MATLAB PROJECT 4

Please read the Instructions located on the Assignments page prior to working on the Project.
BEGIN with creating a Live Script **Project4.**
Note: All exercises in this project have to be completed in the Live Script using the Live Editor. Please refer to the MATLAB video that explains how to use the Live Script: https://www.mathworks.com/videos/using-the-live-editor-117940.html?s_tid=srchtitle
The final script has to be generated by exporting the Live Script to PDF.
Each exercise has to begin with the line
Exercise#
You should also mark down the parts such as (a), (b), (c), and etc. This makes grading easier.
Important: we use the default `format short` for the numbers in all exercises unless it is specified otherwise. We do not employ `format rat` since it may cause problems with running the codes and displaying matrices in the Live Script. If `format long` had been used in an exercise, please make sure to return to the default `format` in the next exercise.

Part I. Eigenvalues, Eigenvectors & Diagonalization

Exercise 1 (5 points) **Difficulty: Hard**
In this exercise, we will work with the eigenvalues and eigenspaces of an $n \times n$ matrix A and construct a diagonalization of A where possible. First, we will output all eigenvalues of A. Then, we will consider distinct eigenvalues and find orthonormal bases for the corresponding eigenspaces. Next, we will check if A is diagonalizable by applying the general theory. If A is diagonalizable, we output an invertible matrix P and a diagonal matrix D, such that, $A = PDP^{-1}$, or, equivalently, AP = PD and P is invertible.
For help with this exercise, please refer to Module 24 of the Lecture Notes.
**Create a function in MATLAB that begins with

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

The input A is an *n* x *n* matrix. The output `L` is the vector of the eigenvalues of A which we will obtain after a sequence of commands outlined in Part 1. The outputs P and D will be matrices that form a diagonalization of A, or P and D will stay empty if A is not diagonalizable.

Include in your function `[L,P,D]=eigendiag(A)` all the commands outlined below.

Part 1. Output the vector L of all eigenvalues of A.

To do that, you will go through the steps listed below: suppress all intermediate outputs and display only the final output vector `L`.

**Output a column vector `L` of all eigenvalues, where each eigenvalue repeats as many times as its multiplicity. Use a basic MATLAB command for this part as below:

```
L=eig(A);
```

**Then, sort the entries of `L` in the ascending order, first, with respect to the real part, then, with respect to the imaginary part by using the command:

```
L=sort(L,'ascend','ComparisonMethod','real');
```

** In your code, you will need to ensure that all "repeating" eigenvalues (they correspond to a multiple eigenvalue) show as the same number. Use the function `closetozeroroundoff` with p = 7 to compare the eigenvalues and assign the same value to the ones that are within 10^(-7) margin from each other. To perform this task, you can go through the "sorted" entries of the vector `L` and, if there is a set of entries that are within 10^(-7) from each other, assign the first entry that comes across to all others in the set.
Important Note: we work here with the eigenvalues that are stored in MATLAB – do not round off any eigenvalues on this step.

**We work with complex eigenvalues in general. However, we want to ensure that the complex eigenvalues with negligibly small imaginary parts will show as real numbers. To do that, run the function `closetozeroroundoff` with p = 7 on the difference between each entry of L and its real part, and, if the function above outputs a 0, assign to that entry its real part (you can employ the MATLAB function `real()` here).

**If A is a singular (not invertible) matrix, it has a zero eigenvalue; however, when running your code, a zero eigenvalue may not show as a 0 on MATLAB.
If A is not invertible, run the function `closetozeroroundoff` with p = 12 on the vector of the eigenvalues `L` to ensure that the zero eigenvalues show as zeros, and assign the output of this function to `L` again. To check if a matrix A is not invertible, use a MATLAB command `rank`.

**After performing all of the tasks in this part, display your final output `L` as below:

```
disp('all eigenvalues of A are')
disp(L)
```

Part 2. **Construct an orthonormal basis for each eigenspace**.

**First, output a column vector `M` of the distinct eigenvalues of A (no repetitions are allowed) which you will obtain from the vector `L`. Also output a vector *m* of the multiplicities of the eigenvalues in `M`. You can use here the MATLAB command

```
[m,M]=groupcounts(L);
```

(suppress the outputs).

**Then, display the output `M` with the message as below:

```
disp('the distinct eigenvalues of A are')
display(M)
```

and assign:

```
flag=0;
```

For each distinct eigenvalue `M(k)` (`k=1:length(M)`), your function has to do the following:

**First, display:
```
lambda=M(k)
multiplicity=m(k)
```
and, then, proceed with constructing an orthonormal basis **W** for the eigenspace for `M(k)`.
Note: to output an orthonormal basis for a null space, use a MATLAB command `null()`.
Display W with the message:
```
disp('a basis for the eigenspace for this lambda is')
disp(W)
```

**Output (do not display) the dimension of the eigenspace W, the number `d(k)`.
Next, check if the dimension of the eigenspace `d(k)` is less than the multiplicity of the `lambda`, which is `m(k)`. If so, output a message:
```
disp('dimension of the eigenspace is less than multiplicity of lambda')
```
and assign
```
flag=1;
```

Part 3. **Construct a Diagonalization when possible**.

**If, for at least one `lambda`, the dimension of the eigenspace is less than the multiplicity of that `lambda` (that is, `flag==1`), matrix A is not diagonalizable. If this is the case, display a message
```
disp('A is not diagonalizable')
```
and terminate the code. The outputs for P and D have to be empty.

Otherwise, matrix A is diagonalizable and we continue with constructing a diagonalization:
**First, output matrix P whose *n* columns are formed by combining the constructed bases for the eigenspaces for each distinct `lambda` (do not display P here).

**Next, output (do not display) the diagonal matrix D whose diagonal entries are the eigenvalues corresponding to the columns of P, where each eigenvalue repeats as many times as its multiplicity (you can use the vector `L` here and the MATLAB function `diag()`).

**Then, set up a conditional statement that would verify that your diagonalization is constructed correctly, that is, the two conditions hold: AP = PD and P is invertible.
If both conditions hold, output:
```
disp('A is diagonalized')
display(P)
display(D)
```
Otherwise, output a message: `'Oops! I got a bug in my code!'`, assign
```
P=[];
D=[];
```
and terminate the code.
Note: To verify that AP = PD, use the condition `~any(closetozeroroundoff(A*P-P*D,7),"all")`. To verify that P is invertible, use the command `rank()`.

This is the end of the function `eigendiag`.

BONUS! (1 point)
Theory: An $n \times n$ matrix A is said to be orthogonally diagonalizable if there exists an orthogonal matrix P ($P^{-1} = P^T$, or equivalently, P is an $n \times n$ matrix and $P^T P = I_n$) and a diagonal matrix D, such that, $A = PDP^{-1}$, or equivalently, $A = PDP^T$.
Theorem: An $n \times n$ matrix A is orthogonally diagonalizable if and only if A is symmetric.
(For more information please read Section 7.1 of the textbook.)

If your diagonalization is confirmed, you can proceed with the coding of the bonus part. To get the bonus, add a set of commands indicated below at the end of your function **`eigendiag`** (mark this addition to your function as `%Bonus`).
**First, check if A is symmetric. If so, output a message
`disp('matrix A is symmetric')`
**Then, verify the orthogonal diagonalization: check if the matrix P is orthogonal by running the function `closetozeroroundoff` with p = 7 on the matrix `P'*P-eye(n)`.
If P is orthogonal, output a message:
`disp('the orthogonal diagonalization is confirmed')`
otherwise, output something like:
`disp('Wow! A symmetric matrix is not orthogonally diagonalizable?!')`

This is the end of your function `eigendiag` with the `%Bonus` part if included.

**Print the functions `eigendiag`, `closetozeroroundoff`, `jord` in the Live Script.
Note: a code for the function `jord` was given in Exercise 5 of Project 1.

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

```
%(a)
A=[3 3; 0 3]
[L,P,D]=eigendiag(A);
%(b)
A=[2 4 3;-4 -6 -3;3 3 1]
[L,P,D]=eigendiag(A);
%(c)
A=[4 0 1 0; 0 4 0 1; 1 0 4 0; 0 1 0 4]
[L,P,D]=eigendiag(A);
%(d)
A=jord(4,3)
[L,P,D]=eigendiag(A);
%(e)
A=ones(4)
[L,P,D]=eigendiag(A);
%(f)
A=[4 1 3 1;1 4 1 3;3 1 4 1;1 3 1 4]
[L,P,D]=eigendiag(A);
%(g)
A=[3 1 1;1 3 1;1 1 3]
[L,P,D]=eigendiag(A);
%(h)
A=magic(4)
[L,P,D]=eigendiag(A);
```

```
%(i)
A=magic(5)
[L,P,D]=eigendiag(A);
%(j)
A=pascal(4)
[L,P,D]=eigendiag(A);
%(k)
A=[0 0 .33;.18 0 0;0 .71 .94]
[L,P,D]=eigendiag(A);
%(l)
A=[0 -1;1 0]
[L,P,D]=eigendiag(A);
%(m)
A=[0 0 .33;.3 0 0;0 .71 .94]
[L,P,D]=eigendiag(A);
```

Part II. QR Factorization & Orthonormal Basis

According to the theory, we can create an orthonormal basis for the column space of a matrix whose columns are linearly independent by using the **Gram-Schmidt orthogonalization**. However, a direct computer realization of the Gram-Schmidt process involves round-off errors that build up when the vectors are calculated, one by one. In this exercise, we will program a Modified Gram-Schmidt process that involves normalization.

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

Theory: Let $A = [\mathbf{a}_1 \ \dots \ \mathbf{a}_n]$ be an *m* x *n* matrix whose columns are linearly independent and let $U = \{\mathbf{u}_1, \dots, \mathbf{u}_n\}$ be an orthonormal basis constructed by the Gram-Schmidt orthogonalization along with normalization. According to the Decomposition Theorem (Lecture 28), the vectors $\mathbf{a}_1, \dots, \mathbf{a}_n$ can be uniquely represented through the basis U as below: (Notice that $\mathbf{a_k} \cdot \mathbf{u_{k+1}} = \dots = \mathbf{a_k} \cdot \mathbf{u_n} = \mathbf{0}$ for $1 \le \mathrm{k} \le \mathrm{n} - 1$.)

$$\mathbf{a}_1 = (\mathbf{a}_1 \cdot \mathbf{u}_1)\mathbf{u}_1$$
$$\mathbf{a}_2 = (\mathbf{a}_2 \cdot \mathbf{u}_1)\mathbf{u}_1 + (\mathbf{a}_2 \cdot \mathbf{u}_2)\mathbf{u}_2$$
$$\vdots$$
$$\mathbf{a}_n = (\mathbf{a}_n \cdot \mathbf{u}_1)\mathbf{u}_1 + (\mathbf{a}_n \cdot \mathbf{u}_2)\mathbf{u}_2 + \cdots + (\mathbf{a}_n \cdot \mathbf{u}_n)\mathbf{u}_n$$

Or, equivalently,

$$A = [\mathbf{a_1} \ \mathbf{a_2} \ \dots \ \mathbf{a_n}] = [\mathbf{u}_1 \ \mathbf{u_2} \ \dots \ \mathbf{u_n}] \begin{bmatrix} \mathbf{a_1} \cdot \mathbf{u_1} & \mathbf{a}_2 \cdot \mathbf{u_1} & \cdots & \mathbf{a}_n \cdot \mathbf{u_1} \\ 0 & \mathbf{a}_2 \cdot \mathbf{u}_2 & \cdots & \mathbf{a}_n \cdot \mathbf{u}_2 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & & \mathbf{a}_n \cdot \mathbf{u}_n \end{bmatrix} = UV \qquad (1)$$

NOTE: We will perform normalization of each vector in the orthogonal basis while running the Gram-Schmidt process and choose a unit vector in the same (vs. opposite) direction as the vector in the basis. In this case, the diagonal entries of the matrix V, $\mathbf{a}_k \cdot \mathbf{u}_k$ $(\mathrm{k} = 1{:}\,\mathrm{n})$, will be positive.

The matrix U in (1) is an *m* x *n* matrix with orthonormal columns that form a basis for Col A and we will obtain it by using Gram-Schmidt process. The matrix V is an *n* x *n* upper-triangular invertible matrix that can be calculated by the formula:

$$V = U^T A \qquad (2)$$

Indeed,

$$UV = A \;\Rightarrow\; U^T UV = U^T A \text{ and } U^T U = I \;\Rightarrow\; V = U^T A$$

For more details on the method, please refer to Lecture 30 and Section 6.4 of the textbook.

**Create a function that begins with the following lines:

```
function [U,V,tolerance]=qrschmidt(A)
format
[~,n]=size(A);
U=[];
V=[];
tolerance=[];
```

The input is a matrix A with *n* columns. The outputs are either empty if the columns of A are linearly dependent, or U is a matrix whose columns form an orthonormal basis for the Col A generated by the Gram-Schmidt process, V is an *n* x *n* invertible upper-triangular matrix calculated by the formula (2), and `tolerance` is the accuracy of the constructed orthonormal basis U. The matrices U and V define an "economy-size" QR factorization of A, A=UV (following MATLAB terminology).

Continue your function as follows:
**The Gram-Schmidt orthogonalization requires for the columns of A to be linearly independent. If it is not the case, orthogonalization cannot be performed, and we will terminate the code with a message as below (the empty outputs for the variables will stay):

```
disp('columns of A are linear dependent and cannot be orthogonalized')
```

(Use the function `rank()` to check on linear dependence.)

If the columns of A do form a basis for Col A, we proceed with orthogonalization by Gram-Schmidt along with normalization.
**Assign initially

```
U=A;
```

Then, follow the Gram-Schmidt process and be sure to have the following presented in your code:
1) To the first column of the output U, we assign the normalized first column of A.
2) A consecutive column (in the range `2:n`) is calculated by the formulas in terms of the orthogonal projections on subspaces presented in the Theorem in Module 30, Part II, Page 195 of the Lecture Notes.
3) The projections are calculated by means of the formulas involving the product of a matrix and its transposed presented in the Theorem in Module 30, Part I, Page 192 of the Lecture Notes.
4) Each consecutive column (in the range `2:n`) is being normalized after it is calculated.

Note: To normalize a column, please see the definition of normalization in Lecture 27 and use MATLAB function `norm()`, which is the 2-norm.

After generating an orthonormal basis for Col A, which we will also call U, you will verify that it is constructed correctly: first, check if U is a basis for Col A and, second, check if it is orthonormal with a minimum tolerance (which will be assigned).

**To verify the first condition, you can use the MATLAB function `rank()` and the matrix A. Be careful with this check: since U contains the same number of columns as the matrix A, *n*, which is also the dimension of the Col A, to show that U is a basis for Col A (by using the Basis Theorem) we need to verify that U spans Col A, that is, the columns of A are in Col U. If U does not pass this test, output a message:

```
disp('U is not a basis for Col A? Check the code!')
```

(Do not terminate the code here.)

**To check the second condition, we will refer to the theory: if an *m* x *n* matrix U has an orthonormal set of columns, then `U'*U=eye(n)`. Due to round-off errors in MATLAB computations, this condition will be satisfied with some tolerance. Initially, you will check if the constructed basis satisfies the minimum tolerance requirement (for inputs in this exercise), which we set as following: the function `closetozeroroundoff(U'*U-eye(n),0)` has to return the zero matrix. If this condition does not hold, output a message:

```
disp('no orthonormalization? Check the code!')
```

**If at least one of the two conditions above is not satisfied, assign:

```
U=[];
```

and terminate the code.
Otherwise, we will have a nonempty output U which is an orthonormal basis for the Col A.

**Finally, output the actual tolerance that we will define as below:

```
tolerance=norm(U'*U-eye(n),1);
```

and the matrix

```
V=U'*A;
```

This is the end of your function `qrschmidt`.

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

```
%(a)
I=eye(3);M=magic(3);I(:,3)=M(:,1);
A=I
[U,V,tolerance]=qrschmidt(A)
%(b)
A=[1 2 -1 4 1;0 1 3 -2 2;0 0 0 2 -2]
[U,V,tolerance]=qrschmidt(A)
%(c)
A=magic(4)
[U,V,tolerance]=qrschmidt(A)
%(d)
A=magic(5)
[U,V,tolerance]=qrschmidt(A)
%(e)
A=magic(4);A=orth(A)
[U,V,tolerance]=qrschmidt(A)
```

```
%(f)
A=randi(10,6,4)
[U,V,tolerance]=qrschmidt(A)
%(g)
A=hilb(5)
[U,V,tolerance]=qrschmidt(A)
%(i)
A=pascal(5)
[U,V,tolerance]=qrschmidt(A)
```

Exercise 3 (6 points) **Difficulty: Hard**

In this exercise, we will use various approaches to QR factorization of an *m* x *n* matrix A.

Part 1.

In this part, we will generate the "full-size" orthogonal-triangular decomposition of an *m* x *n* matrix A by using Givens rotations. Referring to Exercise 2 of Project 1, we can employ Givens rotations to create an upper-triangular matrix from a matrix A preserving the norms of the columns of A. Here, we will use this approach and work with an *m* x *n* matrix A to generate an *m* x *n* upper-triangular matrix r along with an *m* x *m* orthogonal matrix q, which will be, in fact, the standard matrix of the composition of Givens rotations. The matrices q and r will define the "full-size" QR factorization of a matrix A, A=qr.

An explanation of the method is below:

In our code for Exercise 2 of Project 1, we initialized

```
R=A;
```

and, then, calculated a rotation matrix `G` and used it in the recurrence formula for R as

```
R=G'*R;
```

Suppose we arrived at an upper triangular matrix R after p rotations. If we denote a rotation matrix as $G_i(i = 1{:}\,p)$, then

$$R = \left(G_p^T \ldots \ G_1^T\right)A \qquad (1)$$

Notice that each $G_i(i = 1{:}\,p)$ is an orthogonal matrix, that is, $G_i^T = G_i^{-1}$. Left-multiplying both sides of the equation (1) consecutively by $G_p, \ \ldots \ , G_1$ and using $G_i G_i^T = I$, we have

$$A = \left(G_1 \ldots G_p\right)R = QR \qquad (2)$$

Here, we denoted

$$Q = G_1 \ldots G_p \qquad (3)$$

and obtained a factorization of the matrix A:

$$A = QR \qquad (4)$$

where Q is an orthogonal matrix (as a product of orthogonal matrices) and R is an upper triangular matrix.

**Create a new function in MATLAB that begins with:

```
function [q,r] = qrgivens(A)
format
[m,n]=size(A);
r=A;
q=eye(m);
k=min(m,n);
```

**Continue your function with copying and pasting into your code the "for" loop from the function `uppertrian` (created in Exercise 2 of Project 1) which generates an upper triangular matrix R by means of Givens rotations, and, then, modifying it in the following way:

I. Replace R with r.

II. Output the matrix `q` (it is the matrix Q in formula (3)) by accumulating right products of the matrices $G_i(i = 1{:}p)$, that is, the following command has to be placed into the inner loop after outputting the matrix `G`:

```
q=q*G;
```

Note: the initial iteration for `q` has been assigned previously as `q=eye(m);`.

After the loop is completed, you will have the output matrices `q` and `r`. Do not display them.

**Place the command below into your code that will assign zeros to the small by magnitude entries of `r`:

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

Next, we will proceed with a verification that the decomposition A=qr is constructed correctly.

Note: You need to code the output messages only for the cases when there is a problem with the outputs - do not terminate the code after any check.

Proceed as follows:

Assign initially,

```
test=0;
```

and go through all checks below.

**First, we will verify that `r` is, indeed, an upper triangular matrix. If not, output:

```
disp('r is not upper-triangular?!')
test=1;
```

The MATLAB command `~istriu(r)` can be used here.

**Next, we will verify that `q` is an orthogonal matrix by employing the function `closetozeroroundoff(q'*q-eye(m),7)`. If your output does not pass this test, output:

```
disp('q is not orthogonal?!')
test=1;
```

**Finally, we check if we have a QR factorization, that is, $A = qr$, by using the function `closetozeroroundoff(A-q*r,7)`. If not, output:

```
disp('QR factorization is not working?!')
test=1;
```

For the two last checks, you can employ the logical command `any( ,"all")`.

**After all the conditions have been checked, place the following into your code:

```
if test
    r=[];
    q=[];
end
```

Be sure to correct your code is any empty outputs are received after you run your function.

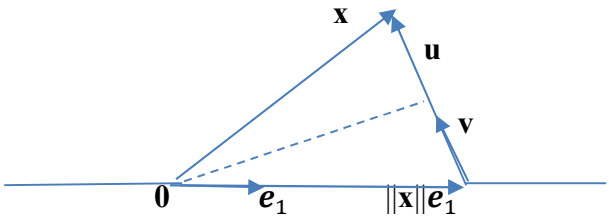
This is the end of the function `qrgivens`.

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

```
%(a)
A=ones(2)
[q,r] = qrgivens(A)
%(b)
A=magic(3)
[q,r] = qrgivens(A)
%(c)
A=magic(4)
[q,r] = qrgivens(A)
%(d)
A=[magic(3),ones(3,2)]
[q,r] = qrgivens(A)
%(e)
A=[magic(3);ones(2,3)]
[q,r] = qrgivens(A)
%(f)
A=triu(magic(5))
[q,r] = qrgivens(A)
%(g)
A=[magic(3);hilb(3)]
[q,r] = qrgivens(A)
%(h)
A=[1 1 2 0;0 0 1 3;0 0 2 4;0 0 3 5]
[q,r] = qrgivens(A)
%(i)
A=pascal(4)
[q,r] = qrgivens(A)
```

Part 2

In this part, we will use modified Householder reflections to construct a QR factorization of an $m \times n$ matrix. For some details on the method, please refer to https://en.wikipedia.org/wiki/QR_decomposition and to the documentation on MATLAB website.



Description of the Algorithm:

We will be working here with an $m \times n$ matrix A and use modified Householder reflections to decompose A into a product of an orthogonal $m \times m$ matrix q and an upper triangular $m \times n$ matrix r with non-negative entries on the main diagonal, such that, $A = qr$. We will refer to this factorization as a "full-size" decomposition.

Let $I_m = [e_1 \ e_2 \ \dots \ e_m]$ be an $m \times m$ identity matrix. We will begin with the first column of A, which we denote here as a vector **x**. If **x** is a non-zero vector, we will reflect it across the line that is the bisector of the angle between the vectors **x** and e_1 (see the chart above). Let Q be the standard matrix of this transformation.

Note: our method is slightly different from the one described in Wikipedia, that is why we call it a "modified" Householder reflection.

In our algorithm, we will consider two matrices *r* and *R*, where *R* is a submatrix of *r*. They will be initialized as *r* = A and *R* = *r*. The next iteration for *r* will be obtained by left-multiplying *r* by Q. This will convert *r* into a matrix whose column 1 is the vector $\|\mathbf{x}\|\boldsymbol{e}_1$, that is, the first column of *r* has entry $\|\mathbf{x}\|$ on the top and all the entries below are zeros. Then, we will consider a submatrix *R* of *r* obtained by removing row 1 and column 1 from *r,* and we will generate another matrix Q, such that, left-multiplication of *r* by Q converts the first column of the submatrix R into the norm of the column multiple of $\boldsymbol{e}_1$, where $\boldsymbol{e}_1$ is the first column of the identity matrix $I_{\boldsymbol{m-1}}$, without changing the first column of *r*. If the first column of a submatrix R is a zero column, it will stay unchanged. The next submatrix R will be obtained by removing rows 1,2 and columns 1,2 from *r* and we will create our next matrix Q in the way outlined above.
Continuing this way, we will generate a sequence of matrices Q, which we denote here as $Q_1, Q_2, \ldots$, and, after p steps, we will arrive at an upper-triangular matrix *r* with non-negative entries on the main diagonal, such that

$$r = (Q_p \ldots Q_1)A \qquad (1)$$

Some notes on the matrices Q_j $(j = 1{:}\,p)$:
In the Algorithm above, **x** is the column 1 of the initial iteration *R* = *r*, where *r* = A. Let $I = [\boldsymbol{e}_1 \ \boldsymbol{e}_2 \ \ldots \ \boldsymbol{e}_m]$ and let Q be the matrix of the transformation described in the Algorithm. Then,

$$Q\mathbf{x} = \|\mathbf{x}\|\boldsymbol{e}_1$$

and $\mathbf{u} = \mathbf{x} - \|\mathbf{x}\|\boldsymbol{e}_1$ (see the chart on the previous page). Thus, $Q\mathbf{x} = \mathbf{x} - \mathbf{u}$.
Let **v** be the unit vector in the direction of a non-zero vector **u**, that is,

$$\mathbf{v} = \mathbf{u}/\|\mathbf{u}\| \qquad (2)$$

Notice that $\|\mathbf{u}\|/2$ is the magnitude of the orthogonal projection of **x** onto **u** (or onto **v**). Thus,

$$\frac{\|\mathbf{u}\|}{2} = \|\mathbf{x}\|\cos(\mathbf{x},\mathbf{v}).$$

Then,

$$\mathbf{u} = \|\mathbf{u}\|\mathbf{v} = 2\mathbf{v}\frac{\|\mathbf{u}\|}{2} = 2\mathbf{v}\|\mathbf{x}\|\cos(\mathbf{x},\mathbf{v}) = 2\mathbf{v}(\mathbf{v}\cdot\mathbf{x}) = 2\mathbf{v}(\mathbf{v}^T\mathbf{x}) = 2(\mathbf{v}\mathbf{v}^T)\mathbf{x}$$

Thus,

$$Q\mathbf{x} = \mathbf{x} - \mathbf{u} = I\mathbf{x} - 2(\mathbf{v}\mathbf{v}^T)\mathbf{x} = (I - 2\mathbf{v}\mathbf{v}^T)\mathbf{x}$$

Therefore,

$$Q = I - 2\mathbf{v}\mathbf{v}^T \qquad (3)$$

The formula (3) can be used in the code for each Q_j, however, it has to be adjusted with each iteration since the sizes of the submatrices *R* are decreasing with each iteration and so is the sizes of the identity matrices *I*. Also, after calculating Q_j by the formula (4), we will need to expend it to a size of an $m \times m$ matrix which we will perform by embedding it into an $m \times m$ identity matrix as a lower right block. If Q_j is adjusted this way, left multiplication of *r* by Q_j will only modify the corresponding submatrix *R* of *r* in the way described in the Algorithm.

Moreover, each matrix Q_j $(j = 1{:}\,p)$ is orthogonal and symmetric, that is, $Q_j^{-1} = Q_j^T = Q$, therefore, from (1) it follows that

$$A = (Q_p \dots Q_1)^{-1} r = \left(Q_1^{-1} \dots Q_p^{-1}\right) r = (Q_1 \dots Q_p) r = qr,$$

where

$$q = Q_1 \dots Q_p \qquad (4)$$

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

```
function [q,r] = hreflections(A)
format
[m,n]=size(A);
q=eye(m);
Q=eye(m);
r=A;
k=min(m,n);
```

The matrices `q` and `r` were described in the Algorithm. The matrix Q, which is initialized as `Q=eye(m)`, is the one that will be computed by the formula (3), expanded to the size of an $m \times m$ matrix, and then used to modify the matrix *r*.

Continue your function with a **"for" loop** that will run through the columns of the matrix *r* from 1 to k (`i=1:k`).

**First, place the command below that generates the submatrices:

```
R=r(i:end,i:end);
```

**Next, check if the first column of `R` is a zero vector. If it is the case, pass the control to the next iteration skipping all remaining commands in the body of the "for" loop - a MATLAB command `continue` can be used here.
If the first column of `R` is a non-zero vector, follow the Description of the Algorithm and, first, output the identity matrix `I`, whose size is adjusted to the iteration (for `i=1` matrix `I=eye(m)` and then the size of its rows/columns drops by 1 with each iteration), and, next output:

```
colnorm=norm(R(:,1));
u=R(:,1)-colnorm*I(:,1);
u=closetozeroroundoff(u,7);
```

If the output **u is a zero vector, pass the control to the next iteration of *i*. If **u** is a non-zero vector, output the vector **v** and the matrix *Q* according to the formulas (2) and (3). Then, embed the matrix *Q* into the size $m \times m$ identity matrix as described in the Algorithm and assign the resulting matrix to *Q* again.
Note: to check if a vector is a zero vector, you can use the command `~any()`.

**Next, place the commands below that output consecutive iterations for *r* and *q* according to the formulas (1) and (4) (remember that *r* and *q* were initialized as `r=A` and `q=eye(m)`):

```
r=Q*r;
q=q*Q;
r=closetozeroroundoff(r,7);
```

This is the **end of the "for" loop**.

After the loop is completed, your outputs will be matrices *q* and *r*. Do not display them here.

**Next, we check if *q* and *r* indeed define an orthogonal-triangular decomposition of A by conducting the three separate tests as below.
Note: A message needs to be displayed only if the required condition does not hold. Do not terminate the code after any check.

Assign initially:
`test=0;`
**First, we will verify that `r` is an upper triangular matrix. You can use the condition `~istriu(r)` and, if it is "True", output:
```
disp('r is not upper-triangular?!')
test=1;
```
**Next, we will verify that the square matrix `q` is orthogonal. Use the condition `any(closetozeroroundoff(q'*q-eye(m),7),"all")` and, if it is "True", output:
```
disp('q is not orthogonal?!')
test=1;
```
**Finally, we check if we got QR factorization. Use the condition `any(closetozeroroundoff(A-q*r,7),"all")` and, if it is "True", output:
```
disp('QR factorization is not working?!')
test=1;
```
**After completing the three tests above, place the following into your code:
```
if test
    q=[];
    r=[];
end
```

Note: the function `hreflections` will be used in the next function that we will create and in the future exercises - it is very important to have the correct code; if you are receiving any empty outputs, please consider revision of your code.

This is the end of the function `hreflections`

**Print the function `hreflections` in the Live Script.
**Run the function on the choices (a)-(j) as indicated below:
```
%(a)
A=zeros(2,4)
[q,r] = hreflections(A)
%(b)
A=ones(2)
[q,r] = hreflections(A)
%(c)
A=zeros(4);A(2,[2 4])=ones(2,1)
[q,r] = hreflections(A)
%(d)
A=pascal(4)
[q,r] = hreflections(A)
%(e)
A=magic(4)
[q,r] = hreflections(A)
```

```
%(f)
A=[magic(3),ones(3,2)]
[q,r] = hreflections(A)
%(g)
A=[magic(3);ones(2,3)]
[q,r] = hreflections(A)
%(h)
A=triu(magic(4))
[q,r] = hreflections(A)
%(i)
A=[magic(3);hilb(3)]
[q,r] = hreflections(A)
%(j)
A=[1 1 2 0;0 0 1 3;0 0 2 4;0 0 3 5]
[q,r] = hreflections(A)
```

Part 3

Theory: In this part, we will use the "full-size" qr factorization of an *m* x *n* matrix A constructed in Part 2, which produces an *m* x *m* orthogonal matrix *q* and an *m* x *n* upper triangular matrix *r*, such that $A = qr$, and construct an "economy-size" QR factorization of A. Notice that, if $m > n$, matrix *r* has rows of zeros from row $n + 1$ to m. We will construct a QR factorization that discards those zero rows of the matrix *r* and the corresponding columns of the matrix q. We will refer to this factorization as an "economy-size" QR decomposition of A. If the columns of the matrix A are linearly independent, by using this method we will decompose A into a product of an $m \times n$ matrix Q, whose columns form an orthonormal basis for the Col A, and an $n \times n$ upper-triangular invertible matrix R with the positive entries on the main diagonal.

Below is an explanation:
Let A be an *m* x *n* matrix. If $m > n$ and $A = qr$, where $q = [q_1 \ q_2]$ with an *m* x *n* matrix q_1 and $r = \begin{bmatrix} r_1 \\ 0 \end{bmatrix}$ with an *n* x *n* upper triangular matrix r_1 and (*m* – *n*) x *n* zero matrix, then

$$A = qr = [q_1 \ q_2]\begin{bmatrix} r_1 \\ 0 \end{bmatrix} = q_1 r_1$$

Thus, if we assign $Q = q_1$ and $R = r_1$, we will have A = QR which gives us an "economy-size" QR factorization of A.
If the columns of A are linear independent, then Q is an *m* x *n* matrix with orthonormal columns and R is an *n* x *n* upper-triangular invertible matrix (it can be shown that R is invertible when A has linearly independent columns). When using modified Householder reflections to generate matrix *r* for a matrix A with linearly independent columns, the matrix R will have positive entries on the main diagonal. Comparing Q and R with the matrices U and V obtained by using the Gram-Schmidt orthogonalization (see Exercise 2 of this Project), it can be seen that Q and R have to match the matrices U and V, respectively. Thus, the columns of the matrix Q form an orthonormal basis for the Col A.
**Create a function in a file that begins with:

```
function [Q,R] = hbasis(A)
[m,n]=size(A);
rankA=rank(A);
```

The input is an m x n matrix A. The outputs Q and R are the matrices that form an "economy-size" QR factorization of A obtained from the "full-size" qr factorization generated by the function `hreflections`.

**Continue your function `hbasis` with placing the line below into the code:
```
[Q,R] = hreflections(A);
```

Next, **if the condition $m > n$ holds, make the following assignments:
re-assign to Q the matrix formed by first n columns of the output Q and re-assign to R the matrix formed by the top n rows of R (suppress these outputs).
Note: if $m \leq n$, the outputs Q and R of the function `hreflections` will not be modified.

Then, verify that your outputs Q and R, indeed, define a QR factorization of A, that is, A=QR.
**Construct a conditional statement with the condition
```
any(closetozeroroundoff(A-Q*R,7),"all")
```
and, if the condition above is "True", output a message:
```
disp('an economy-size factorization is not working?')
```
and assign:
```
Q=[];
R=[];
```
Please be sure to verify your code if you received this message and empty outputs!

For a matrix with <u>linearly independent columns</u>, that is, **if** `rankA==n,` continue your code with a verification that your output matrices Q and R match the outputs U and V, respectively, of the function `qrschmidt` (see Exercise 2 of this Project).

Proceed as follows:
**Place the function
```
[U,V]=qrschmidt(A);
```
in your code (suppress the outputs) and use conditional statements and the function `closetozeroroundoff` with p=7 to verify that Q matches U and R matches V within the given precision. Code the output messages only if there is a mismatch.
**If the matrices Q and U do not match, code a message:
```
disp('the basis Q does not match the Gram-Schmidt basis U')
```
and, if the matrices R and V do not match, code a message:
```
disp('R does not match upper-triangular V from the Gram-Schmidt process')
```
**If there is at least one mismatch, output:
```
disp('Check the code!')
Q=[];
R=[];
```
If you receive any empty outputs after running the function, please be sure to verify the code.
This is the end of the function `hbasis`.

**Print the functions, `qrschmidt, hbasis` and `hreflections` in the Live Script.
**Run the function on the choices (a)-(h) as indicated below:
```
%(a)
A=magic(4)
[Q,R] = hbasis(A)
%(b)
```

```
A=[1 2 3;2 4 3;2 4 2]
[Q,R] = hbasis(A)
%(c)
A=[1 2 3;2 4 6;2 4 6]
[Q,R] = hbasis(A)
%(d)
A=rand(3,5)
[Q,R] = hbasis(A)
%(e)
A=randi([-6 6],5,3)
[Q,R] = hbasis(A)
%(f)
A=[magic(4),pascal(4)]
[Q,R] = hbasis(A)
%(g)
A=magic(3)
[Q,R] = hbasis(A)
%(h)
A=[magic(3);hilb(3)]
[Q,R] = hbasis(A)
```

Part III. Orthogonal Projections & Solutions of the Systems

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

In this exercise, you will create a function that outputs the orthogonal projection, **p**, of a vector **b** onto the column space of an $m \times n$ matrix A, and it will also output the vector **z** which is the component of **b** orthogonal to the Col A.

For help with this exercise, please refer to Lectures 28-31.

Theory: The ***orthogonal projection*** **p** of a vector $\mathbf{b} \notin$ Col A onto the Col A can be found by, first, producing an orthonormal basis Q for the Col A and, then, using the formula for the projection:

$$\mathbf{p} = \mathrm{Q}\mathrm{Q}^T\mathbf{b} \qquad (1)$$

If **p** is the orthogonal projection of a vector **b** onto Col A, then there exists a unique vector **z**, called the ***component of b orthogonal to Col A***, that can be computed by the formula:

$$\mathbf{z} = \mathbf{b} - \mathbf{p} \qquad (2)$$

Also, the 2-norm of the vector **z** is the ***distance*** from **b** to the Col A.

Another way of computing the projection **p** of a vector **b** onto the Col A, which we will use for a verification, is

$$\mathbf{p} = A\mathbf{x} \qquad (3)$$

where **x** is a least-squares solution of the system $\mathrm{A}\mathbf{x} = \mathbf{b}$ or, equivalently, a solution of the normal equations:

$$(\mathrm{A}^T\mathrm{A})\mathbf{x} = \mathrm{A}^T\mathbf{b} \qquad (4)$$

Note: For two special cases – when **b** is in Col A and when **b** is orthogonal to Col A – the vectors **p** and **z** can be output without computations (please refer to Lecture 30 for details).

Your program should allow a possibility that the columns of A are not linearly independent. In order for our algorithm to work, we will use the function `shrink()` to output a new matrix formed by the pivot columns of A (they form a basis for the Col A) and assign that matrix to A again. 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 [pivot,B]=shrink(A)
[~,pivot]=rref(A);
B=A(:,pivot);
end
```

**Create a function that begins with the lines:

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

The inputs are a matrix A with *m* rows and a column vector **b**. If the numbers of rows in A and **b** match, your outputs **p** and **z** will be, respectively, the orthogonal projection of **b** onto the Col A and the component of **b** orthogonal to the Col A.

First, your function has to check whether the input vector **b has exactly *m* entries, where *m* is the number of rows of A. If it is not the case, the program terminates with a message `'No solution: sizes of A and b disagree'`, and the empty outputs for **p** and **z** will stay.

If b has exactly *m* entries, your function will continue:
Determine whether it is the case that **b** ∈ Col A – use here a MATLAB function `rank()`.

****If b ∈ Col A**, output a message `'b is in Col A'`, make assignments to the vectors **p** and **z** and display them (no computations are needed - see a reference to this case in the **Theory** above). After that, the program terminates.
Note: Please be sure to make the correct assignments to **p** and **z**. For example, if you wish to assign a zero vector, it has to be a vector of zeros with the size specified.

****If b ∉ Col A**, your function will continue with the following tasks:
Set up a **conditional statement** to determine whether it is the case that **b** is orthogonal to Col A. You can employ here a matrix-vector multiplication and the function `closetozeroroundoff` with p = 7 to check if this condition holds.

If **b is orthogonal to Col A, output a message `'b is orthogonal to Col A'`, make assignments to the vectors **p** and **z** and display them (no computations are needed!). Do not terminate the code here.

Otherwise, if **b is not orthogonal to Col A, we will output the vectors **p** and **z** using an orthonormal basis Q for Col A and, then, we will check if **p** and **z** are computed correctly. Proceed with outputting an orthonormal basis Q by placing the line below into your code:

```
[Q,~] = hbasis(A);
```

Then, use formulas (1) and (2) to output the vectors **p** and **z** (do not display them here).

Next, verify that **p** and **z** are computed correctly. Proceed as follows:
First, use the normal equations (4) and MATLAB backslash operator to output the least-squares solution **x:
```
x=(A'*A)\(A'*b);
```
and, then, output the projection of the vector **b**, a vector **p**1, by means of the formula (3):
```
p1=A*x;
```
Then, verify that **p and **p**1 match by using the function `closetozeroroundoff` with p=7. If yes, output:
```
disp('the projection of b onto Col A is')
p
```
Otherwise, output:
```
disp('Oops! p is not a projection!?')
p=[];
```

Next, verify that the vector **z is, indeed, orthogonal to the Col A: use a matrix-vector multiplication and the function `closetozeroroundoff` with p=7. If your code confirms that **z** is orthogonal to Col A, display z with the message as below:
```
disp('the component of b orthogonal to Col A is')
z
```
Otherwise, output:
```
disp('What?! z is not orthogonal to Col A!')
z=[];
```

This is the end of your conditional statement.

Finally, if both outputs p and z are non-empty (use MATLAB command `~isempty()`) output the distance `d` from **b to the Col A by using a MATLAB built-in function **`norm()`** (see the **Theory** above). Display `d` with the message:
```
fprintf('the distance from b to Col A is %i',d)
```

This is the end of your function `proj`.

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

**Run the function on the choices (a)-(i) as indicated below:
```
%(a)
A=[1 2 3;2 4 3;2 4 2], b=ones(3,1)
[p,z]=proj(A,b);
%(b)
A=[1 2 3;2 4 6;2 4 6], b=ones(3,1)
[p,z]=proj(A,b);
%(c)
A=magic(4), b=A(:,4)
[p,z]=proj(A,b);
%(d)
A=magic(5), b=(1:4)'
[p,z]=proj(A,b);
%(e)
A=magic(4), b=randi(10,4,1)
[p,z]=proj(A,b);
```

```
%(f)
A=magic(5), b = rand(5,1)
[p,z]=proj(A,b);
%(g)
A=rand(4,3), b=ones(4,1)
[p,z]=proj(A,b);
%(h)
A=ones(4); A(:)=1:16, b=[1;0;1;0]
[p,z]=proj(A,b);
%(i)
B=ones(4); B(:)=1:16, A=null(B,'r'), b=ones(4,1)
[p,z]=proj(A,b);
```

BONUS! (1 point)
% Analyze the outputs in part (f) and write a comment that would explain a reason why a random vector **b** belongs to the Col A.
% For part (i), based on the input matrix A and the outputs, state to which vector space relative to the matrix A the vector **b** has to belong.

**You will need to run some commands in your Live Script to support your responses. Supply them with corresponding output messages and/or comments.

Exercise 5 (4 points) **Difficulty: Moderate**
Theory: In this exercise, we will be solving a system $A\mathbf{x} = \mathbf{b}$. If the system is consistent, we will find an "exact" solution and, if it inconsistent, we will find a least-squares solution.

We will consider two separate cases – they require different approaches.

Case 1. If a matrix A is invertible, we will output the unique solution by using the inverse matrix of A. The inverse of A will be computed by LU factorization as an output of the function `[invA,~] = eluinv(A)` which was created in Exercise 5, Part 2 of Project 3.
Case 2. In all other situations, we will find an "exact" solution or a least-squares solution by employing the QR "economy-size" factorization, where `Q` and `R` are the outputs of the function `[Q,R] = hbasis()` which was created in Exercise 3, Part 3 of this Project. If a matrix A has linearly independent columns, the columns of the output matrix `Q` form an orthonormal basis for Col A and `R` is an upper triangular invertible matrix.

The chain below justifies a use of the formula $R\mathbf{x} = Q^T\mathbf{b}$ for finding a solution **x** of a system A**x** = **b** when the columns of A are linearly independent. In this case, the unique least-squares solution is the solution of the normal equations. For a consistent system, the least-squares solution coincides with the "exact" solution, since either of the solutions is an "exact" solution of $A\mathbf{x} = \hat{\mathbf{b}}$, where $\hat{\mathbf{b}}$ is the orthogonal projection of **b** onto the Col A, and $\hat{\mathbf{b}} = \mathbf{b}$ for a consistent system. Thus, the unique solution (least-squares or "exact") can be found as:

$$A^TA\mathbf{x} = A^T\mathbf{b}\,, A = QR \quad \Leftrightarrow \quad (QR)^T(QR)\mathbf{x} = (QR)^T\mathbf{b} \quad \Leftrightarrow \quad R^T(Q^TQ)R\mathbf{x} = R^TQ^T\mathbf{b}$$
$$\Leftrightarrow \; R^TR\mathbf{x} = R^TQ^T\mathbf{b} \quad \Leftrightarrow \quad R\mathbf{x} = Q^T\mathbf{b}$$

If the set of columns of a matrix A is not linearly independent, we will, first, "shrink" it into a basis for the Col A by leaving only the pivot columns of A, then, find the solution **y** with the new matrix by applying row reduction algorithm to the system $R\mathbf{y} = Q^T\mathbf{b}$, and, finally, adjust the output **y** to satisfy the original system by using the following approach:
The solution **y** outputs the weights on the pivot columns of A to produce $\hat{\mathbf{b}} = \text{proj}_{\text{ColA}}\mathbf{b}$.
When the original matrix A has both pivot and non-pivot columns, we can assign zero weights on the non-pivot columns of A and the combined vector of weights on all columns of A, which produces $\hat{\mathbf{b}}$, will form a solution **x** of the original system.

**Create a function in a file that begins with

```
function x=solveall(A,b)
format
[m,n]=size(A);
x=zeros(n,1);
consist=0;
uniq=0;
```

The inputs are an *m* x *n* matrix A and an *m* x 1 vector **b**. The output **x** is an "exact" solution for a consistent system or a least-squares solution for an inconsistent system.
The output vector **x** is pre-allocated as a zero vector.
Within our code, we will check on the two **Cases** (see above): if a matrix A is invertible (then the system is consistent and the solution is unique), and if A is not invertible (then we will need to determine whether the system is consistent and whether the solution is unique - we have initialized these conditions as "False").

Continue with **a conditional (IF … Else) statement** that will check if a matrix A is invertible, that is, if both conditions hold: A is square and the rank of A is equal to the number of rows/columns.
****IF A is invertible**, assign:

```
consist=1;
```

and display a message

```
disp('the matrix is invertible - there is a unique "exact" solution')
```

**Then, place the function below into your code that outputs the inverse of the matrix A:

```
[invA,~] = eluinv(A);
```

and use the matrix `invA` to output the solution **x** of the matrix equation $A\mathbf{x} = \mathbf{b}$ (do not display **x** here).

****Else (A is not invertible)**,
we will do several checks on the types of the solution set, and, finally, output a solution by using QR factorization. Proceed as follows (you can construct a separate conditional statement for each of the checks below):
**First, check if the system is consistent.
If yes, re-assign:

```
consist=1;
```

and display a message:

```
disp('the system is consistent - look for "exact" solution')
```

Otherwise, display a message

```
disp('the system is inconsistent - look for least-squares solution')
```

**Then, check if there is a unique solution or there are infinitely many solutions (we refer to a "solution" here as either "exact" solution or a least-squares solution). Recall: the system has a unique solution if and only if the columns of A are linearly independent. Use the command `rank()` here.
If the solution is unique, re-assign:
```
uniq=1;
```
and display
```
disp('the system has a unique solution')
```
Otherwise, display:
```
disp('the system has infinitely many solutions')
```

**Next, we will output the matrix B of the pivot columns of A (they form a basis for Col A). Proceed as follows:
If there are infinitely many solutions, that is, `uniq=0,` place the function
```
[pivot_c,B]=shrink(A);
```
which outputs the vector of indexes of pivot columns of A and the matrix B formed by the pivot columns of A (we suppress these outputs).
Otherwise (if the solution is unique), assign:
```
B=A;
```

**Continue with constructing an "economy-size" QR factorization of the matrix B by placing the function below into your code:
```
[Q,R] = hbasis(B);
```
Next, use the outputs Q and R to find the solution **y of $R\mathbf{y} = Q^T\mathbf{b}$ (see **Theory** above) by employing the row reduction algorithm - due to the form of the matrix R, this algorithm does not require too many arithmetic operations. Proceed as follows:
**Place the set of commands below into your code:
```
aug=[R Q'*b];
aug=rref(aug);
y=aug(:,end);
```

If there are infinitely many solutions, that is, `uniq==0`, output a solution **x as
```
x(pivot_c,1)=y;
```
(the weights on the non-pivot columns of A are zeros).
Otherwise, if the solution is unique (all columns are pivot), output
```
x=y;
```
Do not display **x** here.
This is the end of the conditional (IF … Else) statement.

Continue your code with a check if a solution **x**, which is the output of the conditional statement above, is found correctly. Consider the following two cases:
**If `consist==1`, use the condition `~any(closetozeroroundoff(A*x-b,7))` to verify if a solution is found correctly. If this condition is "True", output the solution with a message as below:
```
disp('an "exact" solution of the system is')
disp(x)
```
Otherwise, output
```
disp('Check code for consistent system (including when A is invertible)')
x=[];
```

**If the system is inconsistent, use the condition `~any(closetozeroroundoff((A'*A)*x-A'*b,7))` to check if a least-squares solution is found correctly. If this condition is "True", output a solution with the message as below:

```
disp('a least-squares solution is')
disp(x)
```

Otherwise, output

```
disp('Check code for inconsistent system')
x=[];
```

If you are receiving any empty output when running the function on the choices below, please make corrections in your code.

This is the end of the function `solveall`.

**Print the functions `solveall, closetozeroroundoff, shrink, hbasis, eluinv` in your Live Script.

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

```
%(a)
A=magic(5), H=60*hilb(5); b=H(:,1)
x=solveall(A,b);
%(b)
A=magic(4), b=ones(4,1)
x=solveall(A,b);
%(c)
A=magic(4), b=rand(4,1)
x=solveall(A,b);
%(d)
A=[1 2 3;2 4 6;2 4 6], b=ones(3,1)
x=solveall(A,b);
%(e)
A=[magic(5);zeros(1,5)], b=rand(6,1)
x=solveall(A,b);
%(f)
A=[pascal(5);randi([-5 5],2,5)], b=sum(A,2)
x=solveall(A,b);
%(g)
A=[magic(3),ones(3,2)], b=sum(A,2)
x=solveall(A,b);
%(h)
A=[1 2 3;2 4 3;2 5 2], b=ones(3,1)
x=solveall(A,b);
```

Part IV. Applications

In this part of the project, you will write a code that outputs the best least-squares fit polynomial for the given data. It will be defined by a **parameter vector c,** which is calculated by using 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.

Theory: Assume that we are given m data points (x_i, y_i) $(i = 1{:}\,m)$. A choice of an equation of a curve of the best least-squares fit to the data points is, in general, arbitrary. In this exercise, we will be looking for a polynomial of degree n of the form

$$P = c_n x^n + c_{n-1} x^{n-1} + \ldots + c_1 x + c_0$$

for some consecutive values of n, starting with $n = 1$.

You will need to generate the design matrix X within your code. Its form depends on the degree n of the polynomial. In general,

$$X = \begin{bmatrix} x_1^n & x_1^{n-1} & \ldots & x_1 & 1 \\ x_2^n & x_2^{n-1} & \ldots & x_2 & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ x_m^n & x_m^{n-1} & \ldots & x_m & 1 \end{bmatrix}, \text{ where } \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \mathbf{x} \text{ is the vector of the x-coordinates of the}$$

data points. A parameter vector **c** is the vector of the coefficients of a polynomial of the best fit, and it is calculated as a least-squares solution of the system $X\mathbf{c} = \mathbf{y}$, where

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

is the observation vector formed by the y-coordinates of the data points.

To output a parameter vector $c = (c_n, c_{n-1}, \ldots, c_0)$, we will be using the function `solveall` created in Exercise 5 of this Project.

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

**First, create the following function in MATLAB (the code is 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 coefficients, written in the descending order according to the degree, form the row vector `p`. The function `polyplot` will be used in the function that we will create next.

**Create a function that begins with:

```
function [c,X,N]=lstsqpoly(x,y,n)
format
m=length(x);
c=[];
N=[];
```

The inputs of this function are: a vector **x** of the x-coordinates of the data points, a vector **y** of the y-coordinates of the data points, and a degree n of the polynomial of the best least-squares fit. The outputs are: a parameter vector **c**, the design matrix X, and a number N which is the 2-norm of the residual vector **e**.

Continue your function `lstsqpoly` with the following:
**Assign to the endpoints of the interval, the numbers a and b (they are inputs of the function `polyplot`), the first and the last entry of the vector x, respectively.

Use the vector **x of the x-coordinates of the data points to output the $m \times (n+1)$ design matrix X (see **Theory** above) and display it with a message:
```
disp('the design matrix is')
X
```
Hint: you can employ here either a "for" loop or a vectorized statement (see Project 0, Exercise 2, Part 3).

**Place the line below into your code that outputs a solution c of the system $X\mathbf{c} = \mathbf{y}$:
```
c=solveall(X,y);
```

**Next, we will verify that our output c matches a least-squares solution of $X\mathbf{c} = \mathbf{y}$ calculated by using MATLAB backslash operator.
First, output the vector:
```
c1=X\y;
```
and, then, compose a conditional statement with the condition
```
any(closetozeroroundoff(c-c1,7)).
```
If the condition above is "True", display a message below and assign an empty output to c:
```
disp('Check the code!')
c=[];
```
after that, terminate the program.

If your function passes the checkpoint above, output and display the 2-norm of the residual vector **e, number N, as below:
```
N=norm(y-X*c);
fprintf('the 2-norm of the residual vector e is %i\n',N)
```

**Next, include the following two lines into your code which will plot the data points and the polynomial of the best least-squares fit:
```
plot(x,y,'*'),hold on
polyplot(a,b,c')
```

**Then, output the polynomial of the best least-squares fit with a message as indicated below:
```
fprintf('the polynomial of degree %i of the best least-squares fit is\n',n)
P=vpa(poly2sym(c),4)
```
(this command outputs and displays the polynomial whose coefficients are evaluated with 4 decimal digit accuracy).

**Complete your function with the last command
```
hold off
```

This is the end of the function `lstsqpoly`.

**Print the functions `polyplot`, `solveall`, and `lstsqpoly` in your Live Script.

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

```
%(a)
x = [1;2;3;4;5;6;8], y = [1;2;3;5;6;7;4]
n=1
[c,X,N]=lstsqpoly(x,y,n);
n=2
[c,X,N]=lstsqpoly(x,y,n);
n=3
[c,X,N]=lstsqpoly(x,y,n);
n=4
[c,X,N]=lstsqpoly(x,y,n);
n=5
[c,X,N]=lstsqpoly(x,y,n);
n=6
[c,X,N]=lstsqpoly(x,y,n);
%(b)
m=10;
x=(1:m)'/m; y=10*rand(m,1);
n=1
[c,X,N]=lstsqpoly(x,y,n);
n=2
[c,X,N]=lstsqpoly(x,y,n);
n=3
[c,X,N]=lstsqpoly(x,y,n);
n=4
[c,X,N]=lstsqpoly(x,y,n);
n=5
[c,X,N]=lstsqpoly(x,y,n);
n=6
[c,X,N]=lstsqpoly(x,y,n);
```

BONUS! (1 point)
% Analyze the outputs of the choice (a) for n=6. Based on your outputs and general theory, explain a reason why the best least-squares fit polynomial, in fact, interpolates the 7 data points.

Part V. Applications to Differential Equations

In this part, you will work with an application of eigenvalues and eigenvectors to the continuous analogues of difference equations, specifically, to the dynamical systems involving differential equations.

Theory: Suppose that $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is a vector of differentiable functions of a single variable t in $\mathbb{R}$ with the vector of its derivatives $\mathbf{x}' = (x_1', x_2', \ldots, x_n')$. Suppose also that A is an $n \times n$ matrix and $\mathbf{x}0$ is a (numerical) $n \times 1$ vector.

The ***initial value problem*** for a system of differential equations defined by a matrix A and an initial vector $\mathbf{x}0$ can be written in the form:

$$\mathbf{x}'(t) = A\mathbf{x}(t), \quad \mathbf{x}(0) = \mathbf{x}0 \qquad (1)$$

or, in expanded form,

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \\ \vdots \\ x_n'(t) \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{21} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ x_n(t) \end{bmatrix}, \qquad \begin{bmatrix} x_1(0) \\ x_2(0) \\ \vdots \\ x_n(0) \end{bmatrix} = \mathbf{x}0 \qquad (2)$$

From the general theory it follows that, if a matrix A is diagonalizable, that is A has *n* linearly independent eigenvectors, the general solution of the system $\mathbf{x}'(t) = A\mathbf{x}(t)$ has a form

$$\mathbf{x}(t) = c_1\mathbf{v_1}e^{\lambda_1 t} + c_2\mathbf{v}_2e^{\lambda_2 t} + \dots + c_n\mathbf{v}_ne^{\lambda_n t} \qquad (3)$$

where $\lambda_1, \lambda_2, \dots, \lambda_n$ are the eigenvalues of A, the vectors $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$ are the corresponding eigenvectors, and $\mathbf{v}_1e^{\lambda_1 t}, \mathbf{v}_2e^{\lambda_2 t}, \dots, \mathbf{v}_ne^{\lambda_n t}$ are the ***eigenfunctions*** for A which form a basis for the solution set of $\mathbf{x}'(t) = A\mathbf{x}(t)$.

Moreover, for any initial condition **x**(0) = **x**0**,** there exists a unique set of weights $C = (c_1, c_2, \dots, c_n)$ that defines the solution (3) of the initial value problem (1). The vector of weights C can be found by solving the system of equations below which we obtaine by substituting $t = 0$ into the formula (3) and counting the initial condition **x**(0)= **x**0:

$$c_1\mathbf{v_1} + c_2\mathbf{v_2} + \dots + c_n\mathbf{v_n} = \mathbf{x}0 \qquad (4)$$

For the case when n=2 and a diagonalizable matrix A has **real nonzero** eigenvalues, we can identify the **origin** as an attractor, or repeller, or a saddle point of a continuous dynamical system $\mathbf{x}'(t) = A\mathbf{x}(t)$.
Indeed, from the form of the general solution for n = 2

$$\mathbf{x}(t) = c_1\mathbf{v}_1e^{\lambda_1 t} + c_2\mathbf{v}_2e^{\lambda_2 t} \qquad (5)$$

it follows that:
I. The origin is a repeller if both eigenvalues are positive; the direction of the greatest repulsion is through **0** and the eigenvector corresponding to the largest eigenvalue.
II. The origin is an attractor if both eigenvalues are negative; the direction of the greatest attraction is through **0** and the eigenvector corresponding to the eigenvalue of the largest magnitude.
III. The origin is a saddle point if one eigenvalue is negative and the other eigenvalue is positive; the direction of the greatest attraction is through **0** and the eigenvector corresponding to the negative eigenvalue and the direction of the greatest repulsion is through **0** and the eigenvector corresponding to the positive eigenvalue.
For more details, please read Section 5.7 of the Textbook.

For the case when n=2 and the eigenvalues are imaginary numbers

$$\lambda_1 = a - ib\ ,\ \lambda_2 = a + ib \quad (b \neq 0),$$

we could have used the representation (5) for the general solution; however, it is more convenient to work with the real-valued functions.

Below is an explanation how to construct a real basis for the solution set of $\mathbf{x}'(t) = A\mathbf{x}(t)$ when the eigenvalues of A are two complex conjugate numbers.

We can take one of the eigenvalues, say $\lambda_2 = a + ib$, and an eigenvector for λ_2, $\mathbf{V} = \mathbf{v}_2$, and use them to construct an eigenfunction basis for the solution set, denoted $\{\mathbf{y}_1(t), \mathbf{y}_2(t)\}$, by isolating the real and imaginary parts of the eigenfunction $\mathbf{v}_2 e^{\lambda_2 t} = \mathbf{V}e^{(a+ib)t}$:

We can write

$$\mathbf{V} = \mathrm{Re}\mathbf{V} + i\ \mathrm{Im}\mathbf{V} \text{ and } e^{(a+ib)t} = e^{at}(\cos(bt) + i\sin(bt)).$$

then,

$$\mathbf{V}e^{(a+ib)t} = (\mathrm{Re}\mathbf{V} + i\ \mathrm{Im}\mathbf{V})(\cos(bt) + i\sin(bt))\ e^{at}.$$

If we denote

$$\mathbf{y}_1(t) = \mathrm{Re}\{\mathbf{V}e^{(a+i\ )t}\} \text{ and } \mathbf{y}_2(t) = \mathrm{Im}\{\mathbf{V}e^{(a+ib)t}\}$$

then

$$\mathbf{y}_1(t) = (\cos(bt)\ \mathrm{Re}\mathbf{V} - \sin(bt)\ \mathrm{Im}\mathbf{V})\ \exp(at) \qquad (6)$$
$$\mathbf{y}_2(t) = (\sin(bt)\ \mathrm{Re}\mathbf{V} + \cos(bt)\ \mathrm{Im}\mathbf{V})\ \exp(at)$$

$(t \in \mathbf{R})$

The set $\{\mathbf{y}_1(t), \mathbf{y}_2(t)\}$ is linearly independent and forms a basis for the solution set of $\mathbf{x}'(t) = A\mathbf{x}(t)$. Thus, the general solution of a continuous dynamical system can be written as

$$\mathbf{x}(t) = c_1\mathbf{y}_1(t) + c_2\mathbf{y}_2(t) \qquad (7)$$

The unique solution of the initial value problem $\mathbf{x}'(t) = A\mathbf{x}(t)$, $\mathbf{x}(0) = \mathbf{x}0$ is defined by the vector of weights $C = (c_1, c_2)$ that we will obtain by solving the system

$$\mathbf{x}(0) = c_1\mathbf{y}_1(0) + c_2\mathbf{y}_2(0)$$

which can be reduced to the system below after substituting $t = 0$ into the formulas in (6) and counting the initial condition $\mathbf{x}(0) = \mathbf{x}0$:

$$c_1\mathrm{Re}\mathbf{V} + c_2\mathrm{Im}\mathbf{V} = \mathbf{x}0 \qquad (8)$$

For the dynamical system above, **the origin** can be classified as a ***spiral point*** if $a \neq 0$. Moreover,

I. the trajectories spiral outward if $a > 0$;

II. the trajectories spiral inward if $a < 0$.

If $a = 0$, the trajectories form ***ellipses*** around the origin.

For more details on this topic and Examples, please refer to the Section 5.7 of the textbook.

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

In this exercise, we will generate an eigenfunction basis for a continuous dynamical system $\mathbf{x}'(t) = A\mathbf{x}(t)$ with a diagonalizable matrix A. When **n = 2**, we will plot the trajectories of the dynamical system for the various choices of the initial vector $\mathbf{x}0$.

Start with creating an auxiliary function that will output an eigenvector basis for C^n. The function is a modification of the function `eigendiag`, which was created in Exercise 1 of this Project. The details how the function `eigendiag` should be modified are listed below.

**Create the function that begins with:

```
function [L,P,D]=eigenbasis(A)
```

**Continue with copying the body of the function `eigendiag` and pasting it into the function `eigenbasis`.

**Then, modify the code as indicated below:
(1) replace the command `null( )` with `null( ,'r')` to create a "rational" basis for an eigenspace;
(2) delete all output messages except for the two listed below:

```
disp('A is not diagonalizable')
disp('A is diagonalized')
```

(3) delete all commands that display the variables, such as, `display()` and `disp()`;
(4) suppress all outputs (place semicolon after each command that outputs a variable).
Save your new function in the Current folder in MATLAB with the name `eigenbasis.m`

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

```
function []=invalprob(A,X0)
format
[~,n]=size(A);
```

It takes as inputs an $n \times n$ matrix `A` and an $n \times 1$ initial vector `X0`.

**Continue with placing into your code the function `eigenbasis` as below:

```
[L,P,~]=eigenbasis(A);
```

**Then, place the following lines:

```
if isempty(P)
    return
end
```

- this will terminate the function in the case when the matrix A is not diagonalizable or when there is a problem with the function `eigenbasis`.

**Continue with displaying the eigenvalues and eigenvector basis:

```
fprintf('all eigenvalues of A are\n')
disp(L)
fprintf('an eigenvector basis is\n')
P
```

**Then, introduce a symbolic variable *t*. Type:

```
syms t
```

Continue your function with a conditional (if** …**else**) statement that separates two cases: when **all eigenvalues are real** and when **eigenvalues are imaginary** – you can use the command `isreal(L)` here.

****If L is real**, output a symbolic matrix $E = [\mathbf{v}_1 e^{\lambda_1 t} \quad \mathbf{v}_2 e^{\lambda_2 t} \ \ldots \ \mathbf{v}_n e^{\lambda_n t}]$, whose columns form an eigenfunction basis for the solution set. Display it with the message:

```
fprintf(['an eigenfunction basis for the solution set' ...
        ' is formed the columns of'])
disp(E)
```

Hint: to output E, you can construct a symbolic vector whose entries are the exponential functions of the entries of the vector L scaled by `t`, then, output a diagonal matrix with the constructed vector on its main diagonal, and, finally, use the property that right multiplication of the matrix P = $[\mathbf{v}_1 \ \mathbf{v}_2 \ \ldots \ \mathbf{v}_n]$ by a diagonal matrix scales the column of P by the corresponding entries on the main diagonal of the diagonal matrix.

If $n = 2$, your code (for the real eigenvalues) will continue with the following tasks:

**First, output (do not display):

```
V1=P(:,1);
V2=P(:,2);
flag=0;
```

**Then, if A does not have a zero eigenvalue (use the command `rank()` to check this condition) proceed with a coding of the cases when the origin is an attractor, a repeller, or a saddle point of a continuous dynamical system. Display a corresponding message for each of the three cases and output and display the direction of the greatest attraction for an attractor, the direction of the greatest repulsion for a repeller, and both the direction of greatest attraction and the direction of greatest repulsion for a saddle point - each output has to be supplied with a corresponding message.
**Flag the case when the origin is an attractor (assign: `flag=1;`) – we will use specific settings for the limits on the variable *t* when plotting solutions for this case.

Important Notes: please read the Theory above on the definitions of an attractor, a repeller, and a saddle point for a continuous dynamical system – they are different from the ones for a discrete dynamical system presented in Lecture 26.
The direction of greatest attraction/repulsion has to be given in terms of the eigenvector `V1` or `V2`. Please also recall that the real eigenvalues have been sorted in the ascending order by the function `eigenbasis`.

For all cases of real eigenvalues (including a zero eigenvalue) continue as below:
**Place the following into your code (this gives us more input options for X0):

```
X0 = [V1,V2,-V1,-V2,X0];
N=size(X0,2);
```

**Set up a "for" loop that will plot the parametric functions for various choices of the initial vectors `X0` as below (these commands are implications of the formulas (4) and (5)):

```
for i=1:N
    C=[V1 V2]\X0(:,i);
    xt = @(t) C(1)*V1(1)*exp(L(1)*t)+C(2)*V2(1)*exp(L(2)*t);
    yt = @(t) C(1)*V1(2)*exp(L(1)*t)+C(2)*V2(2)*exp(L(2)*t);
    if flag
        fplot(xt,yt,[-1 1])
    else
        fplot(xt,yt,[-.3 .3])
    end
    hold on
end
hold off
```

%Write a comment in the Live Script on each of the line of the code given above.

****Else (L is imaginary)** - you will work with the case **n = 2** only. Place into your code:

```
V=P(:,2);
a=real(L(2));
b=imag(L(2));
ReV=real(V);
ImV=imag(V);
a=closetozeroroundoff(a,7);
```

**Set up a conditional statement that will output messages stating if `'the origin is a spiral point with inward trajectories'` or `'the origin is a spiral point with outward trajectories'` or `'the trajectories form ellipses around the origin'`

**Then, type:

```
X0 = [ReV,ImV,X0];
N=length(X0);
```

and place the set of commands that will plot the trajectories for various choices of the initial vectors `X0` (these commands are implications of the formulas (6) - (8)):

```
for i=1:N
      C=[ReV ImV]\X0(:,i);
      xt = @(t) (C(1)*(cos(b*t)*ReV(1)-
      sin(b*t)*ImV(1))+C(2)*(sin(b*t)*ReV(1)+cos(b*t)*ImV(1))).*exp(a*t);
      yt = @(t) (C(1)*(cos(b*t)*ReV(2)-
      sin(b*t)*ImV(2))+C(2)*(sin(b*t)*ReV(2)+cos(b*t)*ImV(2))).*exp(a*t);
      fplot(xt,yt)
      hold on
end
hold off
```

This is the end of your conditional (if …else) statement and the end of your function `invalprob`.

**Print the function `invalprob and eigenbasis` in your Live Script.

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

```
%(a)
A=ones(2)
X0=[[1;-2],[1;2],[-2;5],[0;3],[-2;0]];
invalprob(A,X0)
%(b)
A=[-1.5 .5;1 -1]
X0=[[5;4],[1;1],[-5;-2],[0;4],[0;-4],[1;.8],[-.5;.5],[.5;-.5]];
invalprob(A,X0)
%(c)
A=[4 -5;-2 1]
X0=[[2;1],[0;2],[1;1.2],[-5;0],[-1;-2],[2.9;2.6]];
invalprob(A,X0)
%(d)
A=[2 0; 0 3]
X0=[[0;0],[1;1],[1;-1],[-1;-1],[-1;1],[1;.1],[-1;.1],[-1;-.1],[1;-.1]];
invalprob(A,X0)
%(e)
A=[-2 -2.5;10 -2]
X0=[3;3];
invalprob(A,X0)
%(f)
A=[.8 .5; -.1 1.0]
X0=[[-3;0],[-3;-1],[0;-3],[3;1],[-3;3]];
invalprob(A,X0)
%(g)
```

```
A=[-2 -5;1 4]
x0=[[1;2],[-4;-1]];
invalprob(A,x0)
%(h)
A=[3 3;0 3]
x0=[1;3];
invalprob(A,x0)
%(i)
A=diag([1,2,2,3,3,3])
x0=ones(6,1);
invalprob(A,x0)
%(j)
A=[.5 -.6;.75 1.1]
X0=[[.5;.5],[-1;-1],[1;-1],[-.5;-.5],[.5;-.5]];
invalprob(A,X0)
%(k)
A=[-6 4;-10 6]
X0=[1;2];
invalprob(A,X0)
```

BONUS! (2 point):
% Analyze the outputs and the plots for the choice (a). Write comments on the pattern of the trajectories for the solution $\mathbf{x}(t)$ depending on the choice of the initial vector `X0`.
Specifically, use the representation of the solution of the initial value problem and the plots to describe the trajectories for the cases when `X0=V1,-V1`, when `X0=V2,-V2`, and when `X0` has other assignments different from `V1,-V1`, `V2,-V2`.

YOU ARE DONE WITH THE PROJECTS!!!