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.

Elementary Row-Operations

Exercise 1 (5 points) **Difficulty: Hard**
In this exercise you will write a function that reduces an *m* x *n* matrix A to the reduced echelon form by implementing the Row Reduction Algorithm in the way it will be outlined below.
**Create a function in MATLAB that begins with:

```
function R=rredf(A)
format
[m,n]=size(A);
rankA=rank(A);
A=sym(A);
R=A;
```

Notes: To increase accuracy in calculations, we work here with symbolic form of the input matrix A, `A=sym(A)`, which we have assigned to the matrix R, and, after completing the Row Reduction algorithm on R, we will convert the output to a double precision matrix.
We will follow the steps in the description of the Row Reduction Algorithm in Module 2 of Lecture Notes making some adjustments that will allow us to use the least possible number of row operations and obtain a function that can be used in other functions later on.

**Continue your function with the line:

```
%Forward Phase
```

and begin with setting up a **"for" loop** with `k=1:m` that will work on submatrices of the consecutive iterations for R, which are formed by the rows of R in the range from `k:m` (and all columns), starting with the original matrix R (for `k=1`), and, for `k>1`, the submatrices of R are formed by deleting the top rows with the pivot positions that have been already completed after steps 1-3. This is described in Lecture 2 under Row Reduction Algorithm as Step 4:
"4. Cover (or ignore) the row containing the pivot position and cover all rows, if any, above it."

Next, we will apply Steps 1-3 of the Row Reduction Algorithm to submatrices within the loop:
**First, output the index of the left-most non-zero column (you can use command `any()` here), which is a pivot column and the pivot position is on its top.

**Then, to select a non-zero entry in a pivot column that will be a pivot, output the index of the row with the largest by absolute value entry in the column (you can use here the MATLAB function `[~, ]= max(abs())`), and, if that entry is not in the pivot position, use row interchanging operation to move it into the pivot position - this is called partial pivoting (see Numerical Note on Page 17 of textbook).
**If a consecutive submatrix has more than one row (`k<m`), iterate down the pivot column and create zeros below the pivot position. Proceed as follows:
If the entry in a consecutive row is not a 0, first, output the factor *r* that will be used in the formula for the row replacement operation when the row is replaced with the sum of itself and *r*-multiple of the row with the pivot position, and, then, place the command that will do the row replacement using scalar *r*.
**After completing a consecutive pivot column, place the following command into your code:

```
R=closetozeroroundoff(R,7);
```

to ensure that the entries of R that are in the range of 10^(-7) from zero will be a zero.
This is the end of the "for" loop and the end of the `Forward Phase`.

Important Note: Do not use operation of scaling in the Forward Phase - only two operations, row interchanging and row replacement, are needed to reduce a matrix to an echelon form. After completing the Forward Phase as above, the output matrix R will be an echelon form of A obtained by using partial pivoting and the least possible number of row operations.

After you are done with the Forward Phase, place into your code:

```
%Backward Phase
```

and set up a "for" loop with `k=rankA:-1:1` that would iterate through the rows of the echelon form starting with the row whose index is `rankA` (the last non-zero row) and going up to row 1 inclusively. For a consecutive row `k` do the following:
**First, output the index of the column that has a pivot position.
**Next, output the scalar *h* that will be used in the formula for scaling of this row to get the number 1 in the pivot position. Then, place a command that will do the scaling using scalar *h*.
**Then, if the index of a consecutive row is greater than 1 (`k>1`), create zeros above the pivot using row replacement operation: first, output the factor *r* that is used in the formula for the row replacement when the row is replaced with the sum of itself and *r*-multiple of the row with the pivot, and, then, place the command that will do row replacement using scalar *r* - please remember that your pivot here is the number 1.
**After completing a consecutive row, place the following command into your code:

```
R=closetozeroroundoff(R,7);
```

This is the end of the "for" loop and the end of the `Backward Phase.`

Note: Suppress all intermediate outputs in the Row Reduction Algorithm.

**After you completed Row Reduction Algorithm, output and display your matrix R as below:

```
disp('the constructed matrix R is')
disp(double(R))
```

Next, verify that R is, indeed, the reduced echelon form of A. Proceed as follows:
**Place the following set of commands into your code:

```
rf=rref(A);
if closetozeroroundoff(R-rf,7)==0
    disp('R is the reduced echelon form of A')
```

```
    R=double(R);
else
    disp('Something went wrong!')
    R=[]
end
```

Note: If you will receive the last message and the empty output for R after running your function on the choices below, please consider making corrections in the code. It is very important to have a correct code since this function will be used in some other functions that we will be creating.

This is the end of the function `rredf`.

**Print the functions `rredf` and `closetozeroroundoff` in your Live Script.

**Run the function `rredf` on the choices (a)-(i) as indicated below:

```
%(a)
A=[2 0 1 3]
R=rredf(A);
%(b)
A=[2 4 1;1 2 3;1 2 1]
R=rredf(A);
%(c)
A=[zeros(4),magic(4)]
R=rredf(A);
%(d)
A=pascal(3)
R=rredf(A);
%(e)
A=ones(3,6); A(:,1:2:5)=magic(3)
R=rredf(A);
%(f)
A=zeros(3,6); A(:,2:2:6)=magic(3)
R=rredf(A);
%(g)
A=[magic(4);hilb(4)]
R=rredf(A);
%(h)
A=[magic(4),hilb(4)]
R=rredf(A);
%(i)
A=randi(10,5,3);A=[A, sum(A,2)]
R=rredf(A);
```

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

In this exercise, you will write three functions that create the three types of elementary matrices and use these functions to reduce a matrix to the reduced echelon form.

Theory: To create an elementary *m* x *m* matrix, we apply a row operation, such as row replacement, row interchanging, or scaling, to the *m* x *m* identity matrix. This will produce an elementary matrix of one of the three types `E1`, `E2`, or `E3`, respectively. Moreover, a left-multiplication of an *m* x *n* matrix A by an elementary matrix will perform the same operation on the matrix A as the one that created that elementary matrix from the identity matrix.

Before you start working on this Exercise, please review the material of Lecture 12.

Part I

**Create three functions in MATLAB which output elementary *m* x *m* matrices `E1`, `E2`, and `E3`:

```
function E1=ele1(m,i,j,r)
function E2=ele2(m,i,j)
function E3=ele3(m,i,h)
```

In each of the functions, start with the identity matrix `eye(m)` and proceed as follows:
For `E1=ele1(m,i,j,r)`:
matrix `E1` is obtained from `eye(m)` by replacing (row *i*) with [(row *i*) + *r* · (row *j*)].
For `E2=ele2(m,i,j)`:
matrix `E2` is obtained from `eye(m)` by interchanging rows *i* and *j*.
For `E3=ele3(m,i,h)`:
matrix `E3` is obtained from `eye(m)` by multiplying row *i* by *h*.
(Here, *i, j* are the indexes of the rows in the range from 1 to *m*, and *r*, *h* are scalars.)

**Print the created functions `ele1, ele2, ele3` in your Live Script:

Next, use row reduction algorithm to reduce the matrix A given below to the reduced echelon form **by hand on paper**.
Important: Follow in your hand computations all the steps outlined in Exercise 1 of this Project, which include using partial pivoting and not using scaling in the Forward phase.

```
      0     2    -4     0
A =   2     2    -4     4
      1     1     2    -2
```

After you completed hand calculations, return to the Live Script and proceed with reducing matrix A to the reduced echelon form by means of the functions `ele1`, `ele2`, and `ele3`.
Proceed as follows:
**Type in the Live Script

```
format
```

input (and display) matrix A:

```
A=[0 2 -4 0; 2 2 -4 4; 1 1 2 -2]
```

initialize:

```
R=A;
```

and, then, follow the row operations performed by hand and use left-multiplication of the matrix R by the corresponding function `ele1`, `ele2`, or `ele3`, depending on the operation that you used, to output a consecutive iteration for R. Continue this way until you arrive at the reduced echelon form. Display all outputs for R including the intermediate ones. The last output for R has to be the reduced echelon form of A.

You will be able to verify the correctness of each intermediate output by comparing it with the matrix obtained by hand. Moreover, you will also verify the correctness of the reduced echelon form using the conditional statement below (place and run it in the Live Script):

```
rf=rref(A);
if isequal(R,rf)
    disp('the reduced echelon form of A is')
    disp(R)
```

```
else
    disp('I need to check my coding!')
    R=[]
end
```

Next, we proceed with reversion of the Row Reduction Algorithm which would take us back from the reduced echelon form R of the matrix A to the original matrix A.
Begin with the (non-empty!) output R and proceed in the Live Script with a step-by-step inversion of the operations that were performed by consecutively re-assigning to R the left product of R by the elementary function that performs the inverse operation until you arrive at the original matrix A. **Suppress all intermediate outputs in that sequence.
Recall: the inverse of an elementary matrix is of the same type as the elementary matrix itself.

Next run a check whether the row operations were reversed correctly.
**Place the following conditional statement in the Live Script:

```
if isequal(A,R)
    disp('the row reduction operations reversed correctly')
else
    disp('the inversion did not work! Hmm...?')
end
```

If you received the last message, please be sure to make corrections.

Part II

As we have seen in **Part I**, we can perform an elementary row operation on a matrix by left-multiplying it by the corresponding elementary matrix. We will write a function that uses this technique to reduce a matrix to the reduced echelon form.
**Create another function in a file that begins with:

```
function R=redef(A)
```

**Proceed with copying and pasting the body of the function `rredf` from Exercise 1 of this Project into the function `redef`, and, then, make the following changes in the function `redef`:
Replace each command that performs a row operation on a matrix in the function `rredf` with the left product of the matrix by the corresponding elementary function `ele1`, `ele2`, or `ele3` - do not delete the original command but place % sign in front of it so that the original command would not be executed but still presented in the function.

**Print the function `redef` in the Live Script.

**Test the function on the matrix A from Part I. Run in the Live Script commands given below and compare your output with the one received in Part I.

```
%test
A=[0 2 -4 0; 2 2 -4 4; 1 1 2 -2]
R=redef(A);
```

**Run the function on the choices (a)-(i) as below (the inputs are the same as in Exercise 1).

```
%(a)
A=[2 0 1 3]
```

```
R=redef(A);
%(b)
A=[2 4 1;1 2 3;1 2 1]
R=redef(A);
%(c)
A=[zeros(4),magic(4)]
R=redef(A);
%(d)
A=pascal(3)
R=redef(A);
%(e)
A=ones(3,6); A(:,1:2:5)=magic(3)
R=redef(A);
%(f)
A=zeros(3,6); A(:,2:2:6)=magic(3)
R=redef(A);
%(g)
A=[magic(4);hilb(4)]
R=redef(A);
%(h)
A=[magic(4),hilb(4)]
R=redef(A);
%(i)
A=randi(10,5,3);A=[A, sum(A,2)]
R=redef(A);
```

Solving Equations

Exercise 3 (5 points) **Difficulty: Moderate**

In this exercise, we will be solving a non-homogeneous system A**x** = **b** (**b** ≠ **0**), where A is an $m \times n$ matrix and **b** ∈ $\mathbf{R}^m$. The case when there are infinitely many solutions is included.

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

```
function x=nonhomogen(A,b)
format
[~,n]=size(A);
x=[];
```

First, you will determine whether the system is consistent and, if yes, whether there is a unique solution or there are infinitely many solutions. Use here the MATLAB function `rank()` and code the three possible cases listed below (you can either compose a conditional statement in an appropriate way or use the MATLAB command `return`).

If the equation Ax**=**b** is inconsistent, output a message `'The system is inconsistent'` and (the stored empty output for **x** will stay).

If the equation Ax**=**b** is consistent and the solution is unique, output a message `'The system has a unique solution'` and use MATLAB backslash operator, \ , to output and display the solution **x**.

If the equation Ax**=**b** is consistent and there are infinitely many solutions, output a message:

```
disp('There are infinitely many solutions')
```

and your code will continue with all the tasks outlined below in order to represent the general solution **x** of a non-homogeneous system as a sum of the general solution of the corresponding homogeneous system and particular solution **p** of the non-homogeneous system.

Referring to Exercise 3 of Project 1, we can represent the general solution of a homogeneous system A**x** = **0** as the Span of the columns of a matrix C, where C is the output matrix of the function `homobasis(A)`. If A**x** = **0** has non-trivial solutions, the columns of the matrix C form a basis for the solution set (or a basis for the Nul A). We will use the notation `Col(C)` for the Span of the columns of C (called Column Space of C). According to the theory, we can represent the general solution of a nonhomogeneous system A**x** = **b** (**b** ≠ 0) as `Col(C)` translated by the vector **p**, where **p** is particular solution of the nonhomogeneous system.
**Proceed with placing the following line into your code:

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

which will output the matrix C and the list of the free variables of the system A**x** = **0**.
**First, verify that you work with a valid output C. Place the following into your code:

```
if isempty(C)
      p=[]
      return
end
```

which will terminate the code for invalid C.

If C is valid, proceed with constructing particular solution of the non-homogeneous system. To justify the code fragment given below, you may need to solve a sample non-homogeneous system by hand and represent a solution in a parametric vector form. You may notice that the ordered entries of **p** whose indexes correspond to the non-pivot columns of A are zero, and the ordered entries of **p** whose indexes correspond to the pivot columns of A come from the last column of the matrix `B=rref([A b])` with deleted zero rows.
**Place the following lines into your code that output particular solution p:

```
[B,pivot]=rref([A b]);
p=zeros(n,1);
p(pivot,1)=B(1:rank(A),end);
```

Continue your code with a verification that **p is indeed a solution of A**x** = **b**: use a conditional statement and the function `closetozeroroundoff(A*p-b,7)`.
If the function above outputs the zero vector, display **p** with a message:

```
disp('particular solution of non-homogeneous system is vector')
p
```

Otherwise, output

```
disp('Check the code!')
p=[]
```

and terminate the code.
Receiving an empty output for **p** should prompt you to verify your code.

Complete **the function with these lines:

```
syms Col(C), syms p
fprintf(['the general solution of non-homogeneous system Ax=b is\n' ...
          'the Col(C) translated by the vector p:'])
x=Col(C)+p
```

The first line in the fragment above introduces symbolic variables, the second and third lines display a message, and the last line represents the output **x** using symbolic variables.

This is the end of your function `nonhomogen`.

Print the function **nonhomogen, homobasis, closetozeroroundoff in your Live Script.

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

```
%(a)
A=[1 2 -3], b=randi(10,1,1)
x=nonhomogen(A,b);
%(b)
A=magic(3), b=randi(10,3,1)
x=nonhomogen(A,b);
%(c)
A=magic(4), b=randi(10,4,1)
x=nonhomogen(A,b);
%(d)
A=magic(4), b=ones(4,1)
x=nonhomogen(A,b);
%(e)
B=[0 1 2 3;0 2 4 6]; A=[B; eye(4)], b=sum(A,2)
x=nonhomogen(A,b);
%(f)
A=[0 1 0 2 0 3; 0 2 0 4 0 6; 0 4 0 8 0 6], b=ones(3,1)
x=nonhomogen(A,b);
%(g)
A=[0 1 0 2 0 3; 0 2 0 4 0 6; 0 4 0 8 0 6], b=sum(A,2)
x=nonhomogen(A,b);
%(h)
A=[0 0 1 2 3;0 0 2 4 5], b=A(:,end)
x=nonhomogen(A,b);
%(i)
A=[0 0 1 2 3;0 0 2 4 6], b=A(:,end)
x=nonhomogen(A,b);
```

Here is an example how the output for part (i) could be printed in your Live Script if you choose an option “Output inline”:

```
%(i)
A=[0 0 1 2 3;0 0 2 4 6], b=A(:,end)
```

```
A = 2×5
     0     0     1     2     3
     0     0     2     4     6


b = 2×1
     3
     6
```

```
x=nonhomogen(A,b);
```

```
There are infinitely many solutions
the homogeneous system has non-trivial solutions
a free variable is x1
a free variable is x2
a free variable is x4
a free variable is x5
columns of C form a basis for solution set of the homogeneous system
```

```
C = 5×4
     1     0     0     0
     0     1     0     0
     0     0    -2    -3
     0     0     1     0
     0     0     0     1
particular solution of non-homogeneous system is vector
p = 5×1
     0
     0
     3
     0
     0
the general solution of non-homogeneous system Ax=b is
the Col(C) translated by the vector p:
x = p + Col(C)
```

Area, Volume, and Graphics in MATLAB

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

Theory: we will use here a representation of a vector as a position vector, that is, its initial point is at the origin and the coordinates of its terminal point are the entries of the vector.

The area of a parallelogram in $\mathbb{R}^2$ built on non-collinear vectors $\mathbf{v}_1, \mathbf{v}_2$ in $\mathbb{R}^2$ or the volume of a parallelepiped in $\mathbb{R}^3$ built on non-coplanar vectors $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ in $\mathbb{R}^3$ is $|\det A|$ (`abs(det(A))`), where $A = [\mathbf{v}_1 \ \ \mathbf{v}_2]$ or $A = [\mathbf{v}_1 \ \ \mathbf{v}_2 \ \ \mathbf{v}_3]$, respectively (see Section 3.3 of the textbook).

From Calculus we also know that the area of a parallelogram built on two vectors in $\mathbb{R}^3$ is the magnitude of their Cross Product and the volume of a parallelepiped built on three vectors in $\mathbb{R}^3$ is the absolute value of their Mixed Product (or Scalar Triple Product).

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

In this exercise, we will consider two vectors in $\mathbb{R}^2$ ($\mathbf{v}_1$ and $\mathbf{v}_2$) and three vectors in $\mathbb{R}^3$, ($\mathbf{v}_1$, $\mathbf{v}_2$, and $\mathbf{v}_3$) that will form the columns of a matrix A, and, if a parallelogram or a parallelepiped can be built on these vectors, we will output the area or the volume, respectively.

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

```
function D=areavol(A)
format
D=0;
```

It takes as the input a matrix `A`, whose columns are vectors in $\mathbb{R}^2$ or $\mathbb{R}^3$, and the output `D` is either the area or the volume. If the indicated geometrical figures cannot be built using the columns of A, the output `D` will be a zero.

First, your function has to check whether the given set of vectors is linearly independent. Use a conditional statement here and the MATLAB command `rank()`.

If the set is linearly dependent, program the following:
**If the vectors are in $\mathbb{R}^2$, a parallelogram cannot be built. In this case, output the corresponding message and the zero output for D, assigned previously, will stay.

**If the vectors are in $\mathbb{R}^3$, we will consider the following cases:
Case 1. If there is a non-collinear pair of vectors in the set (a linearly independent subset containing two vectors), then we can build a parallelogram on those vectors and find its area. If such a pair exists, first, display a message
```
disp('parallelepiped in R^3 cannot be built but parallelogram can')
```
then, identify a pair of linearly independent columns of A - take the pivot columns of A - and output the area of the parallelogram in $\mathbb{R}^3$ built on these vectors by using their Cross Product (see the **Theory** above). Assign your output to D and display it with a message:
```
fprintf('area of parallelogram built on pivot columns of A is %i\n',D)
```
Hint: to identify the pivot columns, use the command
```
[~,pivot]=rref(A);
```
Case 2. If all three vectors in $\mathbb{R}^3$ are collinear, output a message
```
disp('either parallelogram or parallelepiped in R^3 cannot be built')
```
and the zero output for D will stay.

If the set of vectors is linearly independent, code the following:
If the vectors are in $\mathbb{R}^2$, first, output the area D using the determinant (see **Theory above). Do not display D here.
Next, output the area by using Cross Product and assign that output to S – you will need to make adjustments to the input matrix A in order to use Cross Product (vectors must be in $\mathbb{R}^3$). Then, verify that the outputs D and S match (use function `closetozeroroundoff` with p=7). If your code confirms it, output the area as below:
```
fprintf('The area of the parallelogram is %i\n',D)
```
Otherwise, output the following:
```
disp('What can go wrong?!')
D=[]
```
**If the vectors are in $\mathbb{R}^3$, first, output the volume using the determinant and assign it to D. Next, output the volume using Mixed Product and assign that output to V, Then, verify that the output D and V match (use function `closetozeroroundoff` with p=7). If your code confirms it, output the volume as below:
```
fprintf('The volume of the parallelepiped is %i\n',D)
```
Otherwise, outputs will be:
```
disp('Something did go wrong!')
D=[]
```

Receiving empty outputs for D should prompt you to make corrections in your code.

Hint: suggested MATLAB functions to use in this exercise: `abs()`, `norm()`, `dot()`, `cross()`, `det()`, `rank()`, `rref()`.
This is the end of your function `areavol`.

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

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

```
%(a)
A=randi(10,2)
D=areavol(A);
%(b)
A=hilb(2)
D=areavol(A);
%(c)
A=fix(10*rand(3))
D=areavol(A);
%(d)
A=pascal(2)
D=areavol(A);
%(e)
A=magic(3)
D=areavol(A);
%(f)
A=hilb(3)
D=areavol(A);
%(g)
B=randi([-10,10],2,1); A = [B,3*B]
D=areavol(A);
%(h)
X=randi([-10,10],3,1);Y=randi([-10,10],3,1);A=[X,Y,X-Y]
D=areavol(A);
%(i)
A=[1 2 3;2 4 6;3 6 9]
D=areavol(A);
```

Exercise 5 (3 points) **Difficulty: Easy**

In this exercise, you will plot figures in the x_1x_2-plane and consider their images under the transformations of the plane such as reflections, shears, and rotations.

Part I (Shear & Reflections)

**First, type and display in the Live Script the standard matrices of the indicated transformations of the x_1x_2-plane (for help, please refer to the textbook, Section 1.9):

RX (reflection across the x_1-axis)
RY (reflection across the x_2-axis)
VS (vertical shear with $k = 3$)
HS (horizontal shear with $k = 3$)
RS (reflection across the line $x_2 = x_1$)
RA (reflection across the line $x_2 = -x_1$)

**Then, create the function 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 command line of the function `transf` and write comments in the Live Script.

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

Next, we will 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 (or copy and paste) in the Live Script the set of commands below and Run Section.

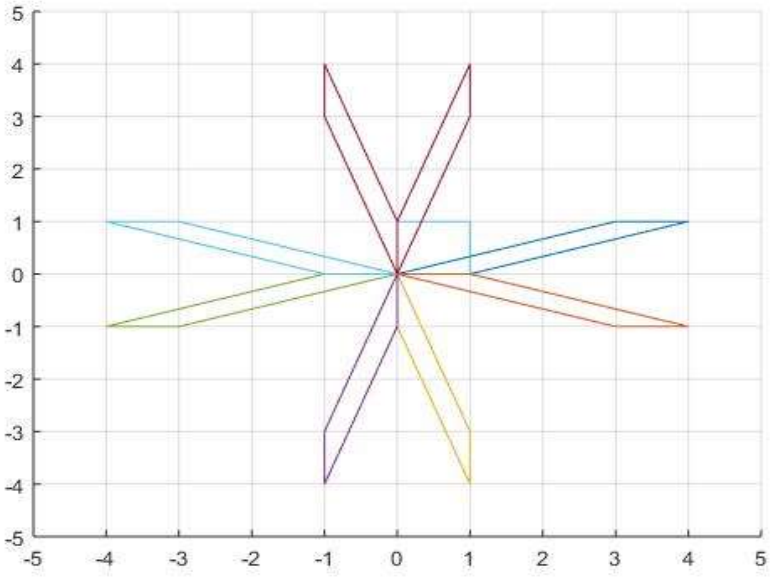
```
E=[0 1 1 0 0;0 0 1 1 0];
A=eye(2);
E=transf(A,E)
hold on
grid on
```

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, perform a sequence of the transformations: on each step, you will assign one of the matrices RX, RY, VS, HS, RS, or RA to the matrix A, and run the function

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

until your plot will become as one on the figure below:



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

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

and 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.

Complete **Part I by typing in the Live Script:

```
hold off
```

Note on the Outputs: After completing all necessary transformations, you will have in your Live Script displayed only the final plot (which is shown above) and all matrices E (including the intermediate ones). The colors on your plot do not need to match the ones on the figure.

Part II (Shear & Rotation)

Insert Section Break in your Live script and start working on Part II of the Exercise which is dealing with a horizontal shear and transformations of the plane defined by Givens rotations.

**Create a function in a file that begins with the lines:

```
function [] = rotation(Q,n)
hold on
grid on
theta=(2*pi)/n;
```

The input Q is a matrix that defines a parallelogram Q, which we will plot and, then, rotate through the angle `theta`. The input *n* is the number of consecutive Givens rotations of the parallelogram Q that we will use to arrive at the original plot of Q.

**Continue your function `rotation` with placing the function `givens()`, which was created in Exercise 2 of Project 1, with the specified variables `m,i,j`, such that, the rotation will be performed in $\mathbb{R}^2$ through the angle `theta`. Assign the output of this function to the matrix A (suppress this output).

**Next, place the "for" loop into your function:

```
for k=1:n
    Q=transf(A,Q);
end
```

This loop will generate *n* consecutive rotations of Q through the angle `theta=(2*pi)/n`.

**Finally, type in the function:

```
hold off
```

This is the end of your function `rotation`.

Print the functions `givens` and **`rotation` in your Live Script.

**Type (or copy and paste) the following in the Live Script:

```
HS=[1 2;0 1]
A=HS;
P=[0 1 1 0 0;0 0 1 1 0];
%(a)
Q=transf(A,P);
n=4;
rotation(Q,n)
%(b)
Q=transf(A,P);
n=16;
rotation(Q,n)
%(c)
Q=transf(A,P);
n=40;
rotation(Q,n)
hold off
```

Note: The matrix HS defines a horizontal shear. We assign HS to the matrix A. Running the command `Q=transf(A,P)` applies the horizontal shear to the unit square P and transforms P into the parallelogram Q. The command `rotation(Q,n)` performs *n* consecutive Givens rotations of Q.
**Run Section. Your outputs will be the Givens rotation matrices G and the three plots – each will hold *n* consecutive rotations of the parallelogram Q for the specified input variable *n*.

Cofactors, Determinants, and Inverse Matrices

In this part of the project, you will write three functions. The first function will output a matrix of cofactors of a given $n \times n$ matrix **a,** the second function will output the determinant of **a**, and the third function will output the inverse of an invertible matrix **a**.
Theory: The determinant of **a** can be computed by using a cofactor expansion across any row or down any column of **a**. For a reference, see Section 3.1 of the textbook.
The inverse of an $n \times n$ invertible matrix **a** can be computed by using the formula:

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

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

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

Part 1
In this part, you will create a function that generates a matrix of cofactors, C, which will be used in the two other functions below. The input for this function is an *n* x *n* matrix **a**.
** Create a function that begins with

```
function C=cofactor(a)
format
[~,n]=size(a);
```

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 the $n \times n$ matrix

$$C = [C(i,j)]\ (i,j = 1{:}n),$$

whose entries are cofactors of the matrix **a** calculated by 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 a message

```
disp('the matrix of cofactors of a is')
C
```

Part 2
In this part, you will create a function that outputs the determinant `D` of a matrix **a**.
**The function begins with

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

It accepts as inputs the matrix **a** and the matrix C which is the output of the function `cofactor(a)`. The output `D` is the determinant of the matrix **a**.

First, your function has to check if the determinant of **a is 0: use a MATLAB function `rank` (you cannot use here the function `det()`).
If the determinant is zero, output:

```
disp('the determinant of the matrix a is')
D=0
```

and terminate the program.

If the determinant is not zero, you will calculate the cofactor expansion across each row and down each column of the matrix **a** using the output matrix C from **Part 1**.
**Proceed as follows:
First, pre-allocate in your code an $n \times 2$ matrix `E` as

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

Then, recalculate the entries of the matrix E in the following way: the *n* entries of the first column of `E` are the cofactor expansions across the *n* **rows** of **a**, respectively, and the *n* entries of the second column of `E` are the cofactor expansions down the *n* **columns** of **a**, respectively. Output the matrix `E` (do not display it).

According to the general theory, each entry of the matrix `E` has to be equal to the determinant of the matrix **a**. Use this fact to verify in your code that the entries of the matrix E are calculated correctly.
**Proceed as follows:
First, output the determinant of the matrix **a** using the MATLAB built-in function `det()`.

```
d=det(a);
```

Then, check if each entry of the matrix `E` is equal to `d`. Use here a conditional statement and the function `closetozeroroundoff` with p=7.
If the code confirms that all entries of the matrix E are calculated correctly, output and display `D` as indicated below:

```
D=E(1,1);
disp('the determinant of the matrix a is')
D
```

Otherwise, if at least one of the entries of `E` does not match the reference value `d`, output a message `'Something went wrong!` and the empty output for `D` will stay.

Important: please be sure that you will not receive empty outputs when running the function `determine` on the variables presented in **Part 4** – the value of `D` will be used in the function which you will create next. Receiving the last message, should prompt you to make corrections in your code.

Part 3
Create a function that outputs matrix $B = \mathbf{a}^{-1}$ for an invertible matrix **a**. It accepts as inputs: matrix **a**, matrix `C` which is the output of the function `cofactor(a)`, and the determinant `D` of the matrix **a** which is the non-empty output of the function `determine(a,C)`.
**The function begins with:

```
function B=inverse(a,C,D)
B=[];
```

First, your function has to check if the matrix **a is invertible – use the output `D` of your function `determine` in this part – you cannot use MATLAB functions `rank()` or `det()` here.
If **a** is not invertible, output the corresponding message and terminate the program – the empty output for `B` will stay.
If **a** is invertible, code the corresponding message and output (do not display) the inverse of the matrix **a**, matrix B, by using of the formula

```
B=C'/D                                    (1)
```

In this formula, `D` is the non-empty and non-zero output of the function `determine(a,C)` (it is the determinant of an invertible matrix **a**) and `C'` is the transpose of the matrix `C` of cofactors of **a**, which is the output of the function `cofactor(a)`.

Next, check whether your matrix `B` matches the output of the MATLAB function `inv()`.
**Proceed as follows:
First, output (do not display) the matrix

```
F=inv(a);
```

Then, use a conditional statement and the function `closetozeroroundoff` with p=7 to check if `B` and `F` match. If it is the case, output a message

```
disp('the inverse is calculated correctly and it is the matrix')
B
```

If `B` and `F` do not match, output a message that could be something like `'What is wrong?!'`
Please be sure to make corrections in your code if you receive the last message.

Part 4
**Print functions `closetozeroroundoff`, `cofactor`, `determine`, `inverse` in the Live Script.

**Run the functions on the choices (a)-(f) as indicated below:

```
%(a)
a=diag([1,2,3,4,5])
C=cofactor(a);
D=determine(a,C);
B=inverse(a,C,D);
%(b)
a=ones(4)
C=cofactor(a);
D=determine(a,C);
B=inverse(a,C,D);
%(c)
a=magic(3)
C=cofactor(a);
D=determine(a,C);
B=inverse(a,C,D);
%(d)
a=magic(4)
C=cofactor(a);
D=determine(a,C);
B=inverse(a,C,D);
%(e)
a=hilb(4)
C=cofactor(a);
```

```
D=determine(a,C);
B=inverse(a,C,D);
%(f)
a=rand(2)
C=cofactor(a);
D=determine(a,C);
B=inverse(a,C,D);
```

Part 5 (BONUS: 1 point!)
In this part, we will work in the Live Script with the input **a** and output C **i**n choice (f) above. We will demonstrate that the method of calculating the inverse using cofactors for an n x n ($n > 2$) invertible matrix is an extension of the method of calculating the inverse of a 2 x 2 invertible matrix presented in the lecture for Module 12.
According to the Theorem in that lecture, if we denote

$$\mathrm{a} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \ \mathrm{S} = \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \qquad (2)$$

then

$$\mathrm{a}^{-1} = \mathtt{S/det(a)} \qquad (3)$$

Use as the input the matrix **a from choice (f) in Part 4 and write a set of commands in the Live Script that will output the matrix S according to the formula in (2). Do not display S.

**Next, compare the formula for the inverse matrix (3) with the formula (1) in Part 3 and run a conditional statement in the Live Scrip that would verify that your output matrix S is actually the matrix `C'`, where C is the output matrix of the function `cofactor` in choice (f).
If it is the case, code a message that the method of calculating the inverse of a 2 x 2 matrix is agreeable with the method of calculating the inverse matrix using cofactors.
Please make sure that you will receive this message after running your commands.

Application: The Leontief "Input-Output" Model

In this part of the Project, we will work with the "input-output" model of an economy with the open sector.

Some Notes on the Theory: Suppose the economy is divided into n productive sectors. Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ be a **production vector** that lists the output of each sector for one year. Suppose the other part of the economy (called open sector) does not produce goods or services but only consumes them. Let the vector **d** in $\mathbf{R}^n$ be the **final demand vector** that lists the values of the goods and services demanded from the n productive sectors by the non-productive part of the economy.
The Leontief "Input-Output" model assumes that, for each jth productive sector ($j = 1{:}n$), there is a **unit consumption vector** $\boldsymbol{c}_j$ that lists the inputs (demands, orders) from the productive sectors needed per unit of its output.

As the sectors produce goods to meet consumer demand, the producers create additional **intermediate demand** for goods they need for their own productions. For example, if the jth sector decides to produce x_j units, then its intermediate demand of manufacturing is $x_j \boldsymbol{c}_j$,

where $\boldsymbol{c}_j$ is the unit consumption vector. Then, the total intermediate demand from all productive sectors is given by

$$x_1\boldsymbol{c}_1 + x_2\boldsymbol{c}_2 + \ldots + x_n\boldsymbol{c}_n = C\mathbf{x}.$$

Matrix $C = [\boldsymbol{c}_1\ \boldsymbol{c}_2\ \ldots\ \boldsymbol{c}_n]$ is called the **consumption matrix**.
Note: All of the input and output units are measured in millions of dollars, not in quantities such as tons or bushels. And the prices for goods and services are held constant.

Counting the final demand **d** from the open sector, the total demand for that production will be $C\mathbf{x} + \mathbf{d}$. Leontief asked if there is a production level **x**, such that the amounts produced (or "supplied") will exactly balance the total demand for that production. It gave a rise to the Leontief's model:

$$\mathbf{x} = C\mathbf{x} + \mathbf{d} \qquad (1)$$

The equation (1) can be written as

$$(I - C)\mathbf{x} = \mathbf{d} \qquad (2)$$

where *I* is the *n* x *n* identity matrix.

The production vector **x** has to be economically feasible, that is, its entries must be non-negative numbers. Theorem 11 in Section 2.6 of the textbook states the conditions on the consumption matrix C and the final demand vector **d** under which the matrix $(I - C)$ is invertible and the production vector **x**, which will be the unique solution of the equation (2), has non-negative entries.

Exercise 7 (4 points) **Difficulty: Moderate**
To complete this Exercise, you will need to read Section 2.6, Pages 134-137 of the textbook and review the settings and formulas in Exercises 13-15 on Page 139 of the textbook.
**Create a function in MATLAB that begins with

```
function x = production(C,d)
format
n=size(C,2);
x=[];
```

It takes as inputs an *n* x *n* matrix C and a vector **d** with *n* entries. The output, for valid inputs, will be the non-empty production vector **x**.

First your function has to check whether C and **d are valid inputs for which it is guaranteed the existence of unique economically feasible output **x** (refer to Theorem 11 in Section 2.6). If either C or **d** or both are not valid inputs, output the corresponding message, or messages, that have to specify a reason for each input to be invalid. If at least one of the inputs is not valid, the program terminates and the empty output for **x** will stay.

If both inputs, C and **d**, are valid, your function will continue:
First, output the production vector **x using the equation (2) (see above). You can employ the MATLAB function `inv()` for this part. Do not display **x** here.
Then, verify that **x is an economically feasible output, that is, its entries are non-negative numbers. If your code does not confirm that **x** is economically feasible, output:

```
disp('check the code!')
x=[];
```

and terminate the program.

If your code confirms that **x** is a valid output, display the vector **x** with a message:
`disp('the unique production vector is')`
`x`
and your function will continue with one more task:
Instead of the equation (2), you will be using now **the recurrence formula in Exercise 15 on Page 139** to output the production vector with a desired accuracy. For a reference value, you will take the output **x**. Your code will also output (and display) the number of iterations, `k`, needed to match the reference value.
(Prior to working on this part, you may find it useful to read Pages 136-137 of the textbook.)

**Proceed with this task in the following way:
Assign initially:
```
x0=d;
k=1;
```
(`x0` is your first iteration), and set up a "while" loop for the consecutive iterations using the recurrence formula given in Exercise 15. To make the code simple, assign each consecutive iteration to `x0` again and your loop will run while the condition
`any(closetozeroroundoff(x0-x,1))`
holds, that is, while there is at least one mismatch in the entries of the vectors up to the first decimal place. Count the number of iterations `k`.
Assign the last iteration to the vector `x1` and output it with the message:
`disp('the production vector calculated by recurrence relation is')`
`x1`
Display the number of iterations, `k`, needed to archive this accuracy:
`fprintf('the number of iteration to match the output x is %i\n',k)`
This is the end of the function `production`.

Print the function **`production` in your Live Script:

First, run the function on the matrix C and vector **d** from Examples 1 and 2 in Section 2.6. You can verify some of your outputs by comparing them with the ones in the textbook.
**Type the following in the Live Script and Run Section:
```
%(a)
C = [.5 .4 .2; .2 .3 .1; .1 .1 .3]
d = [50; 30; 20]
x = production(C,d);
```

Next, we will work with the inputs listed in Exercises 13 and 14 on Page 139 and with some additional ones. The inputs for this part will be imported into your Live Script from the csv files, which are located under the Resources, **Exercise7 Files (csv)**, on the Canvas page Assignments, Project 2. You will need to save these files in your Current Folder in MATLAB. We will use MATLAB command `importdata` to store the data as matrices in the Live Script.
**After you saved the data files in MATLAB, run the following code fragment in the Live Script and answer the questions with the reference to Exercise 13 on Page 139:
```
%(b)
C = importdata('consumption.csv')
d = importdata('demand.csv')
x = production(C,d);
```

% What is the meaning of the (4, 3)-entry of the matrix C?
% How many units does Sector (6) need to produce in order to satisfy the total demand for its production?

**Next, run the function on the choices (c)-(f) as indicated below:

```
%(c)
C = [.5 .4 .2; .2 .3 .1; -.1 .1 .3]
d = [50; 30; 20]
x = production(C,d);
%(d)
C = importdata('consumption.csv')
d = importdata('demand_1.csv')
x = production(C,d);
%(e)
C = importdata('consumption_1.csv')
d = importdata('demand_1.csv')
x = production(C,d);
%(f)
C = importdata('consumption_1.csv')
d = importdata('demand_2.csv')
x = production(C,d);
```

THIS IS THE END OF PROJECT 2