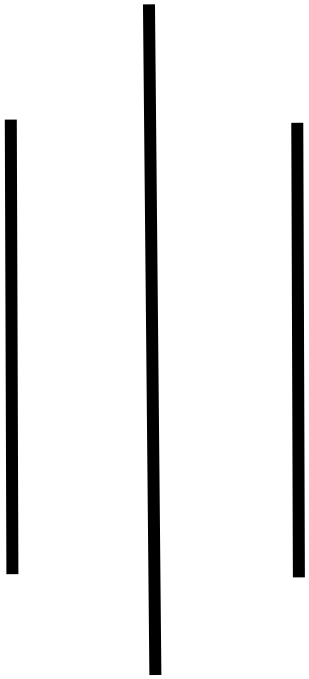
Assignment – III

Linear Algebra and Matrix Theory



Submitted by: Bikram Giri

Submitted to: Samriddha Pathak

Submission Date: 2025/07/24

Table of Contents

I. Cover Page ……. 1

II. Table of Contents ……. 2

III. Problem Statement …… 3

1. Definition & Verification of Linear Transformation
2. NumPy Function to Check Linearity of a Matrix
3. System Consistency via Matrix Rank
4. Solving a System using np.linalg.solve and np.linalg.lstsq
5. Matrix Operations: Transpose, Inverse, Determinant (NumPy)
6. LU Decomposition with SciPy and Interpretation
7. Solving Overdetermined System using QR Decomposition
8. Geometric Interpretation of Linear Systems with Visualization
9. Importance of Matrix Invertibility in Linear Systems
10. Classify System Solutions using Python and NumPy
11. Basis of Vector Space vs Column Space (with Example)
12. Check Basis in $\mathbb{R}^3$ using NumPy Vectors
13. Rank-Nullity Theorem with Example Matrix
14. Compute and Explain Rank of a Matrix
15. Proof: Rank Equals Number of Pivot Columns
16. Create a 3×3 Rank-1 Matrix & Verify with NumPy
17. Generate Random 4×4 Matrices and Count Full Rank
18. Prove Independent Vectors Form Basis in $\mathbb{R}^n$ (Numerical Check)
19. Rank & Basis Columns for a Given Matrix
20. Python Script: Find Basis for Column Space of Any Matrix

IV. Conclusion …….. 28

1. Define a linear transformation. Show whether the function T(x,y)=(2x,3y) is a linear transformation by checking the two required properties.

Ans: A linear transformation is a function T: $\mathbb{R}^n \rightarrow \mathbb{R}^m$ that satisfies the following two properties for all vectors u, v $\in \mathbb{R}^n$ and all scalars c $\in \mathbb{R}$:

1. Additivity (Preservation of Vector Addition):
   T(u + v) = T(u) + T(v)
2. Homogeneity (Preservation of Scalar Multiplication):
   T(cu) = cT(u)

Check if T(x, y) = (2x, 3y) is a Linear Transformation

Let's verify the two properties:

<u>Additivity:</u>

Let
$u = (x_1, y_1)$,
$v = (x_2, y_2)$

Then,
$T(u + v) = T(x_1 + x_2, y_1 + y_2)$
$= (2(x_1 + x_2), 3(y_1 + y_2))$
$= (2x_1 + 2x_2, 3y_1 + 3y_2)$

Now,
$T(u) = (2x_1, 3y_1)$
$T(v) = (2x_2, 3y_2)$
$T(u) + T(v) = (2x_1 + 2x_2, 3y_1 + 3y_2)$

Since both results are equal, additivity is satisfied.

<u>Homogeneity:</u>

Let c $\in \mathbb{R}$, and u = (x, y)

Then,
T(cu) = T(cx, cy)
= (2cx, 3cy)
= c(2x, 3y)

Also,
cT(u) = c(2x, 3y)

Since both results are equal, homogeneity is satisfied.

Thus, both additivity and homogeneity properties hold, T(x, y) = (2x, 3y) is a linear transformation.

2. Write a NumPy function that checks if a given transformation matrix satisfies additivity and scalar multiplication preservation(i.e., linearity).

Ans:

```
import numpy as np

def check_linearity(T):
    # Create two sample vectors and a scalar
    u = np.array([[1], [2]])
    v = np.array([[3], [4]])
    c = 5

    # Additivity: T(u + v) == T(u) + T(v)
    left_add = T @ (u + v)
    right_add = (T @ u) + (T @ v)
    additivity = np.allclose(left_add, right_add)

    # Homogeneity: T(c * u) == c * T(u)
    left_hom = T @ (c * u)
    right_hom = c * (T @ u)
    homogeneity = np.allclose(left_hom, right_hom)

    # Final result
    if additivity and homogeneity:
        print("The transformation is linear.")
    else:
        print("The transformation is NOT linear.")

# Example usage:
```

```
T = np.array([[2, 0], [0, 3]])  # This is the matrix for T(x, y) =
(2x, 3y)
check_linearity(T)
```

Output:

```
Python/New folder/Asignment-III.py"
The transformation is linear.
```

3. Classify the following system as consistent / inconsistent and dependent / independent: x+2y=3 & 2x+4y=6. Justify your answer using matrix rank.

Ans:

```
import numpy as np

# Coefficient matrix A and augmented matrix [A|b]
A = np.array([[1, 2],
              [2, 4]])

b = np.array([[3],
              [6]])

augmented = np.hstack((A, b))

# Compute ranks
rank_A = np.linalg.matrix_rank(A)
rank_aug = np.linalg.matrix_rank(augmented)
n = A.shape[1]  # number of variables

# Classification based on rank
if rank_A == rank_aug:
    if rank_A == n:
        print("The system is consistent and independent (unique
solution).")
    else:
        print("The system is consistent and dependent (infinitely many
solutions).")
else:
    print("The system is inconsistent (no solution).")
```

Output:

```
Python/New folder/Asignment-III.py"
The system is consistent and dependent (infinitely many solutions).
```

4. Solve the system of equations using both NumPy's np.linalg.solve and np.linalg.lstsq. Compare the results: 2x+3y=8 & 5x+4y=13.

Ans:

```python
import numpy as np

# Coefficient matrix A and constant vector b
A = np.array([[2, 3],
              [5, 4]])

b = np.array([8, 13])

# Solve using np.linalg.solve (for exact solution)
try:
    solution_solve = np.linalg.solve(A, b)
    print("Solution using np.linalg.solve:", solution_solve)
except np.linalg.LinAlgError:
    print("np.linalg.solve: No unique solution.")

# Solve using np.linalg.lstsq (least squares, works for all systems)
solution_lstsq, residuals, rank, s = np.linalg.lstsq(A, b, rcond=None)
print("Solution using np.linalg.lstsq:", solution_lstsq)
```

Output:

```
Python/New folder/Asignment-III.py"
Solution using np.linalg.solve: [1. 2.]
Solution using np.linalg.lstsq: [1. 2.]
```

Comparison:

- Both methods give the same result [x = 1, y = 2] because the system is square and has a unique solution.
- np.linalg.solve() is faster and accurate for square, full-rank systems.

- np.linalg.lstsq() is more general; it also works for overdetermined or underdetermined systems.

5. Given the matrixA= [[1 2], [3 4]], compute the following using NumPy: Transpose, Inverse, Determinant, & Verify that AA^-1≈I.

Ans:

```
import numpy as np

# Define the matrix A
A = np.array([[1, 2],
              [3, 4]])

# Transpose
transpose_A = A.T
print("Transpose of A:\n", transpose_A)

# Determinant
det_A = np.linalg.det(A)
print("\nDeterminant of A:", det_A)

# Inverse (only if determinant is not 0)
if det_A != 0:
    inverse_A = np.linalg.inv(A)
    print("\nInverse of A:\n", inverse_A)

    # Verify A * A^-1 ≈ Identity matrix
    identity = np.dot(A, inverse_A)
    print("\nA * A^-1:\n", identity)
    print("\nIs A * A^-1 ≈ Identity?:", np.allclose(identity,
np.eye(2)))
else:
    print("\nMatrix A is not invertible (determinant is 0).")
```

Output:

```
Python/New folder/Asignment-III.py"
Transpose of A:
 [[1 3]
 [2 4]]

Determinant of A: -2.0000000000000004

Inverse of A:
 [[-2.   1. ]
 [ 1.5 -0.5]]

A * A^-1:
 [[1.0000000e+00 0.0000000e+00]
 [8.8817842e-16 1.0000000e+00]]

Is A * A^-1 ≈ Identity?: True
```

6. Perform LU decomposition on a 3×3 matrix using SciPy. Interpret the resulting matrices L, U, Pand describe their utility in solving linear systems.

Ans:

```python
import numpy as np
from scipy.linalg import lu

# Define a 3x3 matrix
A = np.array([[4, 3, 2],
              [2, 1, 5],
              [6, 7, 8]])

# Perform LU decomposition
P, L, U = lu(A)

print("Permutation matrix P:\n", P)
print("\nLower triangular matrix L:\n", L)
print("\nUpper triangular matrix U:\n", U)
```

Output:

```
Python/New folder/Asignment-III.py"Python\New folder>
Permutation matrix P:
 [[0. 1. 0.]
 [0. 0. 1.]
 [1. 0. 0.]]

Lower triangular matrix L:
 [[1.         0.         0.        ]
 [0.66666667 1.         0.        ]
 [0.33333333 0.8        1.        ]]

Upper triangular matrix U:
 [[ 6.          7.          8.        ]
 [ 0.         -1.66666667 -3.33333333]
 [ 0.          0.          5.        ]]
```

Utility in Solving Linear Systems:

To solve Ax=b:

1. Use LU decomposition: PA=LU
2. Rewrite as LUx=Pb
3. Solve Ly=Pb for y by forward substitution (since L is lower triangular)
4. Solve Ux=y for x by backward substitution (since U is upper triangular)

This breaks the original problem into simpler steps, improving computational efficiency and stability.

7. Solve the following over determined system using QR decomposition: A= [[1 2], [3 4], [5 6]] , b= [1 2 3]. Use NumPy to perform decomposition and back-substitution.

Ans:

```python
import numpy as np

# Given matrix A and vector b
A = np.array([[1, 2],
              [3, 4],
              [5, 6]])
b = np.array([1, 2, 3])

# Perform QR decomposition of A
```

```
Q, R = np.linalg.qr(A)

# Compute Q^T * b
Qt_b = np.dot(Q.T, b)

# Since R is upper-triangular (2x2), solve Rx = Q^T b by back-substitution
# np.linalg.solve works for square systems
x = np.linalg.solve(R, Qt_b)

print("Solution x:", x)
```

Output:

```
Python/New folder/Asignment-III.py"
Solution x: [1.50129554e-16 5.00000000e-01]
```

8. Explain the geometric interpretation of consistent and inconsistent linear systems. Create and solve one example of each using NumPy, then visualize in2D using matplotlib.

Ans:

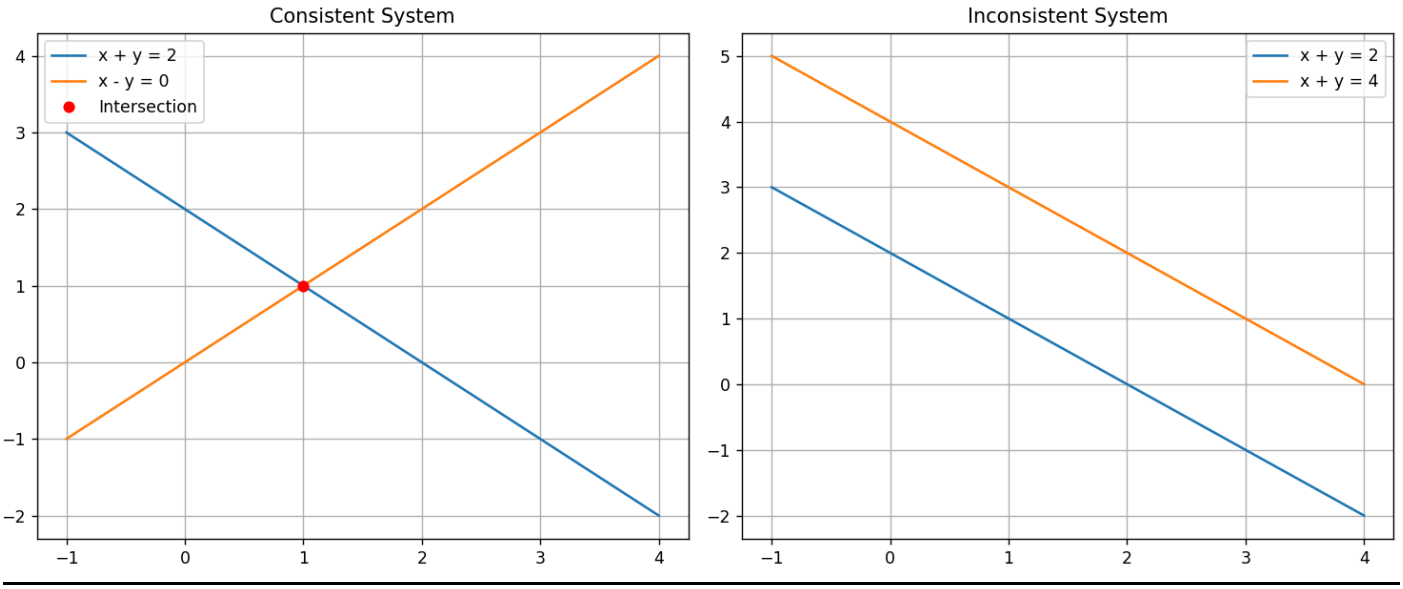
Geometric Interpretation:

- Consistent system (has at least one solution):
  In 2D, this means the lines intersect at a point or lie on top of each other (infinite solutions).
- Inconsistent system (no solution):
  The lines are parallel and distinct, so they never meet.

```
import numpy as np
import matplotlib.pyplot as plt

# Consistent system
# x + y = 2
# x - y = 0
A1 = np.array([[1, 1], [1, -1]])
```

```
b1 = np.array([2, 0])
x1 = np.linalg.solve(A1, b1)

# Inconsistent system
# x + y = 2
# x + y = 4 (parallel lines, different intercepts)
A2 = np.array([[1, 1], [1, 1]])
b2 = np.array([2, 4])
# Least-squares approach (gives best approx. even if no exact
solution)
x2, residuals, rank, s = np.linalg.lstsq(A2, b2, rcond=None)

# Plotting
x_vals = np.linspace(-1, 4, 100)
fig, axs = plt.subplots(1, 2, figsize=(12, 5))

# Plot Consistent
axs[0].plot(x_vals, 2 - x_vals, label="x + y = 2")
axs[0].plot(x_vals, x_vals, label="x - y = 0")
axs[0].plot(*x1, 'ro', label='Intersection')
axs[0].set_title("Consistent System")
axs[0].legend()
axs[0].grid(True)

# Plot Inconsistent
axs[1].plot(x_vals, 2 - x_vals, label="x + y = 2")
axs[1].plot(x_vals, 4 - x_vals, label="x + y = 4")
axs[1].set_title("Inconsistent System")
axs[1].legend()
axs[1].grid(True)

plt.tight_layout()
plt.show()
```

Output:



Output Interpretation:

- Consistent System: Lines intersect at point (1, 1) — unique solution.
- Inconsistent System: Lines are parallel (same slope, different intercepts) — no solution.

9. Why is matrix invertibility important in solving linear systems? Give an example of an on-invertible matrix and interpret the result in terms of system solutions.

Ans:

Matrix invertibility plays a crucial role in solving linear systems of the form:

$Ax=b$

where:

- A is a square matrix (n × n),
- x is the vector of unknowns,
- b is a known vector.

If Matrix A is Invertible:

You can find a unique solution to the system using:

$x=A^{-1}\,b$

This means:

- The matrix has full rank.
- All rows (and columns) are linearly independent.
- The system has exactly one solution.

If Matrix A is Not Invertible (Singular):

Then:

- A^{-1} does not exist.
- The system may have no solution (inconsistent) or infinitely many solutions (dependent).
- The matrix has linearly dependent rows/columns (i.e., determinant = 0).

Example of a Non-Invertible Matrix:

Let's consider the matrix: A=[[1 2], [2 4]], b=[3 6]

Notice that row 2 is just 2× row 1, so the matrix is singular (not invertible).

Now solve using Python:

```
import numpy as np

A = np.array([[1, 2],
              [2, 4]])
b = np.array([3, 6])

# Check determinant
det = np.linalg.det(A)

# Try solving using lstsq (since solve will fail)
x, residuals, rank, s = np.linalg.lstsq(A, b, rcond=None)

print("Determinant:", det)
print("Rank:", rank)
print("Solution (least squares):", x)
```

Output:

```
Python/New folder/Asignment-III.py"
Determinant: 0.0
Rank: 1
Solution (least squares): [0.6 1.2]
```

Interpretation:

- Determinant = 0 ⟹ matrix is non-invertible.
- Rank < number of variables ⟹ rows are linearly dependent.
- The system has infinitely many solutions (dependent system) if b lies in the column space of A, as in this case.
- If b didn't match this ratio (e.g., [3, 7]), then it would have no solution.

10. Write a Python script using NumPy that classifies a given system AX=b as:

- **Consistent with a unique solution**
- **Consistent with infinite solutions**
- **Inconsistent**

Ans:

```python
import numpy as np

def classify_system(A, b):
    A = np.array(A, dtype=float)
    b = np.array(b, dtype=float).reshape(-1, 1)

    # Augmented matrix [A | b]
    Ab = np.hstack((A, b))

    # Compute ranks
    rank_A = np.linalg.matrix_rank(A)
    rank_Ab = np.linalg.matrix_rank(Ab)
    n_vars = A.shape[1]

    # Classification
    if rank_A == rank_Ab:
        if rank_A == n_vars:
```

```
            return "Consistent with a unique solution"
        else:
            return "Consistent with infinite solutions"
    else:
        return "Inconsistent system"

# Example 1: Unique Solution
A1 = [[2, 3], [1, 2]]
b1 = [8, 5]
print("System 1:", classify_system(A1, b1))

# Example 2: Infinite Solutions
A2 = [[1, 2], [2, 4]]
b2 = [3, 6]
print("System 2:", classify_system(A2, b2))

# Example 3: Inconsistent
A3 = [[1, 2], [2, 4]]
b3 = [3, 7]
print("System 3:", classify_system(A3, b3))
```

Output:

```
Python/New folder/Asignment-III.py"
System 1: Consistent with a unique solution
System 2: Consistent with infinite solutions
System 3: Inconsistent system
```

11. Explain the difference between “basis of a vector space” and “basis of a column space” with a concrete example.

Ans:

Aspect	Basis of a Vector Space	Basis of a Column Space
Definition	A set of vectors that span the entire vector space and are linearly independent.	A set of linearly independent vectors that span the column space of a matrix.
Scope	Applies to entire vector spaces like $\mathbb{R}^2$, $\mathbb{R}^3$, etc.	Applies to the set of all linear combinations of matrix columns.
Example Space	$\mathbb{R}^2$ — the set of all 2D vectors	Column space of matrix A = \[1,2\[1, 2, 3,63, 6] (a subspace of $\mathbb{R}^2$)
Basis Example	Standard basis: {[1, 0], [0, 1]}	From A above: only one independent column → {[1, 3]} is the basis
Dimension (Rank)	Equal to the number of basis vectors for the full space (e.g., 2 in $\mathbb{R}^2$)	Equal to the number of independent columns (rank of matrix)

12. Use NumPy to check whether the following vectors form a basis for R3:

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

Ans:

```
import numpy as np

# Define the vectors
v1 = [1, 0, 0]
v2 = [0, 1, 0]
v3 = [0, 0, 1]

# Stack as columns of a matrix
A = np.column_stack((v1, v2, v3))

# Compute the rank
rank = np.linalg.matrix_rank(A)

# Check if rank is 3
if rank == 3:
    print("The vectors form a basis for R^3.")
else:
    print("The vectors do NOT form a basis for R^3.")
```

Output:

```
Python/New folder/Asignment-III.py"
The vectors form a basis for R^3.
```

13. State and explain the rank-nullity theorem. Provide a matrix example with full explanation.

Ans:

The Rank-Nullity Theorem is a fundamental result in linear algebra that relates the dimensions of the domain of a linear transformation to the dimensions of its range and kernel (null space).

If A is an m×n matrix (with n columns), then:

Rank(A) + Nullity(A) = n

Where:

- Rank(A) is the number of linearly independent columns (dimension of the column space).
- Nullity(A) is the number of free variables (dimension of the null space).
- nnn is the total number of columns in the matrix AAA.

The theorem tells us:

- Out of all variables (columns), some contribute to solutions (rank), and others can vary freely (nullity).
- The total number of variables is always the sum of these two.

Example:

Let's consider the following matrix:

A=[[1 2 3], [4 5 6]]

This is a 2×3 matrix → n = 3 (3 columns)

Step 1: Compute the Rank

```
import numpy as np

A = np.array([[1, 2, 3],
              [4, 5, 6]])

rank = np.linalg.matrix_rank(A)
print("Rank =", rank)
```

Output:

```
Python/New folder/Asignment-III.py"
Rank = 2
```

So, Rank(A) = 2

Step 2: Use Rank-Nullity Theorem

Rank(A) + Nullity(A) = 3 ⇒ 2 + Nullity(A) = 3 ⇒ Nullity(A) =1

Interpretation:

- The matrix has 2 pivot columns, meaning 2 variables are leading (basic variables).
- 1 variable is free, so the null space is 1-dimensional.
- The solution to Ax=0Ax = 0Ax=0 lies on a line in R3\mathbb{R}^3R3.

Why It Matters:

- If nullity = 0, the system has a unique solution (only the trivial solution for homogeneous).
- If nullity > 0, the system has infinitely many solutions (there are free variables).
- This theorem helps in understanding solution structures of linear systems.

14. Compute the rank of the following matrix using NumPy: A = [[1 2 3], [2 4 6], [1 1 1]]. Explain why the rank is less than 3.

Ans:

```
import numpy as np

A = np.array([[1, 2, 3],
              [2, 4, 6],
              [1, 1, 1]])

rank = np.linalg.matrix_rank(A)
print("Rank of matrix A =", rank)
```

Output:

```
Python/New folder/Asignment-III.py"
Rank of matrix A = 2
```

The matrix is a 3×3 matrix, so the maximum possible rank is 3. However, the actual rank is 2. Here's why:

1. Row 2: [2, 4, 6] = 2 × [1, 2, 3] → Linearly dependent on Row 1
2. So, only two rows contribute new information.
3. The third row, [1, 1, 1], is not a linear combination of the other two, but since one row is dependent, the rank drops from 3 to 2.

15. Prove that the rank of a matrix equals the number of pivot columns in its row echelon form. Illustrate with an example matrix.

Ans:

Theorem:

The rank of a matrix equals the number of pivot columns in its row echelon form (REF).

Definitions:

Concept	Meaning
Rank	Number of linearly independent rows/columns in a matrix.
Pivot Column	A column that contains a leading 1 (non-zero entry) in REF.
Row Echelon Form (REF)	A form of a matrix where each row has more leading zeros than the previous row, and below each pivot, all entries are zero.

Proof (Outline):

1. When a matrix is transformed to row echelon form, it retains the same row space.
2. The number of pivot positions (i.e., first non-zero entries in rows) corresponds to linearly independent rows.
3. So, the rank, defined as the dimension of the row space, equals the number of pivot columns.

Example:

Let’s take the matrix:

A = [[1 2 3], [2 4 6], [3 6 9]]

Step 1: Convert to Row Echelon Form (manually or via NumPy)

```
import numpy as np
from scipy.linalg import lu

A = np.array([[1, 2, 3],
              [2, 4, 6],
              [3, 6, 9]], dtype=float)

# Row Echelon Form via LU Decomposition
_, L, U = lu(A)

print("Row Echelon Form (U):\n", np.round(U, 2))

# Count pivot columns
rank = np.linalg.matrix_rank(A)
print("Rank of A:", rank)
```

Output:

```
Python/New folder/Asignment-III.py"
Row Echelon Form (U):
 [[3. 6. 9.]
 [0. 0. 0.]
 [0. 0. 0.]]
Rank of A: 1
```

Explanation:

- In REF, only 1 pivot in the first column.
- Columns 2 and 3 are linear combinations of column 1.
- So, the rank = 1 and 1 pivot column → They are equal.

Conclusion:

Rank of a matrix = Number of pivot columns in its Row Echelon Form, since each pivot corresponds to a linearly independent row (or column), and reflects the dimension of the matrix's row (or column) space.

16. Construct a 3×3 matrix of rank 1. Use NumPy to verify that it has only one linearly independent column.

Ans:

```
import numpy as np

# Create a rank-1 matrix where all columns are multiples of [1, 2, 3]
col = np.array([[1], [2], [3]])
A = np.hstack([col, 2*col, -5*col])

# Display the matrix
print("Matrix A:\n", A)

# Compute rank
rank = np.linalg.matrix_rank(A)
print("Rank of A:", rank)

# Check linear dependence
print("\nColumn 2 = 2 × Column 1?")
print(np.allclose(A[:, 1], 2 * A[:, 0]))

print("Column 3 = -5 × Column 1?")
print(np.allclose(A[:, 2], -5 * A[:, 0]))
```

Output:

```
Python/New folder/Asignment-III.py"Python\New folder>
Matrix A:
 [[  1   2  -5]
 [  2   4 -10]
 [  3   6 -15]]
Rank of A: 1

Column 2 = 2 × Column 1?
True
Column 3 = -5 × Column 1?
True
```

Interpretation:

Check	Result
Rank of the matrix	1
Linearly independent cols	Only Column 1
Other columns	Dependent

So the matrix has only one linearly independent column, confirming it is of rank 1.

17. Write a Python function to generate 10 random 4×4 matrices. For each, compute its rank and determine how many are full rank. Report the percentage.

Ans:

```
import numpy as np

def check_full_rank_matrices(n=10, size=4):
    full_rank_count = 0

    for i in range(n):
        matrix = np.random.randint(-10, 10, size=(size, size))
        rank = np.linalg.matrix_rank(matrix)
        print(f"Matrix {i+1}:\n{matrix}")
        print(f"Rank: {rank}\n")
        if rank == size:
            full_rank_count += 1

    percentage = (full_rank_count / n) * 100
    print(f"Full rank matrices: {full_rank_count} out of {n}")
    print(f"Percentage of full rank matrices: {percentage:.2f}%")

# Run the function
check_full_rank_matrices()
```

Output:

```
th Python/New folder/Asignment-III.py"
Matrix 1:
[[-5  9 -3  3]
 [-7  9  5  7]
 [ 5  6 -5 -8]
 [ 8  8  9 -1]]
Rank: 4

Matrix 2:
[[  0  -5   9  -4]
 [  7   0 -10   7]
 [ -5  -1  -1  -9]
 [  2  -7   7   4]]
Rank: 4

Matrix 3:
[[ 6 -6  4  4]
 [ 3 -4  9  2]
 [-9 -3  4  0]
 [ 6  3 -9  8]]
Rank: 4

Matrix 4:
[[ -1   0   9 -10]
 [  2   0   2   9]
 [ -6  -5   7   8]
 [  4   7   2   3]]
Rank: 4

Matrix 5:
[[-2  2  3 -8]
 [ 0 -3 -2  0]
 [-3  6  3 -4]
 [-9 -1  1  9]]
Rank: 4
```

```
Matrix 6:
[[ 3  8 -2  3]
 [-7 -5 -5  0]
 [-4 -3  1 -2]
 [ 9  5  3  4]]
Rank: 4

Matrix 7:
[[  6   5 -10  -6]
 [ -6   5   4   6]
 [  6  -5   8  -6]
 [ -3   2   3 -10]]
Rank: 4

Matrix 8:
[[-4  4 -8 -5]
 [-8 -9 -8  6]
 [-5  7 -5  9]
 [-9 -9  6  0]]
Rank: 4

Matrix 9:
[[  9  -1   7  -1]
 [ -1   6  -2  -9]
 [ -8  -3 -10   4]
 [  9  -6  -3   1]]
Rank: 4

Matrix 10:
[[ -8   3   5  -5]
 [ -8  -7  -6   2]
 [ -9   3  -6 -10]
 [ -3   5  -5   0]]
Rank: 4

Full rank matrices: 10 out of 10
Percentage of full rank matrices: 100.00%
```

18. Prove that any n linearly independent vectors in R^n form a basis. Verify this numerically with three vectors in R^3.

Ans:

Theoretical Proof:

Statement: Any n linearly independent vectors in R^n form a basis of R^n.

Why?

- A basis of a vector space must:
  1. Span the space.
  2. Be linearly independent.
- In R^n, a set of n linearly independent vectors will automatically span R^n, because:
  - The maximum number of linearly independent vectors in R^n is n.
  - So, if we have nnn vectors and they are independent, they must span R^n.
  - Hence, they form a basis.

Numerical Verification in Python (for R^3)

Let's verify this with 3 linearly independent vectors in R^3.

```
import numpy as np

# Define three vectors in R^3
v1 = np.array([1, 0, 0])
v2 = np.array([0, 1, 0])
v3 = np.array([0, 0, 1])

# Stack vectors column-wise into a matrix
A = np.column_stack((v1, v2, v3))

# Check rank
rank = np.linalg.matrix_rank(A)

# Display matrix and result
print("Matrix A (formed by vectors as columns):\n", A)
print("Rank of A:", rank)
```

```
if rank == 3:
    print("The vectors are linearly independent and form a basis of
R^3.")
else:
    print("The vectors do not form a basis.")
```

Output:

```
Matrix A (formed by vectors as columns):
 [[1 0 0]
 [0 1 0]
 [0 0 1]]
Rank of A: 3
The vectors are linearly independent and form a basis of R^3.
```

Conclusion

Criteria	Satisfied?
3 vectors in R^3	Yes
Linearly independent	Yes
Span R^3	Yes
Form a basis	Confirmed

19. Consider the matrix: A = [[1 0 1], [0 1 1], [0 1 1]]. Find the rank of A and determine which columns form a basis for its column space.

Ans:

Step 1: Compute the Rank using NumPy

```
import numpy as np

# Define the matrix
A = np.array([
    [1, 0, 1],
    [0, 1, 1],
    [0, 1, 1]
])

# Compute the rank
rank = np.linalg.matrix_rank(A)
print("Rank of A:", rank)
```

Output:

```
/Asignment-III.py"
Rank of A: 2
```

So, rank(A) = 2, which means only 2 columns are linearly independent**.**

Step 2: Determine Linearly Independent Columns

Let's look at the columns of A:

col_1 = [1 0 0], col_2 = [0 1 1], and col_3 = [1 1 1]

Observe: $col_3 = col_1 + col_2 \Rightarrow$ Linearly dependent on col_1 and col_2

Therefore, the linearly independent columns are: col_1, and col_2

20. Write a Python script that takes any matrix as input and outputs a set of linearly independent columns (i.e.,a basis for the column space).

Ans:

```
import numpy as np
from numpy.linalg import matrix_rank

def find_column_space_basis(A):
    """
    Returns a basis for the column space of matrix A
    by selecting linearly independent columns.
    """
    A = np.array(A, dtype=float)
    rank = matrix_rank(A)
    n_cols = A.shape[1]

    basis_columns = []
    for i in range(n_cols):
        # Add the i-th column to the current basis set
        candidate = A[:, basis_columns + [i]]
        if matrix_rank(candidate) > len(basis_columns):
            basis_columns.append(i)
```

```
        # Stop if we've collected enough independent columns
        if len(basis_columns) == rank:
            break

    basis = A[:, basis_columns]
    return basis, basis_columns

# Example matrix
A = np.array([
    [1, 0, 1],
    [0, 1, 1],
    [0, 1, 1]
])

basis, cols = find_column_space_basis(A)
print("Original Matrix:\n", A)
print("\nBasis for the Column Space:\n", basis)
print("\nIndices of Basis Columns:", cols)
```

Output:

```
/Asignment-III.py"
Original Matrix:
 [[1 0 1]
 [0 1 1]
 [0 1 1]]

Basis for the Column Space:
 [[1. 0.]
 [0. 1.]
 [0. 1.]]

Indices of Basis Columns: [0, 1]
```

Conclusion:

In this comprehensive assignment on "**Linear Algebra and Matrix Theory"**, we explored foundational and practical concepts that are essential in both theoretical mathematics and applied computational tasks. Starting from the definition and verification of linear transformations, we moved on to programmatically checking linearity properties using NumPy, analyzing systems of equations using matrix rank, and solving them using powerful numerical methods such as np.linalg.solve, np.linalg.lstsq, LU decomposition, and QR decomposition. These exercises not only strengthened the understanding of solution existence and uniqueness but also illuminated how tools like NumPy and SciPy enable efficient problem-solving in linear systems. The geometric interpretation of solutions and visualization with Matplotlib further enriched our grasp of the consistency and structure of linear systems.

Furthermore, the assignment delved into essential matrix operations such as computing the inverse, determinant, and transpose, and examined their implications in system solvability. Key theoretical constructs like the Rank-Nullity Theorem**,** matrix invertibility, and the basis of a vector space vs. column space were addressed both conceptually and computationally. Through hands-on Python scripts, we verified properties like rank, pivot columns, and linear independence, culminating in the ability to extract a basis for any matrix's column space. This blend of theory and computation underscores the real-world utility of linear algebra in data science, machine learning, and scientific computing, providing a strong foundation for further exploration and application.