# MATLAB PROJECT 4

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

**BEGIN** with creating Live Script **Project4.**
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. If `format long` or `format rat` has been used, please make sure to return to the default `format` in the next exercise.**

## Part I. Eigenvalues, Eigenvectors & Diagonalization

**Exercise 1** (5 points) **Difficulty: Hard**
In this exercise, you will work with the eigenvalues of a given $n \times n$ matrix A. First, you will find all eigenvalues and, then, you will take distinct eigenvalues and find orthonormal bases for the corresponding eigenspaces and the dimensions of the eigenspaces. Next, you will check if A is diagonalizable by applying the general theory. If a matrix A is diagonalizable, the output has to contain an invertible matrix P and the diagonal matrix D, such that, $A = PDP^{-1}$, or, equivalently, $AP = PD$ and P is invertible. Next, for diagonalizable matrices, you will run a built-in MATLAB function, which also performs diagonalization, and you will compare its outputs with the outputs P and D of your function.

**Create a function in MATLAB that begins with

```
function [P,D]=eigen(A)
format compact
[~,n]=size(A);
```

Your function `[P,D]=eigen(A)` will have a set of commands which generates the following outputs for an $n \times n$ matrix A. Each output has to be supplied with a corresponding message - you could use the commands `disp` and `fprintf`.

**Part 1**. Output the vector **L** of eigenvalues of the matrix.
To do that, you will go through several steps. Please suppress all intermediate outputs and display only the final output vector **L**.
**Find a row vector **L** of all eigenvalues, where each eigenvalue repeats as many times as its multiplicity. A basic MATLAB command for this part `L=eig(A)` returns a column vector of all eigenvalues of A, and we use the function `transpose()` to get a row vector.
**You will also need to **sort** the entries of **L** – use the MATLAB command `sort()` to output the entries of **L** in ascending order.

To output vector L correctly, you will go through two more steps:
**In your code, you have to ensure that the multiple eigenvalues show as the same number. You will use `closetozeroroundoff` function with the parameter p = 7 to compare the eigenvalues and assign the same value to the ones that are within 10^(-7) tolerance of each other. To perform this task, you can go through the sequence of the sorted eigenvalues and, if there is a set of eigenvalues that are within 10^(-7) from each other, you will assign the first eigenvalue that comes across to all of them in that set.
Note: please work with the eigenvalues that are stored in MATLAB – we do not round off any eigenvalues on this step.
**A singular matrix A has a zero eigenvalue; however, when running your code, a zero eigenvalue may not show as a zero due to round-off errors in MATLAB. If a matrix A is not invertible, you will run the function `closetozeroroundoff` with p = 7 on the vector of the eigenvalues to ensure that the zero eigenvalues show as a zero. To check whether a matrix is not invertible, please use a MATLAB command `rank()`.

After performing all the tasks outlined above, you will display your final output **L** – a row vector of the sorted eigenvalues of A where all multiple eigenvalues equal to each other and the zero eigenvalues (for singular matrices) show as a zero.

**Part 2**. Construct an orthonormal basis W for each eigenspace.
**Output a row vector **M** of all distinct eigenvalues (no repetitions are allowed). The MATLAB command `unique()` can be used here.

For each distinct eigenvalue `M(i)`, your function has to do the following:
**Find the multiplicity, `m(i)`, of the eigenvalue `M(i)`. Output it with the message:
```
fprintf('Eigenvalue %d has multiplicity %i\n',M(i),m(i))
```
**Output an orthonormal basis **W** for the eigenspace for the eigenvalue `M(i)`. Display it with the message:
```
fprintf('A basis for eigenvalue lambda = %d is:\n',M(i))
    W
```
Hint: An appropriate command in MATLAB that creates an orthonormal basis for the null space of a matrix is `null()`.
**Calculate the dimension `d(i)` of **W**. Output it with the message:
```
fprintf('Dimension of eigenspace for lambda = %d is %i\n',M(i),d(i))
```

**Part 3**. Construct a Diagonalization when possible.
(For help with this part, please review the material of Lecture 14)
**First, we determine if A is diagonalizable. You will work with the multiplicity of the eigenvalue `M(i)`, `m(i)`, and the dimension of the corresponding eigenspace, `d(i),` for every i.
If A is not diagonalizable, output a corresponding message, assign the empty outputs
```
P=[];D=[];
```
and terminate the program.
If A is diagonalizable, output a corresponding message, and your function will continue with constructing a diagonalization:
**Output and display a matrix P constructed from the combined bases W for the eigenspaces.
**Output and display a diagonal matrix D with the corresponding eigenvalues on its main diagonal.

**Verify that both conditions hold: AP=PD and P is invertible. To verify the first condition, you will need to use the function `closetozeroroundoff` with the parameter p = 7. To verify the second condition, please use the command `rank()`. If both conditions hold, output a message: `'Great! I got a diagonalization!'` Otherwise, output a message: `'Oops! I got a bug in my code!'` and terminate the program.

**Part 4**. Compare your outputs with the corresponding ones of a MATLAB built-in function. If the diagonalization is confirmed, your code will continue with the following task.
**There is a MATLAB built-in function that runs as
`[U,V]=eig(A)`
which, for a diagonalizable matrix A, generates a diagonal matrix V, with the eigenvalues of A on its main diagonal, and an $n \times n$ invertible matrix U of the corresponding eigenvectors, such that AU=UV. Place this function in your code and display the outputs U and V.

**Compare the output matrix U with the matrix P: write a set of commands that would check on the following case:

> The sets of columns of P and U are the same or match up to a scalar (-1), where the order of the columns does not matter.

If it is the case, output a message `'Sets of columns of P and U are the same or match up to scalar (-1)'`.
If it is not the case (which is possible), the output message could be `'There is no specific match among the columns of P and U'`
Note: You will need to use the function `closetozeroroundoff` with p = 7 when comparing the sets of columns of the matrices P and U.

**Verify that the matrices D and V have the same set of diagonal elements (the order does not count either). If it is the case, output a message `'The diagonal elements of D and V match'`. If it is not the case, output something like: `'That cannot be true!'`
Hint: To perform this task, you can "sort" the diagonal elements of V and compare them with the vector L using the function `closetozeroroundoff` with p = 7.

**This is the end of the function `eigen`.**

**Print the functions `eigen`, `closetozeroroundoff`, `jord` in the Live Script.
(The function `jord` was created in Exercise 1 of Project 1.)
**Type in the Live Script
`format`
**Run the function `[P,D]=eigen(A);` on the following matrices:

```
%(a)
A=[3 3; 0 3]
%(b)
A=[4 0 0 0; 1 3 0 0; 0 -1 3 0; 0 -1 5 4]
%(c)
A=jord(5,5)
%(d)
A=diag([3, 3, 3, 2, 2, 1])
%(e)
A=magic(4)
```

```
%(f)
A=ones(4)
%(g)
A=magic(5)
%(h)
A=hilb(7)
%(k)
A=[5 8 -4;8 5 -4;-4 -4 -1]
```

**Exercise 2** (3 points) **Difficulty: Easy**

In this exercise, we will construct an orthogonal diagonalization of an $n \times n$ symmetric matrix.

Theory: A square matrix A is called symmetric if $A^T = A$.

A matrix A is said to be orthogonally diagonalizable if there exists an orthogonal matrix $P$ ($P^{-1} = P^T$) and a diagonal matrix $D$, such that $A = PDP^{-1}$, or equivalently, $A = PDP^T$.

An $n \times n$ matrix A is orthogonally diagonalizable if and only if A is symmetric.

**Create a function in MATLAB

```
function []=symmetric(A)
```

**First, your function has to check whether A is symmetric. If not, output a message that the matrix is not symmetric and terminate the program.

**If a matrix is symmetric, you will construct an orthogonal diagonalization using a built-in MATLAB function

```
[P,D]=eig(A)
```

in your code. The outputs have to be an orthogonal matrix P of the eigenvectors of A and a diagonal matrix D with the corresponding eigenvalues on its main diagonal. Place this function in your code and display the outputs P and D.

**Next, you should verify in your code that you have got an orthogonal diagonalization, that is, both conditions hold: AP=PD and P is an orthogonal matrix. If it is the case, output a corresponding message. Otherwise, a message could be `'What is wrong?!'`

Note: you will need to use the function `closetozeroroundoff` with p = 7 in your code for verification of each of the two conditions above.

(For the help with this exercise, please refer to Lecture 29.)

**This is the end of the function `symmetric`.**

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

**Run the function on the given matrices as indicated below:

```
%(a)
A=[2 -1 1;-1 2 -1;1 -1 2]
symmetric(A)
%(b)
A=[2 -1 1;-1 2 -2;1 -1 2]
symmetric(A)
%(c)
B=A*A'
symmetric(B)
```

```
%(d)
A=[3 1 1;1 3 1;1 1 3]
symmetric(A)
%(e)
A=[5 8 -4;8 5 -4;-4 -4 -1]
symmetric(A)
%(f)
A=[4 3 1 1; 3 4 1 1 ; 1 1 4 3; 1 1 3 4]
symmetric(A)
```

## Part II. Orthogonal Projections & Least-Squares Solutions

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

In this exercise, you will create a function `proj(A,b)` which will work with the projection of a vector **b** onto the Column Space of an $m \times n$ matrix A.
For a help with this exercise, please refer to Lectures 28, 30, and 31.

**Theory**: When the columns of A are linearly independent, a vector **p** is the orthogonal projection of a vector $\mathbf{b} \notin$ Col A onto the Col A if and only if $\mathbf{p} = A\hat{\mathbf{x}}$, where $\hat{\mathbf{x}}$ is the unique least-squares solution of $A\mathbf{x} = \mathbf{b}$, or equivalently, the unique solution of the ***normal equations***

$$A^T A\mathbf{x} = A^T\mathbf{b}.$$

Also, if **p** is the orthogonal projection of a vector **b** onto Col A, then there exists a unique vector **z**, such that, $\mathbf{z} = \mathbf{b} - \mathbf{p}$ and **z** is orthogonal to Col A.

**Your program should allow a possibility that the columns of A are not linearly independent. In order for the algorithm to work, we will use the function `shrink()` to create a new matrix, also denoted A, whose columns form a basis for Col A. Thus, we assign in our code:

```
A=shrink(A);
```

The function `shrink()` was created in Exercise 2 of Project 3 and you should have it in your Current Folder in MATLAB. If not, please see the code below:

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

You will create a function `proj` as defined below. The inputs are an $m \times n$ matrix A and a column vector **b**. The outputs will be the projection **p** of the vector **b** onto the Col A and the vector **z** which is the component of **b** orthogonal to the Col A.

**Your function will begin with:

```
function [p,z]=proj(A,b)
format compact
A=shrink(A);
m=size(A,1);
```

**First, the program checks whether the input vector **b** has exactly *m* entries, where *m* is the number of rows in A. If it doesn't, the program breaks with a message `'No solution: dimensions of A and b disagree'`, outputs and displays the empty vectors **p** and **z**.

If **b** has exactly *m* entries, we proceed to the next step:
**Determine whether it is the case that **b** ∈ Col A. You should use a MATLAB function `rank()`. If **b** ∈ Col A, you will just assign the values to the vectors **p** and **z** based on the general theory (no computations are needed!) and output a message `'b is in Col A'`. After that, the program terminates.

If **b** ∉ Col A, proceed to the next step:
**Determine whether it is the case that **b** is orthogonal to Col A. Due to round-off errors in MATLAB computations, you will need to use `closetozeroroundoff` with p = 7 to check orthogonality. If **b** is orthogonal to Col A, you should just assign the values to the vectors **p** and **z** (no computations are needed!) and output a message `'b is orthogonal to Col A'`. After that, the program terminates.

**If **b** is not orthogonal to Col A, proceed to the next steps:
Find the solution of the ***normal equations*** (see the **Theory** above), a vector **x** (use the inverse matrix or the ***backslash operator***, \ ). Output the least squares solution, **x**, with a message:
`'the least squares solution of the system is'`
(display **x)**.
**Then, calculate and output a vector
`x1=A\b`
and check, by using the function `closetozeroroundoff` with p=12, whether the vectors **x** and **x1** match within the given precision. If yes, output a message: `'A\b returns the least-squares solution of an inconsistent system Ax=b'`
**Next, calculate the vectors **p** and **z** (see the **Theory** above).
**Check whether the vector **z** is, indeed, orthogonal to the Col A – the function `closetozeroroundoff` with p=7 should be used here. If your code confirms that, display a message: `'z is orthogonal to Col A. Great job!'` Otherwise, output a message like: `'Oops! Is there a bug in my code?'` and terminate the program.
**Finally, use the vector **z** to compute the distance `d` from **b** to Col A. Output `d` with the message:
`fprintf('the distance from b to Col A is %i',d)`
Hint: use a MATLAB built in function **`norm()`.**

**This is the end of the function `proj`.**

**Print the functions `closetozeroroundoff, shrink, proj` in your Live Script.
**Run the function `[p,z]=proj(A,b)`on the following choices of matrices A and vectors **b**. Notice that some of the matrices are created in several steps and some intermediate outputs have been suppressed.

```
%(a)
A=magic(4), b=sum(A,2)
%(b)
A=magic(4); A=A(:,1:3),b=(1:4)'
```

```
%(c)
A=magic(6), E=eye(6); b=E(:,6)
%(d)
A=magic(6), b=(1:5)'
%(e)
A=magic(5), b = rand(5,1)
%(f)
A=ones(4); A(:)=1:16, b=[1;0;1;0]
%(g)
B=ones(4); B(:)=1:16, A=null(B,'r'), b=ones(4,1)
```

%Analyze the outputs in parts (e) and write a comment that would explain a reason why a random vector **b** belongs to the Col A.

**BONUS!** (1 point)
%For part (g), based on the input matrix A and the outputs, state to which vector space the vector **b** has to belong.

**You will need to present some additional computation in your Live Script to support your responses. Supply them with the corresponding output messages and/or comments.

## The Exact and Least-Squares Solutions

Please review the section Theory below very carefully – you will be using these facts when working on Exercise 4.
**Theory**: An $m \times n$ matrix $U$ has orthonormal columns if and only if $U^T U = I_n$, where $I_n$ is an $n \times n$ identity matrix. If $U$ is a square matrix with orthonormal columns, it is called orthogonal.

We will work with the system A**x**=**b**, where the columns of A are linearly independent.

When solving a consistent system A**x**=**b** (that is, $\mathbf{b} \in$ Col A), the unique "exact" solution can be found by using the backslash operator: **x** =A**b**.

When solving a consistent system A**x**=**b**, where A is a matrix with orthonormal columns, we can find the unique solution **x** in two ways as indicated below:
(1) using the backslash operator, \ , that is, **x** =A**b**;
(2) using the Orthogonal Decomposition theorem (see Lecture 28).

When solving an inconsistent system A**x**=**b** (that is, $\mathbf{b} \notin$ ColA), we will be looking for the least-squares solution, which can be found either by solving the normal equations, $A^T A\mathbf{x} = A^T\mathbf{b}$, or by using the backslash operator, A**b** (see Exercise 3 of this Project).

Another way of calculating the least squares solution of an inconsistent system A**x**=**b** is to employ an orthonormal basis U for the Col A. If the columns of an $m \times n$ matrix U form an orthonormal basis for Col A, then, for any vector $\mathbf{b} \in \mathbb{R}^m$, we can find its projection $\hat{\mathbf{b}}$ onto the

Col A by the formula $\hat{\mathbf{b}} = UU^T\mathbf{b}$, and, then, calculate the least-squares solution $\hat{\mathbf{x}}$ as the "exact" solution of a consistent system $A\mathbf{x} = \hat{\mathbf{b}}$.
Using the projection of **b** onto the Col A, which could be represented either as $A\hat{\mathbf{x}}$ or as $\hat{\mathbf{b}}$, we can calculate the least-squares error of approximation of the vector **b** by the elements of the Col A as $norm(\mathbf{b} - A\hat{\mathbf{x}})$ or as $norm(\mathbf{b} - \hat{\mathbf{b}})$ (either of them is also the distance from **b** to Col A – see Exercise 3 of this Project).

**Exercise 4** (5 points) **Difficulty: Hard**
In this exercise, you will work with the matrix equation $A\mathbf{x} = \mathbf{b}$, where the columns of A are linearly independent. You will look for the unique "exact" solution, for a consistent equation ($\mathbf{b} \in$ Col A); and you will look for the unique least-squares solution, for an inconsistent equation ($\mathbf{b} \notin$ Col A).

**Create a function in MATLAB that begins with

```
function X=solvemore(A,b)
format compact
format long
[m,n]=size(A);
```

The inputs are an $m \times n$ matrix A, whose columns form a basis for Col A, and a vector $\mathbf{b} \in \mathbb{R}^m$. Some of the input matrices A will be created by running the function `shrink()` – it will leave only linearly independent columns. You should have already had the function `shrink()` created in a file in MATLAB, if not, please see the code in Exercise 3 of this Project.

**Next step is to determine if $\mathbf{b} \in$ Col A or $\mathbf{b} \notin$ Col A – please use a MATLAB function `rank()` – and you will employ a logical "if … else" statement to work with the two possible outcomes as indicated below.

**This is the beginning of "if … else" statement:**

## "If" b is in Col A

first, we output a message: `'The system is consistent - look for the exact solution'` and go through the following steps:
**Determine whether the matrix A has orthonormal columns and, if it is the case, determine if A is also orthogonal.
Hint: To conduct this test in MATLAB, you will need to use the function `closetozeroroundoff()` with p = 7.

**If A does not have orthonormal columns, output a corresponding message and find the unique solution `x1` of $A\mathbf{x} = \mathbf{b}$ using the backslash operator, \. In this case, we assign `X=x1` and terminate the program.

**If A does have orthonormal columns, output a corresponding message, and your code will continue with the following tasks:
First, it will check whether A is orthogonal. If yes, it outputs a message `'A is orthogonal'`.

If not, the output message should be `'A has orthonormal columns but is not orthogonal'`.
Next, output the solution of $A\mathbf{x} = \mathbf{b}$ in two ways (do not display the outputs `x1` and `x2` here):
(1) using the backslash operator, \ , denote this solution `x1`,
(2) using the Orthogonal Decomposition theorem (Lecture 28), denote this solution `x2`.
Hint: for part (2), you can either use a "for loop" to calculate the entries of the vector `x2`, one by one, or you can employ the transpose of A and output `x2` by using a MATLAB operation.

## else (b is not in Col A)

we display a message `'The system is inconsistent: look for the least-squares solution'` and, then, we will find the least-squares solution in both ways as indicated below:
**First way: we will find the unique least-squares solution directly by either solving the normal equations or using the backslash operator. We denote the solution `x1` and display it with a message that it is the least-squares solution of the system.
**Second way: we will use an orthonormal basis for Col A to find the least-squares solution. It might be possible that the matrix A has orthonormal columns and, if not, we will create an orthonormal basis for Col A.
Check if the matrix *A* has orthonormal columns. If yes, your outputs will be:

```
disp('A has orthonormal columns: orthonormal basis for Col A is U=A')
U=A
```

If the matrix *A* does not have orthonormal columns, find an orthonormal basis `U` for Col A using a MATLAB function `orth()`. Output and display `U` with a message

```
disp('A does not have orthonormal columns: orthonormal basis for Col A is')
U=orth(A)
```

Next, using the orthonormal basis `U` for Col A, calculate the projection of **b** onto Col A, denote it `b1`. Output and display vector `b1` with the corresponding message.
Next, find the least-squares solution `x2` as the "exact" solution of the equation A**x** = `b1` (you can use the backslash operator here). Output `x2` with the message: `'The least-squares solution using the projection b1 is'` (output and display `x2`).

**Use the least-squares solution `x1` to find a least-squares error of approximation of **b** by the elements of Col A, denote it `n1`.
Display `n1` with a message:

```
fprintf('Error of approximation of b by vector A*x1 of Col A is\n')
n1
```

**Now, we use the projection `b1` to find the least-squares error of approximation of **b** by the elements of Col A, denote it `n2`.
Display `n2` with a message:

```
fprintf('Error of approximation of b by vector b1 of Col A is\n')
n2
```

**Next, input a vector `x=rand(n,1)`. Compute an error `n3=norm(b-A*x)` of approximation of the vector **b** by a vector Ax of the Col A for a random vector `x`.
Output it with a message:

```
fprintf('error of approximation of b by A*x of Col A for random x is\n')
n3
```

**This is the end of your "if … else" statement**

## After completing all of the above

your code will proceed with composing the output `X` for the cases when we have two solutions `x1` and `x2` for both consistent and inconsistent systems, but, first, we will check if the corresponding entries of the two solutions `x1` and `x2` are sufficiently close to each other. We will run the function `closetozeroroundoff()` with p = 12 on the vector of the difference between `x1` and `x2`, and compare the output with the zero vector. If your code confirms that the corresponding entries of the solutions are in the range of 10^(-12) from each other, output a message `'solutions x1 and x2 are sufficiently close to each other'`, and assign (and display) `X=[x1,x2]`. Otherwise, the output message should be `'Check the code!'` and the empty matrix should be assigned to the output `X`.

****Print the functions `closetozeroroundoff, shrink, solvemore` in your Live Script.

**Run the function `X=solvemore(A,b)` on the following matrices. Please type (or copy and paste) the matrices and the vectors exactly as they appear below.

```
%(a)
A=magic(4); b=A(:,4), A=orth(A)
%(b)
A=magic(5); A=orth(A), b=rand(5,1)
%(c)
A=magic(6); A=shrink(A), b=ones(6,1)
%(d)
A=magic(6); A=shrink(A), b=rand(6,1)
%(e)
A=magic(4); A=orth(A), b=rand(4,1)
```

% Compare the outputs `n1` and `n2` and write a comment whether they are close to each other.
% Next, compare `n1` with `n3`. Write a comment whether the least-squares solution, `x1` (or `x2`) may, indeed, minimize the distance between a vector **b** and the vectors A**x** of the Col A **(**which has to be true according to the general theory).

## Part III. Application to Polynomials

### Applications to Linear Models - Regression Lines

In this part of the project, you will write codes that output the least-squares fit equation for the given data. The equation is determined by the **parameter vector c** (that has to be calculated), the **design matrix** X, and the **observation vector y,** such that, the 2-norm of the residual vector **e** = **y** – *X***c** is the minimum.

Assume that we are given the data points $(x_i, y_i)$, where $i = 1:m$. A choice of the equation of a curve which gives the least-squares fit to the data is, in general, arbitrary. First, we will take a look at the equation of a straight line $y = c_1 x + c_0$ that minimizes the sum of squares of the residuals – it is called the least-squares regression line. The parameter vector **c** can be determined by computing the least-squares solution of $X\mathbf{c} = \mathbf{y}$, where

$$X=\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_m & 1 \end{bmatrix},\ \mathbf{c}=\begin{bmatrix} c_1 \\ c_0 \end{bmatrix},\ \mathbf{y}=\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}.$$

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

**First, create the following function in MATLAB that will be used in the function below:

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

**Create another function in MATLAB that begins with the following lines:

```
function c=lstsqline(x,y)
hold off
format
format compact
x=x';
y=y';
a=x(1);
m=length(x);
b=x(m);
disp('the design matrix is')
X=[x,ones(m,1)]
disp('the parameter vector is')
c=lscov(X,y)
disp('the norm of the residual vector is')
N=norm(y-X*c)
plot(x,y,'*'),hold on
polyplot(a,b,c');
fprintf('the least-squares regression line is\n')
P=poly2sym(c)
```

Note: The data vectors **x** and **y** (inputs) are the row vectors – we use the transpose function in the code above to convert them into the column vectors. The parameter vector `p` in the function `polyplot(a,b,p)` has to be a row vector – in the code above it appears as `c'`.

**The MATLAB command `c=lscov(X,y)` calculates the parameter vector `c` which is, in fact, the least-squares solution of the inconsistent system $X\mathbf{c}=\mathbf{y}$. You will continue the function `lstsqline` with a verification of this fact: output (do not display) the least-squares solution `c1` of the system $X\mathbf{c}=\mathbf{y}$, calculated by any of the method from Exercise 4 of this Project, and, then, employ `closetozeroroundoff()` with p=7, to verify that `c` and `c1` match. If it is the case, output the message `'c is the least-squares solution'`
(Please make sure that you will receive this message after running the function)

**Complete your function **`lstsqline`** with the commands:

```
hold off
end
```

**Print the functions `polyplot` and `lstsqline` in your Live Script.
**Input the vectors:
```
x = [0,2,3,5,6], y = [1,4,3,4,5]
```
**Run the function
```
c=lstsqline(x,y);
```

Your outputs have to be: the design matrix `X`, the parameter vector `c`, the 2-norm of the residual vector `N`, the equation of the least-squares regression line `P`, the plot that contains both the data points and the line of the best fit, and the message confirming that `c` is, indeed, the least-squares solution.

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

In this Exercise you will find a polynomial of degree `n` of the best fit for the given data points.

**Create a new function in a file
```
function c=lstsqpoly(x,y,n)
```
which takes as inputs the data vectors `x` and `x` and a positive integer number `n`.

The function `lstsqpoly(x,y,n)` is a modification of the function `lstsqline(x,y)` from Exercise 5 in the way that it has to output the least-squares polynomial of degree *n*.

You will take the function `c=lstsqline(x,y)`and make the three changes in it as indicated below (everything else will stay the same):
**Add one more input variable `n` and re-name the function as `lstsqpoly`.
**Replace the matrix `X` with the new matrix, also named `X`, whose form depends on the degree *n* of the polynomial that we use to fit the data. For example, when `n=3`, the least-squares polynomial will be of degree 3, and the design matrix `X` will have a form:

$$X = \begin{bmatrix} x_1^3 & x_1^2 & x_1 & 1 \\ x_2^3 & x_2^2 & x_2 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ x_m^3 & x_m^2 & x_m & 1 \end{bmatrix}.$$

**Replace the output message for the polynomial `P` with a message
```
fprintf('the polynomial of degree %i of the best least-squares fit is\n',n)
```

Please make sure that you have the function `polyplot` in your Current Folder in MATLAB - it was used in the function `lstsqline` and it will be used in the function `lstsqpoly` as well.

**Print the functions `polyplot` and `lstsqpoly` in the Live Script.
**Run the function `c=lstsqpoly(x,y,n)` on the vectors from Exercise 5:
```
x = [0,2,3,5,6], y = [1,4,3,4,5]
```
for each n = 1, 2, 3, 4.

For each n, the outputs have to be: the design matrix `X`, the parameter vectors `c`, the 2-norm of the residual vector `N`, the equation of the least-squares polynomial `P`, the plot containing both

the data points and the graph of the polynomial `P`, and the message confirming that `c` is, indeed, the least-squares solution.

.

% Verify that your code in Exercise 6 for n = 1 is consistent with the code in Exercise 5, that is, the outputs of the functions `c=lstsqline(x,y)` and `c=lstsqpoly(x,y,1)` match. Write a comment about it.

**BONUS!** (1 point)

% Analyze the outputs for n = 4. Justify the fact that the least-squares polynomial of degree 4 is, actually interpolates the 5 data points.

## Part III. Application to Dynamical Systems

In this part of the Project, you will be working with application of eigenvalues and eigenvectors to the description of evolution of a discrete dynamical system.

**Exercise 7** (5 points) **Difficulty: Very Hard**

**Theory**: We will work with a 2x2 matrix A, whose entries are real numbers, and the related linear difference equation

$$\mathbf{x}_{k+1} = A\mathbf{x}_k \quad (k = 0,1,2,...) \qquad (1)$$

that describes evolution of a discrete dynamical system with an initial vector $\mathbf{x}_0$ in $\mathbb{R}^2$.

If A is diagonalizable, there exists an eigenvector basis $\{\mathbf{v}_1, \mathbf{v}_2\}$ for $\mathbb{R}^2$, and, for any initial vector $\mathbf{x}_0$ in $\mathbb{R}^2$, there is a unique vector of weights $C = (c_1, c_2)$, such that,

$$\mathbf{x}_0 = c_1\mathbf{v}_1 + c_2\mathbf{v}_2 . \qquad (2)$$

Moreover, the solution of the difference equation (1) is represented in an explicit form as

$$\mathbf{x}_k = c_1(\lambda_1)^k \mathbf{v}_1 + c_2(\lambda_2)^k \mathbf{v}_2 \quad (k = 0,1,2,...) \qquad (3)$$

where $\lambda_1$ and $\lambda_2$ are the eigenvalues of A corresponding to the eignevectors $\mathbf{v}_1$ and $\mathbf{v}_2$, respectively.

To represent the solution geometrically:

we will use equation (3) to calculate $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, ...$, when working with the matrices whose eigenvalues are positive numbers;

and we will use the recurrence relation (1) to find the consecutive $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, ...$, when working with the matrices whose eigenvalues are complex (non-real) numbers.

The graph of $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, ...$ is called the trajectory of the dynamical system.

We will accept the following description of the origin in relation to the patterns of the trajectories of a dynamical system:

I. All trajectories tend towards the origin. In this case, the origin is called an attractor, and it happens when both eigenvalues are less than 1 in magnitude. The direction of the greatest attraction is through the origin and the eigenvector corresponding to the eigenvalue of the smaller magnitude.

II. All solutions, except for the constant (zero) solution, are unbounded, and their trajectories tend away from the origin. In this case, the origin is called a repeller, and it happens when both eigenvalues are greater than 1 in magnitude. The direction of the greatest repulsion is through the origin and the eigenvector corresponding to the eigenvalue of the larger magnitude.

III. Some trajectories tend towards the origin and the others tend away from the origin. In this case, the origin is called a saddle point, and it happens when one of the eigenvalues is less than 1 in magnitude and the other one is greater than 1 in magnitude. The direction of the (greatest) attraction is through the origin and the eigenvector corresponding to the eigenvalue of the smaller magnitude; and the direction of the greatest repulsion is through the origin and the eigenvector corresponding to the eigenvalue of the larger magnitude.

For more details, please refer to Lecture 26 and Section 5.6 of the Textbook.

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

```
function []=trajectory(A,X0,N)
format
format compact
L=eig(A);
```

The inputs are: a 2x2 real-valued matrix A, a matrix `X0` whose columns are the initial vectors in $\mathbb{R}^2$, and the number `N` of the vectors $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, ..., \mathbf{x}_N$ which we will calculate.

**We will use a logical 'if … else" statement to break up the whole code into two parts:**

Part 1

**"if" matrix A has real eigenvalues**

first, we display a message `'the eigenvalues of A are real'` and proceed with the following tasks:

**Check if A has a zero eigenvalue. If it is the case, output a message `'A has a zero eigenvalue'` and terminate the program – we will not graph the solutions in this case.

**You will continue your code for `trajectory` by modifying the function `eigen`, which was created in Exercise 1 of this Project. The details how the function `eigen` should be modified to be included into the function `trajectory` are listed below – please remove/suppress all the tasks/outputs of the function `eigen` that are not on the list.

Important: Please notice that you have n=2 in your code and you also need to replace the command `null( )`, used in the function `eigen`, with `null( ,'r')` to output a "rational" basis for an eigenspace.

After adjusting the function `eigen` for your code, you will add some more commands that are listed below in these instructions.

The function `trajectory` has to contain the following parts coming from the function `eigen`:

**Output a row vector **L** of the sorted eigenvalues of A, where the multiple eigenvalues are all equal to each other: the entries of **L** have to be written in the ascending order and each eigenvalue repeats as many times as its multiplicity. Display L with a message:

```
fprintf('all sorted eigenvalues of A are\n')
L
```

**Output (do not display) the matrix P whose columns are bases for the eigenspaces corresponding to the sorted eigenvalues in L.
**Check if A is diagonalizable.
If A is not diagonalizable, output a message: `'A is not diagonalizable: there is no eigenvector basis for R^2'`
and terminate the code.
If A is diagonalizable, output the following:

```
fprintf('A is diagonalizable: there exists an eigenvector basis for R^2\n')
fprintf('it is formed by V1,V2 corresponding to sorted eigenvalues in L\n')
```

(assign to `V1` and `V2` the columns 1 and 2 of P, respectively, and display `V1` and `V2` here.)

**This is the end of the part of your code which relates to the function `eigen`.**

Continue your function `trajectory` with the following:
**Check if all eigenvalues of A are positive numbers. If at least one of the eigenvalues of A is negative, we terminate the code with a message `'A has a negative eigenvalue'`
**Next (for the matrices whose eigenvalues are positive numbers), we will determine whether the origin is an attractor, a repeller, or a saddle point. (See the **Theory** above.) Program an output message for each of the three cases and also output a message that will state the direction of the greatest attraction/repulsion.
An example of the output messages when the origin as an attractor, is below:

```
disp('the origin is an attractor')
fprintf('a direction of greatest attraction is through 0 and\n')
V1
```

(Remember, `V1` and `V2` are the vectors in an eigenvector basis for $\mathbb{R}^2$ corresponding to the positive eigenvalues of A sorted in the ascending order.)
**Then, we will check if at least one of the eigenvalues of A is the integer number 1. If it is the case, we display a message `'A has an eigenvalue 1'` and go to the next step (do not terminate the code here).

Next, we will create plots of the trajectories of a dynamical system.
**First, you will introduce a new matrix of the initial vectors, which is a horizontal concatenation of the vectors `V1,V2,-V1,-V2` with the input matrix `X0`. Type in your code:

```
X0 = [V1,V2,-V1,-V2,X0];
```

**Next, type the following lines in your code:

```
n=size(X0,2);
X=zeros(2,N+1);
```

The first line counts the total number of the initial vectors, and the second line stores a matrix X, whose entries will be re-calculated by using a double "for loop" as indicated below:

**For each initial vector `X0(:,i)`, where `i=1:n`, calculate the coefficient vector C according to the formula (2). Use the vector C to calculate by formula (3) the N+1 vectors $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N$ and output them as N+1 columns of the matrix `X`. Proceed with graphing the trajectories.
Type (see also Exercise 5 of Project 2):

```
x=X(1,:);y=X(2,:);
plot(x,y,'*'), hold on
plot(x,y)
```

and program the viewing window as
`v=[-1 1 -1 1];`
for the two cases: when the origin is an attractor and when A has an eigenvalue 1;
and program the viewing window as
`v=[-5 5 -5 5];`
for all other cases.

After that, type:
`axis(v)`
**This is the end of the double "for loop".**

**This is the end of the "IF" part of your code - proceed to the "ELSE" part.**

Part 2

**else** (**matrix A has complex conjugate (non-real) eigenvalues**)
first, we output a message
```
disp('the eigenvalues of A are complex conjugate (non-real) numbers')
```
**Next, we will graph the trajectories by calculating consecutive iterations $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, ..., \mathbf{x}_N$, for each initial vector $\mathbf{x}_0$ (a column of the input `X0`) by using the recurrence equation (1) (see the **Theory**).
Begin this part with the commands:
```
L=eig(A)
magn=abs(L)
n=size(X0,2);
X=zeros(2,N+1);
```
Entries of the vector `magn` are the magnitudes (moduli) of the complex eigenvalues in L.

**Then, use a double "for loop" to graph the trajectory:
for each initial vector `X0(:,i)`, where `i=1:n,` calculate the consecutive iterations $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N$ by formula (1), and output them as the N+1 columns of `X`.
Plot the trajectory by the following lines:
```
x=X(1,:);y=X(2,:);
plot(x,y,'*'), hold on
plot(x,y)
```
**This is the end of a double "for loop".**

**This is the end of the "ELSE" part of your "if … else" statement.**

**The <u>last commands</u> in your function `trajectory` are:
```
hold off
end
```

**Print the function `trajectory` in your Live Script.
**Run the function `trajectory(A,X0,N)` on the following sets of variables.
```
%(a)
A=[2 0;0 .5]
X0=[[.1;5],[-.1;5],[-.1;-5],[.1;-5]];
N=10;
```

```
%(b)
A=[2 0; 0 3]
X0=[[0;0],[1;1],[1;-1],[-1;-1],[-1;1],[1;.1],[-1;.1],[-1;-.1],[1;-.1]];
N=10;
A=[.80 0;0 .64]
X0=[[1;1],[-1;1],[-1;-1],[1;-1],[.5;1],[-.5;1],[-.5;-1],[.5;-1]];
N=10;
%(c)
A=[.80 0;0 .64]
X0=[[1;1],[-1;1],[-1;-1],[1;-1],[.5;1],[-.5;1],[-.5;-1],[.5;-1]];
N=10;
%(d)
A=[.64 0;0 .64]
X0=[[1;1],[-1;1],[-1;-1],[1;-1],[.5;1],[-.5;1],[-.5;-1],[.5;-1]];
N=10;
%(e)
A=[5 0; 1 5]
X0=[[1;1],[-1;1],[-1;-1],[1;-1]];
N=10;
%(f)
A=[1 0;0 .64]
X0=[[.5;1],[-.5;1],[-.5;-1],[.5;-1]];
N=10;
%(g)
A=[.90 .04;.10 .96]
X0=[[.2;.8],[.1;.9],[.9;.1],[.6;.4],[.5;.5]];
N=10;
%(h)
A=[0 -1;1 0]
X0=[[.1;.1],[.5;.5],[.8;0],[1;-1]];
N=10;
%(i)
A=[.5 -.6;.75 1.1]
X0=[[.1;.1],[.5;.5],[-1;-1],[1;-1]];
N=10;
%(j)
A=[.8 .5; -.1 1.0]
X0=[[1;0],[0;-1]];
N=100;
%(k)
A=[1.01 -1.02;1.02 1.01]
X0=[[1;0],[0;-1]];
N=10;
%(l)
A=[.3 .4;-.3 1.1]
X0=[[0;.5],[1;1],[-1;-1],[0;-.5],[-1;-.8],[1;.8],[-.5;.5],[.5;-.5]];
N=10;
%(m)
A=[.5 .6; -.3 1.4]
X0=[[.1;2],[4;0],[-4;0],[-.1;-2]];
N=30;
```

```
%(n)
A=[.8 .3; -.4 1.5]
X0=[[0;1],[-1;0],[0;-1],[1;0],[0;0],[3;0],[-3;0]];
N=50;
%(p)
A=[[1 2;2 4]]
X0=[];
N=10;
%(q)
A=[-.64 0;0 1.36]
X0=[];
N=10;
```

**BONUS!** (1 point)
%Analyze the outputs for the choice (d) and use equation (3) to explain why all trajectories are straight lines.
%Explain the pattern of behavior of the solutions for part (f).

**BONUS!** (1 point)
In part (g), we have a regular stochastic matrix A. Slightly modify your function `trajectory` and the vector `X0` to output additionally the unique steady-state vector `q` that describes the evolution of the given Markov Chain. Save your new function in MATLAB as `trajectory_1`
**Print the function `trajectory_1`
**Run `trajectory_1(A,X0,N)` with `N=100` on the stochastic matrix A given in (g) and your own input `X0`. Output and display the steady-state vector `q`.

**BONUS!** (1 point)
%Analyze the outputs for choices (h),(i),(j), and (k), where the eigenvalues are complex conjugates. Use the equation (3) and the output for `magn` to explain the pattern of the trajectories for each system.

## You are done with the Projects!!!