MATLAB PROJECT 1

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

BEGIN with creating a Live Script **Project1.**
Note: All exercises in this project will be completed in the Live Script using the Live Editor. Please refer to the MATLAB video that explains how to use the Live Script:
https://www.mathworks.com/videos/using-the-live-editor-117940.html?s_tid=srchtitle
The final script has to be generated by exporting the Live Script to PDF.
Each exercise has to begin with the line:
Exercise#
You should also mark down the parts such as (a), (b), (c), and etc. This makes grading easier.
Important: we use the default `format short` **for the numbers in all exercises unless it is specified otherwise**. We do not employ `format rat` since it may cause problems with running the codes and displaying matrices in Live Script. If `format long` has been used, please make sure to return to the default `format` in the next exercise.

EXISTENCE AND UNIQUENESS THEOREM

EXERCISE 1 (4 points) **Difficulty: Moderate**
In this exercise, we work with a system of equations A**x**=**b**, where A is an *m* x *n* matrix. You will answer the questions whether a solution exists and, if yes, whether it is unique or whether there are infinitely many solutions. For a consistent system, you will output a solution.
You will use the **Existence and Uniqueness Theorem** from Lecture 2 and you will not be allowed to use in your code its equivalent form, **Rouche-Capelli** Theorem, which employs the **rank** of a matrix. That is why your function will be called `usenorank`.
Theory: The Existence and Uniqueness Theorem states that a system A**x**=**b** is consistent if and only if the last column of the augmented matrix [A **b**] is not a pivot column (**condition (1)**), or equivalently, if there is no row in an echelon form of the augmented matrix whose all entries are zero except for the last entry which is a non-zero number (**condition (2)**). The Theorem also states that a consistent system A**x**=**b** has a unique solution if and only if there are no free variables in the system, that is, all columns of A are pivot columns (**condition (3)**, and we can consider an equivalent condition for the uniqueness of the solution that `m>=n` and the square block matrix formed by the first *n* rows and the first *n* columns of the reduced echelon form of [A **b**] is the *n* x *n* identity matrix (**condition (4)**) – the last condition is an implication of the fact that A has a pivot position in every column.
**Create a function in a file that begins with:

```
function [R,x]=usenorank(A,b)
format
x=[];
[m,n]=size(A);
fprintf('A is %i by %i matrix\n',m,n)
[R,pivot]=rref([A b]);
```

The inputs are an *m* x *n* matrix A and an *m* x 1 vector **b**. The output **x** is a solution of A**x**=**b** if the system is consistent, and, if there is no solution, **x** will stay empty.

Note: The MATLAB command `[R,pivot]=rref([A b]);` outputs an `m` x (`n+1`) matrix `R`, which is the reduced echelon form of the matrix `[A b]`, and a row vector `pivot`, which lists the indexes of the pivot columns of `[A b]`.
**Display these outputs with the messages as below:

```
disp('the reduced echelon form of [A b] is')
disp(R)
disp('the vector of indexes of the pivot columns of [A b] is')
disp(pivot)
```

**Continue your function with the following command:

```
N=numel(pivot);
```

This command outputs the number of elements in the vector `pivot`, thus, `N` is the number of the pivot columns of the matrix [A **b**], or equivalently, the number of the pivot positions in [A **b**]. Also notice that `N` is the number of the non-zero rows of the matrix R.

Continue your code with testing in two ways if the system A**x**=**b** is consistent.
**We will employ here conditional statements and the variables `test1` and `test2`. Proceed as follows. Initialize:

```
test1=1;
test2=1;
```

If the **condition (1)** in the **Theory** above does not holds, assign:

```
test1=0;
```

If the **condition (2)** in the **Theory** does not hold, assign:

```
test2=0;
```

Hints: To check if the **condition (1)** does not holds, you can use the output `pivot` and a MATLAB command `ismember()`.
To check whether the **condition (2)** does not hold, you can set up a "for" loop that will iterate through the rows of the matrix R, starting from the top, and check if there is a row whose first *n* entries are 0's and the last entry (`n+1`) (or `end`) is a non-zero number - you can employ a logical operator `any()` here. If such a row is found, you will assign `test2=0;` and terminate "for" loop - a MATLAB command `break` can be used to terminate a loop.
**Outputs and display the variables as below:

```
test1
test2
```

After completing the tasks above, your code will check if the two methods agree, and, if yes, it will display a message whether the system is consistent or not.
**Proceed with placing the following conditional statement in your code:

```
if isequal(test1,test2,1)
    disp('the system is consistent')
elseif isequal(test1,test2,0)
    disp('the system is inconsistent')
    return
else
    disp('test1 and test2 disagree - something is not quite right!')
    return
end
```

Notice that, according to the statement above, your code terminates when either the system is inconsistent or the two methods disagree. If the system is inconsistent, the empty output for **x** will stay. If you receive a message that tests disagree, please be sure to make corrections in your code.

Your code will continue (for a consistent system only).
Proceed with testing in two ways if the system A**x**=**b** has a unique solution or if there are infinitely many solutions.
**We will use here conditional statements and the variables `test3` and `test4`. Proceed as follows. Initialize

```
test3=0;
test4=0;
```

If the **condition (3)** in the **Theory** above holds, assign:

```
test3=1;
```

If the **condition (4)** in the **Theory** holds, assign:

```
test4=1;
```

Hints: you can check if the **condition (3)** holds by using the output `N`. To check if the **condition (4)** holds, you can use logical comparison statements and the matrix `eye(n)`.
**Output and display the variables as below:

```
test3
test4
```

After completing the tasks above, check if the two methods agree, and, if yes, display a message whether the system has a unique solution or it has infinitely many solutions.
**Place the following conditional statement in your code:

```
if isequal(test3,test4,1)
    disp('the solution is unique')
elseif isequal(test3,test4,0)
    disp('there are infinitely many solutions')
else
    disp('test3 and test4 disagree - something is definitely wrong!')
    return
end
```

Notice that according to this statement, your code terminates only when the two methods disagree. Receiving the message that the tests disagree should prompt you to verify your code.

Continue your code with outputting just one solution (or the only one solution) of A**x**=**b**.
We will work here with the matrix R, which is the reduced echelon form of [A **b**], with the zero rows eliminated.
**First, output the matrix formed by the non-zero rows of R (the top N rows) and assign that output to R again.
**Next, use the variable `test3` in a conditional statement and the new matrix R to output a solution x (do not display x here) for the two cases:
(1) If a solution is unique, output x as the last column of R.
(2) If there are infinitely many solutions, we will output a solution x in the following way: assign to the basic variables, which correspond to the pivot columns of A, the ordered entries of the last column of R, and assign 0's to the free variables. This approach can be justified by using the parametric-vector form of the general solution of A**x**=**b** where the parameters (free variables) are set equal to 0. Proceed with coding of this part as follows:

**Pre-allocate:
`x=zeros(n,1);`
and, then, assign:
`x(pivot,1)=R(:,end);`

Next, we will verify that a solution **x** is found correctly, that is, x satisfies the equation A**x**=**b**.
**Proceed as follows:
Write a conditional statement with the function `closetozeroroundoff(A*x-b,7)`. If the output of the function `closetozeroroundoff` is a zero vector, display a solution x with a message as below:

```
disp('a solution of the system Ax=b is')
x
```

Otherwise, output (and display) the following (and please be sure to check your code):

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

Note: the function `closetozeroroundoff` was created in Project 0.

This is the end of the function `usenorank`.
Save it in your current folder in MATLAB with the name **`usenorank`** and the extension .m

**Print the functions `usenorank` and `closetozeroroundoff` in the Live script by commands:

```
type usenorank
type closetozeroroundoff
```

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

```
%(a)
A=magic(5)
b=rand(5,1)
[R,x]=usenorank(A,b);
%(b)
A=magic(5)
b=zeros(5,1)
[R,x]=usenorank(A,b);
%(c)
A=magic(4)
b=rand(4,1)
[R,x]=usenorank(A,b);
%(d)
A=magic(4)
b=ones(4,1)
[R,x]=usenorank(A,b);
%(e)
A=[1 2 3 4;-1 -2 -3 -4;2 4 -3 -1;1 2 -1 5;3 6 1 2]
b=sum(A,2)
[R,x]=usenorank(A,b);
%(f)
A=[magic(5),randi(10,5,2)]
b=rand(5,1)
[R,x]=usenorank(A,b);
%(g)
```

```
A=[magic(5);zeros(2,5)]
b=[rand(5,1);zeros(2,1)]
[R,x]=usenorank(A,b);
%(h)
A=[magic(5);randi(10,2,5)]
b=rand(7,1)
[R,x]=usenorank(A,b);
%(i)
A=randi([-5 5],5,3)
b=sum(A,2)
[R,x]=usenorank(A,b);
%(j)
A=ones(5)
b=sum(A,2)
[R,x]=usenorank(A,b);
```

%Write a comment that would explain why the number N, which was defined in the Instructions as

`N=numel(pivot);`

is also the number of non-zero rows of the matrix R that was output by the command `[R,pivot]=rref([A b])`.

BONUS! (1 point)

%State and justify the exact relation between the number N and the rank of the matrix A for each of the two cases: when the system A**x**=**b** is consistent and when A**x**=**b** is inconsistent (when writing this comment, you can refer to the Rouche-Capelli Theorem in Lecture 2).

EXERCISE 2 (5 points) **Difficulty: Hard**

Part I

Theory: A transformation in $\mathbb{R}^2$ defined by multiplication of a vector **x** in $\mathbb{R}^2$ by the matrix

$$G = \begin{bmatrix} cos\,\theta & -sin\,\theta \\ sin\,\theta & cos\,\theta \end{bmatrix} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \qquad (1)$$

rotates **x** in x_1x_2-plane through the angle θ radians. The same type of a "plane" rotation in x_ix_j-plane can be performed in $\mathbb{R}^m$ $(m \geq 2, \quad 1 \leq i < j \leq m)$, and it is called a Givens rotation. Givens rotations can be used to zero out individual isolated entries of a matrix without changing the norms of the columns. We will employ Givens rotations in this Project to transform an *m* x *n* matrix A into an upper triangular matrix R preserving the 2-norm (magnitude) of each column of A. In a later Project, we will construct QR factorization of a matrix by means of consecutive Givens rotations.

Description: A Givens rotation in $\mathbb{R}^m$ is represented by a matrix G that can be defined by the following parameters: m $(m \geq 2)$, $theta = \theta$ (the angle of rotation), and i, j $(1 \leq i < j \leq m)$. We will create a matrix G starting with the $m \times m$ identity matrix, `G=eye(m)`, then, outputting

```
c=cos(theta);
s=sin(theta);
```

and, finally, assigning the entries of the matrix on the right-hand side of (1) to the entries of `G` that are located at the intersections of rows i, j with columns i, j $(i < j)$ as below:

```
G(i,i)=c; G(i,j)=-s; G(j,i)=s; G(j,j)=c;
```
(2)

Geometrical Meaning: Matrix-vector multiplication of G and a column vector **x** in $\mathbb{R}^m$ performs rotation of the vector **x** in the $x_i x_j$-plane through θ radians. This rotation can only modify rows i,j in **x**. Thus, left-multiplication of an m x n matrix A=$[a_1 \ \ a_2 \ \dots \ \ a_n]$ by G, which is defined as

$$GA = G[a_1 \ \ a_2 \ \dots \ \ a_n] = [Ga_1 \ \ Ga_2 \ \dots \ \ Ga_n] ,$$

which rotates the columns of A through θ radians, can only modify the rows i and j in A.

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

```
function G=givens(m,i,j,theta)
format
G=[];
```

The function will produce an $m \times m$ Givens matrix G under the conditions:

$$1 \le i < j \le m \quad \text{and} \quad m \ge 2 \qquad (3)$$

**Continue your function with a conditional statement that will check if all conditions in (3) hold. If yes, output the matrix G following the Description above and display it with a message:

```
disp('the Givens rotation matrix G is')
disp(G)
```

If at least one of the conditions in (3) does not hold, display a message that a Givens rotation matrix cannot be constructed. The empty output for G assigned previously will stay.

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

```
type givens
```

**Run the function on the choices (a) - (e) as indicated below:
(In MATLAB, we type pi for π.)

```
%(a)
m=1;i=1;j=2;theta=pi
G=givens(m,i,j,theta);
%(b)
m=4;i=3;j=2;theta=pi/2
G=givens(m,i,j,theta);
%(c)
m=5;i=2;j=4;theta=pi/4
G=givens(m,i,j,theta);
%(d)
m=2;i=1;j=2;theta=-pi/2
G=givens(m,i,j,theta);
%(e)
m=3;i=1;j=2;theta=pi
G=givens(m,i,j,theta);
```

The rest of Part I has to be done in the Live Script using the output G from choice (e):
**Based on the Geometrical Meaning of the product of G and a vector, predict what would be the matrix (output it as GA) whose columns are the images of the vectors a_1, a_2, a_3, respectively, under the Givens rotation defined in choice (e), where

$$\boldsymbol{a}_1 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad \boldsymbol{a}_2 = \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}, \quad \boldsymbol{a}_3 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}.$$

**Type and display the matrix A= $[a_1 \ \ a_2 \ \ a_3]$ and your matrix GA in the Live Script.
**Then, output the actual product of G and the matrix A and display it as

```
GpA=G*A
```

**Run a conditional statement that will verify that your predicted matrix GA and the observed matrix `GpA` match within the margin 10^(-7). If the matrices match, output a message that the predicted matrix matches the observed one – please make sure you will receive this message.
Note: use the function `closetozeroroundoff` with p = 7 when comparing two matrices.

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

```
type closetozeroroundoff
```

Part II

According to the Geometrical Meaning, left-multiplication of an *m* x *n* matrix A by the matrix `G=givens(m,i,j,theta)` can only modify the rows *i* and *j* of the matrix A. Applying consecutive Givens rotations to an *m* x *n* matrix, we can transform it into an upper triangular matrix by generating zeros below the main diagonal and preserving the norm (magnitude) of each column vector (the norm of a vector does not change under a rotation transformation).
Theory: to map a vector $(r, 0)$ in $\mathbb{R}^2$ $(r > 0)$ onto a vector (a, b), where $r = \sqrt{a^2 + b^2}$, we would rotate it through the angle θ, where $\cos(\theta) = \frac{a}{r}$ and $\sin(\theta) = \frac{b}{r}$. We can construct a Givens rotation matrix adjusted to a vector (a, b) without using the angle θ but instead working directly with its cosine and sine.
**Create a function in a file that begins with:

```
function G=givensrot(m,i,j,a,b)
G=eye(m);
r=hypot(a,b);
c=a/r;
s=b/r;
```

Note: compared with the function `givens`, we do not input `theta` but, instead, we calculate c and s using the inputs `a` and `b`. We also employ the function `hypot` to output r (in order to avoid a possible overflow when calculating the square root directly).

**Complete your function `givensrot` with the assignments given in (2):

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

Do not display the output matrix G.

To understand a purpose of the next function, we will take a look at the example. In order to transform the vector $\begin{bmatrix} a \\ b \end{bmatrix}$ back to the vector $\begin{bmatrix} r \\ 0 \end{bmatrix}$ $(r > 0)$ (see **Theory**), we rotate it through the angle $-\theta$. Substituting $-\theta$ for θ into the matrix `G` in (1), we will obtain `G'` (transpose of `G`). Indeed, since

$$G = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}$$

then,

$$\begin{bmatrix} \cos(-\theta) & -\sin(-\theta) \\ \sin(-\theta) & \cos(-\theta) \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = G'$$

Thus, `G'*` $\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$. This transformation converts the bottom entry of the vector $\begin{bmatrix} a \\ b \end{bmatrix}$ to a 0 preserving the magnitude of the vector (it stays equal to r).

Algorithm: To transform a matrix A into an upper triangular matrix using Givens rotations, we will iterate through the columns of A and zero out the entries below the main diagonal as follows:
For a consecutive column j of the matrix A, we are choosing pairs (***a***, ***b***), such that ***b*** is iterating through the rows, starting with the last row and going up, until it reaches the entry right below the diagonal entry, – the entries that we are assigning to **b** will be zeroed out by performing Givens rotations defined by the matrix `G'`. And, to the variable ***a*** , we are assigning the entry on the main diagonal.
Notice also that, when running this algorithm on a column j (j > 1), the columns to the left already have zero entries below the main diagonal, and Givens rotations constructed using the entries of the column j will not modify the columns to the left of the jth column.

**Create another function in a file that will transform an *m* x *n* matrix A into an upper triangular matrix R by performing rotations of its columns by using the matrices `G'`, where `G` will be generated by the function `givensrot`.

The new function begins with the lines:
```
function R = uppertrian(A)
format
[m,n]=size(A);
R=A;
k=min(m,n);
```

Continue your function with the nested loops, where the outer loop is a "for" loop and the inner - a "while" loop.
**The outer "for" loop has to iterate through the columns of R from column 1 to column k.
The inner "while" loop will iterate through the rows of a consecutive column starting with the last row (row *m*) and going up the column "while" the index of the row stays greater than the index of the column (see **Algorithm above).
Within the inner loop we, first, check if the entry in the consecutive row is not a 0 and, if so, assign that entry to ***b***, and we assign to ***a*** the entry located on the main diagonal. This pair of variables ***a*** and ***b*** will be used in the function `givensrot`, and the other input variables i and j in the function `givensrot` have to be adjusted to your loop accordingly (they will be the ordered indexes of the rows of R that are affected by the rotation).
Place the function `givensrot` into your inner loop to generate the matrix `G` (suppress this output) followed by the command below that will perform the rotation:
```
R=G'*R;
```
**After the nested loop is completed, it outputs the matrix R. We will run the function `closetozeroroundoff` with p=7 to ensure that the small by magnitude numbers will show as zeros and display R with a message as below:
```
R=closetozeroroundoff(R,7);
disp('the output matrix R is')
disp(R)
```

Continue your function with a check whether the matrix R is constructed correctly.
**Assign initially
```
test1=1;
test2=1;
```

**First, we will verify if R is, indeed, an upper triangular matrix. Use a conditional statement with the condition `~istriu(R)`. If the condition is "True", assign:

```
test1=0;
```

**Then, check if the transformation preserves the norms of the column vectors in A. Set up a "for" loop that iterates through all columns of the matrices A and R (`i=1:n`) and, for each `i` it checks if the following condition holds:

```
closetozeroroundoff(norm(A(:,i))-norm(R(:,i)),7)~=0
```

If your code comes across a column for which the above condition is "True", assign

```
test2=0;
```

and "break" the loop. (You can use the command `break` to terminate a "for" loop).

**Next, place the following lines into your code:

```
if test1 & test2
    disp('A has been reduced correctly to an uppertriangular matrix R')
else
    disp('the output matrix R is not what was expected?!')
end
```

This is the end of the function `uppertrian`.

**Print the functions `givensrot` and `uppertrian` in your Live Script.

**Run the function `uppertrian` on the choices (a)-(j) as indicated below:

```
%(a)
A=ones(2)
R = uppertrian(A);
%(b)
A=magic(3)
R = uppertrian(A);
%(c)
A=magic(4)
R = uppertrian(A);
%(d)
A=[magic(3),ones(3,2)]
R = uppertrian(A);
%(e)
A=[magic(3);ones(2,3)]
R = uppertrian(A);
%(f)
A=triu(magic(5))
R = uppertrian(A);
%(g)
A=tril(magic(3))
R = uppertrian(A);
%(h)
A=[1 1 2 0;0 0 1 3;0 0 2 4;0 0 3 5;1 0 -2 3]
R = uppertrian(A);
%(i)
A=hilb(4)
R = uppertrian(A);
%(j)
A=[1 3 4 -1 2;2 6 6 0 -3;1 3 1 2 -1;1 3 0 3 0]
R = uppertrian(A);
```

SOLVING HOMOGENEOUS SYSTEMS

EXERCISE 3 (4 points) **Difficulty: Moderate**

Theory: In this exercise, you will work with the solution set of a homogeneous linear system $A\mathbf{x} = \mathbf{0}$, where A is an $m \times n$ matrix and $\mathbf{x} \in \mathbb{R}^n$. If the system has only the trivial solution (there are no free variables), a spanning set for the solution set is the zero vector of $\mathbb{R}^n$. If the system has non-trivial solutions (there is at least one free variable), we can obtain a spanning set for the solution set by solving the system using row reduction algorithm – a spanning set that we obtain is also linearly independent and, therefore, forms a basis for the solution set. Later on, we will call the solution set of a homogeneous system $A\mathbf{x} = \mathbf{0}$ the Null space of the matrix A. Thus, in this exercise, we are creating a spanning set for the solution set of $A\mathbf{x} = \mathbf{0}$ which will be a basis for the Null space of A when the Null space contains non-zero vectors. To justify the algorithm presented below, you may find it useful to solve some sample homogeneous systems by hand: work with the reduced echelon forms of the matrices in choices (a)-(j) and represent the solution sets which contain non-zero vectors in a parametric vector form (as a span of a set of vectors) using the free variables as parameters.

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

```
function C = homobasis(A)
format
[~,n]=size(A);
R=rref(A);
rankA=rank(A);
```

Continue with composing a conditional statement that will go through the following two cases:
**Check if the system $A\mathbf{x} = \mathbf{0}$ has only the trivial solution, $\mathbf{x} = \mathbf{0}$. It happens if and only if there are no free variables in the system or, equivalently, all columns of A are pivot columns. If this is the case, output:

```
disp('the homogeneous system has only the trivial solution')
C=zeros(n,1);
```

and your code will terminate here.
Hint: use here a built-in MATLAB function `rank()`.
**If the system has nontrivial solutions, output a message:

```
disp('the homogeneous system has non-trivial solutions')
```

and your code will continue with constructing a basis for the solution set, that is, constructing a basis the Null space of A.

In order to proceed with this task, we need to output the ordered sets of indexes of the pivot and non-pivot columns of A.
**Include into your code all of the commands listed below:
A command that outputs the ordered set of indexes of the pivot columns:

```
[~,pivot]=rref(A);
```

A command that outputs the ordered set of indexes of the non-pivot columns:

```
nonpivot=setdiff(1:n,pivot);
```

The set of commands that output messages listing the free variables:

```
q=numel(nonpivot);
j=1:q;
fprintf('a free variable is x%i\n',nonpivot(j))
```

After we have identified the free variables and non-pivot columns, we begin constructing a basis for the solution set which will be formed by the columns of an $n \times q$ matrix C.
**Pre-allocate matrix C as
`C=zeros(n,q);`
Then, place the following set of commands into your code:
```
R=R(1:rankA,nonpivot);
C(pivot,:)=-R;
C(nonpivot,:)=eye(q);
```
Here, we made assignments to the rows of C using matrices `R=rref(A)` and `eye(q)`: different assignments were made to the rows of C whose indexes are from the set `pivot` and to the rows of C whose indexes are from the set `nonpivot` (analyze your hand calculations!).

Finally, we will check if the columns of C form a basis for the solution set of the homogeneous system. Since the number of columns of C is equal to the number of the free variables in the system $A\mathbf{x} = \mathbf{0}$, we only need to verify that the following conditions hold:
(1) the set of the columns of C is linearly independent (use the command `rank)`;
(2) the columns of C are solutions of A**x** = **0,** or, equivalently, a matrix `A*C` is the zero matrix. (use the function `closetozeroroundoff` with p=5).
**Proceed with constructing a conditional statement in the way that, if both conditions (1) and (2) hold, display a message
`disp('columns of C form a basis for solution set of homogeneous system')`
otherwise, assign:
`C=[];`
Note: If you receive an empty output for C after running the function on the choices below, please be sure to make corrections in your code.

This is the end of the function `homobasis`.

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

```
%(a)
A=[1 -1 -1 2;-2 5 4 4]
C=homobasis(A)
%(b)
A=[1 2 -3]
C=homobasis(A)
%(c)
A=magic(3)
C=homobasis(A)
%(d)
A=[magic(3), ones(3,1)]
C=homobasis(A)
%(e)
A=magic(4)
C=homobasis(A)
%(f)
A=[0 1 2 3;0 2 4 6]
C=homobasis(A)
%(g)
```

```
A=[0 1 0 2 0 3; 0 2 0 4 0 6; 0 4 0 8 0 6]
C=homobasis(A)
%(h)
A=[1 0 2 0 3;2 0 5 0 6]
C=homobasis(A)
%(i)
A=[1 0 0 2 3;2 0 0 4 6]
C=homobasis(A)
%(j)
A=hilb(4)
C=homobasis(A)
```

MORE ON MATRICES; APPLICATION

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

In this exercise, you will create a function which produces, when possible, a polynomial that interpolates the given data points.

Theory: We will work with the polynomials of degrees at most $(n-1)$ written in the form

$$P(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x + a_0, \qquad (1)$$

where $x \in \mathbb{R}$, $n \geq 1$ is a natural number, and the coefficients $a_{n-1},\ a_{n-2},\ \ldots\ ,\ a_0$ are real numbers.

Suppose that we have *m* data points

$$(x_1, y_1),\ (x_2, y_2),\ \ldots\ ,\ (x_m, y_m) \qquad (2)$$

where $m \geq 1$, and, for a given *n*, we are looking for a polynomial of the form (1) interpolating the data points (2), that is, we are looking for a polynomial that satisfies the conditions

$$P(x_i) = y_i \quad (i = 1{:}m) \qquad (3)$$

if such a polynomial exists.

The set of *m* conditions in (3) gives a rise to a linear system of *m* equations (according to the number of data points) in *n* variables $(a_{n-1}, a_{n-2}, \ldots, a_0)$:

$$\begin{cases} x_1^{n-1}a_{n-1} + x_1^{n-2}a_{n-2} + \cdots + x_1a_1 + x_1^0a_0 = y_1 \\ x_2^{n-1}a_{n-1} + x_2^{n-2}a_{n-2} + \cdots + x_2a_1 + x_2^0a_0 = y_2 \\ \vdots \\ x_m^{n-1}a_{n-1} + x_m^{n-2}a_{n-2} + \cdots + x_ma_1 + x_m^0a_0 = y_m \end{cases} \qquad (4)$$

Notice that $x_1, x_2, \ldots, x_m$ are numbers and their powers in (4) are numerical coefficients of the system of linear equations.

If we denote X and Y to be the *m* x 1 vectors of the x- and y-coordinates of the data points in (2), respectively, then, the system (4) can be written in a compact form as

$$\mathbf{Ac=Y}, \qquad (5)$$

where matrix A is an *m* x *n* matrix whose columns are formed by the elementwise powers of the vector X in the descending order from (n -1) to 0, and c=$(a_{n-1},\ a_{n-2},\ \ldots\ ,\ a_0)$ is the vector of coefficients of an interpolating polynomial if such a polynomial exists.

Thus, the problem has been reduced to determining whether the linear system Ac=Y is consistent and, if yes, finding a solution c which will define an interpolating polynomial.

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

```
function [A,P]=interpolate(m,n,X,Y)
format
P=[];
```

The inputs *m* and *n* are natural numbers ($m, n \geq 1$). The inputs `X` and `Y` are *m* x 1 vectors whose entries are the x- and y-coordinates of the *m* data points, respectively.
The output A is an *m* x *n* matrix for the system of equations (5) (see **Theory** above). The output `P` is an interpolating polynomial whose coefficients are defined by a solution c of the system (5) if the system is consistent, and, if the system is inconsistent, the output for `P` will stay empty.

Continue your function with outputting the matrix A:
**Pre-allocate it as

```
A=ones(m,n);
```

and, then, re-calculate its entries according to the system (4). Output and display A with a message:

```
disp('the matrix A of the system Ac=Y for finding coefficient vector c is')
disp(A)
```

Hint: the columns of the matrix A are elementwise powers of the vector X in the descending order from (n – 1) to 0. You can output A by using either a single "for" loop or a vectorized statement (see Project 0, Exercise 2, Part 3).

Next, we check whether the system Ac**=Y is consistent. Use a conditional statement and the Rouche-Capelli Theorem (see Module 2 in the Lecture Notes) along with the MATLAB function `rank()` and program the following cases:

If the system Ac=Y is inconsistent, output a message `'there is no solution'` and terminate the program: you can either compose a condition statement in an appropriate way or use the command `return`. For an inconsistent system, the empty output for P will stay.
If the system Ac=Y is consistent, first code a message

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

and, then, use a conditional statement to check if the solution is unique or there are infinitely many solutions. If the solution is unique, code a message `'the solution is unique'`, otherwise, output `'there are infinitely many solutions'`.
**Next, output one of the solutions (or the only one) by the MATLAB backslash operator:

```
c=A\Y;
```

Note: we are suppressing the output c here.

Your code will continue (for a consistent system!) with a few more tasks:

Verify that your output vector c is indeed a solution of the system A**c**=**Y**.
**Use a conditional statement and work with the output of the function

```
closetozeroroundoff(A*c-Y,7)
```

If the function above returns the zero vector, code a message `'a solution is found correctly'`**; otherwise**, code a message which could be something like `'what is wrong?!'`, and terminate the program (the empty output for P will stay).

Note: The last message shouldn't be received and, if so, please make corrections in your code!

Next, make arrangements to output the polynomial `P` in a (symbolic) form (1):
First, run the function `closetozeroroundoff` with p=12 on the vector **c to convert into a zero those entries of the vector **c** whose absolute values are less than 10^(-12). Output and display c with a set of commands:

```
c=closetozeroroundoff(c,12);
disp('the vector c of the coefficients of a polynomial P is')
disp(c)
```

**Then, output and display an interpolating polynomial `P` as below:

```
disp('an interpolating polynomial P is')
P=vpa(poly2sym(c),4)
```

Note: the command above outputs and displays the polynomial in symbolic form whose coefficients are evaluated with 4 decimal digit accuracy.

Continue your function with plotting the output polynomial P by using the vector of its coefficients `c`.
**First, create a separate function in file (the code is given below):

```
function []=polyplot(a,b,p)
x=(a:(b-a)/50:b)';
y=polyval(p,x);
plot(x,y);
end
```

This function plots on the interval [a, b] the polynomial whose row vector of coefficients is p.

**Then, include the following lines into your code for the function `interpolate`:

```
plot(X,Y,'*'),hold on
a=X(1);
b=X(end);
polyplot(a,b,c');
hold off
```

The first line of the code fragment plots the data points, the three lines below it make assignments to a and b and plot the polynomial whose (row) vector of the coefficients is `c'`.

This is the end of your function `interpolate`.

**Print the functions `interpolate, polyplot, closetozeroroundoff` in your Live Script.
**Run the function on the choices (a)-(h) as indicated below:

```
%(a)
m=4;n=4;
X=(1:m)'/m
Y=randi(10,m,1)
[A,P]=interpolate(m,n,X,Y);
%(b)
m=4;n=4;
X=(1:m)'/m
Y=X
[A,P]=interpolate(m,n,X,Y);
%(c)
m=5;n=4;
X=(1:m)'
```

```
Y=X
[A,P]=interpolate(m,n,X,Y);
%(d)
m=4;n=6;
X=(1:m)'
Y=randi(10,m,1)
[A,P]=interpolate(m,n,X,Y);
%(e)
m=6;n=4;
X=(1:m)'/m
Y=randi(10,m,1)
[A,P]=interpolate(m,n,X,Y);
%(f)
m=10;n=10;
X=(1:m)'
Y=randi(10,m,1)
[A,P]=interpolate(m,n,X,Y);
%(g)
m=10;n=11;
X=(1:m)'
Y=randi(10,m,1)
[A,P]=interpolate(m,n,X,Y);
%(h)
X=unique(sort(randi(10,10,1)))
m=numel(X);n=10;
Y=randi(10,m,1)
[A,P]=interpolate(m,n,X,Y);
```

EXERCISE 5 (5 points) **Difficulty: Hard**

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

**Write a function in a file that begins with

```
function [S1,S2,L,R]=stochastic(A)
L=[];
R=[];
fprintf('the vector of sums down each column is\n')
S1=sum(A)
fprintf('the vector of sums across each row is\n')
S2=sum(A,2)
```

It accepts as the input a square matrix A with nonnegative entries. Outputs `L` and `R` will be either empty matrices or left and right stochastic matrices, respectively, generated according to the instructions below. The function also outputs and displays vectors `S1` and `S2` with the corresponding messages as indicated above.

Continue your function with a conditional statement and include all of the cases below in your code. Use your outputs `S1` and `S2`. You may find it helpful to employ in this part the MATLAB logical command `all`.

**First, your function has to check whether a matrix A contains both a zero column and a zero row. If yes, output a message "A is neither left nor right stochastic and cannot be scaled to either of them". The empty outputs, assigned previously to `L` and `R`, will stay.
**Then, we will check if A is only left stochastic, or only right stochastic, or both left and right stochastic, that is, doubly stochastic. You will need to program each of the three cases, output a corresponding message on the type of the matrix, and make the assignments listed below:
(1) If A is doubly stochastic, we assign:

```
L=A;
R=A;
```

(2) If A is only left stochastic, we assign:

```
L=A;
```

(`R` stays empty).
(3) If A is only right stochastic, we assign:

```
R=A;
```

(`L` stays empty).
Here is an example how the output message and the assignment for case (2) may look:

```
disp('A is only left stochastic')
L=A;
```

Note: we do not display outputs `L` and `R` here since they are either the matrix A itself or empty.

**Finally, we consider the case when A is neither left nor right stochastic but can be *scaled* to a left stochastic and/or to a right stochastic matrix. In this case, we output a message `'A is neither left nor right stochastic but can be scaled to stochastic'`

Operation of *scaling*:
To scale A to a left stochastic matrix `L`, when possible, we multiply each column of A by the reciprocal of the corresponding entry of S1. To scale A to a right stochastic matrix `R`, when possible, we multiply each row of A by the reciprocal of the corresponding entry of S2.

Proceed with the ***scaling*** in the ways outlined below:
**If neither S1 nor S2 has a zero entry, we are scaling A to the left stochastic matrix `L` and to the right stochastic matrix `R`.
Then, we check if the matrices `L` and `R` are equal (use the function `closetozeroroundoff` with p=7). If it is the case, then A has been scaled to a doubly stochastic matrix, and we output and display one of the matrices, say `L`, with the message as below:

```
disp('A has been scaled to a doubly stochastic matrix:')
disp(L)
```

If `L` and `R` are not equal, display each of them as indicated below:

```
disp('A is scaled to a left stochastic matrix:')
L
disp('and A is scaled to a right stochastic matrix:')
R
```

**If S1 does not have a zero entry but S2 does, we can only scale A to the left stochastic matrix `L` (`R` stays empty). Scale A to left stochastic matrix `L` and display it as below:

```
disp('A can be scaled to left stochastic matrix only:')
L
```

**And, if S2 does not have a zero entry but S1 does, we can only scale A to the right stochastic matrix `R` (`L` stays empty). Scale A to right stochastic matrix `R` and display it as below:

```
disp('A can be scaled to right stochastic matrix only:')
R
```

Note: division by a zero will not be accepted as a valid operation – you need to avoid it in your code when scaling a matrix.

This is the end of your function `stochastic`.

**Create another function in a file called `jord` (the code is below). The function generates a *Jordan Block* square matrix of the size $n \times n$ $(n \geq 2)$ with a scalar *r* on its main diagonal, 1's on the diagonal right above it, and with all other entries zero. We will use it in our Projects.

```
function J=jord(n,r)
J=ones(n);
J=tril(triu(J,1),1)+diag(r*ones(n,1));
end
```

**Print the functions `stochastic`, `jord`, and `closetozeroroundoff` in your Live Script.

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

```
%(a)
A=[0.5,  0,  0.5; 0,  0,  1; 0.5,  0,  0.5]
[S1,S2,L,R]=stochastic(A);
%(b)
A = transpose(A)
[S1,S2,L,R]=stochastic(A);
%(c)
A=[0.5,  0,  0.5; 0,  0,  1; 0,  0,  0.5]
[S1,S2,L,R]=stochastic(A);
%(d)
A=transpose(A)
[S1,S2,L,R]=stochastic(A);
%(e)
A=[0.5,  0,  0.5; 0,  0.5,  0.5; 0.5,  0.5,  0]
[S1,S2,L,R]=stochastic(A);
%(f)
A=magic(4)
[S1,S2,L,R]=stochastic(A);
%(g)
B=[1 2;3 4;5 6]; A=B*B'
[S1,S2,L,R]=stochastic(A);
%(h)
A=jord(3,4)
[S1,S2,L,R]=stochastic(A);
%(i)
A=randi(10,4);A(:,1)=0;A(1,:)=0
[S1,S2,L,R]=stochastic(A);
```

EXERCISE 6 (3 points) **Difficulty: Easy**

In this exercise, you will use some functions created in the previous exercises to output the *equilibrium prices* for a sample Leontief "exchange" ("input-output") model. Please review Section 1.6 of the textbook and Module 6 of the Lecture Notes before proceeding with coding.

We assume that our economy is divided into *n* sectors. Starting with a random *n* x *n* matrix A as the input, we will generate a left stochastic "consumption" matrix L (its columns are probability vectors) and use this matrix to create the "exchange" table between the sectors that would show how each sector's output is divided (exchanged) among the other sectors of the economy. Then, we will construct a homogeneous system to find the equilibrium prices.
**Create a function in MATLAB that begins with the lines:

```
function [] = economy(n)
format
A=randi(10,n);
S1=sum(A);
```

All the tasks outlined below have to be included in your function `economy`.

**First, place a command that will modify A to output a left stochastic matrix `L` by *scaling* each column of A by the reciprocal of the corresponding entry of the vector `S1` (please refer to Exercise 5 of this Project). Do not display `L` (suppress this output).

**Next, generate and display the "exchange" table for our economy using the set of commands (include the commands below into your function):

```
Sector=L;
T=array2table(Sector)
```

Then, output the matrix `B` by using the matrix `L` that defines a homogeneous system Bx**=**0** whose solution **x** gives the equilibrium prices for our economy.
Hint: For help with constructing `B` from the "consumption" matrix `L`, please refer to the first Example in Lecture 6.

Next, output a basis for the solution set of Bx**=**0**. There will be one vector in the basis which is the output C of the function `homobasis` created in Exercise 3 of this Project - you will need to include into your code the following line as indicated below:

```
C=homobasis(B)
```

Notes:
We work with the matrix B, not with the reduced echelon form of B.
The output matrix C will have only one column.
The system B**x**=**0** has n variables: x1, x2, … , xn. The function `homobasis` will list the only one free variable that the system has, which is xn, (with n specified in the output message).

Finally, display a solution **x in a (symbolic) parametric-vector form using the free variable xn as a parameter - we will denote the free variable p. The output **x** has to be coded as below:

```
syms p
fprintf('vector of equilibrium prices with parameter p=x%i is\n',n)
x=vpa(p*C,4)
```

Note: The command `syms p` creates a symbolic variable `p`. The command `x=vpa(p*C,4)` displays a symbolic vector with the entries of `C` are evaluated with 4 decimal digit accuracy.

This is the end of your function `economy`.

**Print the functions `economy` and `homobasis` in the Live Script.

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

```
%(a)
n=2;
economy(n)
%(b)
n=4;
economy(n)
%(c)
n=7;
economy(n)
```

EXERCISE 7 (5 points) **Difficulty: Moderate**

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

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

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

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

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

We will consider two functions in this exercise:

$F(x) = arctan(x) + 2x - 1$ and $G(x) = x^3 - (2/3)x - 1$.

First, we will create a **function handle.** For help, refer to the documentation in MATLAB: https://www.mathworks.com/help/matlab/matlab_prog/creating-a-function-handle.html?searchHighlight=handle&s_tid=doc_srchtitle#buvu9u8-1

Below we are creating handles to anonymous functions and calculating the first derivatives of the function handles.

**Type in the Live Script:

```
format
syms x
F = @(x) atan(x) + 2*x - 1
F1 = eval(['@(x)' char(diff(F(x)))])
G=@(x) x.^3-(2/3)*x-1
G1=eval(['@(x)' char(diff(G(x)))])
```

Notes: in the code fragment above, the MATLAB command `syms x` defines a symbolic variable x. `F` and `G` are the function handles and `F1` and `G1` are the function handles of the first derivatives of F and G, respectively.

Next, we output 2-D plots of the functions F, G, and the zero function. This will allow us to visualize the x-intercepts and choose initial approximations x_0 close to the x-intercepts that we are approximating.

**Type (or copy and paste) in the Live Script the code fragment given below – it will output the graphs of F and G together with the graph of the function y=0.

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

Notice that here we created another symbolic function `yzero`.

It is obvious from the properties of the function `F(x)` that it has only one real zero which we will approximate.
From the properties of the function `G`, which is a polynomial of degree 3, it follows that it has at least one real zero. We can use MATLAB commands to show that it has only one real zero that we are going to approximate: we use a variable p for `G` in the code fragment below, output all zeros of the polynomial p, and isolate the real one.
**Type and Run in the Live Script:

```
syms x
p=x^3-(2/3)*x-1;
r=sym2poly(p);
R=roots(r);
disp('all zeros of the polynomial are')
disp(R)
for k=1:numel(R)
    if closetozeroroundoff(R(k)-real(R(k)),12)==0
        R(k)=real(R(k));
        disp('a real zero of the polynomial is')
        disp(R(k))
    end
end
```

Here, we, first, output a vector `r` of the coefficients of p by the command `sym2poly(p)`. Then, we used the function `roots(r)` to output all zeros of the polynomial p. And, finally, we isolated the real zero by using the condition that, if the imaginary part of a complex number is negligibly small by the magnitude, we can consider the number to be real.

**Continue with creating a function in MATLAB that approximates a real zero. The function begins with:

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

The inputs `fun` and `dfun` are a function and its first derivative, respectively, and `x0` is an initial approximation. The output `root` will be our approximation of the real zero of a function.

**Continue your function `newtons` with the line below that outputs a MATLAB approximation of a real zero:

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

Display `x` with a message:

```
disp('a MATLAB approximation of the real zero of the function is')
x
```

Next, we will calculate consecutive iterations according to the Newton's method and assign to the output `root` the iteration which is the first one that falls into the range of 10^(-12) from the MATLAB approximation `x` of that zero. We will also count the number of iterations, N.
**Assign initially
```
N=0;
```
and employ a "while" loop to get the iterations according to the formula in (1).
The loop runs while the absolute value of the difference between a consecutive iteration and `x` is greater than or equal to (≥) `10^(-12)`.
(Suppress (do not display) all intermediate iterations in the loop.)

Note: if the initial iteration is already the `root`, then the output for N has to be a 0.

**After the "while" loop breaks, assign your last iteration to the output `root` and display it with a message:
```
disp('our approximation of the real zero of the function is')
root
```
**Output and display the number of iterations N with a message as below:
```
disp('the number of iterations to archive the required accuracy is')
N
```

This is the end of your function `newtons`.

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

**Test the function.
Type and Run in the Live Script the following:
```
%Test
H=@(x) x-1
H1=eval(['@(x)' char(diff(H(x)))])
x=linspace(-2,2);
plot(x,H(x),x,yzero(x));
fun=H;
dfun=H1;
x0=0
root=newtons(fun,dfun,x0);
x0=1
root=newtons(fun,dfun,x0);
```
BONUS! (1 point)
Write a comment that would explain a reason for the output N to have the values that you received for the given function and the given choices of the initial approximations.

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

Part I
You will work with the function F in this Part.

**Input the function handles:
```
fun=F;
dfun=F1;
```

**Use the graph of the functions `F` and `y=0` to choose three different values of the initial approximation `x0` of the real zero of `F`. Input (and display) each initial value `x0` in your Live Script and, for each `x0`, run the function

```
root=newtons(fun,dfun,x0);
```

Part II

You will work with the function G in this part.

**First, input the corresponding function handles.

**Then, run `root=newtons(fun,dfun,x0);` on the choices (a)-(h) as indicated below:

```
%(a)
x0=3
root=newtons(fun,dfun,x0);
%(b)
x0=2
root=newtons(fun,dfun,x0);
%(c)
x0=1.2
root=newtons(fun,dfun,x0);
%(d)
x0=1.1
root=newtons(fun,dfun,x0);
%(e)
x0=1
root=newtons(fun,dfun,x0);
%(f)
x0=sqrt(2)/3
root=newtons(fun,dfun,x0);
%(g)
x0=0.4714
root=newtons(fun,dfun,x0);
%(h)
x0=0
root=newtons(fun,dfun,x0);
```

BONUS! (2 points)

% Analyze the consecutive iterations throughout the choices (a)-(h) in **Part II** and write a comment on the patterns of behavior of sequences of iterations depending on the choices of initial approximations. Specifically, look into how close is an initial iteration to the x-intercept of the function and what is happening when an initial iteration is chosen to be very close to a real zero of the first derivative of the function.

Hint: for the **bonus** part, you may temporarily display all intermediate iterations that come from the "loop". Please be sure to put the semicolon back in when preparing the Exercise for submission!

You may also need to review in more detail the Newton's Method from Calculus.

THIS IS THE END OF PROJECT 1