run in the Live Script:

```
R=rref(A'), sym_colbasis2=colspace(sym(A))
```

%Analyze the outputs R and `sym_colbasis2` and write a comment in the Live Script on the path that the function `colspace(sym(A))` takes to generate a symbolic matrix whose columns form a basis for the Col A. Justify that this approach is acceptable for creating a basis for Col A.

**Then, run the command below that converts a symbolic matrix `sym_colbasis2` to a double precision array (display the output):

```
colbasis2=double(sym_colbasis2)
```

Part 2: Null Space

**Create a function in a file that begins with

```
function [C,p] = noll(A)
n=size(A,2);
rankA=rank(A);
```

The function `noll` takes as the input a matrix A with *n* columns. We will use the function `homobasis` created in Exercise 3 of Project 1 within the function `noll` to output the matrix C whose columns span the Nul A: **C is either (1) a matrix with a single zero column or (2) a matrix whose columns form a basis for the Nul A.**
The function `noll` will also output the dimension of the Nul A, number p.

**First, use the Rank Theorem to output the dimension p of the Nul A by employing the variable `rankA`. Display p with a message:

```
fprintf('Nul A is a %i-dimensional subspace of R^%i\n',p,n)
```

**Next, place the function as below into your code:

```
C = homobasis(A)
```

which will output the matrix C.

**Then, construct a conditional statement to program the following:
If the Nul A is the zero subspace of $\mathbb{R}^n$, output:

```
disp('Nul A is spanned by the column of C')
```

Otherwise, output a message:

```
disp('a basis for Nul A is formed by the columns of the matrix C')
```

This is the end of the function `noll`.

**Print the function `noll` and `homobasis` in the Live Script.

Test your function `noll`.
**Input the matrix:
```
A=magic(5)
```
Run the function
```
[C,p]=noll(A);
```
Your outputs should match the case (1).

**Input the matrix:
```
A=magic(4)
```
Run the function
```
[C,p]=noll(A);
```
Your outputs should match the case (2).

There is a built-in MATLAB function `null(A,'r')` that outputs a "rational" basis for Nul A.
**Run in the Live Script:
```
N=null(A,'r')
isequal(C,N)
```
%The logical command above has to output "true". Write a comment in the Live Script on the way of generating a basis for the Nul A by the MATLAB function `null(A,'r')`.

**Run the function on the choices (a)-(d) as below:
```
%(a)
A=[2 -4 -2 3; 6 -9 -5 8; 2 -7 -3 9; 4 -2 -2 -1; -6 3 3 4]
[C,p]=noll(A);
%(b)
A=ones(5)
[C,p]=noll(A);
%(c)
A=[magic(4),ones(4,1)]
[C,p]=noll(A);
%(d)
A=[pascal(4);ones(2,4)]
[C,p]=noll(A);
```

Part 3: Row Space
Next, you will perform the following tasks in the Live Script to create a basis for the Row A in two different ways as described below:

**First, input the matrix:
```
A=[2 -4 -2 3; 6 -9 -5 8; 4 -5 -3 5; 4 -2 -2 -1; -6 3 3 4]
```

**Next, use the function `[ , ]=shrink()` to output and display a matrix, `rowbasis1`, whose columns form a basis for the Row A. The function has to also output and display the vector `pivot1` of the indexes of the rows of A which form the basis, that is, the indexes of the rows of A which are both linear independent and span the Row A.

**Then, output and display the matrix `B=rref(A)` and use this matrix and the function `[,]=shrink` () to output and display the matrix `rowbasis2` whose <u>columns</u> form a basis for the Row A as well. The function has to also output and display the vector `pivot2` of the indexes of non-zero rows of `B` that form the basis.

%Analyze your outputs `pivot1` and `pivot2` and write a comment on a possible effect of row operations on linear dependence/independence relations among the rows of a matrix.

Part II. Isomorphism & Change of Basis

Exercise 3 (5 points) **Difficulty: Hard**
In this exercise, you will be given a set B of *n* polynomials. The polynomials in B are from the subspace P_{n-1} of the polynomials whose degrees do not exceed $(n-1)$. The standard basis for P_{n-1} is $E = \{x^{n-1}, x^{n-2}, \dots, x, 1\}$, where *x* is in $\mathbb{R}$.
You will determine whether the given set B forms a basis for P_{n-1}. That could be done by employing isomorphism from P_{n-1} onto $\mathbb{R}^n$ in the following way: first, we create a matrix P of the E-coordinate vectors of the polynomials in B and, then, we check if the set of the columns of P forms a basis for $\mathbb{R}^n$. According to the isomorphism, the set of *n* polynomials in B forms a basis for the subspace P_{n-1} if and only if the set of the *n* columns of P forms a basis for $\mathbb{R}^n$.
If the set B is not a basis for P_{n-1}, you will construct a new basis B by replacing some polynomials in the original set B with the polynomials from the standard basis E.
Then, you will work with the basis B (original or constructed) and perform two more tasks:
(1) You will find the B-coordinate vector, `q`, of a polynomial Q that is given in symbolic form through the standard basis E;
(2) you will output the polynomial R in symbolic form through the standard basis E, given its B-coordinate vector `r`.

For help with this exercise, you may find it useful to review the second Example in the Lecture Notes for Module 19.

We will be using the function `closetozeroroundoff()` with p = 7 within our code. This function was created in Project 0 and you should have it in your Current Folder in MATLAB.

**Create a function in a file that begins with

```
function P=polyspace(B,Q,r)
format
n=numel(B);
```

The input $B = [B(1),\ B(2),\ \dots\ , B(n)]$ is a vector whose *n* components are polynomials from the vector space P_{n-1}, the input `Q` is a single polynomial from the same space P_{n-1}, and `r` is a numerical column vector with *n* entries. The output P is a matrix whose columns are either the E-coordinate vectors of the polynomials in B if the polynomials form a basis for P_{n-1} or a new matrix constructed using the E-coordinate vectors of the polynomials in B and some other polynomials such that its columns form a basis for $\mathbb{R}^n$.
<u>Note</u>: The number *n* introduced above is also the dimension of the vector space P_{n-1}; thus, P_{n-1} is isomorphic to the Euclidean space $\mathbb{R}^n$.

Note on the format of the input polynomials: The input polynomials will be written through the standard basis E in the descending order according to the degree. For the purpose of this program, it is required that the coefficient of the leading term x^{n-1} of a polynomial must not be zero. However, the zero leading coefficient is accepted by the definition of the subspace P_{n-1}, and some of the input polynomials do not have term x^{n-1}, that is, the coefficient of x^{n-1} is a zero. To be able to work with such polynomials, we insert the coefficient 10^(-8) of x^{n-1}, and, then, we will convert it to a 0 by running the function `closetozeroroundoff` with p = 7 on the matrix P.

**Continue your function `polyspace` with outputting the matrix P:
First, pre-allocate a matrix
```
P=zeros(n);
```
and, then, re-assign to the *i*th column of P, `P(:,i)`, the vector of coefficients of the polynomial `B(i)` using the command `sym2poly(B(i)) (i=1:n)`.
Note: in general, the command `sym2poly()` when applied to a polynomial written through the basis E outputs the row vector of the coefficients of the polynomial.
Your output for this part will be the matrix P (do not display it here).

**Then, convert to a 0 all entries of the matrix P that are in the range of 10^(-7) from a 0 by running the function:
```
P=closetozeroroundoff(P,7);
```
and display the matrix P with a message:
```
fprintf('matrix of E-coordinate vectors of the polynomials in B is\n')
P
```

Next, **continue your code with a conditional statement** that will check if the columns of P form a basis for $\mathbb{R}^n$ - use here the command `rank()`.
Note: do not employ command `det()` since the determinant of a matrix may be calculated in MATLAB with a round off error.

**If the columns of P form a basis for $\mathbb{R}^n$, output a message:
```
fprintf('the polynomials in B form a basis for P%d\n',n-1)
```

**Otherwise, if the columns of P do not form a basis for $\mathbb{R}^n$, then, due to the isomorphism, the polynomials in B do not form a basis for P_{n-1}. In this case, output a message:
```
fprintf('the polynomials in B do not form a basis for P%d\n',n-1)
```
and continue your code for this case with constructing a basis for P_{n-1} by removing some polynomials from B and adding to B some polynomials from the standard basis E.

**To perform this task, run the function `shrink` as below:
```
[~,P]=shrink([P eye(n)]);
```
Notice that the columns of the new matrix P form a basis for $\mathbb{R}^n$. This matrix P will be the matrix of the E-coordinates of the new set B of the polynomials in P_{n-1}, which, due to the isomorphism, forms a basis for P_{n-1}.
Display the matrix P as below:
```
disp('the new matrix P is')
P
```

**Proceed with constructing the new set B in the following way:
Use a "for" loop `(i=1:n)` and the function `poly2sym` that you will run on an `i`th column of P to output the polynomial `B(i)`. The whole set of these polynomials will form a new basis `B`.
Output and display the constructed basis `B` as below:

```
disp('the constructed basis is')
B
```

This is the end of the conditional statement.

After completing the conditional statement above, you will have as an output the matrix P of the E-coordinate vectors of the polynomials in the basis B, where B is either the original set or the constructed basis. The columns of P form a basis for $\mathbb{R}^n$.

**You will use the matrix P to proceed with two more tasks:
(1) Given a polynomial Q written in symbolic form through the standard basis E, output the B-coordinate vector, `q`, of Q.
Hint: First, use the MATLAB command `sym2poly()` that outputs the E-coordinate (row) vector of the polynomial Q. Then, transpose it to a column vector and run the function `closetozeroroundoff` with p = 7 on that vector to convert the leading entry to a 0 if needed. After that, you can proceed with outputting the B-coordinate vector `q` of Q using the Change-of-Coordinates (matrix) equation.
Your displayed outputs for this part will be a message and the vector `q` as below:

```
fprintf('the B-coordinate vector of Q is\n')
q
```

(2) Given the B-coordinate vector, `r`, of a polynomial R, output the polynomial R in symbolic form through the standard basis E.
Hint: first, output the E-coordinate vector of the polynomial R by using the Change-of-Coordinates equation; next, run the function `closetozeroroundoff` with p = 7 on the output vector to convert to zero the entries that are within the given margin from a 0; and, finally, use the obtained vector and the command `poly2sym()` to output the required polynomial R.
The displayed outputs for this part is a message and the polynomial R as below:

```
fprintf('the polynomial whose B-coordinates form the vector r is\n')
R
```

This is the end of the function `polyspace`.

**Print the functions `closetozeroroundoff`, `shrink`, and `polyspace` in the Live Script.
**Then, type in the Live Script

```
syms x
```

This command introduces a symbolic variable *x*. It will allow you to input the polynomials in the variable *x* by typing (or copying and pasting) the inputs B and Q as they are given below.

** Run the function `P=polyspace(B,Q,rB);` on the choices (a)-(f) as indicated below:

```
%(a)
B=[x^3+3*x^2,10^(-8)*x^3+x,10^(-8)*x^3+4*x^2+x,x^3+x]
Q=10^(-8)*x^3+x^2+6*x+3
r=[2;-1;3;-2]
P=polyspace(B,Q,r);
%(b)
B=[x^3-1,10^(-8)*x^3+2*x^2,10^(-8)*x^3+x,x^3+x]
```

```
P=polyspace(B,Q,r);
%(c)
B=[x^3+1,10^(-8)*x^3+x^2+1,10^(-8)*x^3+x+1,10^(-8)*x^3+1]
Q=10^(-8)*x^3+3*x^2+x+6
r=[1;-4;2;3]
P=polyspace(B,Q,r);
%(d)
B=[x^4+x^3+x^2+1,10^(-8)*x^4+x^3+x^2+x+1,10^(-8)*x^4+x^2+x+1,10^(-
8)*x^4+x+1,10^(-8)*x^4+1]
Q=10^(-8)*x^4+3*x^3-1
r=diag(pascal(5))
P=polyspace(B,Q,r);
%(e)
B=[x^3+3*x^2,10^(-8)*x^3+x,x^3+3*x^2,2*x^3+6*x^2+x]
Q=10^(-8)*x^3+x^2+6*x+3
r=[-1;3;2;4]
P=polyspace(B,Q,r);
%(f)
B=[x^3+2*x^2+3*x+1,2*x^3+4*x^2+6*x+2,3*x^3+6*x^2+9*x+3,10^(-8)*x^3-2*x^2+1]
Q=10^(-8)*x^3+x^2+6*x+3
r=[-2;1;-3;5]
P=polyspace(B,Q,r);
```
Note: `Q` and `r` in choice (b) are the same as in choice (a).

Part III. Matrix Factorization

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

Theory: In this exercise, we will work with QR factorization of an $m \times n$ matrix. First, for an $m \times n$ matrix A, we will use a MATLAB built-in function to output an $m \times m$ unitary (or orthogonal) matrix Q and an $m \times n$ upper-triangular matrix R, such that, $A = Q * R$.

Definition: A matrix Q is called unitary (or orthogonal) if Q is a square $m \times m$ matrix and $Q^T * Q = eye(m)$.

Note: from the Invertible Matrix Theorem it follows that Q is an orthogonal matrix if and only if $Q^{-1} = Q^T$.

When A is a square $m \times m$ matrix, we will use the procedure outlined below to create a sequence of similar matrices $A, A_1, A_2, \ldots$ and use them to approximate the eigenvalues of A based on the fact that the similar matrices have the same eigenvalues and with the same multiplicities. To do that, you will go through the following steps:
(1) decompose A into a product $Q * R$ and interchange the factors to create $A_1 = R * Q$;
(2) decompose A_1 into a product $Q * R$ (with the new matrices Q and R) and interchange the factors to create $A_2 = R * Q$; and so on.

For certain square matrices, such as, symmetrical, tridiagonal, Hessenberg matrices, this process produces a sequence of matrices $A, A_1, A_2, \ldots$, all similar to A, such that after a number of iterations, the matrices in the sequence become close (within some margin) to an upper triangular matrix whose diagonal entries are used to approximate the eigenvalues of A.

**Create a function in MATLAB that begins with:

```
function [Q,R] = quer(A)
format
[m,n]=size(A);
```

**Generate a QR factorization of A by running the MATLAB built-in function as:

```
[Q,R]=qr(A)
```

(`Q` and `R` have to be displayed).
**Then, your function has to verify that you have got an orthogonal-triangular decomposition of A. The following conditions need to be verified:
(1) that we got a factorization, that is, `closetozeroroundoff(A-Q*R,7)` outputs zero matrix;
(2) that Q is unitary, that is, `closetozeroroundoff(Q'*Q-eye(m),7)` outputs zero matrix;
(3) R is an upper triangular matrix, that is, `istriu(R)` holds.
If all conditions (1) - (3) are "true", output a message:

```
disp('Q*R forms an orthogonal-triangular decomposition of A')
```

Otherwise, output something like `'What is wrong?!'` and <u>terminate</u> the program.

If $m \neq n$ your code will terminate here.

Otherwise, if A is a square matrix, you will continue with creating a sequence of matrices $A, A_1, A_2, \ldots$ all similar to A, as described in the **Theory**. Proceed as follows:
**Initialize

```
k=0;
```

and set up a "while" loop to get the consecutive iterations: to make the code simpler, you can re-assign each consecutive iteration to A again, and your loop will run "while" the condition

```
any(closetozeroroundoff(A-triu(A),9),'all')
```

holds.
Assign your last iteration to the matrix B and count the number of the iterations, k.
<u>Note</u>: If a matrix A was originally an upper triangular matrix, then your output B has to be the matrix A itself, and the number of iterations k has to be 0.
(Please make sure that you will suppress all intermediate iterations in the loop.)

**Display your outputs as below:

```
fprintf(['the matrix which is both close to upper triangular\n'...
    'and similar to A is\n'])
B
fprintf('the number of iterations is %i\n',k)
disp('the vector of eigenvalues of matrix A is approximated by')
E=diag(B)
```

**Next, you will verify that your approximations of the eigenvalues are correct. Place the following lines into your code to compare your outputs and outputs of a MATLAB function:

```
E=sort(E);
L=eig(A);
L=sort(L);
if ~any(closetozeroroundoff(E-L,7))
    disp('Great! The eigenvalues are found correctly!')
else
    disp('Oh no! I need to check the code!')
end
```

Please be sure to check your code if you have received the last message.

This is the end of your function `quer`.

**Print the functions `closetozeroroundoff` and `quer` in your Live Script.
**Run the function `[Q,R]=quer(A);` on the choices (a)-(h) as indicated below:

```
%(a)
A=randi(10,4,5)
[Q,R] = quer(A);
%(b)
A=ones(5,4);
[Q,R] = quer(A);
%(c)
A=diag([1,2,3,4,5])
[Q,R] = quer(A);
%(d)
A=triu(magic(4))
[Q,R] = quer(A);
%(e)
A=tril(magic(4),1)
[Q,R] = quer(A);
%(f)
A=triu(magic(5),-1)
[Q,R] = quer(A);
%(g)
A=tril(A,1)
[Q,R] = quer(A);
%(h)
A=[1 1 4;0 -4 0;-5 -1 -8]
[Q,R] = quer(A);
```

Note: the input in choice (g) is the matrix A from choice (f).

Exercise 5 (6 points) **Difficulty: Moderate**

In this exercise, we will construct LU factorization of an $m \times n$ matrix A. The MATLAB command `[L,U] = lu(A)` returns a permuted lower-triangular matrix L and an upper-triangular matrix U, such that, $A = L * U$. The output U is an echelon form of matrix A or resembles an echelon form of A. The output L is a matrix whose rows can be rearranged (permuted) to a lower-triangular matrix with 1's on the main diagonal.

In our algorithm, we will create LU factorization, which, in some cases, will be different from the one produced by MATLAB. We will also output the permutation matrix P which has the property that left-multiplication of the matrix L by P returns a lower triangular matrix with 1's on the main diagonal (called a *unit lower triangular matrix*). Our matrix P, in some cases, will match the output P of the MATLAB function `[L,U,P] = lu(A)`.

To learn about another algorithm that can be used for LU factorization, which is based on the "inspection" of the row reduction algorithm performed on a matrix, please read Section 2.5 of the textbook and the corresponding section in the "Study Guide" which you can access through the link "Tools for Success" on MyLab & Mastering.

Part 1
In this part of the Exercise, we will write a function that generates LU factorization of a matrix A by using as a basis the function `redef` constructed in Exercise 2 of Project 2.

Description of the Algorithm:
An $m \times n$ matrix A can be reduced to an echelon form R by using only row replacement and row interchanging operations. We also know that an elementary row operation can be performed on a matrix by left-multiplying it by the corresponding elementary matrix. We can construct a sequence of elementary row operations that involves *partial pivoting* (see Numerical Note on Page 17 of the textbook) and reduce A to an echelon form R without using scaling. If we denote E_j an elementary matrix that performs one of the two types of the elementary operations in that sequence (row replacement or row interchanging), then, after p steps, we will arrive at an echelon form R of the matrix A, that is,

$$R = \left(E_p \ldots E_1\right)A.$$

Therefore,

$$A = (E_p \ldots E_1)^{-1}R = \left({E_1}^{-1} \ldots {E_p}^{-1}\right)R = LU$$

Here, we denoted $U = R$ and

$$L = {E_1}^{-1} \ldots {E_p}^{-1} \qquad (1)$$

Note: The matrix U is an $m \times n$ upper triangular matrix which is an echelon form of A.
Here are some notes on the matrix L:
We know that the inverse of an elementary matrix is of the same type as the elementary matrix itself; moreover, the inverse matrix of an elementary matrix of Type 2 (which performs row interchanging) is the elementary matrix itself. If we were able to perform the row reduction algorithm by using only row replacements, that is, all elementary matrices E_j were of the Type 1, then, as it can be seen from the representation (1), L would be a lower triangular matrix with 1's on the main diagonal. However, it is not always possible to avoid row interchanges and, if you remember, we were using partial pivoting when running the row reduction algorithm: we were interchanging rows to move into the pivot position the entry with the largest absolute value in its column if that entry is not located in the pivot position. When we are performing the row interchanging operation (permutation), it places an elementary matrix of Type 2 into the product ${E_1}^{-1} \ldots {E_p}^{-1} = L$. We can keep track on those permutations and construct a matrix P which accumulates the left products of the elementary matrices of Type 2, which are the same as their inverses, in the order they were placed into the product ${E_1}^{-1} \ldots {E_p}^{-1}$. In this way, left multiplication of the output matrix L by P will permute L into a unit lower triangular matrix - this explains the reason of calling L permuted lower triangular matrix. We can also count the number of permutations used and employ that number when computing the determinant.
**Create a function in MATLAB that begins with

```
function [L,U,N,L1,P]=elu(A)
format
[m,n]=size(A);
rankA=rank(A);
A=sym(A);
R=A;
L=eye(m);
P=eye(m);
N=0;
```

The function takes as an input an $m \times n$ matrix A. The outputs L and U will form LU factorization of the matrix A, the output P will be the permutation matrix that converts L into a unit lower triangular matrix L1, and N is a number of permutations for this transformation.

Continue your function with copying and pasting the compete `Forward Phase` from the function `redef`, which reduces matrix R to an echelon form, and make the following additions to the code (see the Description of the Algorithm):
**There is a line in your code where you used an elementary matrix of Type 2 (`ele2`) to perform a row interchange - place two more commands right below that line: one command reassigns to L the right product of L by `ele2` and the second command reassigns to P the left product of P by `ele2`. Also, increase by 1 the number of permutations N.
**There is another line in your code where you used an elementary matrix of Type 1 (`ele1`) to perform a row replacement – place one more command right below that line: reassign to L the right product of L by the inverse of the matrix `ele1`.

After completing this part, you will have the following output matrices: matrix R which is echelon form of A obtained by using partial pivoting and the least possible number of row operations, matrix L which is a permuted lower triangular matrix, and matrix P which is a permutation matrix.
**Next, output (do not display):
```
U=R;
L1=P*L;
```

If the rank of A is less than m and U is an echelon form of the matrix A, then U has at least one row of zeros on the bottom. We will adjust the matrices L and U by deleting the zero rows from U and the corresponding columns from L.
**Compose a conditional statement in a way such that, if `rankA<m`, your code will reassign to U a matrix formed by the nonzero rows of U, which are in the range `1:rankA`, and re-assign to L a matrix formed by the columns of L that are in the range `1:rankA`. Suppress all these outputs.

Further, we proceed with a verification that our LU factorization is constructed as correctly.
**First, verify that we do have a factorization, that is `A=L*U` within some margin, using the condition `~any(closetozeroroundoff(A-L*U,7),"all")`. If the condition holds, output
```
U=double(U);
```
otherwise, assign:
```
L=[];
U=[];
```
(do not terminate the code here).
**Then, verify that `L1` is a unit lower triangular matrix, that is, both conditions hold:
`L1` is lower triangular and it has 1's on the main diagonal.
If it is not the case, output:
```
L1=[];
P=[];
```
Note: We are suppressing all outputs in the function **`elu`**.

This is the end of the function `elu`.

**Print the functions `elu` and `closetozeroroundoff` in your Live script.
**Run the function on the choices (a)-(k) as indicated below:

```
%(a)
A=[0 0 1;1 0 0;0 1 0]
[L,U,N,L1,P]=elu(A)
%(b)
A=magic(3)
[L,U,N,L1,P]=elu(A)
%(c)
A=magic(4)
[L,U,N,L1,P]=elu(A)
%(d)
A=[1 1 4 3;0 -4 0 2;-5 -1 -8 1]
[L,U,N,L1,P]=elu(A)
%(e)
A=[1 1 4;0 -4 0;-5 -1 -8; 2 3 -1]
[L,U,N,L1,P]=elu(A)
%(f)
A=[1 2 3;2 4 6;3 6 4]
[L,U,N,L1,P]=elu(A)
%(g)
A=magic(5)
[L,U,N,L1,P]=elu(A)
%(h)
A=randi(10,6,3)
[L,U,N,L1,P]=elu(A)
%(i)
A=randi(10,3,6)
[L,U,N,L1,P]=elu(A)
%(j)
A=pascal(5)
[L,U,N,L1,P]=elu(A)
%(k)
A=hilb(3)
[L,U,N,L1,P]=elu(A)
```

Please notice: the function `elu` will be used in other functions that we will create later – it is very important to have a correct code. Please be sure to perform verification of your outputs correctly, and, if you are receiving any empty outputs, please consider revision of the code.

Part 2
Theory: MATLAB uses LU factorization, $A = L * U$, to output the inverse matrix of A by, first, inverting L and U and, then, computing $A^{-1} = U^{-1} * L^{-1}$.
By using the outputs L, U of the function `elu` created in Part 1, we will be able to compute the inverse of an invertible matrix A in the way MATLAB does it and compute the determinant of A using the outputs U, N.

**Create a function in MATLAB that begins with the commands:

```
function [invA,detA] = eluinv(A)
[~,n]=size(A);
```

The input of this function is an $n \times n$ matrix A. The output `detA` is the determinant of A and, for an invertible matrix A, the output `invA` is the inverse matrix of A. If A is not invertible, the output for `invA` will be empty and the `detA` will be a 0.
To obtain the required outputs for an invertible matrix A, we will be using the function `[L,U,N,~,~]=elu(A)` constructed in Part 1.

Continue your function with a check if A is invertible (use the command `rank`).
**If A is not invertible, output the following:

```
fprintf('A is not invertible')
invA=[];
detA=0;
```

and terminate the program.

If A is invertible, your code will continue.
**Place the line below into your code:

```
[L,U,N]=elu(A);
```

**Then, compute the inverses of the matrices L and U, `invL` and `invU`, by applying MATLAB function `rref()` to the matrices `[L eye(n)]` and `[U eye(n)]` and assigning to `invL` and `invU` the last *n* by *n* blocks of the reduced echelon forms of the augmented matrices above, respectively. (Due to the structure of the matrices L and U, these computations will not take too many arithmetic operations.) Do not display `invL` and `invU`.

Output (do not display) the inverse matrix of A, `invA`, using the constructed matrices `invL` and `invU` (see the **Theory above).
**Run the MATLAB command as indicated below to output the reference matrix:

```
F=inv(A);
```

**Verify that the matrices `invA` and `F` match by using the function `closetozeroroundoff` with p=7. If `invA` and `F` do not match within the given precision, assign:

```
invA=[];
```

(do not terminate the code here).

**Next, compute the determinant of the matrix A by using the matrix U, the number of permutations N (both are outputs of the function `[L,U,N]=elu(A)`), and the MATLAB function `prod()`. Assign your output to `detA` (do not display it).

**Run the MATLAB command as indicated below to output the reference value:

```
d=det(A);
```

**Verify that `detA` and `d` match by using the function `closetozeroroundoff` with p=7. If `detA` and `d` do not match within the given precision, assign:

```
detA=[];
```

Note: receiving an empty output for an invertible matrix A after running the function on the choices given below should prompt you to make corrections in your code.

This is the end of the function `eluinv`.

**Print the function `eluinv` in your Live Script.
**Run the function on the choices (a)-(f) as indicated below:

```
%(a)
A=[1 1 4;0 -4 0;-5 -1 -8]
[invA,detA] = eluinv(A)
%(b)
A=magic(4)
[invA,detA] = eluinv(A)
%(c)
A=[2 1 -3 1;0 5 -3 5;-4 3 3 3;-2 5 1 3]
[invA,detA] = eluinv(A)
%(d)
A=magic(5)
[invA,detA] = eluinv(A)
%(e)
A=hilb(3)
[invA,detA] = eluinv(A)
%(f)
A=pascal(4)
[invA,detA] = eluinv(A)
```

Part 3

LU factorization can be successfully used for solving simultaneously p matrix equations ($p \geq 2$) with the same invertible matrix A. This algorithm significantly reduces the number of arithmetic operations, or "flops" (floating point operations) in comparison with the other methods, provided L and U have been calculated.

Suppose that we have p matrix equations:

$$A\mathbf{x} = \mathbf{b}_1, \quad A\mathbf{x} = \mathbf{b}_2, \quad \dots, \quad A\mathbf{x} = \mathbf{b}_\text{p}.$$

Let $B = [\mathbf{b}_1 \ \mathbf{b}_2 \ \dots \ \mathbf{b}_p]$. To solve simultaneously the p systems above, we can find a solution X of the matrix equation

$$AX = B,$$

where $X = [\mathbf{x}_1 \ \ \mathbf{x}_2 \ \ \dots \ \ \mathbf{x}_p]$ is the matrix whose columns are the solutions of the p systems, respectively, that is, the following holds:

$$AX = A[\mathbf{x}_1 \ \ \mathbf{x}_2 \ \ \dots \ \ \mathbf{x}_p] = [A\mathbf{x}_1 \ A\mathbf{x}_2 \ \ \dots \ \ A\mathbf{x}_p] = [\mathbf{b}_1 \ \ \mathbf{b}_2 \ \ \dots \ \ \mathbf{b}_p] = B.$$

If A is invertible, each system has a unique solution and we will use three methods to find the solution.

Our <u>first</u> method for finding the solution will be using the MATLAB inverse matrix:

$$X = \text{inv}(A) * B \qquad (1)$$

The <u>second</u> method will be using the MATLAB "backslash" operator:

$$X = A\backslash B \qquad (2)$$

And our <u>third</u> method will be based on the LU factorization of A:

Assume that we have LU factorization of an invertible matrix A: $A = LU$. The logical chain below demonstrates the procedure of using LU factorization to find the solution X:

$$AX = B \quad \Leftrightarrow \quad L(UX) = B \quad \Leftrightarrow \quad LY = B \ \ \& \ \ UX = Y \quad \overset{\text{output}}{\Rightarrow} \quad X \qquad (3)$$

That is, to output X, we, first, solve $LY = B$ for Y, and, then, we use Y to solve $UX = Y$ for X.

**Create a function in MATLAB:

```
function [X1,X2,X] = msystem(A,B)
[~,n]=size(A);
[~,p]=size(B);
X1=[];
X2=[];
X=[];
```

The inputs are an $n \times n$ matrix A and $n \times p$ matrix B. The outputs `X1,X2,X` will be uniquely defined solutions calculated by the three methods above if A is invertible, or they will stay empty if A is not invertible.

**Check if A is not invertible. If it is the case, output a message

```
disp('A is not invertible and there is no unique solution')
```

and terminate the code. The empty outputs for `X1,X2,X` assigned previously will stay.

If A is invertible, your code will continue.
**First, output the solution $X1$ of $AX = B$ according to the formula in (1). Display it with a message:

```
disp('the solution calculated by using the inverse of A is')
disp(X1)
```

**Second, output the solution $X2$ of $AX = B$ according to the formula in (2). Display it with a message:

```
disp('the solution calculated by using the backslash operator is')
disp(X2)
```

**Third, output the solution X of $AX = B$ by using the algorithm in (3) which employs LU factorization. Proceed in the following way:
Place into your code the following command:

```
[L,U] = elu(A);
```

which outputs the matrices L and U.
Then, output the solution Y (which is a matrix with p columns) of the system $LY = B$ by using the row reduction algorithm – you will employ the command `rref()` in your code. Then, output the solution X of the equation $UX = Y$ (Y is the output from the previous step) using the row reduction algorithm as well. The output matrix X must have p columns (remember, we are solving simultaneously p matrix equations). Display the solution X with a message:

```
disp('the solution calculated by using LU factorization is')
disp(X)
```

**Next, we will run the function `closetozeroroundoff` with p=7 on each of the matrices (X1-X2) and (X1-X) to verify that the solutions calculated by different methods match. If both outputs are zero matrices, display the message:

```
'The solutions calculated by different methods match'
```

Otherwise, output something like:

```
'What! They do not match! Something is off...'
```

This is the end of the function `msystem`.

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

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

```
%(a)
A=magic(4), B=pascal(4);
[X1,X2,X] = msystem(A,B);
%(b)
A=[1 1 4;0 -4 0;-5 -1 -8], B=hilb(3)
[X1,X2,X] = msystem(A,B);
%(c)
A=magic(5), B=randi(10,5,3)
[X1,X2,X] = msystem(A,B);
%(d)
A=pascal(3),B=randi(10,3,5)
[X1,X2,X] = msystem(A,B);
%(e)
A=magic(3), B=[magic(3),eye(3)]
[X1,X2,X] = msystem(A,B);
```

BONUS: (1 point)
%Analyze the output X for the choice (e) and write comments in the Live Script on the matrices that form the left 3-by-3 block and the right 3-by-3 block of the solution X. You will need to run in the Live Script some logical commands to support your statements.

Part IV. Applications

In this exercise, you will work with Markov Chains. Please read the **Theory** below and perform the indicated tasks.

Theory: A vector with nonnegative entries that add up to 1 is called a ***probability vector***. A ***stochastic matrix*** is a square matrix whose columns are probability vectors.
Important Note: in this Exercise, the definition of the stochastic matrix matches the definition of the left-stochastic matrix in Exercise 5 of Project 1.
A ***Markov chain*** is a sequence of probability vectors $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$ together with a stochastic matrix *P*, such that

$$\mathbf{x}_1 = P\mathbf{x}_0, \quad \mathbf{x}_2 = P\mathbf{x}_1, \quad \mathbf{x}_3 = P\mathbf{x}_2, \quad \ldots .$$

Thus, a Markov chain is described by the first-order difference equation

$$\mathbf{x}_{k+1} = P\mathbf{x}_k, \ \ k = 0,1,2,\ldots$$

The *n* entries of a vector $\mathbf{x}_k$ list, respectively, a probability that the system is in one the *n* possible states. For this reason, $\mathbf{x}_k$ is called a ***state vector***. If *P* is a *stochastic matrix*, then a ***steady-state vector*** for *P* is a *probability vector* **q** which is a solution of the equation $P\boldsymbol{q} = \boldsymbol{q}$, or, equivalently, a solution of the equation $(P - eye(n))\boldsymbol{q} = \mathbf{0}$.
A *stochastic matrix P* is called ***regular*** if some matrix power P^k contains only strictly positive entries.

Theorem: If *P* is an $n \times n$ *regular stochastic* matrix, then *P* has a *unique steady-state* vector **q**. Further, if $\mathbf{x}_0$ is any initial state and $\mathbf{x}_{k+1} = P\mathbf{x}_k$ for $k = 0,1,2,\ldots$, then the Markov chain $\{\mathbf{x}_k\}$ converges to **q**, as $k \to \infty$.
For more on Markov Chains, read the textbook Section 4.9, Applications to Markov Chains.

Exercise 6 (4 points): **Difficulty: Easy**

**Create a function in a file that begins with

```
function q=markov(P,x0)
format
[n,~]=size(P);
q=[];
```

The input P is an $n \times n$ matrix and the input `x0` is the initial state vector of the size $n \times 1$. First, set a conditional statement to check if the given matrix P, which will have positive entries, is stochastic, that is, left-stochastic.

**If P is not left-stochastic, the display a message

```
disp('P is not a stochastic matrix')
```

and the empty output for `q` will stay.

**If P is left-stochastic (then it will be regular stochastic), display a message

```
disp('P is a stochastic matrix')
```

and, then, output the unique steady-state vector `q`, which is the probability vector that forms a basis for the Null space of the matrix (`P-eye(n)`). Employ here a MATLAB command `null(,'r')` to output a basis for the Null space and, then, scale the vector in the basis to get the required probability vector `q`. Display `q` with a message:

```
disp('the steady-state vector is')
q
```

**Next, initialize

```
k=0;
```

and verify that the Markov chain converges to `q` by calculating consecutive iterations

$$\mathbf{x}_1 = P * \mathbf{x}0, \quad \mathbf{x}_2 = P * \mathbf{x}_1, \quad \mathbf{x}_3 = P * \mathbf{x}_2, \quad \ldots .$$

**To make the code simple, you can set up a "while" loop and assign each consecutive iteration to `x0` again. The loop will run while the condition below holds:

```
any(closetozeroroundoff(x0-q,7))
```

Count the number of iterations `k` that is required to archive the required accuracy and display `k` in your code with a message:

```
fprintf('number of iterations to archive required accuracy is %i\n',k)
```

Note: if `x0=q`, then `k=0`.

This is the end of your function `markov`.

**Print the function `markov` and `closetozeroroundoff` in your Live Script.
**Run the function `q=markov(P,x0);` on the sets of variables as indicated below:

```
%(a)
P=[.6 .3;.5 .7], x0=[.4;.6]
q=markov(P,x0);
%(b)
P=[.5 .3;.5 .7],x0=[.5;.5]
q=markov(P,x0);
%(c)
P=[.9 .2;.1 .8], x0=[.11;.89]
q=markov(P,x0);
%(d)
P=[.9 .2;.1 .8], x0=[.90;.10]
```

```
q=markov(P,x0);
%(e)
P=[.90 .01 .09;.01 .90 .01;.09 .09 .90], x0=[.5; .3; .2]
q=markov(P,x0);
%(f)
P=magic(5); P=P./sum(P), x0=randi(10,5,1);x0=x0./sum(x0)
q=markov(P,x0);
%(g)
x0=q
q=markov(P,x0);
```

Note: in choice (g), the input matrix `P` is same as in (f) and the input vector `x0` is the output `q` from the choice (f).

Exercise 7 (3 points) **Difficulty: Easy**

Theory: In this Exercise you will work with **Green's** Theorem from Vector Calculus and apply it to calculation of the area of a planar region bounded by a closed polygonal curve. We denote the distinct vertices of a polygon with n sides as $(x_1, y_1),\ (x_2, y_2),\ \dots, (x_n, y_n)$, and we set $(x_{n+1}, y_{n+1}) = (x_1, y_1)$. The vertices of the polygon are listed in the counterclockwise order to match the positive orientation of the boundary of the region in **Green's** Theorem.

The area of the polygon can be calculated by the formula:

$$Area = \frac{1}{2}\sum_{i=1}^{n}(x_i y_{i+1} - y_i x_{i+1}) \qquad (1)$$

which is derived from the formula for the area A in terms of the Line Integral of a vector field:

$$A = \frac{1}{2}\oint_C xdy - ydx \qquad (2)$$

The formula in (2) is obtained from the Green's Theorem, where the boundary of the planar region, the curve C, is a simple closed piecewise-smooth curve oriented counterclockwise.

**Create a function that begins with

```
function Area = polygon(E)
A=eye(2);
C=transf_1(A,E);
n=size(E,2)-1;
```

The input E is a $2 \times (n+1)$ matrix whose columns are the coordinates of the vertices of the polygon listed in the counterclockwise order.

The function `C=transf_1(A,E)` is a light modification of the function `C=transf(A,E)` that was created in Project 2, Exercise 5: you will need just to replace the viewing window in the function `transf` with `v=[0 21 0 21];` and save the modified function in your Current Folder in MATLAB as `transf_1`. The output of your function `C=transf_1(A,E);` will be a graph of the polygon whose vertices are defined by the columns of the input matrix E.

**Your function will proceed with outputting of the area of a polygon by using formula (1), where the x-coordinates are the ordered entries of the first row of the input matrix E and the y-coordinates are the ordered entries of the second row of E.

This is the end of the function `polygon`.

**Print the functions `transf_1` and `polygon` in your Live Script.

**Run the function `Area = polygon(E)` on the variables as indicated below (make sure that the graph of the polygon and the output `Area` are displayed):

```
%(a)
E=[2 14 14 2 2;2 2 14 14 2];
Area = polygon(E)
%(b)
E=[2 10 15 11 4 2;8 5 7 12 10 8]
Area = polygon(E)
%(c)
A1=randi(10,1,5);
A1=sort(unique(A1), 'ascend');
B1=randi(10,1,size(A1,2));
B1=sort(B1,'descend');
A2=randi([11 20],1,5);
A2=sort(unique(A2), 'ascend');
B2=randi(10,1,size(A2,2));
B2=sort(B2,'ascend');

A3=randi([11 20],1,5);
A3=sort(unique(A3), 'descend');
B3=randi([11 20],1,size(A3,2));
B3=sort(B3,'ascend');

A4=randi(10,1,5);
A4=sort(unique(A4), 'descend');
B4=randi([11 20],1,size(A4,2));
B4=sort(B4,'descend');

E=[A1 A2 A3 A4 A1(1,1);B1 B2 B3 B4 B1(1,1)]
Area = polygon(E)
```

BONUS (2 points!)
Insert Section Break in the Live Script after running the function `polygon` on the choices (a)-(c) and proceed with writing a code fragment that will approximate the area of a circular disk of radius `R` and center `r0=[x0;y0]` by using inscribed polygons of *n* sides of the equal length.
<u>Note</u>: your code has to be written <u>directly</u> in the Live Scrip without creating a new function.
**Begin the code with the commands:

```
clear
R=4;
r0=[10;10];
n=3;
```

**We start with an inscribed polygon that has 3 distinct vertices (`n=3`) and use a "while" loop inside of which we will be increasing the number of vertices, *n*, until, for the first time, the `Area` of the inscribed polygon falls in the range of 10^(-1) from the area of the circle calculated by the formula

```
A=pi*R^2;
```

(use here the function `closetozeroroundoff()` with p=1 to test the condition).

For each *n* within a "while" loop, starting with `n=3`, do the following:

**First, generate a 2 x (`n+1`) matrix E. The two rows of the matrix E are the x- and y-coordinates, respectively, of the vertices of the polygon, which we will calculate by using a parametric representation of the circle

```
x=x0+Rcos(t)
y=y0+Rsin(t)   (0≤t≤2*pi)
```

where `R` is the radius and `r0=[x0;y0]` is the center of the circle, and we take for `t` the angles

```
t(i)=2*pi*(i-1)/n; (i=1:n+1)
```

which will define the vertices of the polygon that have to be located on the circle.
Notice that the last vertex (at the angle `2*pi`) coincides with the first vertex (at the angle 0).
**Next, run the function `polygon` as indicated below:

```
Area = polygon(E);
```

**Then, test if the condition for the loop termination holds. If not, increase the number of sides, *n*, by 1, and repeat the procedure above until you archive the required accuracy.

Your displayed outputs (after the loop breaks) have to be:
(1) an approximation of the area of the circle, which is the last iteration for the variable **`Area`**;
(2) the minimum number of distinct vertices of an inscribed polygon needed to archive the required accuracy, the number *n*;
(3) the graph of the inscribed polygon of the *n* sides.

Note: the bonus part of Exercise 7 has to be a separate Section in your Live Script.

THIS IS THE END OF PROJECT 3