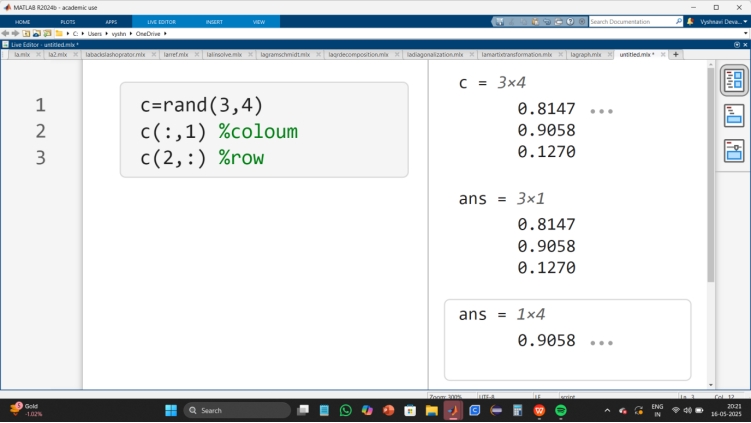
1. Creating a matrix of order 3\*4 with values between 0 to 1 using rand command and printing 1st column and 2nd row.



Description :

c = rand(3,4);

This creates a 3x4 matrix c, filled with random values between 0 and 1.

c(:,1)

The colon (:) in MATLAB means "select all elements."

Here, you're selecting all rows but only column 1.

This returns a 3x1 column vector.

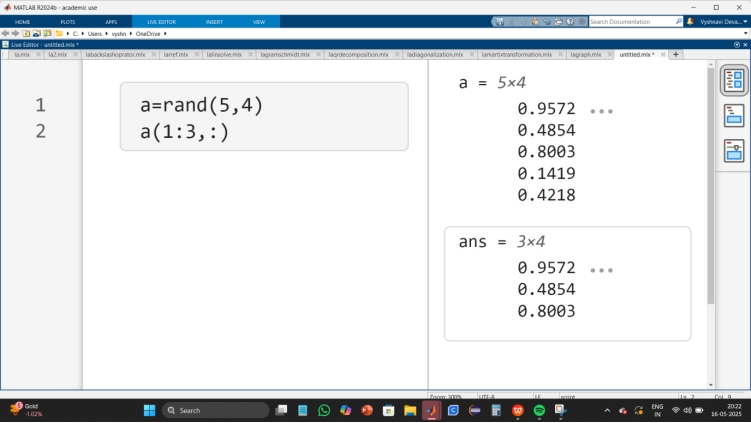
c(2,:)

The number 2 refers to the second row of c.

The colon (:) means "select all elements in that row."

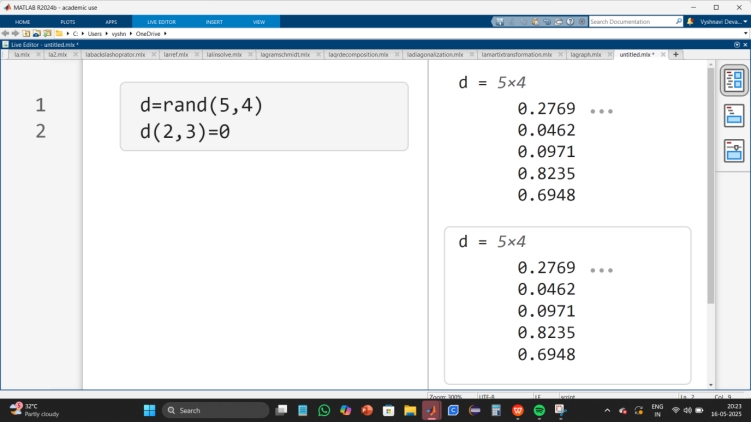
This returns a 1x4 row vector.

1. Creating a matrix of order 5\*4 with values between 0 to 1 using rand command and printing a sub-matrix of order 3\*4.

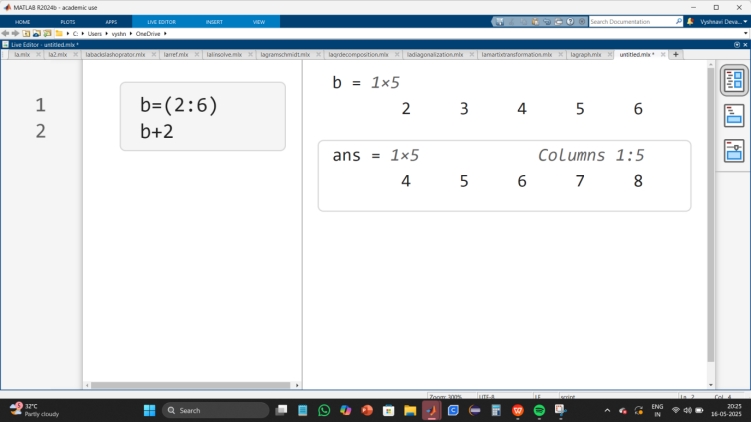


1. Creating a matrix of order 5\*4 with values between 0 to 1 using rand command and

making the value 0 at 2nd row , 3rd column.



1. Creating a row matrix with values from 2 to 6 and adding 2 to each element of the matrix.



Description :

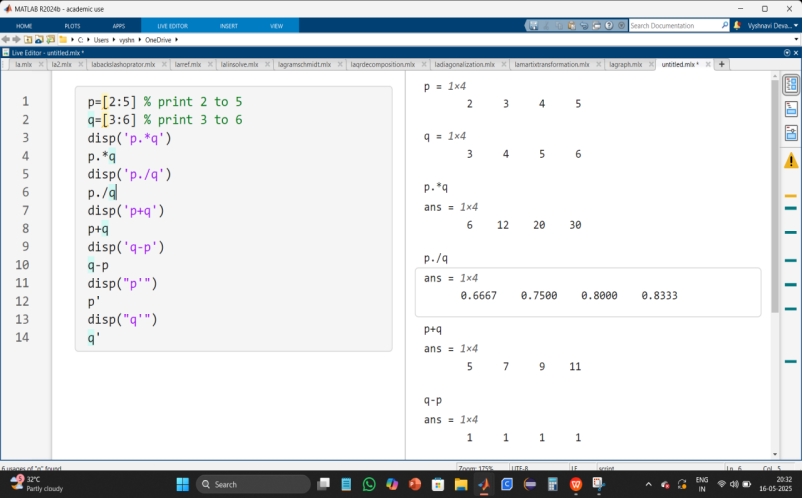
b = (2:6);

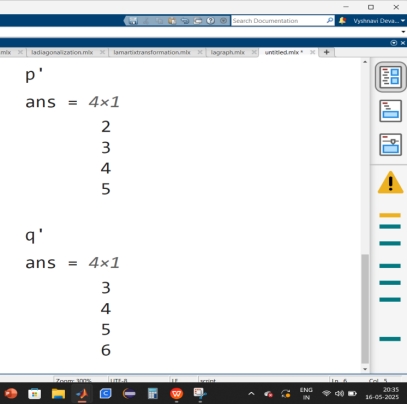
This creates a row vector containing integers from 2 to 6.

b + 2

This adds 2 to each element of b.

1. Creating two row matrices p and q with values from 2 to 5 and 3 to 6 and applying point wise multiplication, division ; element wise addition and subtraction and transpose of the row matrices p and q.





Description :

1. Point-wise multiplication (p .\* q)

Each element in p is multiplied by the corresponding element in q

1. Point-wise division (p ./ q)

Each element in p is divided by the corresponding element in q

1. Element-wise addition (p + q)

Each element of p is added to the corresponding element in q

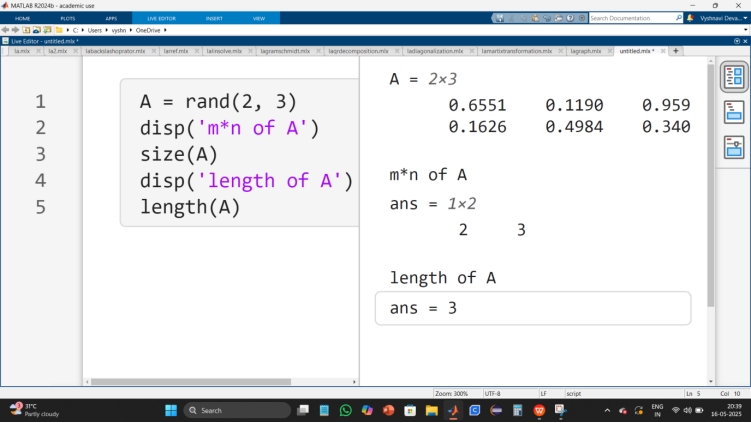
1. Element-wise subtraction (q - p)

Each element in q is subtracted by the corresponding element in p

1. Transpose (p' and q')

The transpose (') converts a row vector into a column vector

6) Creating a matrix of order 5\*4 with values between 0 to 1 using MATLAB rand command and for finding m\*n order of matrix use MATLAB size() , for finding largest dimension of matrix use MATLAB length() commands.



Description :

1. size(A)

This returns the size (dimensions) of matrix A as [5, 4], indicating 5 rows and 4 columns.

1. length(A)

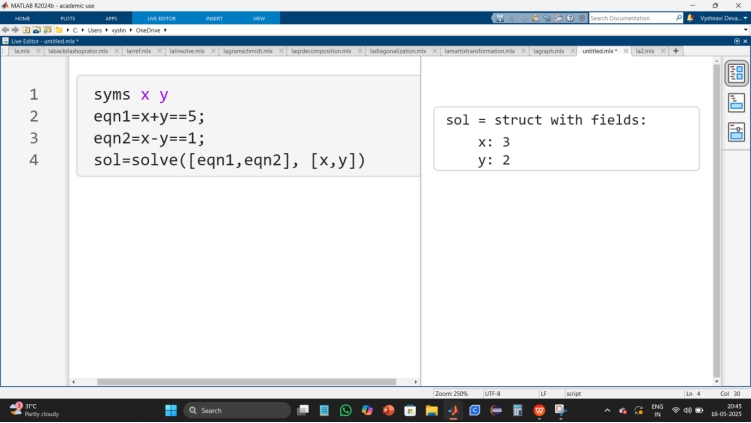
The length function returns the largest dimension of a matrix. Since A has 5 rows and 4 columns, length(A) will return 5.

EXERCISE - 2

AIM : to solve equations of two and three variables using MATLAB command solve and linsolve.

DATE :

1. Finding the solutions that satisfy the two equation of two variables using MATLAB solve command.



Description :

1) syms x y

This declares x and y as symbolic variables, allowing MATLAB to work with them algebraically.

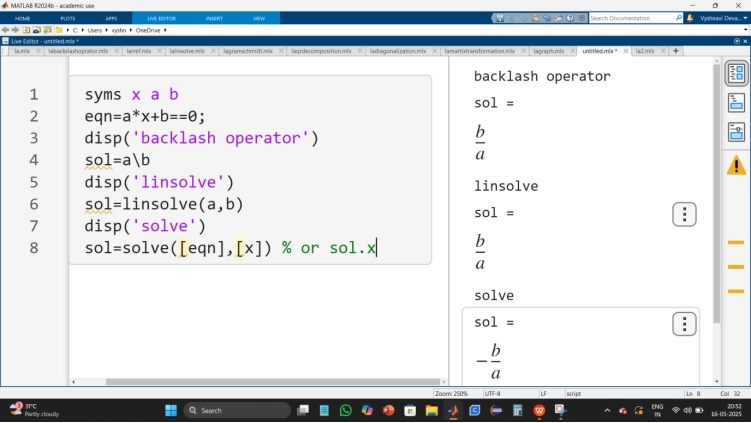
2) solve([eqn1,eqn2], [x,y])

solve() takes these two equations and solves for x and y simultaneously.

1. Finding the solutions that satisfy the two equation of three variables using MATLAB solve command.



1. Using MATLAB backlash operator, linsolve and solve commands to compute the equation.



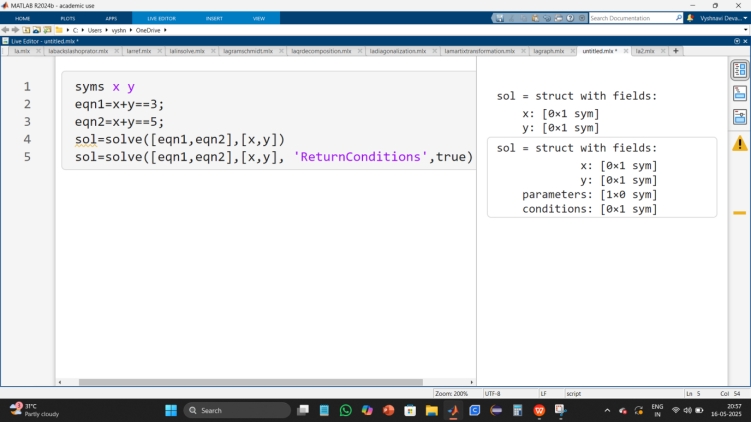
Description :

1) solve() correctly solves for x.

2) a\b is primarily for numerical matrix division.

3) linsolve() is designed for matrix equations.

1. Indicating the general conditions for the solution set of two equations of two variables.



Description :  
1) solve([eqn1, eqn2], [x, y])

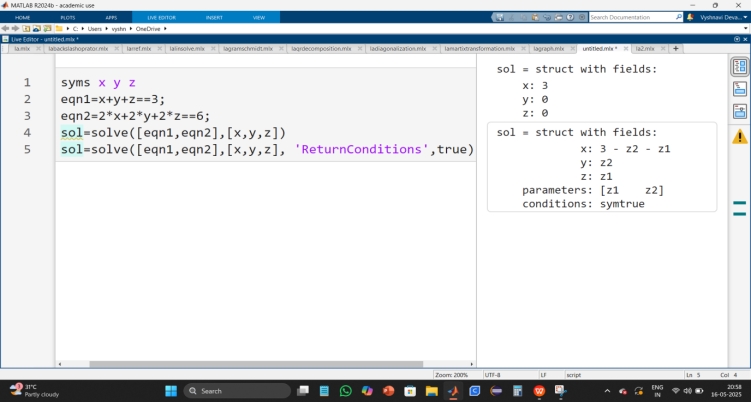
attempts to find values of (x) and (y) that satisfy both equations.

However, these two equations are inconsistent, meaning there is no solution.

2) solve([eqn1, eqn2], [x, y], 'ReturnConditions', true)

typically provides general solutions along with conditions under which they hold.

1. Indicating the general conditions for the solution set two equations of three variables.



Description :

1) second equation identical to the first equation i.e, one equation and three variables which means there are infinitely many solutions.

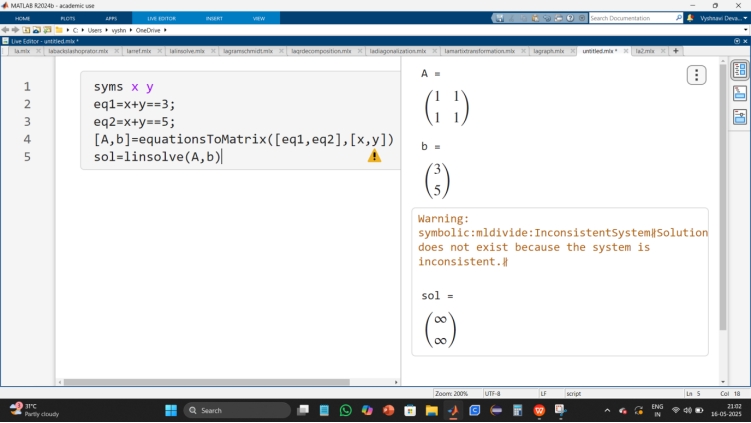
2) solve([eqn1, eqn2], [x, y, z])

will return a parametric solution, expressing two of the variables in terms of a free variable.

3) solve([eqn1, eqn2], [x, y, z], 'ReturnConditions', true)

will return the same solution, but also indicate the general conditions for the solution set.

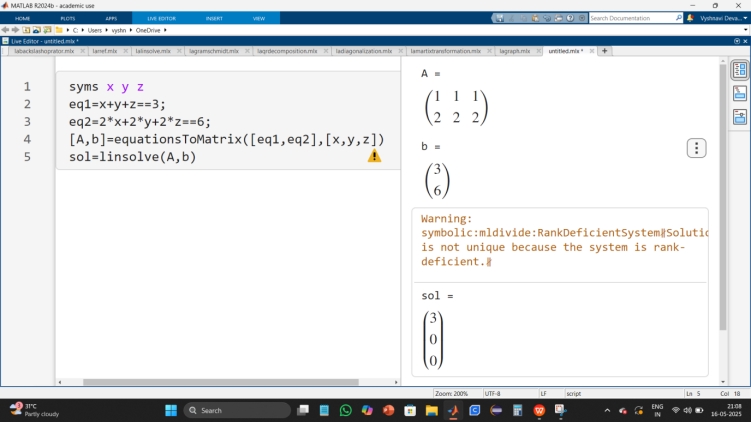
1. Finding the solutions that satisfy the two system of linear equations of two variables using MATLAB linsolve command.



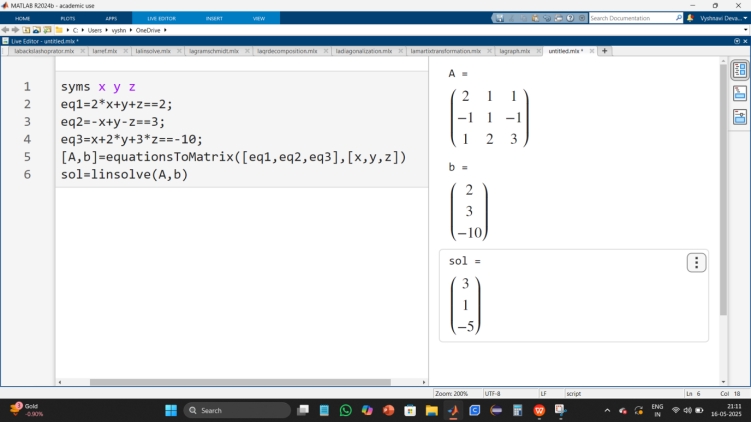
Description :

1) [A, b] = equationsToMatrix([eq1, eq2], [x, y]);   
transforms the system of equations into matrix form.  
 2) X = linsolve(A, B)  
Solves the matrix equation AX = B. A = coefficient matrix, B = right-hand side vector or matrix and X = solution.

1. Finding the solutions that satisfy the two system of linear equations of three variables using MATLAB linsolve command.



1. Finding the solutions that satisfy the three system of linear equations of three variables using MATLAB linsolve command.

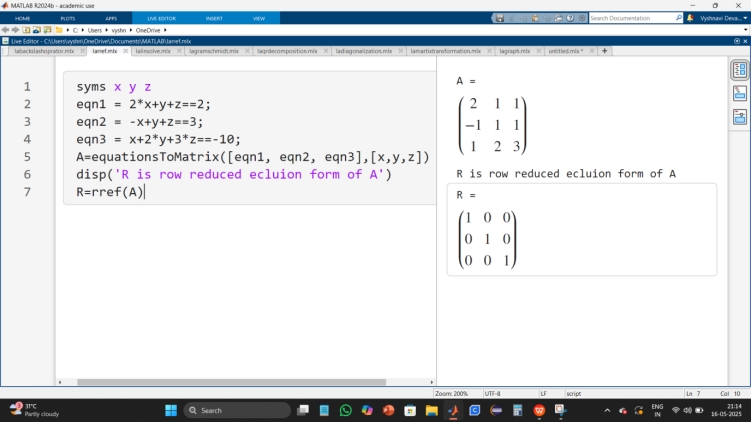


EXERCISE - 3

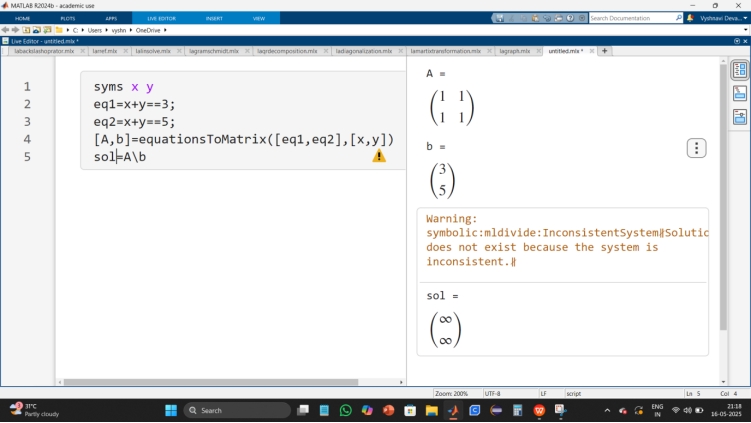
AIM : to convert system of linear equations to matrix form and applying row reduced ecluion form using the MATLAB command rref to find solution of system of linear equations and using MATLAB backlash operator command to solve the system of linear equations.

DATE :

1. Converting equations to matrix form and applying row reduced ecluion form to solve the matrix using the MATLAB command rref.



1. Converting equations to matrix form and using MATLAB backlash operator command to solve the two system of linear equations with two variables.



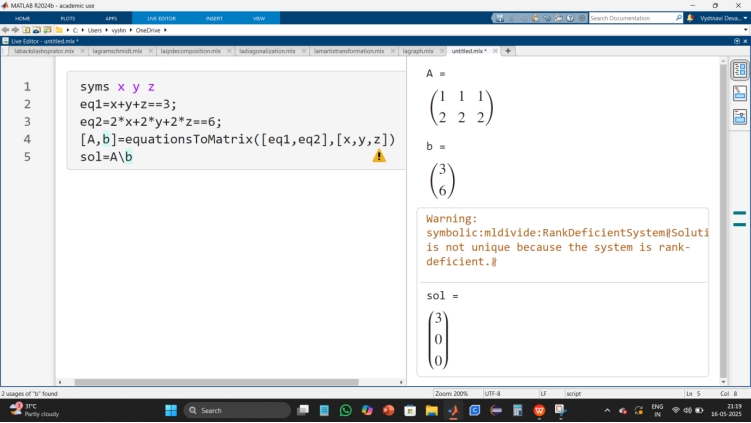
Description :  
1) [A, b] = equationsToMatrix([eq1, eq2], [x, y])

converts the system of equations into matrix form. ( A ) is the coefficient matrix and ( b ) is the constant matrix.

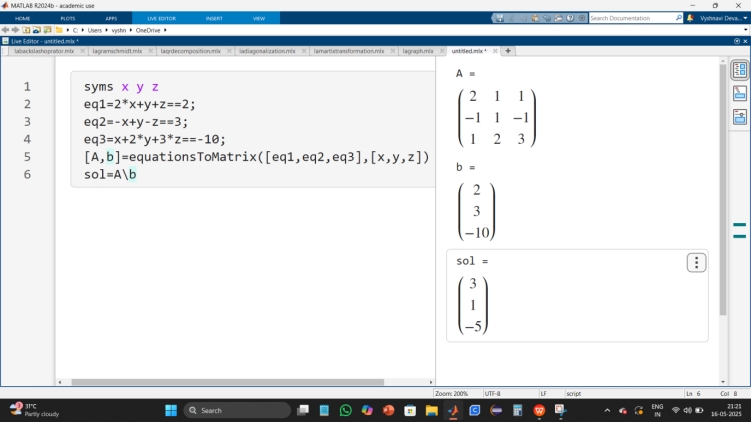
2) sol = A \ b

to solve the system using matrix division.

1. Converting equations to matrix form and using MATLAB backlash operator command to solve the two system of linear equations with three variables.



1. Converting equations to matrix form and using MATLAB backlash operator command to solve the three system of linear equations with three variables.



EXERCISE - 4

AIM : To plot 2D and 3D vector graphs.

DATE :

1. To plot 2D vector graph. Where v1 = [3, 4] and v2 = [-2, 5].

Input:   
% 2d-vector graph

%define vectors

origin=[0,0];

v1=[3,4];

v2=[-2,5];

%plot vectors

quiver(origin(1),origin(2),v1(1),v1(2),0,'r','LineWidth',2);

hold on;

quiver(origin(1),origin(2),v2(1),v2(2),0,'b','LineWidth',2);

%set axis limits

xlim([-5,5]);

ylim([-5,5]);

%labeling

grid on;

xlabel('x-axis');

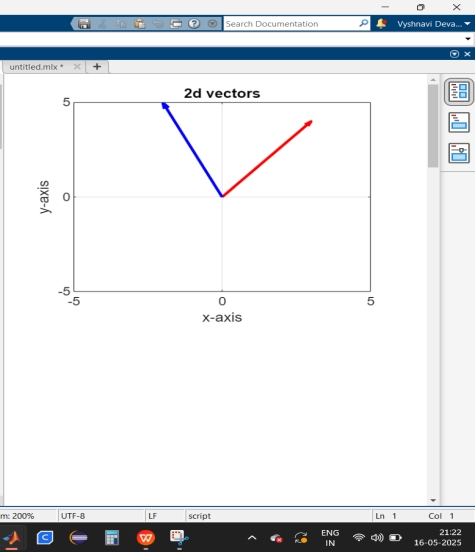
ylabel('y-axis');

title('2d vectors');

legend('vector v1','vector v2');

hold off;

Output:



Description :

1. The quiver() function is used to plot vectors. The parameters are:

origin(1), origin(2): The starting point of the vector.

v1(1), v1(2): The components of the first vector.

0: No automatic scaling.

'r': Red color for the first vector.

'LineWidth',2: Specifies the thickness of the vector.

quiver() is called again to plot the second vector (v2) in blue ('b').

1. xlim([-5,5]); ylim([-5,5]); ensures the graph has appropriate axis limits for visualization.

grid on; enables the grid.

xlabel('x-axis'); ylabel('y-axis'); labels the axes.

title('2d vectors'); sets the graph title.

%legend('vector v1','vector v2'); (commented out) would add a legend labeling the two vectors.

1. hold on; allows multiple plots on the same figure.

hold off; ensures the plotting stops after the vectors are drawn.

1. To plot 3D vector graph. Where v1 = [3, 4, 2] and v2 = [-2, 5, -3].

Input:   
% 3d-vector graph

%define vectors

origin=[0,0,0];

v1=[3,4,2];

v2=[-2,5,-3];

%plot vectors

quiver3(origin(1),origin(2),origin(3),v1(1),v1(2),v1(3),0,'r','LineWidth',2);

hold on;

quiver3(origin(1),origin(2),origin(3),v2(1),v2(2),v2(3),0,'b','LineWidth',2);

%set axis limits

xlim([-5,5]);

ylim([-5,5]);

zlim([-5,5]);

%labeling

grid on;

xlabel('x-axis');

ylabel('y-axis');

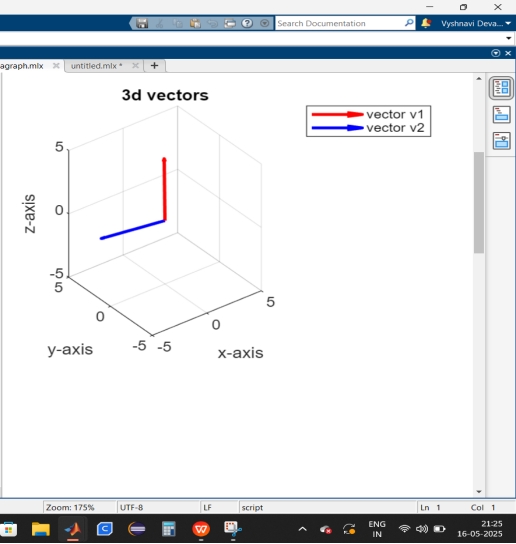
zlabel('z-axis')

title('3d vectors');

legend('vector v1','vector v2');

hold off;

Output:



EXERCISE - 5

AIM : Spanning set of two 2D vectors and 3D vectors.

DATE :

1. Spanning set of two 2D vectors. Where v1 = [3, 1] and v2 = [1, 2].

Input:   
%span(k1v1+k2v2=v)

%define vectors

v1=[3,1];

v2=[1,2];

origin=[0,0];

%grid of scalar coefficients for linear combination

[k1,k2]=meshgrid(-5:0.5:5);

%spanning set

X=k1\*v1(1)+k2\*v2(1);

Y=k1\*v1(2)+k2\*v2(2);

%plot the spanning set

%figure

scatter(X(:),Y(:),10,'filled','MarkerFaceAlpha',0.5);

hold on;

%plot original vector

quiver(origin(1),origin(2),v1(1),v1(2),0,'r','LineWidth',2);

quiver(origin(1),origin(2),v2(1),v2(2),0,'b','LineWidth',2);

%set axis limits

xlim([-20,20]);

ylim([-20,20]);

%labeling

grid on;

xlabel('x-axis');

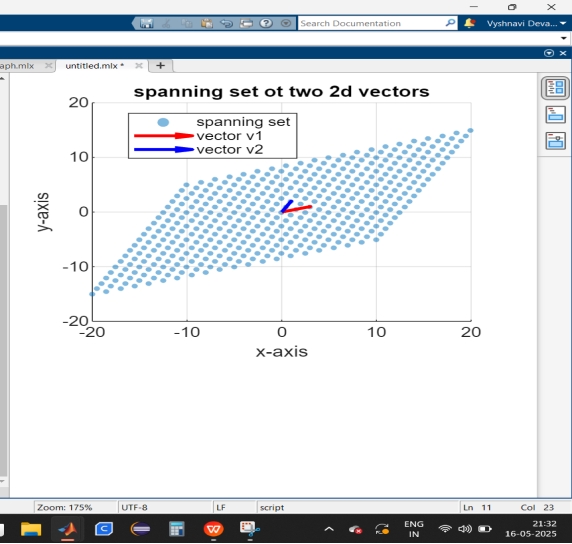
ylabel('y-axis');

title('spanning set ot two 2d vectors');

legend('spanning set','vector v1','vector v2');

hold off;

Output:



Description :

1. [k1,k2] = meshgrid(-5:0.5:5); creates a grid of scalar values ranging from -5 to 5 in steps of 0.5.

X = k1\*v1(1) + k2\*v2(1); computes the x-coordinates for the linear combinations.

Y = k1\*v1(2) + k2\*v2(2); computes the corresponding y-coordinates.

1. scatter(X(:),Y(:),10,'filled','MarkerFaceAlpha',0.5); plots the spanning set as scattered points to visualize all possible linear combinations.
2. quiver(origin(1),origin(2),v1(1),v1(2),0,'r','LineWidth',2); plots v1 in red.

quiver(origin(1),origin(2),v2(1),v2(2),0,'b','LineWidth',2); plots v2 in blue.

1. xlim([-20,20]); ylim([-20,20]); defines the visible range of the plot.

grid on; enables grid lines.

xlabel('x-axis'); ylabel('y-axis'); labels the axes.

title('spanning set of two 2D vectors'); sets the plot title.

legend('spanning set','vector v1','vector v2'); adds a legend.

1. hold on; allows multiple plots on the same figure.

hold off; ensures that further plots do not interfere with this visualization.

1. Spanning set of two 3D vectors. Where v1 = [3, 1, 2] and v2 = [1, 4, -1].

Input:

% Define vectors

v1 = [3, 1, 2];

v2 = [1, 4, -1];

origin = [0, 0, 0];

% Grid of scalar coefficients for linear combination

[k1, k2] = meshgrid(-2:0.5:2);

% Spanning set

X = k1 \* v1(1) + k2 \* v2(1);

Y = k1 \* v1(2) + k2 \* v2(2);

Z = k1 \* v1(3) + k2 \* v2(3);

% Plot the spanning set

figure;

surf(X, Y, Z, 'FaceAlpha', 0.5, 'EdgeColor', 'none');

hold on;

% Plot original vectors

quiver3(origin(1), origin(2), origin(3), v1(1), v1(2), v1(3), 'r', 'LineWidth', 2);

quiver3(origin(1), origin(2), origin(3), v2(1), v2(2), v2(3), 'b', 'LineWidth', 2);

% Set axis limits

xlim([-10, 10]);

ylim([-10, 10]);

zlim([-10, 10]);

% Labeling

grid on;

xlabel('x-axis');

ylabel('y-axis');

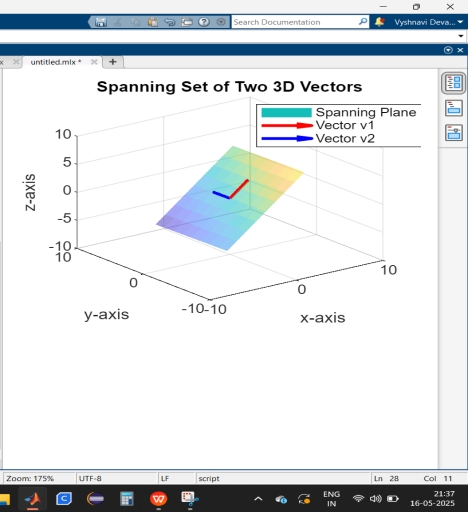
zlabel('z-axis');

title('Spanning Set of Two 3D Vectors');

legend('Spanning Plane', 'Vector v1', 'Vector v2');

hold off;

Output:



EXERCISE - 6

AIM : Gram schmidth orthogonalization in R^3 and R^4.

DATE :

1. Gram schmidth othogonalization in R^3. where v1=[1,1,0], v2=[0,1,1] and v3=[1,0,1]

Input:   
 % gram schmidth orthogonalization in R^3

clc;

clear all;

close all;

v1 = [1; 1; 0];

v2 = [0; 1; 1];

v3 = [1; 0; 1];

A = [v1 v2 v3];

fprintf('Original basis vectors:');

disp(A);

[m, n] = size(A);

Q = zeros(m, n);

% Gram-Schmidt Orthogonalization

for k = 1:n

v = A(:, k);

for j = 1:k-1

q = Q(:, j);

v = v - (q' \* v) / (q' \* q) \* q; % Subtract projection

end

Q(:, k) = v;

end

fprintf('Orthogonal basis vectors:');

disp(Q);

fprintf('Orthogonality check (should be diagonal):');

orth\_check = Q' \* Q;

disp(orth\_check);

% Normalize to get orthonormal basis

Q\_normalized = Q ./ vecnorm(Q);

fprintf('Orthonormal basis vectors:');

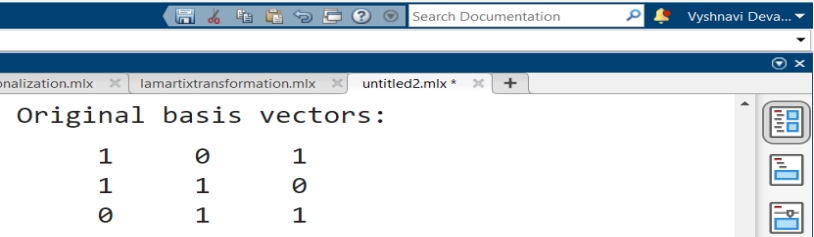
disp(Q\_normalized);

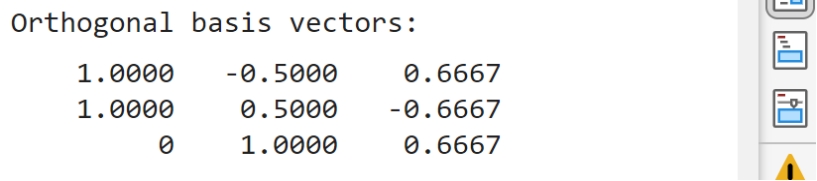
fprintf('Orthonormality check (should be identity):');

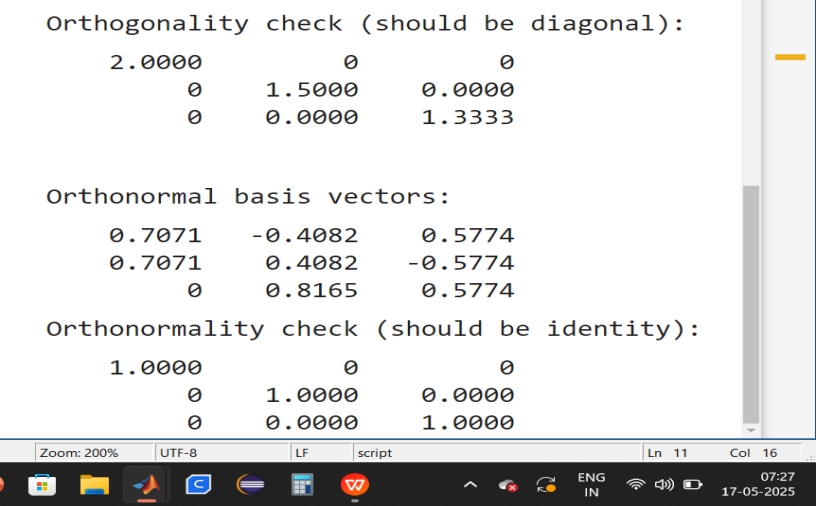
orthonorm\_check = Q\_normalized' \* Q\_normalized;

disp(orthonorm\_check);

Output:







Description :

1. clc; clears the command window.
2. clear all; removes all variables from memory.
3. close all; closes all figure windows.
4. v1 = [1; 1; 0];   
    v2 = [0; 1; 1];   
    v3 = [1; 0; 1];   
    A = [v1 v2 v3];

Three basis vectors v₁, v₂, v₃ are defined in R³.

The matrix A stores these vectors as columns.

1. [m, n] = size(A);   
    Q = zeros(m, n);

m and n store the number of rows and columns of A.

Q is initialized as a zero matrix to store the orthogonalized vectors.

1. for k = 1:n v = A(:, k);   
    for j = 1:k-1 q = Q(:, j);  
    v = v - (q' \* v) / (q' \* q) \* q;   
    end  
    Q(:, k) = v;   
    end

Looping over each column vector v\_k in A.

The inner loop subtracts the projection of v\_k onto the previously computed orthogonal vectors (Q(:, j)).

This ensures each new vector is perpendicular to all previous vectors.

The resulting orthogonal vectors are stored in Q.

1. orth\_check = Q' \* Q;  
    disp(orth\_check);

The dot product matrix Q' \* Q should be diagonal (since orthogonal vectors have zero dot products).

1. Q\_normalized = Q ./ vecnorm(Q);

Each vector in Q is divided by its norm to make it unit-length.

The resulting matrix Q\_normalized contains orthonormal basis vectors.

1. orthonorm\_check = Q\_normalized' \* Q\_normalized;  
    disp(orthonorm\_check);

The dot product matrix of Q\_normalized should be identity, confirming orthonormality.

1. Gram schmidth othogonalization in R^4. where v1=[0,2,1,0], v2=[1,-1,0,0], v3=[1,2,0,-1] and v4=[1,0,0,1].

Input:

% Gram-Schmidt Orthogonalization in R^4

clc;

clear all;

close all;

% Define basis vectors for R^4

v1 = [0; 2; 1; 0];

v2 = [1; -1; 0; 0];

v3 = [1; 2; 0; -1];

v4 = [1; 0; 0; 1];

A = [v1 v2 v3 v4];

fprintf('Original basis vectors:');

disp(A);

% Initialize matrix for orthogonal basis

[m, n] = size(A);

Q = zeros(m, n);

% Gram-Schmidt Orthogonalization

for k = 1:n

v = A(:, k);

for j = 1:k-1

q = Q(:, j);

v = v - (q' \* v) / (q' \* q) \* q; % Subtract projection

end

Q(:, k) = v;

end

fprintf('Orthogonal basis vectors:');

disp(Q);

% Check orthogonality (should be diagonal)

fprintf('Orthogonality check (should be diagonal):');

orth\_check = Q' \* Q;

disp(orth\_check);

% Normalize to get orthonormal basis

Q\_normalized = Q ./ vecnorm(Q);

fprintf('Orthonormal basis vectors:');

disp(Q\_normalized);

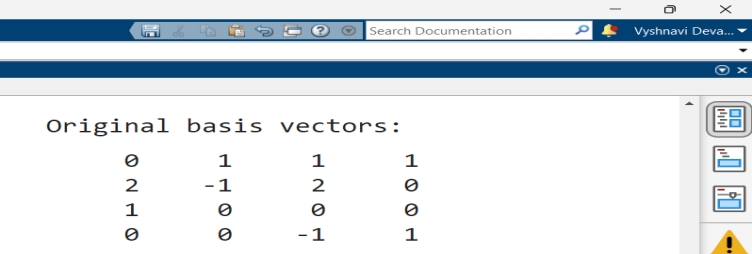
% Check orthonormality (should be identity)

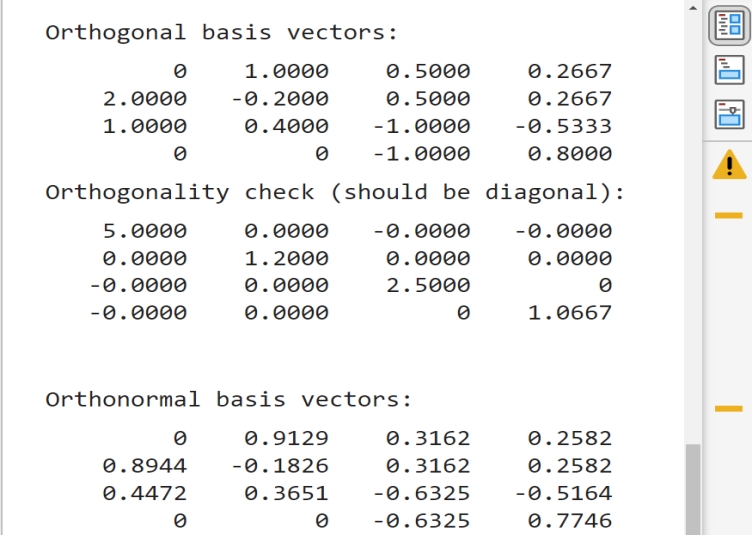
fprintf('Orthonormality check (should be identity):');

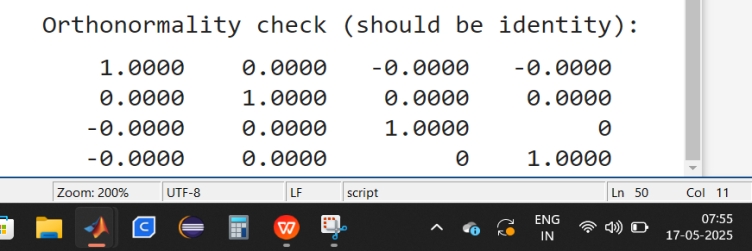
orthonorm\_check = Q\_normalized' \* Q\_normalized;

disp(orthonorm\_check);

Output:







EXERCISE - 7

AIM : QR decomposition in R^3. (A = Q\* R)

DATE :

1. QR decomposition in R^3. where v1 = [1; 1; 0], v2 = [0; 1; 1], v3 = [1; 0; 1].

Input:

% QR decomposition

clc;

clear all;

close all;

% Original basis vectors

v1 = [1; 1; 0];

v2 = [0; 1; 1];

v3 = [1; 0; 1];

A = [v1 v2 v3];

fprintf('Original basis vectors:');

disp(A);

[m, n] = size(A);

Q = zeros(m, n);

R = zeros(n, n);

% Gram-Schmidt Orthogonalization (Modified for Stability)

for k = 1:n

v = A(:, k);

for j = 1:k-1

R(j, k) = Q(:, j)' \* A(:, k); % Projection coefficients

v = v - R(j, k) \* Q(:, j); % Subtract projection

end

R(k, k) = norm(v); % Normalization factor

Q(:, k) = v / R(k, k); % Normalize to get orthonormal Q

end

fprintf('Orthonormal basis vectors Q:');

disp(Q);

fprintf('Upper triangular matrix R:');

disp(R);

fprintf('QR Decomposition Check (A ≈ Q\*R):');

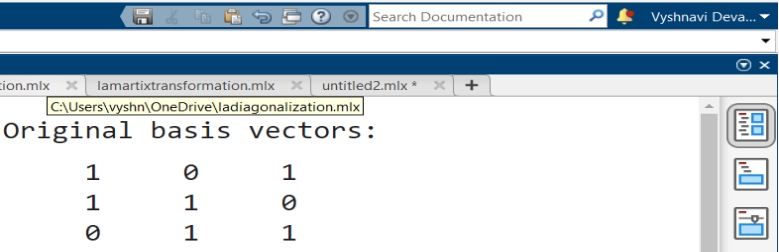
disp(Q \* R); % Should equal A

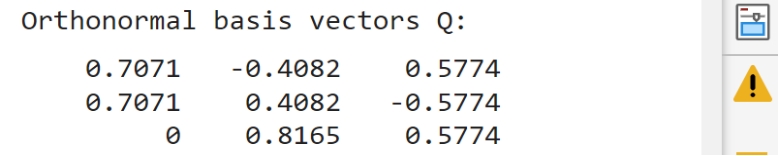
fprintf('Orthonormality check (Q^T \* Q ≈ I):');

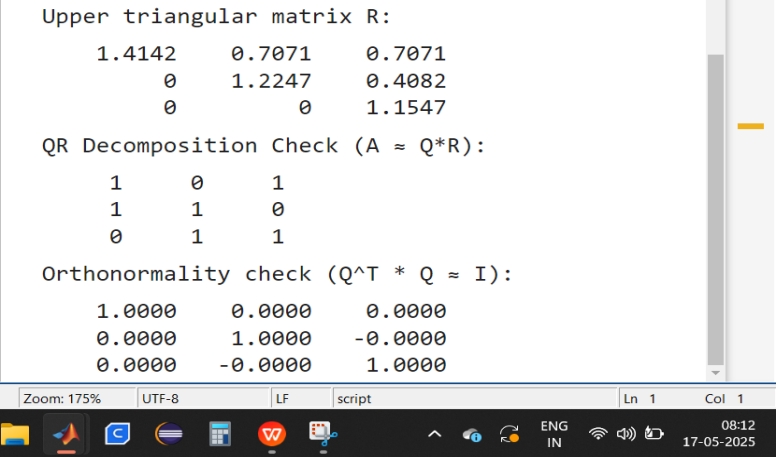
orthonorm\_check = Q' \* Q;

disp(orthonorm\_check);

Output:







Description :

1. clc; clears the command window.
2. clear all; removes all variables from memory.
3. close all; closes all figure windows.
4. v1 = [1; 1; 0];

v2 = [0; 1; 1];

v3 = [1; 0; 1];

A = [v1 v2 v3];

Three independent vectors v₁, v₂, v₃ in R³ are defined.

The matrix A stores these vectors as columns.

1. [m, n] = size(A);

Q = zeros(m, n);

R = zeros(n, n);

m, n store the dimensions of A.

Q (orthonormal matrix) and R (upper triangular matrix) are initialized with zeros.

1. for k = 1:n

v = A(:, k);

for j = 1:k-1

R(j, k) = Q(:, j)' \* A(:, k);

v = v - R(j, k) \* Q(:, j);

end

R(k, k) = norm(v);

Q(:, k) = v / R(k, k);

end

The outer loop iterates over each column of A to build the Q and R matrices.

The inner loop removes components of the current vector that are aligned with previously computed vectors.

R(j, k) stores the projection coefficients.

The final step normalizes Q(:, k) so it becomes unit-length.

1. fprintf('Orthonormal basis vectors Q:');

disp(Q);

Displays the orthonormal basis vectors obtained from Q.

1. fprintf('Upper triangular matrix R:');

disp(R);

Displays the upper triangular matrix R, which stores the coefficients from the Gram-Schmidt process.

1. fprintf('QR Decomposition Check (A ≈ Q\*R):');

disp(Q \* R);

Since QR decomposition states A = Q \* R, this verifies the accuracy.

1. fprintf('Orthonormality check (Q^T \* Q ≈ I):');

orthonorm\_check = Q' \* Q;

disp(orthonorm\_check);

Q' \* Q should be close to the identity matrix (confirming proper orthonormalization).

1. QR decomposition in R^3 using the formulas we discussed in the class. where

v1 = [1; 1; 0], v2 = [0; 1; 1], v3 = [1; 0; 1].

Input:

clc;

clear all;

close all;

v1 = [1; 1; 1];

v2 = [0; 1; 1];

v3 = [0; 0; 1];

A = [v1 v2 v3];

fprintf('Original basis vectors:');

disp(A);

% Get dimensions

[m, n] = size(A);

Q = zeros(m, n);

% Gram-Schmidt Orthogonalization

for k = 1:n

v = A(:, k); % Current vector

for j = 1:k-1

q = Q(:, j); % Previous orthogonal vector

proj = (q' \* v) / (q' \* q) \* q;

v = v - proj; % Subtract projection

end

Q(:, k) = v; % Store orthogonal vector

end

fprintf('Orthogonal basis vectors:');

disp(Q);

% Orthogonality check

fprintf('Orthogonality check (should be diagonal):');

orth\_check = Q' \* Q;

disp(orth\_check);

% Normalize to get orthonormal basis

Q\_normalized = zeros(m, n);

for k = 1:n

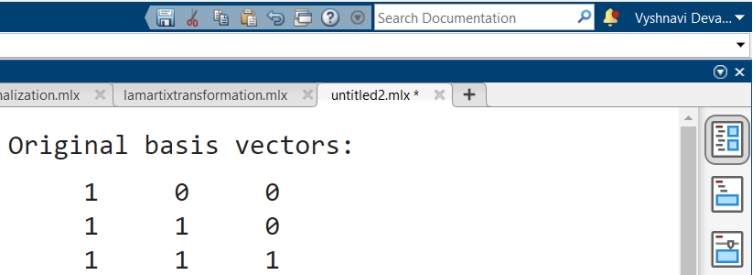
Q\_normalized(:, k) = Q(:, k) / norm(Q(:, k)); % Normalize each vector

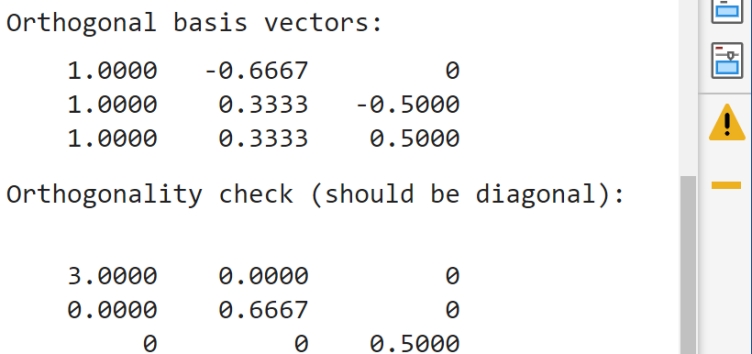
end

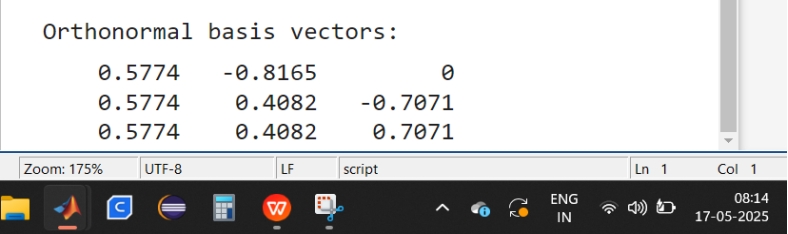
fprintf('Orthonormal basis vectors:');

disp(Q\_normalized);

Output:







Description :

1. clc; clears the command window.
2. clear all; removes all variables from memory.
3. close all; closes all figure windows.
4. v1 = [1; 1; 1];

v2 = [0; 1; 1];

v3 = [0; 0; 1];

A = [v1 v2 v3];

Three independent vectors v₁, v₂, v₃ in R³ are defined.

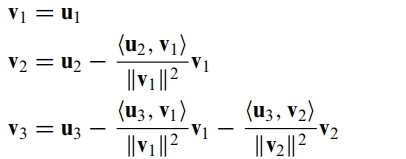
The matrix A stores these vectors as columns.

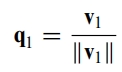
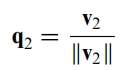
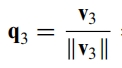
1. [m, n] = size(A);

Q = zeros(m, n);

m, n store the dimensions of A.

Q (orthonormal matrix) are initialized with zeros.

1. Orthogonal vector  
    
2. Orthonormal vector

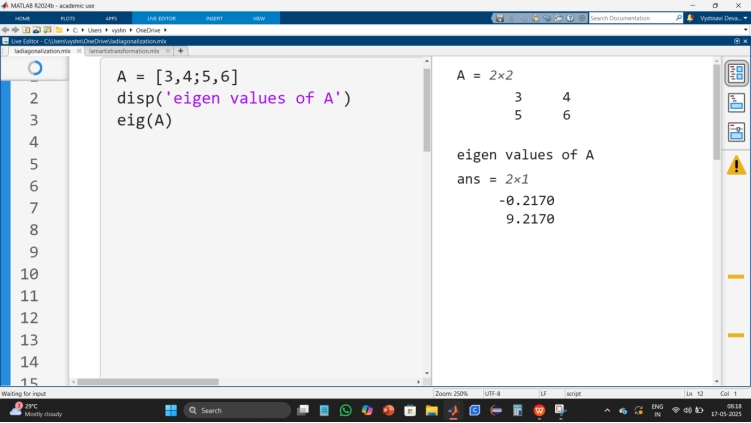
  

EXERCISE - 8

AIM : (Diagonalization ) to find the diagonal matrix (D = P^-1 \*A \*P).

DATE :

1. Finding the eigen values of the matrix A



1. Finding the diagonal matrix if the matrix is diagonalizable.

Input:

clc;

clear;

A = [4 2 2; 2 4 2; 2 2 4]

[m n]=size(A);

[p,d]=eig(A);

eigenvalues=diag(d);

disp('eigen values of A :');

disp(p);

if rank(p)==m

disp("A is diagonalizable");

disp("p :");

disp(p);

disp("p inverse :");

p\_inv=inv(p);

disp(p\_inv);

diagonal\_matrix=p\_inv\*A\*p;

disp("diagonal matrix :");

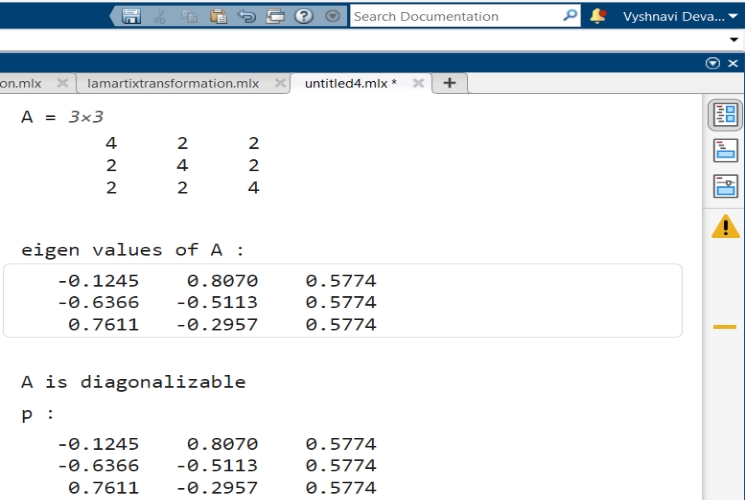
disp(diagonal\_matrix);

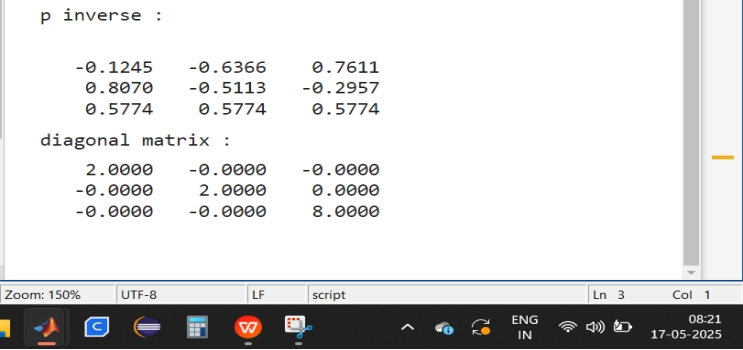
else

disp("matrix not diagonalizable(not linearly independent)");

end

Output:





Description :

1. clc;

clear;

Clears the command window and removes previous variables from memory.

1. [p, d] = eig(A);

computes the eigenvectors and find the eigenvalues of matrix A.

p: Contains the eigenvectors as columns.

d: Is a diagonal matrix where diagonal elements are the eigenvalues.

1. rank(p) == m:

This checks if the eigenvectors in p are linearly independent.

If rank(p) = m (number of rows), the eigenvectors form a basis, meaning A is diagonalizable.

1. If A is diagonalizable:

The inverse of p is computed: p\_inv = inv(p).

1. If A is Not Diagonalizable

If rank(p) ≠ m, the eigenvectors are not linearly independent, meaning A is not diagonalizable.

1. Write a MATLAB code to orthogonally diagonalize the symmetric matrix A = [5 1 0; 1 3 2; 0 2 4].

Input:

clc;

clear;

A = [5 1 0; 1 3 2; 0 2 4]

[m, n] = size(A);

[p, d] = eig(A);

eigenvalues = diag(d);

disp('Eigenvalues of A:');

disp(eigenvalues);

if rank(p)==m

disp("A is diagonalizable");

disp("p");

disp(p);

disp("p transpose");

p\_transpose=p';

disp(p');

diagonal\_matrix=p\_transpose\*A\*p;

disp("diagonal matrix");

disp(diagonal\_matrix);

else

disp("matrix not diagonalizable(not linearly independent)");

end

Description :

1. Clears the MATLAB command window (clc) and removes all variables from the workspace (clear) to ensure a clean environment for execution.
2. [m, n] = size(A);

since A is square (m=n=3) , A is vali for diagonalization.

1. [p, d] = eig(A);

computes eigen values of A

1. eigenvalues = diag(d);

extracts the eigen values into vector

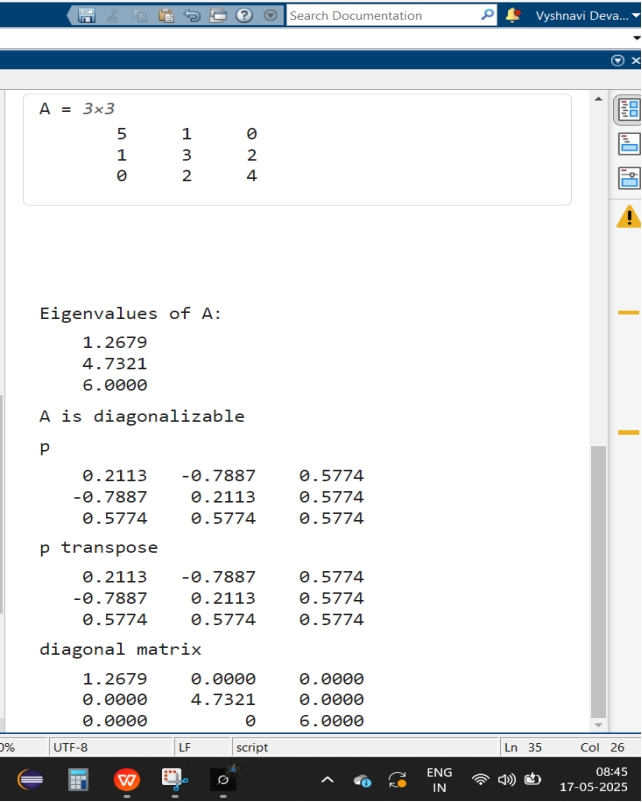
1. if rank(p)==m

diagonal\_matrix = p\_transpose \* A \* p;

compute the diagonal matrix p\_transpose \* A \* p

1. if rank(p)!=m

matrix not diagonalizable(not linearly independent)

Output:  


EXERCISE - 9

AIM : Matrix transformation

DATE :

1. Matrix transformation of unit circle.

Input:

A = [1 2; 3 1];

% create unit circle

theta = linspace(0, 2\*pi, 100);

x = cos(theta);

y = sin(theta);

circle = [x; y];

% apply transformation

transformed = A\*circle;

% plot = for data | flpot = used for function

figure;

subplot(1, 2, 1); % row no, col num, index

plot(circle(1,:), circle(2,:)); % 1st row, 2nd row

title('unit circle');

axis equal; % scaling (make sure (1 unit of x & y are equal))

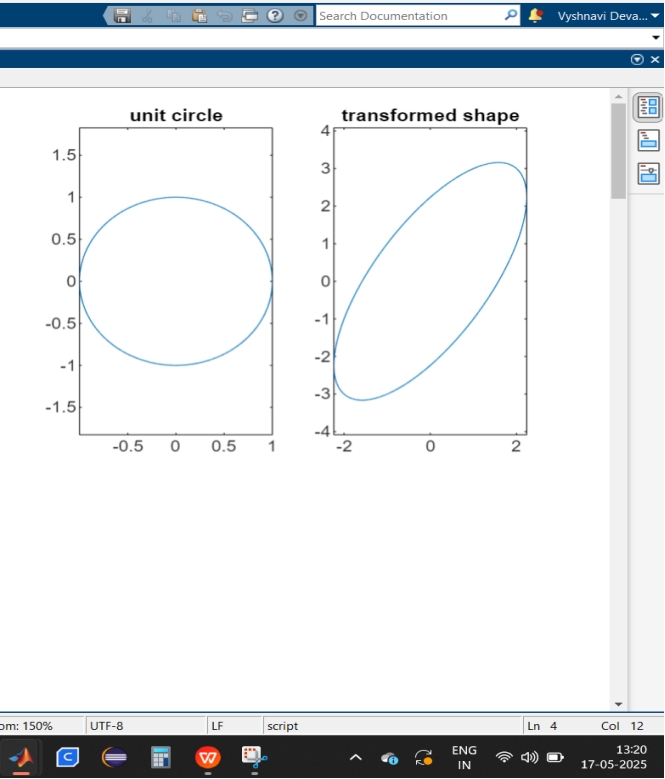
subplot(1, 2, 2);

plot(transformed(1,:), transformed(2,:));

title('transformed shape'); % ellipse

axis equal;

Output:



Description :

1. theta = linspace(0, 2\*pi, 100);

x = cos(theta);

y = sin(theta);

circle = [x; y];

Generates points along a unit circle using cosine and sine functions.

1. linspace(0, 2\*pi, 100): Creates 100 equally spaced points between 0 and 2π.

circle = [x; y]: Stores x and y coordinates of the unit circle.

transformed = A \* circle;

Multiplies the unit circle points by matrix A.

1. subplot(1, 2, 2);

plot(transformed(1,:), transformed(2,:));

title('transformed shape');

axis equal;

Creates a subplot where the unit circle is displayed.

axis equal: Ensures equal scaling for x and y axes.

1. subplot(1, 2, 2);

plot(transformed(1,:), transformed(2,:));

title('transformed shape');

axis equal;

The transformed shape is likely an ellipse due to the linear transformation.

1. Matrix transformation of unit circle (shape expansion with uniform units (no rotation)).

Input:

A = [2 0; 0 2];

theta = linspace(0, 2\*pi, 100);

x = cos(theta);

y = sin(theta);

circle = [x; y];

transformed = A\*circle;

figure;

subplot(1, 2, 1);

plot(circle(1,:), circle(2,:));

title('unit circle');

axis equal;

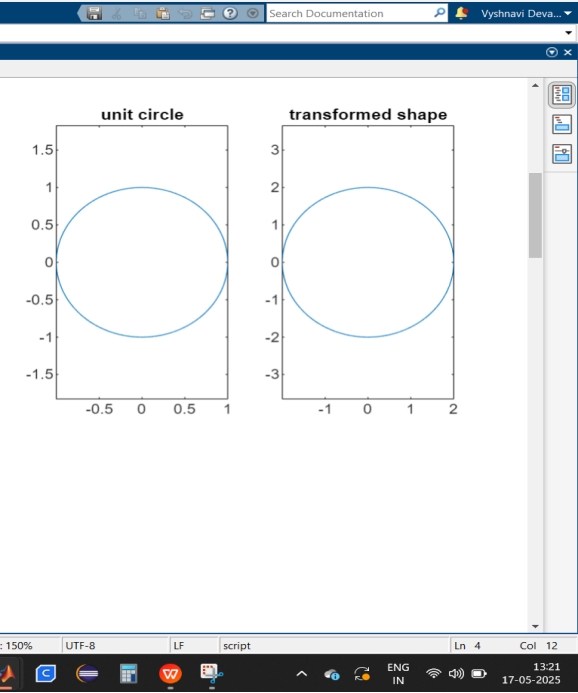
subplot(1, 2, 2);

plot(transformed(1,:), transformed(2,:));

title('transformed shape');

axis equal;

Output:



1. Matrix transformation of unit circle (shape expansion with non uniform units (no rotation)).

Input:

A = [3 0; 0 2];

theta = linspace(0, 2\*pi, 100);

x = cos(theta);

y = sin(theta);

circle = [x; y];

transformed = A\*circle;

figure;

subplot(1, 2, 1);

plot(circle(1,:), circle(2,:));

title('unit circle');

axis equal;

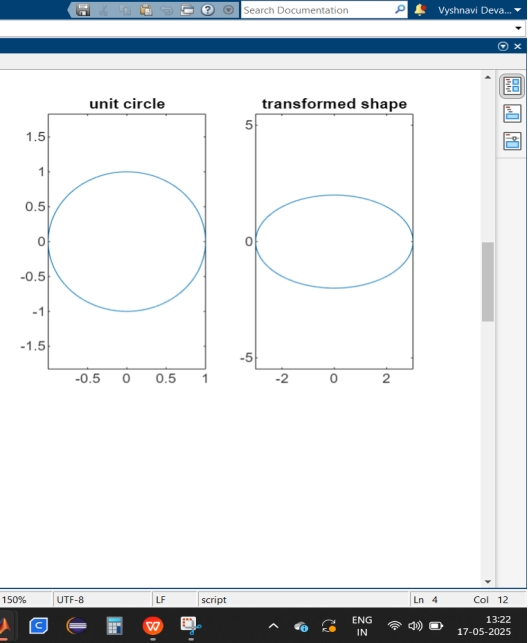
subplot(1, 2, 2);

plot(transformed(1,:), transformed(2,:));

title('transformed shape'); % ellipse

axis equal;

Output:



1. Matrix transformation of square.

Input :

A = [1 0.5; -0.5 1];

square = [0 1 1 0 0; 0 0 1 1 0];

transformed = A\*square;

figure;

subplot(1, 2, 1);

plot(square(1,:), square(2,:), 'b-', 'LineWidth', 2);

title('original square');

axis equal;

grid on;

subplot(1, 2, 2);

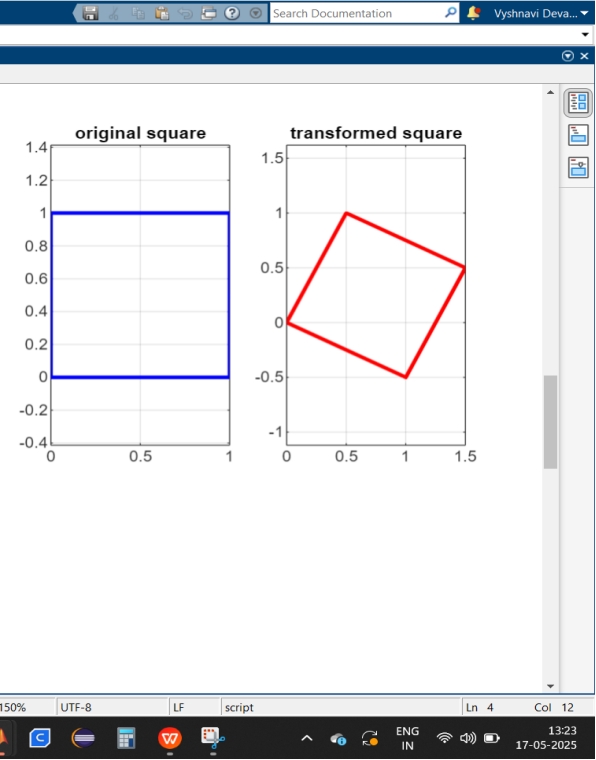
plot(transformed(1,:), transformed(2,:), 'r-', 'LineWidth', 2);

title('transformed square');

axis equal;

grid on;

Output :



Description :

1. square = [0 1 1 0 0; 0 0 1 1 0];

The first row (`[0 1 1 0 0]`) represents x-coordinates of the square.

The second row (`[0 0 1 1 0]`) represents y-coordinates.

This forms a closed square moving through points (0,0) → (1,0) → (1,1) → (0,1) → (0,0).

1. transformed = A \* square;

The transformation matrix A modifies the x and y coordinates of the square.

This results in a sheared and slightly rotated version of the square.

1. subplot(1, 2, 1): Creates the first plot for the original square.
2. axis equal: Ensures correct scaling of x and y axes.
3. subplot(1, 2, 2): Creates the second plot for the transformed square.