Lab #4

Introduction to NumPy and Linear Algebra

Introduction

This lab provides a comprehensive introduction to NumPy, an essential Python library for numerical computations and linear algebra. NumPy is designed for efficient operations with arrays and matrices, forming the foundation for data science, machine learning, and scientific computing. This lab emphasizes linear algebra operations that are crucial for advanced mathematical computations.

Prerequisites

- Basic knowledge of Python (variables, lists, functions)
- Python environment with NumPy installed:
- `!pip install numpy`
- A code editor (VS Code, Jupyter Notebook, or Python IDE)
- Basic understanding of matrices and linear algebra concepts

Objectives

- Create and manipulate NumPy arrays for numerical computations
- Perform linear algebra operations including matrix multiplication, determinants, and inverses
- Apply advanced NumPy functions for scientific computing
- Understand broadcasting, stacking, and array manipulation techniques

Part 1: NumPy Array Fundamentals

1.1 Creating NumPy Arrays

NumPy arrays are the core data structure in NumPy. Unlike Python lists, they are homogeneous (all elements must be the same type) and optimized for numerical operations. Arrays can be one-dimensional (vectors), two-dimensional (matrices), or multi-dimensional (tensors).

Example: Creating Arrays of Different Dimensions

```
import numpy as np

# Create a 1D array from a list
array_1d = np.array([1, 2, 3, 4, 5])
print("1D Array:", array_1d)
```

```
print("Shape:", array_1d.shape)  # Output: (5,)
print("Data type:", array_1d.dtype)  # Output: int64 or int32

# Create a 2D array (3x3 matrix)
array_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("\n2D Array:\n", array_2d)
print("Shape:", array_2d.shape)  # Output: (3, 3)
print("Number of dimensions:", array_2d.ndim)  # Output: 2

# Create an array of zeros (2x3)
zeros_array = np.zeros((2, 3))
print("\nZeros Array:\n", zeros_array)

# Create an array of ones (2x2)
ones_array = np.ones((2, 2))
print("\nOnes Array:\n", ones_array)

# Create an identity matrix (3x3)
identity_matrix = np.eye(3)
print("\nIdentity Matrix:\n", identity_matrix)

# Create an array with a range of values
range_array = np.arange(0, 10, 2)  # Start, stop, step
print("\nRange Array:", range_array)  # Output: [0 2 4 6 8]

# Create an array with evenly spaced values
linspace_array = np.linspace(0, 1, 5)  # Start, stop, number of points
print("\nLinspace Array:", linspace_array)  # Output: [0.   0.25 0.5  0.75 1.
]
```

Key Points:

- **np.array()**: Converts a list or nested list into a NumPy array
- **np.zeros(shape)**: Creates an array filled with zeros
- **np.ones(shape)**: Creates an array filled with ones
- **np.eye(n)**: Creates an n×n identity matrix (1s on diagonal, 0s elsewhere)
- **np.arange(start, stop, step)**: Creates an array with values in a range (similar to Python's range())
- **np.linspace(start, stop, num)**: Creates an array with num evenly spaced values between start and stop
- **.shape**: Returns the dimensions of the array as a tuple
- **.ndim**: Returns the number of dimensions
- **.dtype**: Returns the data type of array elements

1.2 Array Operations

NumPy supports element-wise operations, which means operations are applied to each element individually. This is much faster than using loops in Python.

***Example**: Element-wise Operations*

```
# Element-wise addition
array_1d = np.array([1, 2, 3, 4, 5])
array_1d_plus_2 = array_1d + 2
print("Array + 2:", array_1d_plus_2)  # Output: [3 4 5 6 7]

# Element-wise multiplication
array_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
array_2d_times_3 = array_2d * 3
print("2D Array * 3:\n", array_2d_times_3)

# Element-wise operations between arrays
array_a = np.array([1, 2, 3])
array_b = np.array([4, 5, 6])
print("\nArray addition:", array_a + array_b)  # Output: [5 7 9]
print("Array multiplication:", array_a * array_b)  # Output: [4 10 18]

# Statistical operations
sum_1d = np.sum(array_1d)
mean_1d = np.mean(array_1d)
std_1d = np.std(array_1d)
median_1d = np.median(array_1d)
print("\nSum:", sum_1d)
print("Mean:", mean_1d)
print("Standard Deviation:", std_1d)
print("Median:", median_1d)

# Axis-specific operations
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
print("\nSum along axis 0 (columns):", np.sum(array_2d, axis=0))  # [5 7 9]
print("Sum along axis 1 (rows):", np.sum(array_2d, axis=1))  # [6 15]
print("Mean along axis 0:", np.mean(array_2d, axis=0))  # [2.5 3.5 4.5]
```

Key Points:

- **Broadcasting**: NumPy automatically extends operations between arrays and scalars (e.g., array + 2 adds 2 to each element)
- **Element-wise operations**: +, -, *, /, ** are applied to corresponding elements
- **Statistical functions**: np.sum(), np.mean(), np.std(), np.median(), np.min(), np.max()
- **Axis parameter**: axis=0 operates along columns (vertically), axis=1 operates along rows (horizontally)
- **Note**: Element-wise multiplication (*) is different from matrix multiplication

1.3 Reshaping and Indexing

Understanding how to reshape and index arrays is crucial for data manipulation and preparing data for mathematical operations.

***Example**: Reshaping Arrays*

```
# Reshape a 1D array to a 2D matrix
array_1d = np.array([1, 2, 3, 4, 5, 6])
reshaped_array = array_1d.reshape(2, 3)
print("Original shape:", array_1d.shape)  # (6,)
print("Reshaped to 2x3:\n", reshaped_array)
print("New shape:", reshaped_array.shape)  # (2, 3)

# Reshape to 3x2
reshaped_3x2 = array_1d.reshape(3, 2)
print("\nReshaped to 3x2:\n", reshaped_3x2)

# Flatten a 2D array back to 1D
flattened = reshaped_array.flatten()
print("\nFlattened array:", flattened)

# Transpose a matrix (swap rows and columns)
original = np.array([[1, 2, 3], [4, 5, 6]])
transposed = original.T
print("\nOriginal (2x3):\n", original)
print("Transposed (3x2):\n", transposed)
```

***Example**: Indexing and Slicing*

```
array_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Access single element (row 1, column 2)
element = array_2d[1, 2]
print("Element at [1, 2]:", element)  # Output: 6

# Access a complete row
first_row = array_2d[0, :]
print("First row:", first_row)  # Output: [1 2 3]

# Access a complete column
second_column = array_2d[:, 1]
print("Second column:", second_column)  # Output: [2 5 8]

# Slice a submatrix
submatrix = array_2d[0:2, 1:3]
print("Submatrix (rows 0-1, columns 1-2):\n", submatrix)

# Boolean indexing
mask = array_2d > 5
print("\nBoolean mask (elements > 5):\n", mask)
filtered = array_2d[mask]
print("Filtered elements:", filtered)  # Output: [6 7 8 9]

# Modifying elements
array_2d[array_2d > 5] = 0
```

```
print("After setting elements > 5 to 0:\n", array_2d)
```

Key Points:

- **.reshape(rows, cols)**: Changes array shape (total elements must remain constant)
- **.flatten()**: Converts any array to 1D
- **.T**: Transposes the array (swaps rows and columns)
- **Indexing**: Use [row, column] for 2D arrays; indices start at 0
- **Slicing**: Use [start:stop] syntax; : means "all"
- **Boolean indexing**: Create a mask with conditions and use it to filter elements

Part 2: Linear Algebra with NumPy

Linear algebra is fundamental to many areas of mathematics, physics, engineering, and data science. NumPy's linalg module provides comprehensive linear algebra functionality.

2.1 Matrix Multiplication

Matrix multiplication is different from element-wise multiplication. For two matrices A (m×n) and B (n×p), the product C = AB is an (m×p) matrix where each element C[i,j] is the dot product of row i of A and column j of B.

***Example**: Matrix Multiplication*

```
# Define two matrices
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

print("Matrix A:\n", A)
print("\nMatrix B:\n", B)

# Method 1: Using np.dot()
matrix_product_dot = np.dot(A, B)
print("\nMatrix Product (A @ B) using np.dot():\n", matrix_product_dot)

# Method 2: Using @ operator (Python 3.5+)
matrix_product_at = A @ B
print("\nMatrix Product (A @ B) using @ operator:\n", matrix_product_at)

# Method 3: Using np.matmul()
matrix_product_matmul = np.matmul(A, B)
print("\nMatrix Product (A @ B) using np.matmul():\n", matrix_product_matmul)

# Compare with element-wise multiplication
elementwise_product = A * B
print("\nElement-wise Product (A * B):\n", elementwise_product)

# Matrix multiplication with different dimensions
C = np.array([[1, 2, 3], [4, 5, 6]])  # 2x3 matrix
```

```
D = np.array([[7, 8], [9, 10], [11, 12]])  # 3x2 matrix
product_CD = C @ D  # Result is 2x2
print("\nC (2x3):\n", C)
print("\nD (3x2):\n", D)
print("\nC @ D (2x2):\n", product_CD)
```

Mathematical Explanation:

For matrices A and B:

```
A @ B[i,j] = Σ(A[i,k] * B[k,j]) for all k
```

For the example above:

```
A @ B = [[1*5 + 2*7,  1*6 + 2*8],    [[19, 22],
         [3*5 + 4*7,  3*6 + 4*8]]  =  [43, 50]]
```

Key Points:

- **np.dot(A, B)**, **A @ B**, and **np.matmul(A, B)** perform matrix multiplication
- Matrix multiplication is **not commutative**: A @ B ≠ B @ A in general
- For multiplication to be valid, the number of columns in A must equal the number of rows in B
- Element-wise multiplication (*) is completely different from matrix multiplication

2.2 Determinants

The determinant is a scalar value that provides important information about a square matrix, including whether it's invertible and the scaling factor of transformations.

***Example**: Computing Determinants*

```
# 2x2 matrix
A = np.array([[1, 2], [3, 4]])
det_A = np.linalg.det(A)
print("Matrix A:\n", A)
print("Determinant of A:", det_A)  # Output: -2.0

# 3x3 matrix
B = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
det_B = np.linalg.det(B)
print("\nMatrix B:\n", B)
print("Determinant of B:", det_B)  # Output: ~0 (approximately 0)

# Identity matrix
I = np.eye(3)
det_I = np.linalg.det(I)
print("\nIdentity Matrix:\n", I)
print("Determinant of Identity:", det_I)  # Output: 1.0

# Singular matrix (determinant = 0)
singular = np.array([[2, 4], [1, 2]])
```

```
det_singular = np.linalg.det(singular)
print("\nSingular Matrix:\n", singular)
print("Determinant:", det_singular)  # Output: 0.0
```

Mathematical Explanation:

For a 2×2 matrix:

```
det([[a, b],     = ad - bc
     [c, d]])
```

For a 3×3 matrix:

```
det([[a, b, c],
     [d, e, f],    = a(ei - fh) - b(di - fg) + c(dh - eg)
     [g, h, i]])
```

Key Points:

- **np.linalg.det(A)**: Computes the determinant of a square matrix
- **Determinant = 0**: Matrix is singular (not invertible)
- **Determinant ≠ 0**: Matrix is non-singular (invertible)
- **Determinant of identity matrix = 1**
- The sign of the determinant indicates orientation (positive/negative)

2.3 Matrix Inverses

The inverse of a matrix A, denoted A^{-1}, satisfies the equation $A @ A^{-1} = A^{-1} @ A = I$ (identity matrix). Only square matrices with non-zero determinants have inverses.

Example*: Computing Matrix Inverses*

```
# Invertible matrix
A = np.array([[1, 2], [3, 4]])
print("Matrix A:\n", A)
print("Determinant:", np.linalg.det(A))

# Compute inverse
inverse_A = np.linalg.inv(A)
print("\nInverse of A:\n", inverse_A)

# Verify: A @ A^(-1) should equal identity matrix
identity = A @ inverse_A
print("\nA @ A^(-1):\n", identity)
print("Is this the identity matrix?", np.allclose(identity, np.eye(2)))

# Verify: A^(-1) @ A should also equal identity
identity2 = inverse_A @ A
print("\nA^(-1) @ A:\n", identity2)
```

```
# 3x3 example
B = np.array([[2, 1, 0], [1, 3, 1], [0, 1, 2]])
inverse_B = np.linalg.inv(B)
print("\n3x3 Matrix B:\n", B)
print("\nInverse of B:\n", inverse_B)
print("\nVerification (B @ B^(-1)):\n", B @ inverse_B)

# Attempting to invert a singular matrix (will raise error)
try:
    singular = np.array([[2, 4], [1, 2]])
    inverse_singular = np.linalg.inv(singular)
except np.linalg.LinAlgError as e:
    print("\nCannot invert singular matrix:", e)
```

Mathematical Explanation:

For a 2×2 matrix:

```
A^(-1) = (1/det(A)) * [[d, -b],
                       [-c, a]]

where A = [[a, b],
           [c, d]]
```

Key Points:

- **np.linalg.inv(A)**: Computes the inverse of a square matrix
- Only matrices with **non-zero determinant** have inverses
- **$A @ A^{-1} = A^{-1} @ A = I$** (identity matrix)
- Inverses are used to solve systems of linear equations: **$Ax = b \rightarrow x = A^{-1}b$**
- Use np.allclose() to check equality with floating-point numbers (handles rounding errors)

2.4 Solving Linear Systems

Systems of linear equations can be solved using matrix operations. For a system Ax = b, the solution is $x = A^{-1}b$ (when A is invertible).

***Example**: Solving Linear Systems*

```
# System of equations:
# 2x + y = 8
# x + 3y = 13

# Coefficient matrix A
A = np.array([[2, 1], [1, 3]])

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

print("System of equations:")
print("2x + y = 8")
print("x + 3y = 13")
```

```
print("\nCoefficient Matrix A:\n", A)
print("Constants vector b:", b)

# Method 1: Using inverse
A_inv = np.linalg.inv(A)
x_method1 = A_inv @ b
print("\nSolution using inverse (x = A^(-1) @ b):", x_method1)

# Method 2: Using np.linalg.solve
x_method2 = np.linalg.solve(A, b)
print("Solution using np.linalg.solve():", x_method2)

# Verify the solution
verification = A @ x_method2
print("\nVerification (A @ x):", verification)
print("Original b:", b)
print("Match?", np.allclose(verification, b))

# Larger system (3x3)
print("\n" + "="*50)
print("3x3 System:")
# x + 2y + z = 6
# 2x + y - z = 1
# x - y + 2z = 7

A2 = np.array([[1, 2, 1], [2, 1, -1], [1, -1, 2]])
b2 = np.array([6, 1, 7])

solution = np.linalg.solve(A2, b2)
print("Solution:", solution)
print("Verification:", A2 @ solution)
print("Expected:", b2)
```

Key Points:

- **np.linalg.solve(A, b)**: Solves the system Ax = b efficiently
- More numerically stable than computing A^{-1} and multiplying
- Used for solving systems of linear equations in engineering, physics, and data science
- Always verify solutions by computing A @ x and comparing with b

Part 3: Advanced NumPy Operations

3.1 Broadcasting

Broadcasting allows NumPy to perform operations on arrays of different shapes efficiently, without creating copies of data.

Example*: Broadcasting Operations*

```
# Broadcasting: Add a 1D array to each row of a 2D array
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
array_1d = np.array([10, 20, 30])
```

```
print("2D Array (2x3):\n", array_2d)
print("\n1D Array:", array_1d)

# Broadcasting automatically extends array_1d to match array_2d's shape
broadcast_result = array_2d + array_1d
print("\nBroadcast Result (2D + 1D):\n", broadcast_result)

# Broadcasting with different shapes
column_vector = np.array([[1], [2], [3]])  # 3x1
row_vector = np.array([10, 20])  # 1x2

print("\n" + "="*50)
print("Column vector (3x1):\n", column_vector)
print("\nRow vector (1x2):", row_vector)

# Results in 3x2 matrix
broadcast_result2 = column_vector + row_vector
print("\nBroadcast Result (3x2):\n", broadcast_result2)

# Multiplication broadcasting
array_3x2 = np.array([[1, 2], [3, 4], [5, 6]])
multiplier = np.array([10, 20])

result = array_3x2 * multiplier
print("\n" + "="*50)
print("3x2 Array:\n", array_3x2)
print("\nMultiplier:", multiplier)
print("\nResult:\n", result)
```

Broadcasting Rules:

1. Arrays with different dimensions: prepend 1s to smaller array's shape
2. Arrays are compatible when dimensions are equal or one of them is 1
3. Result shape is the maximum along each dimension

Key Points:

- Broadcasting eliminates the need for explicit loops
- Significantly improves performance and memory efficiency
- Understanding broadcasting is crucial for efficient NumPy code

3.2 Array Stacking and Splitting

Combining and dividing arrays is essential for data manipulation and organizing computations.

Example*: Stacking Arrays*

```
# Define two arrays
array1 = np.array([[1, 2], [3, 4]])
array2 = np.array([[5, 6], [7, 8]])
```

```
print("Array 1:\n", array1)
print("\nArray 2:\n", array2)

# Vertical stacking (vstack): Stack arrays vertically (row-wise)
vstacked = np.vstack((array1, array2))
print("\nVertically Stacked (4x2):\n", vstacked)

# Horizontal stacking (hstack): Stack arrays horizontally (column-wise)
hstacked = np.hstack((array1, array2))
print("\nHorizontally Stacked (2x4):\n", hstacked)

# Depth stacking (dstack): Stack arrays along third dimension
dstacked = np.dstack((array1, array2))
print("\nDepth Stacked (2x2x2):\n", dstacked)
print("Shape:", dstacked.shape)

# Concatenate with axis parameter
concat_axis0 = np.concatenate((array1, array2), axis=0)  # Same as vstack
concat_axis1 = np.concatenate((array1, array2), axis=1)  # Same as hstack
print("\nConcatenate axis=0:\n", concat_axis0)
print("\nConcatenate axis=1:\n", concat_axis1)
```

Example: *Splitting Arrays*

```
# Create a large array
large_array = np.array([[1, 2, 3, 4],
                        [5, 6, 7, 8],
                        [9, 10, 11, 12]])

print("Large Array (3x4):\n", large_array)

# Split horizontally into 2 equal parts
hsplit_result = np.hsplit(large_array, 2)
print("\nHorizontal Split (into 2 parts):")
for i, arr in enumerate(hsplit_result):
    print(f"Part {i+1}:\n{arr}\n")

# Split vertically
vsplit_result = np.vsplit(large_array, 3)
print("Vertical Split (into 3 parts):")
for i, arr in enumerate(vsplit_result):
    print(f"Part {i+1}:\n{arr}\n")

# Split at specific indices
split_at_indices = np.split(large_array, [1, 2], axis=0)
print("Split at indices [1, 2] along axis 0:")
for i, arr in enumerate(split_at_indices):
    print(f"Part {i+1}:\n{arr}\n")
```

Key Points:

- **np.vstack()**: Stacks arrays vertically (increases rows)
- **np.hstack()**: Stacks arrays horizontally (increases columns)
- **np.dstack()**: Stacks arrays along a third dimension

- **np.hsplit()**: Splits array horizontally
- **np.vsplit()**: Splits array vertically
- **np.split(indices, axis)**: Splits at specified indices along given axis

3.3 Matrix Norms and Trace

Norms measure the "size" of a matrix, while the trace is the sum of diagonal elements.

***Example**: Norms and Trace*

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

print("Matrix A:\n", A)

# Frobenius norm (default): sqrt of sum of squared elements
frobenius_norm = np.linalg.norm(A)
print("\nFrobenius Norm:", frobenius_norm)

# Matrix 1-norm (maximum absolute column sum)
norm_1 = np.linalg.norm(A, ord=1)
print("1-Norm:", norm_1)

# Matrix infinity-norm (maximum absolute row sum)
norm_inf = np.linalg.norm(A, ord=np.inf)
print("Infinity-Norm:", norm_inf)

# Matrix 2-norm (spectral norm - largest singular value)
norm_2 = np.linalg.norm(A, ord=2)
print("2-Norm:", norm_2)

# Trace (sum of diagonal elements)
trace_A = np.trace(A)
print("\nTrace of A:", trace_A)  # 1 + 4 = 5

# Trace of identity matrix
I = np.eye(3)
trace_I = np.trace(I)
print("Trace of 3x3 Identity:", trace_I)  # 3
```

Key Points:

- **np.linalg.norm(A)**: Computes various matrix norms
- **Frobenius norm**: $\sqrt{(\Sigma|a_{ij}|^2)}$ - most commonly used
- **np.trace(A)**: Sum of diagonal elements
- Trace has important properties: $tr(A + B) = tr(A) + tr(B)$, $tr(AB) = tr(BA)$

Lab Tasks

Complete the following exercises to practice NumPy and linear algebra operations. Save your code in a Python script (lab_numpy_linear_algebra.py) and include comments explaining each step.

Task 1: Array Creation and Basic Operations

1. Create a 1D NumPy array of numbers from 10 to 50 with a step of 5 using np.arange()
2. Compute the sum, mean, median, and standard deviation of the array
3. Reshape the array into a 2x4 matrix
4. Transpose the matrix and print the result

Task 2: Array Manipulation

1. Create a 4x4 NumPy array with random integers between 1 and 20 using np.random.randint()
2. Replace all values greater than 10 with -1
3. Compute the sum of each row and each column
4. Find the maximum value in each row

Task 3: Matrix Multiplication

1. Create two 3x3 matrices A and B with values of your choice
2. Compute their matrix product using @ operator
3. Compute the element-wise product using * operator
4. Compare the results and explain the difference in comments

Task 4: Determinants and Inverses

1. Create a 2x2 matrix M = [[4, 7], [2, 6]]
2. Calculate its determinant
3. Compute its inverse
4. Verify that M @ M^{-1} equals the identity matrix using np.allclose()
5. Create a singular matrix and attempt to compute its inverse (handle the error)

Task 5: Solving Linear Systems

Solve the following system of equations using NumPy:

$$3x + 2y + z = 10$$
$$2x + 3y + 2z = 14$$
$$x + 2y + 3z = 14$$

1. Set up the coefficient matrix A and constants vector b
2. Solve using np.linalg.solve()
3. Verify your solution by computing A @ x and comparing with b

\

```
# Task 1:
import numpy as np
arr = np.arange(10, 55, 5)  # 10, 15, ..., 50
print("1D Array:", arr)
print("Sum:", np.sum(arr))
print("Mean:", np.mean(arr))
print("Median:", np.median(arr))
print("Standard Deviation:", np.std(arr))
matrix_2x4 = arr.reshape(2, 4)
print("2x4 Matrix:\n", matrix_2x4)
print("Transpose:\n", matrix_2x4.T)

# Task 2:
rand_matrix = np.random.randint(1, 21, size=(4, 4))
print("\nRandom 4x4 Matrix:\n", rand_matrix)
rand_matrix[rand_matrix > 10] = -1
print("After replacing > 10 with -1:\n", rand_matrix)
print("Row sums:", np.sum(rand_matrix, axis=1))
print("Column sums:", np.sum(rand_matrix, axis=0))
print("Max in each row:", np.max(rand_matrix, axis=1))

# Task 3:
A = np.array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
B = np.array([[9, 8, 7],[6, 5, 4],[3, 2, 1]])
matrix_product = A @ B
print("\nMatrix Product (A @ B):\n", matrix_product)
elementwise_product = A * B
print("Element-wise Product (A * B):\n", elementwise_product)

# Task 4:
M = np.array([[4, 7],[2, 6]])
det = np.linalg.det(M)
print("\nDeterminant of M:", det)
M_inv = np.linalg.inv(M)
print("Inverse of M:\n", M_inv)
print("M @ M⁻¹ ≈ I ? ", np.allclose(M @ M_inv, np.eye(2)))
singular = np.array([[2, 4],[1, 2]])
try:
    np.linalg.inv(singular)
except np.linalg.LinAlgError as e:
    print("Error (singular matrix cannot be inverted):", e)

# Task 5:
A = np.array([[3, 2, 1],[2, 3, 2],[1, 2, 3]])
b = np.array([10, 14, 14])
x = np.linalg.solve(A, b)
print("\nSolution (x, y, z):", x)
print("Verification A @ x:", A @ x)
print("Original b:", b)
```