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 will be completed in the Live Script using the Live Editor.

Each exercise has to begin with the line
Exercise#
You should also mark down the parts such as (a), (b), (c), and etc. This makes grading easier.
Important: we use the default `format short` for the numbers in all exercises unless it is specified otherwise. We do not employ `format rat` since it may cause problems with running the codes and displaying matrices in Live Script. **If `format long` has been used, please make sure to return to the default `format` in the next exercise.**

Part I. Subspaces & Bases

Exercise 1 (5 points) **Difficulty: Hard**
In this exercise, you will be given two matrices A and B. You will create a function that works with the column spaces of the matrices. 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 if Col A and Col B have the same dimension, and, if yes, whether Col A = Col B. Obviously, when two subspaces have the same dimension, it might 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 the file that begins with
```
function []=columnspaces(A,B)
m=size(A,1);
n=size(B,1);
```

First, your function has to check if Col A and Col B are subspaces of the same Euclidian vector space, that is, if m and n are equal.
**If Col A and Col B are subspaces of different spaces, you will output the corresponding message and terminate the program.
**If Col A and Col B are subspaces of the same vector space $\mathbb{R}^m$, you will code the corresponding message that also outputs the dimension of the vector space $\mathbb{R}^m$, which is m. An example of the output message is below:
```
fprintf('Col A and Col B are subspaces of R^%i\n',m)
```

**Then, your function will continue with calculating the dimensions of Col A and Col B. Output them with the corresponding messages.
**Next, check if both conditions hold: Col A is the whole $\mathbb{R}^m$ and Col B is the whole $\mathbb{R}^m$. If it is the case, output a message that Col A = Col B = $\mathbb{R}^m$ and <u>terminate</u> the program.

If either Col A or Col B or both are not the whole $\mathbb{R}^m$, your code will continue.
**First, it will check if Col A and Col B have the same dimension. If not, output the message that the dimensions of Col A and Col B are different.
**If the dimensions are the same, the code has to check whether Col A and Col B are the same set. If it is the case, output the message that Col A = Col B.
If it is not the case, output a message that the dimensions of Col A and Col B are the same but Col A ~= Col B.

<u>Note</u>: You <u>cannot</u> use a MATLAB built-in function `colspace(sym())` within the function `columnspaces`.
This is the end of the function `columnspaces`.

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

```
%(a)
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)
%(b)
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'
columnspaces(A,B)
%(c)
A=magic(5)
B=ones(5)
columnspaces(A,B)
%(d)
A=magic(4)
B=eye(4)
columnspaces(A,B)
%(e)
A=magic(4)
B=[eye(3);zeros(1,3)]
columnspaces(A,B)
%(f)
A=magic(3)
B=[hilb(3),eye(3)]
columnspaces(A,B)
```

%Based on the outputs for part (a), write a comment in the Live Script on a possible effect of the elementary row operations on the column space of a matrix.

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

In this exercise, you will create a basis for the Col A in two ways: using a built-in MATLAB function `colspace(sym(A))`and using a function `shrink()` which you will create in a file.

Part 1:

The column space of a matrix A is a subspace spanned by the columns of A. It is possible that the set of columns of A is not linearly independent, that is, not all columns of A will be in a basis. The set of columns of A can be “shrunk” into a basis for Col A by using the function `shrink()`. Here is a code:

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

**Create the function `B=shrink(A)`in a file.
**Print the function `shrink` in your Live Script.
**Input a matrix as indicated below:

```
A=magic(4); A(:,3)=A(:,2)
```

Run the command

```
rref(A)
```

**Then, run the two lines given below (display the outputs):

```
[R,pivot]=rref(A)
B=A(:,pivot)
```

%Write a comment in the Live Script on the outputs for each of the two lines above.

**Next, run the command:

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

%Explain the difference between the outputs for the last command and for the command `[R,pivot]=rref(A)`

**Input the matrices

```
A=[2 -4 -2 3;6 -9 -5 8;2 -7 -3 9;4 -2 -2 -1;-6 3 3 4]
B=shrink(A)
```

**Run the function `columnspaces(A,B)`, which was created in Exercise 1 of this Project, to show that the matrices A and B have the same column space.
**Print the function `columnspaces` in your Live Script

%Explain in the Live Script why the set of the columns of B forms a basis for the column space of A.

Part 2

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

```
A=[2 -4 -2 3;6 -9 -5 8;2 -7 -3 9;4 -2 -2 -1;-6 3 3 4]
```

**Type and run in the Live Script:

```
R=rref((A'))
M=shrink(R')
B=colspace(sym(A))
```

**Next, run the command:

```
D=double(B)
```

which converts a symbolic matrix `B` to a double-precision matrix `D`. Run the logical command indicated below to compare the matrices `D` and `M`.

```
isequal(D,M)
```

Next, run the command `columnspaces(A,B)` created in Exercise 1 of this Project to show that the column spaces of the matrices `A` and `B` are the same.

BONUS: 2 points
% Analyze the outputs for Part 2 of this Exercise and determine the path that the function `colspace(sym())` takes to create a basis for Col A. Justify the statement that the set of the columns of `B` forms a basis for the Col A.

BONUS: 1 point
**Use the results of Part 2 of this exercise to write a new function in a file

```
function []=columnspaces_1(A,B)
```

which takes the function `columnspaces()` and modifies only one part of this function, the one that checks if Col A = Col B; you will need to employ a MATLAB built-in function `colspace(sym())`within your code for the function `columnspaces_1`.
**Print the function `columnspaces_1` in your Live Script.

Note: You can test here that the new function `columnspaces_1()` gives the same outputs as the function `columnspaces()` by running it on the matrices in Exercise 1; however, you do not need to include these tests into Exercise 2 file for the submission.

Exercise 3 (4 points) **Difficulty: Easy**

In this exercise, you will be given an $m \times n$ matrix A. Your program has to create two matrices B and D. The matrix B is formed by the pivot columns of A – the set of columns of B is a basis for Col A which is a subspace of $\mathbb{R}^m$. The set of columns of the matrix D forms a basis for $\mathbb{R}^m$. If Col A $\neq \mathbb{R}^m$, the matrix D is constructed by using all columns of B and some columns of the $m \times m$ identity matrix.

**Create the function in a file which begins with

```
function [B,D]=basis(A)
m=size(A,1);
```

**First, use the function `shrink()`, which was created in Exercise 2 of this project, within the function `basis` to output (and display) the matrix B of the pivot columns of A – supply the output B with a message that `'a basis for Col A is the set of columns of'`
(display B)
**Then, your code has to check whether the set of columns of the matrix B forms a basis for $\mathbb{R}^m$. If yes, the program breaks with the message:

```
fprintf('a basis for R^%i is D=B\n',m)
```

and assigns

```
D=B;
```

**If the set of the columns of B does not form a basis for $\mathbb{R}^m$, you should create a matrix D whose first columns are all columns of the matrix B and the rest of the columns come from the matrix `eye(m)`, such that the set of all columns of D forms a basis for $\mathbb{R}^m$. You can use your function `shrink()` to create D.
**Next, you will write a set of commands within your code to verify whether the set of columns of D is, indeed, a basis for $\mathbb{R}^m$. If your code confirms that, display a corresponding message, for example:

```
fprintf('a basis for R^%i is\n',m)
```

(display D).
Otherwise, an output message could be: `'something definitely went wrong!'`

**Print the function `basis` and `shrink` within the Live Script.
**Run the function `[B,D]=basis(A);` on the following matrices (display the inputs):

```
%(a)
A=[1 0;0 0;0 0;0 1]

%(b)
A=[0 0;2 0;3 0;0 0]

%(c)
A=magic(4)

%(d)
A=magic(5)

%(e)
A=ones(4)
```

Part II. Isomorphism & Change of Basis

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

DESCRIPTION: In this exercise, you will be given a set of n polynomials, which is denoted B. The polynomials in B are from the subspace P_{n-1} of the polynomials whose degrees do not exceed ($n-1$). A standard basis for P_{n-1} is $E=\{x^{n-1}, x^{n-2}, \dots, x, 1\}$. You will determine whether the given set B forms a basis for P_{n-1}. This could be done by employing isomorphism from P_{n-1} onto $\mathbb{R}^n$. You will create a matrix P, which is the matrix of the E-coordinates of the polynomials in B. According to the isomorphism, the set of the polynomials in B forms a basis for the subspace P_{n-1} if and only if the set of the columns of P forms a basis for $\mathbb{R}^n$.
If the set B is a basis for P_{n-1}, your function will continue with two more tasks: (1) you will find a vector **y** of the B-coordinates of a polynomial Q, where Q is given in a symbolic form through the standard basis E, and (2) you will output a polynomial R in a symbolic form (through the standard basis E), given its B-coordinate vector **r**.
(For a help with this exercise, please refer to lecture Module 19.)

We will use the function `closetozeroroundoff` with a parameter p = 7 within the 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
```

An input $B=[B(1), B(2), ..., B(n)]$ is a vector whose components are polynomials from the vector space P_{n-1}, Q is a single polynomial from the same space P_{n-1}, and **r** is a numerical vector with n entries.

Note on the format of input polynomials: 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 given 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 we will convert this leading coefficient into a 0 by running within our code the function `closetozeroroundoff` with p = 7 on the matrix P of the coefficients of the polynomials in B.

**Continue your function `polyspace` with the commands:

```
u=sym2poly(B(1));
n=length(u);
```

The command `sym2poly(B(1))` takes the coefficients of the polynomial B(1), which is written through the standard basis E in the descending order according to the degree, and writes them as a row vector (a $1\times n$ matrix).

Note: The number n is the dimension of the vector space P_{n-1}; thus, P_{n-1} is isomorphic to the Euclidean space $\mathbb{R}^n$. The number n will be used later in this program.

**To use isomorphism, you will create an $n\times n$ matrix P, whose columns are the vectors of the coefficients of the polynomials in the set B – each column is generated by the commands `transpose` and `sym2poly` (as described above) – the columns of P are the E-coordinate vectors of the polynomials in B.

Note: to output P, you can employ a “for loop” in your code (do not display the output P here).

**Then, you will convert to 0 the entries of the matrix P which are in the range of 10^(-7) from a 0 by running the function:

```
P=closetozeroroundoff(P,7)
```

**Display the new matrix P with the message:

```
fprintf('matrix of E-coordinate vectors of polynomials in B is\n')
```

(display P).

Then you will check if the columns of P form a basis for $\mathbb{R}^n$ - use the command `rank()`.

**If the set of the columns of P is not a basis for $\mathbb{R}^n$, then, due to the isomorphism, the set of the polynomials B does not form a basis for P_{n-1}. In this case, you will output a corresponding

message and calculate and output matrix `A=rref(P)`, which is the reduced echelon form of the matrix P (the matrix A should visualize the fact that the columns of P do not form a basis for $\mathbb{R}^n$). After that, the program terminates.
Examples of the output messages for this part are given below:

```
sprintf('the polynomials in B do not form a basis for P%d',n-1)
fprintf('the reduced echelon form of P is\n')
```

(display A).

**If the set of the columns of P forms a basis for $\mathbb{R}^n$, then, due to the isomorphism, you will output a message that the polynomials in B form a basis for the subspace of the polynomials P_{n-1}, and your function will continue with two more tasks:

(1) Given the polynomial Q written in a symbolic form through the standard basis E, calculate the B-coordinate vector **y** of Q. To calculate the vector **y**, you will use the Change-of-Coordinates equation.
Hint: use a MATLAB command `sym2poly()` to output the row vector of the E-coordinates of the polynomial Q. Then, run `closetozeroroundoff` with p = 7 within your code on the E-coordinate vector to convert the leading entry into a 0, when needed. After that, you can proceed with calculation of the B-coordinate vector **y** of Q.

Your outputs for this part will be a message and the vector **y**:

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

(display **y**).

(2) Given the B-coordinate vector **r** of a polynomial R, output the polynomial R written in a symbolic form through the standard basis E.
Hint: you will need to calculate the E-coordinate vector of the polynomial R by the Change-of-Coordinates equation, and, then, use that vector and the command `poly2sym()` to output the required polynomial R.

The output R should be supplied with a message, for example:

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

(display R).

For a help with this exercise, you may find it useful to review the second Example in the Lecture Notes for Module 19.
This is the end of the function `polyspace`.

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

```
syms x
```

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

```
P=polyspace(B,Q,r);
```

on each set of the variables (a)-(c). (Display the inputs in the Live Script.)

```
%(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^3-2*x^2+x-1
r=[1;-3;2;4]
%(b)
B=[x^3-1,10^(-8)*x^3+2*x^2,10^(-8)*x^3+x,x^3+x]
Q=10^(-8)*x^3-2*x^2+x-1
r=[1;-3;2;4]
%(c)
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=x^4-2*x+3
M=magic(5);r=M(:,1)
```

Part III. Application to Calculus

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

In this exercise, you will approximate the definite integral of a function using both Riemann sums and a MATLAB built-in function `integral`.

The code accepts as inputs: a function `fun`, a column vector **n** whose entries are the numbers of subintervals of partitions, and two scalars a, b – the endpoints of the interval of integration. Riemann sum calculations should be performed using partitions of $[a,b]$ by subintervals of the equal length `h(j)` defined as

`h(j)=(b-a)/n(j);`

where `n(j)`is a jth entry of **n**, with `j=1:N` and `N=length(n)`. Each entry of the vector **n,** `n(j)`, is the number of the subintervals of the corresponding partition of $[a,b]$.

Your function has to return a table T whose first column formed by the `N` entries of the vector **n**, where `n(j)` entry of **n** defines the `j`th partition of [a, b]. For a `j`th partition (`j=1:N`), you will calculate the Riemann sums approximations of the definite integral by choosing the left endpoints, the middle points, and the right endpoints of each subinterval of the partition – these sums will be the `j`th entries of the column vectors L, M, and R, respectively. The vectors L, M, and R will form the columns 2, 3, and 4 of the output table T.

**Write a function that begins with

```
function T=reimsum(fun,a,b,n)
format compact
N=length(n);
```

It has to calculate the column vectors L, M, R as described above and form a matrix

`A=[n,L,M,R];`

The following command converts the N-by-4 array A into an N-by-4 table T with the names of the variables as indicated below. This command should be present in your code.

```
T=array2table(A,...
    'VariableNames',{'n','Left','Middle','Right'})
```

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

**Type

```
syms x
format long
```

**Run the function `T=reimsum(fun,a,b,n)` in the way indicated below on the function handles. At the ends of each part (a) and (b), you will also run (and display the output) a MATLAB built-in function

`Int=integral(fun,a,b)`

which calculates definite integral of a function `fun.` The output of this function will allow you to verify if your approximations of the definite integral are correct.

```
%(a)
fun=@(x) x.*tan(x) + x + 1
a=0;b=1;

n=(1:10)';
T=reimsum(fun,a,b,n)

n=[1;5;10;100;1000;10000];
T=reimsum(fun,a,b,n)

Int=integral(fun,a,b)

%(b)
fun=@(x) x.^4 - 2*x - 2
a=0;b=3;

n=(1:10)';
T=reimsum(fun,a,b,n)

n=[1;5;10;100;1000;10000];
T=reimsum(fun,a,b,n)

Int=integral(fun,a,b)
```

%Write a comment in your Live Script for which choice of the points on a subinterval of a partition (left endpoints, middle points, or right endpoints) the Riemann sums give the best approximation of the integral.

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

In this exercise you are given a polynomial, and you will write a code that outputs an antiderivative of the polynomial.

**Write a function in the file that begins with

```
function I=polint(P)
format compact
syms x
```

which accepts as an input a polynomial P. A polynomial will be written in a symbolic form through the standard basis (you may find it helpful to review Exercise 4 of the current Project).

The function has to calculate indefinite integral of the polynomial assigning a value 3 to an arbitrary constant. The output `I` has to be a polynomial written in a symbolic form through the standard basis. Do not use a MATLAB built-in function `int(P)` within the function `polint`.

Suggested commands within your code: `sym2poly`, `poly2sym`, `length`. A single "for loop" or a vectorized statement can be used.
This is the end of the function `polint`.

**Print the function `polint` in your Live Script.
**Type in the Live Script:

```
format
syms x
```

**Run `I=polint(P)` on the polynomials in parts (a) and (b). (You can copy and paste each polynomial in the Live Script.)

(a) `P=6*x^5+5*x^4+4*x^3+3*x^2+2*x+6`
(b) `P=x^5-2*x^3+3*x+5`

**For each part, (a) and (b), after getting (and displaying) the output I, run a logical command

```
isequal(I,int(P)+3)
```

to make sure that your output matches the output of a MATLAB built-in function. If it doesn't, consider making corrections in your code.

Part IV. Application to Markov Chains

Exercise 7 (4 points)**:** **Difficulty: Moderate**
In this exercise, you will work with Markov Chains. Please read the part **Theory** and perform the tasks indicated below.
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 a 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

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

In other words, a Markov chain is described by the first-order difference equation $\mathbf{x}_{k+1} = P\mathbf{x}_k$ for $k = 0, 1, 2, \ldots$

The *n* entries of vectors $\mathbf{x}_k$ 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* $\mathbf{q}$ such that $P\mathbf{q} = \mathbf{q}$.

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.

**Create a function in a file that begins with

```
function q=markov(P,x0)
format
n=size(P,1);
```

**First, the function has to check whether the given $n \times n$ matrix P, whose entries will be positive numbers, is stochastic (that is, left-stochastic). If P is not left-stochastic, the program displays a message `'P is not a stochastic matrix'`, returns `q=[];` and terminates.

**If P is left-stochastic (then it will be a regular stochastic matrix), we do the following:

(1) Find the unique steady-state vector **q**.

Recall: the steady-state vector **q** is a probability vector which is a solution of the equation $P\mathbf{q} = \mathbf{q}$, or, equivalently, the equation $(P - eye(n))\mathbf{q} = \mathbf{0}$. You can find a basis for the solution set of this system (which has to contain only one vector) by using a MATLAB function

```
null(P-eye(n),'r')
```

Then, you will need to output the probability vector **q** that belongs to the solution set. Display **q** with a message that it is the steady-state vector of the system.

(2) Next, verify that the Markov chain converges to **q** by calculating consecutive iterations $\mathbf{x}_1 = P*\mathbf{x}_0$, $\mathbf{x}_2 = P*\mathbf{x}_1$, $\mathbf{x}_3 = P*\mathbf{x}_2$, ... , until, for the first time, $\mathrm{norm}(\mathbf{x}_k - \mathbf{q}) <$ `10^(-7)`

Output the number of iterations *k* that are required to archive this accuracy – supply your output with the corresponding message.

**Type the function `markov` in your Live Script.

**Run the function

```
q=markov(P,x0);
```

on the following choices of the matrix `P` and the vector `x0`. (Display the inputs.)

```
%(a)
P=[.6 .3;.5 .7]
x0=[.3;.7]
%(b)
P=[.5 .3;.5 .7]
```

`x0` is the same as in part (a)

```
%(c)
P=[.9 .2;.1 .8]
x0=[.10;.90]
```

where P is a migration matrix between two regions.

```
%(d)
```

A migration matrix P is the same as in (c)

```
x0=[.81;.19]
```

```
%(e)
P=[.90 .01 .09;.01 .90 .01;.09 .09 .90]
x0=[.5; .3; .2]
```

% Compare the output vectors **q** for parts (c) and (d), which have the same matrix `P` and different vectors `x0`, and write a comment whether a choice of the initial vector `x0` has an effect on the steady-state vector q. Does a choice of `x0` have an effect on the number of iterations k? Write a comment about it as well.

This is the end of Project 3