

Computational Linear Algebra

Siju Swamy

2024-07-16

Table of contents

Preface	4
Introduction	66
Relearning of Terms and Operations in Linear Algebra	66
Matrix Addition and Subtraction in Data Analysis	66
More on Matrix Product and its Applications	81
Matrix Measures of Practical Importance	136
Rank Nullity Theorem	140
Fundamental Subspaces	142
1 Python Libraries for Computational Linear Algebra	153
1.1 Introduction to NumPy	153
1.1.1 Purpose of Using NumPy	153
1.2 Basic Operations in NumPy	154
1.2.1 Define different types of <code>numpy</code> arrays	155
1.2.2 Review Questions	160
1.2.3 Tensors in NumPy	161
1.2.4 Array Indexing: Accessing Single Elements	165
1.2.5 Array Concatenation and Splitting	172
1.2.6 Review Questions	175
1.3 Introduction	177
1.4 LU Decomposition	178
1.4.1 Step-by-Step Procedure	179
1.4.2 Example	180
1.4.3 Python Implementation	181
1.4.4 LU Decomposition Practice Problems with Solutions	181
1.5 LU Decomposition Practice Problems	185
1.6 Matrix Approach to Create LU Decomposition	186
2 Spectral Decomposition	190
2.1 Background	190
2.2 Introduction	191
2.3 Spectral Decomposition: Detailed Concepts	191
2.3.1 Eigenvalues and Eigenvectors	191
2.3.2 Eigenvalue Decomposition (Spectral Decomposition)	194
2.3.3 Geometric Interpretation	195

2.3.4	Importance of Diagonalization	195
2.3.5	Properties of Symmetric Matrices	195
2.4	Mathematical Requirements for Spectral Decomposition	196
2.4.1	Determining Eigenvalues and Eigenvectors	196
2.4.2	Characteristic Polynomial of 2×2 Matrices	196
2.4.3	Problems	197
2.4.4	<code>Python</code> code to find eigen values and eigen vectors	205
2.4.5	Diagonalization of Symmetric Matrices	206
2.4.6	Matrix Functions and Spectral Theorem	208
3	QR Decomposition	209
3.0.1	Practical Uses of QR Decomposition	210
3.0.2	<code>Python</code> method for QR decomposition	211
3.1	Overdetermined Systems	212
3.1.1	Example of an Overdetermined System	212
3.1.2	Challenges in Solving Overdetermined Systems	213
3.1.3	Why We Need QR Decomposition	213
3.1.4	Solving an Overdetermined System using QR Decomposition	214
3.1.5	Problems	215
	References	220

Preface

- - -

I was born in the darkness of ignorance, and my master opened my eyes with the torch of knowledge. I offer my respectful obeisances unto him

Welcome

to
the
course
on
**Com-
pu-
ta-
tional
Lin-
ear
Al-
ge-
bra.**

This
course
is de-
signed
to
pro-
vide
a
prac-
tical
per-
spec-
tive
on
lin-
ear
alge-
bra,
bridg-
ing
the
gap
be-
tween
math-
e-
mati-
cal
the-
ory
and
real-
world
ap-
plica-
tions.
As
we
delve

Welcome

to
the
course
on
**Com-
pu-
ta-
tional
Lin-
ear
Al-
ge-
bra.**

This
course
is de-
signed
to
pro-
vide
a
prac-
tical
per-
spec-
tive
on
lin-
ear
alge-
bra,
bridg-
ing
the
gap
be-
tween
math-
e-
mati-
cal
the-
ory
and
real-
world
ap-
plica-
tions.
As
we
delve

Pseudocode: the new language for algorithm design

Pseudocode is a way to describe algorithms in a structured but plain language. It helps in p

:::{.callout-caution}

There are varieties of approaches in writing pseudocode. Students can adopt any of the standar

:::

Matrix Sum

****Mathematical Procedure:****

To add two matrices A and B , both matrices must have the same dimensions. The sum C of

$$C[i][j] = A[i][j] + B[i][j]$$

****Example:****

Let A and B be two 2×2 matrices:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

The sum C is:

$$C = A + B = \begin{bmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

****Pseudocode:****

```python

FUNCTION matrix\_sum(A, B):

    Get the number of rows and columns in matrix A

    Create an empty matrix C with the same dimensions

    FOR each row i:



```

 FOR each column j:
 Set C[i][j] to the sum of A[i][j] and B[i][j]
 RETURN the matrix C
 END FUNCTION
 ...

```

**\*\*Explanation:\*\***

1. Determine the number of rows and columns in matrix  $A$ .
2. Create a new matrix  $C$  with the same dimensions.
3. Loop through each element of the matrices and add corresponding elements.
4. Return the resulting matrix  $C$ .

### ### Matrix Difference

**\*\*Mathematical Procedure:\*\***

To subtract matrix  $B$  from matrix  $A$ , both matrices must have the same dimensions. The difference is calculated as follows:

$$C[i][j] = A[i][j] - B[i][j]$$

**\*\*Example:\*\***

Let  $A$  and  $B$  be two  $2 \times 2$  matrices:

$$A = \begin{bmatrix} 9 & 8 \\ 7 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

The difference  $C$  is:

$$C = A - B = \begin{bmatrix} 9-1 & 8-2 \\ 7-3 & 6-4 \end{bmatrix} = \begin{bmatrix} 8 & 6 \\ 4 & 2 \end{bmatrix}$$

**\*\*Pseudocode:\*\***

```

```python
FUNCTION matrix_difference(A, B):
    # Determine the number of rows and columns in matrix A
    rows = number_of_rows(A)
    cols = number_of_columns(A)

    # Create an empty matrix C with the same dimensions as A and B
    C = create_matrix(rows, cols)

    # Iterate through each row

```

```

FOR i FROM 0 TO rows-1:
    # Iterate through each column
    FOR j FROM 0 TO cols-1:
        # Calculate the difference for each element and store it in C
        C[i][j] = A[i][j] - B[i][j]

    # Return the result matrix C
    RETURN C
END FUNCTION
```

```

In more human readable format the above pseudocode can be written as:

```

```python
FUNCTION matrix_difference(A, B):
    Get the number of rows and columns in matrix A
    Create an empty matrix C with the same dimensions
    FOR each row i:
        FOR each column j:
            Set C[i][j] to the difference of A[i][j] and B[i][j]
    RETURN the matrix C
END FUNCTION
```

```

**\*\*Explanation:\*\***

1. Determine the number of rows and columns in matrix  $A$ .
2. Create a new matrix  $C$  with the same dimensions.
3. Loop through each element of the matrices and subtract corresponding elements.
4. Return the resulting matrix  $C$ .

**### Matrix Product**

**\*\*Mathematical Procedure:\*\***

To find the product of two matrices  $A$  and  $B$ , the number of columns in  $A$  must be equal

$$C[i][j] = \sum_k A[i][k] \cdot B[k][j]$$

**\*\*Example:\*\***

Let  $A$  be a  $2 \times 3$  matrix and  $B$  be a  $3 \times 2$  matrix:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 7 & 8 \end{bmatrix}$$

The product  $C$  is:

$C = A \cdot B = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}$

**\*\*Pseudocode:\*\***

```
```python
FUNCTION matrix_product(A, B):
    # Get the dimensions of A and B
    rows_A = number_of_rows(A)
    cols_A = number_of_columns(A)
    rows_B = number_of_rows(B)
    cols_B = number_of_columns(B)

    # Check if multiplication is possible
    IF cols_A != rows_B:
        RAISE Error("Incompatible matrix dimensions")

    # Initialize result matrix C
    C = create_matrix(rows_A, cols_B)

    # Calculate matrix product
    FOR each row i FROM 0 TO rows_A-1:
        FOR each column j FROM 0 TO cols_B-1:
            # Compute the sum for C[i][j]
            sum = 0
            FOR each k FROM 0 TO cols_A-1:
                sum = sum + A[i][k] * B[k][j]
            C[i][j] = sum

    RETURN C
END FUNCTION
```
```

A more human readable version of the `pseudocode` is shown below:

```
```python
FUNCTION matrix_product(A, B):
    Get the number of rows and columns in matrix A
    Get the number of columns in matrix B
    Create an empty matrix C with dimensions rows_A x cols_B
    FOR each row i in A:
        FOR each column j in B:
```

```

        Initialize C[i][j] to 0
        FOR each element k in the common dimension:
            Add the product of A[i][k] and B[k][j] to C[i][j]
    RETURN the matrix C
END FUNCTION
```

```

**\*\*Explanation:\*\***

1. Determine the number of rows and columns in matrices  $A$  and  $B$ .
2. Create a new matrix  $C$  with dimensions  $\text{rows}(A) \times \text{columns}(B)$ .
3. Loop through each element of the resulting matrix  $C[i][j]$  and calculate the dot product.
4. Return the resulting matrix  $C$ .

### Determinant

**\*\*Mathematical Procedure:\*\***

To find the determinant of a square matrix  $A$ , we can use the Laplace expansion, which involves

$$\det(A) = A[0][0] \cdot A[1][1] - A[0][1] \cdot A[1][0]$$

For larger matrices, the determinant is calculated recursively.

**\*\*Example:\*\***

Let  $A$  be a  $2 \times 2$  matrix:

$$A = \begin{bmatrix} 4 & 3 \\ 6 & 3 \end{bmatrix}$$

The determinant of  $A$  is:

$$\det(A) = (4 \cdot 3) - (3 \cdot 6) = 12 - 18 = -6$$

**\*\*Pseudocode:\*\***

```

```python
FUNCTION determinant(A):
    # Step 1: Get the size of the matrix
    n = number_of_rows(A)

    # Base case for a 2x2 matrix
    IF n == 2:
        RETURN A[0][0] * A[1][1] - A[0][1] * A[1][0]

```

```

# Step 2: Initialize determinant to 0
det = 0

# Step 3: Loop through each column of the first row
FOR each column j FROM 0 TO n-1:
    # Get the submatrix excluding the first row and current column
    submatrix = create_submatrix(A, 0, j)
    # Recursive call to determinant
    sub_det = determinant(submatrix)
    # Alternating sign and adding to the determinant
    det = det + ((-1) ^ j) * A[0][j] * sub_det

RETURN det
END FUNCTION

FUNCTION create_sub_matrix(A, row, col):
    sub_matrix = create_matrix(number_of_rows(A)-1, number_of_columns(A)-1)
    sub_i = 0
    FOR i FROM 0 TO number_of_rows(A)-1:
        IF i == row:
            CONTINUE
        sub_j = 0
        FOR j FROM 0 TO number_of_columns(A)-1:
            IF j == col:
                CONTINUE
            sub_matrix[sub_i][sub_j] = A[i][j]
            sub_j = sub_j + 1
        sub_i = sub_i + 1
    RETURN sub_matrix
END FUNCTION

```

A human readable version of the same pseudocode is shown below:

```

```python
FUNCTION determinant(A):
 IF the size of A is 2x2:
 RETURN the difference between the product of the diagonals
 END IF
 Initialize det to 0
 FOR each column c in the first row:
 Create a sub_matrix by removing the first row and column c
 Add to det: the product of $(-1)^c$, the element $A[0][c]$, and the determinant of the s

```

```

 RETURN det
END FUNCTION

FUNCTION create_sub_matrix(A, row, col):
 Create an empty sub_matrix with dimensions one less than A
 Set sub_i to 0
 FOR each row i in A:
 IF i is the row to be removed:
 CONTINUE to the next row
 Set sub_j to 0
 FOR each column j in A:
 IF j is the column to be removed:
 CONTINUE to the next column
 Copy the element A[i][j] to sub_matrix[sub_i][sub_j]
 Increment sub_j
 Increment sub_i
 RETURN sub_matrix
END FUNCTION

```

**\*\*Explanation:\*\***

1. If the matrix is  $2 \times 2$ , calculate the determinant directly.
2. For larger matrices, use the Laplace expansion to recursively calculate the determinant.
3. Create submatrices by removing the current row and column.
4. Sum the determinants of the submatrices, adjusted for the sign and the current element.

**### Rank of a Matrix**

**\*\*Mathematical Procedure:\*\***

The rank of a matrix  $A$  is the maximum number of linearly independent rows or columns in  $A$ .

**\*\*Example:\*\***

Let  $A$  be a  $3 \times 3$  matrix:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

After performing Gaussian elimination, we obtain:

$$\text{REF}(A) = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & 0 \end{bmatrix}$$

The rank of  $A$  is the number of non-zero rows, which is 2.

**\*\*Pseudocode:\*\***

```
```python
FUNCTION matrix_rank(A):
    # Step 1: Get the dimensions of the matrix
    rows = number_of_rows(A)
    cols = number_of_columns(A)

    # Step 2: Transform the matrix to row echelon form
    row_echelon_form(A, rows, cols)

    # Step 3: Count non-zero rows
    rank = 0
    FOR each row i FROM 0 TO rows-1:
        non_zero = FALSE
        FOR each column j FROM 0 TO cols-1:
            IF A[i][j] != 0:
                non_zero = TRUE
                BREAK
        IF non_zero:
            rank = rank + 1

    RETURN rank
END FUNCTION

FUNCTION row_echelon_form(A, rows, cols):
    # Perform Gaussian elimination
    lead = 0
    FOR r FROM 0 TO rows-1:
        IF lead >= cols:
            RETURN
        i = r
        WHILE A[i][lead] == 0:
            i = i + 1
            IF i == rows:
                i = r
                lead = lead + 1
            IF lead == cols:
                RETURN
        # Swap rows i and r
        swap_rows(A, i, r)
```

```

    # Make A[r][lead] = 1
    lv = A[r][lead]
    A[r] = [m / float(lv) for m in A[r]]
    # Make all rows below r have 0 in column lead
    FOR i FROM r + 1 TO rows-1:
        lv = A[i][lead]
        A[i] = [iv - lv * rv for rv, iv in zip(A[r], A[i])]
    lead = lead + 1
END FUNCTION

```

```

FUNCTION swap_rows(A, row1, row2):
    temp = A[row1]
    A[row1] = A[row2]
    A[row2] = temp
END FUNCTION
...

```

A more human readable version of the above pseudocode is shown below:

```

```python
FUNCTION rank(A):
 Get the number of rows and columns in matrix A
 Initialize the rank to 0
 FOR each row i in A:
 IF the element in the current row and column is non-zero:
 Increment the rank
 FOR each row below the current row:
 Calculate the multiplier to zero out the element below the diagonal
 Subtract the appropriate multiple of the current row from each row below
 ELSE:
 Initialize a variable to track if a swap is needed
 FOR each row below the current row:
 IF a non-zero element is found in the current column:
 Swap the current row with the row having the non-zero element
 Set the swap variable to True
 BREAK the loop
 IF no swap was made:
 Decrement the rank
 RETURN the rank
END FUNCTION
...

```

**\*\*Explanation:\*\***

1. Initialize the rank to 0.



2. Loop through each row of the matrix.
3. If the diagonal element is non-zero, increment the rank and perform row operations to zero
4. If the diagonal element is zero, try to swap with a lower row that has a non-zero element
5. If no such row is found, decrement the rank.
6. Return the resulting rank of the matrix.

### Practice Problems

Find the rank of the following matrices.

1.  $\begin{pmatrix} 1 & 1 & 3 \\ 2 & 2 & 6 \\ 2 & 5 & 3 \end{pmatrix}$ .
2.  $\begin{pmatrix} 2 & 0 & 2 \\ 1 & 2 & 3 \\ 3 & 2 & 7 \end{pmatrix}$

### Solving a System of Equations

#### Mathematical Procedure

To solve a system of linear equations represented as  $A \mathbf{x} = \mathbf{b}$ , where  $A$  is

#### Example

Consider the system of equations:

$$\begin{cases} x + 2y + 3z = 9 \\ 4x + 5y + 6z = 24 \\ 7x + 8y + 9z = 39 \end{cases}$$

The augmented matrix is:

$$[A \mid \mathbf{b}] = \begin{bmatrix} 1 & 2 & 3 & | & 9 \\ 4 & 5 & 6 & | & 24 \\ 7 & 8 & 9 & | & 39 \end{bmatrix}$$

After performing Gaussian elimination on the augmented matrix, we get:

$$\text{REF}(A) = \begin{bmatrix} 1 & 2 & 3 & | & 9 \\ 0 & -3 & -6 & | & -12 \\ 0 & 0 & 0 & | & 0 \end{bmatrix}$$

Performing back substitution, we solve for  $z$ ,  $y$ , and  $x$ :

$$\begin{cases} z = 1 \\ y = 0 \end{cases}$$

```
x = 3
\end{cases}$$
```

Therefore, the solution vector is  $\mathbf{x} = \begin{bmatrix} 3 \\ 0 \\ 1 \end{bmatrix}$ .

**\*\*Pseudocode:\*\***

```
```python
FUNCTION solve_system_of_equations(A, b):
    # Step 1: Get the dimensions of the matrix
    rows = number_of_rows(A)
    cols = number_of_columns(A)

    # Step 2: Create the augmented matrix
    augmented_matrix = create_augmented_matrix(A, b)

    # Step 3: Transform the augmented matrix to row echelon form
    row_echelon_form(augmented_matrix, rows, cols)

    # Step 4: Perform back substitution
    solution = back_substitution(augmented_matrix, rows, cols)

    RETURN solution
END FUNCTION

FUNCTION create_augmented_matrix(A, b):
    # Combine A and b into an augmented matrix
    augmented_matrix = []
    FOR i FROM 0 TO number_of_rows(A)-1:
        augmented_matrix.append(A[i] + [b[i]])
    RETURN augmented_matrix
END FUNCTION

FUNCTION row_echelon_form(augmented_matrix, rows, cols):
    # Perform Gaussian elimination
    lead = 0
    FOR r FROM 0 TO rows-1:
        IF lead >= cols:
            RETURN
        i = r
        WHILE augmented_matrix[i][lead] == 0:
            i = i + 1
```

```

        IF i == rows:
            i = r
            lead = lead + 1
            IF lead == cols:
                RETURN
            # Swap rows i and r
            swap_rows(augmented_matrix, i, r)
            # Make augmented_matrix[r][lead] = 1
            lv = augmented_matrix[r][lead]
            augmented_matrix[r] = [m / float(lv) for m in augmented_matrix[r]]
            # Make all rows below r have 0 in column lead
            FOR i FROM r + 1 TO rows-1:
                lv = augmented_matrix[i][lead]
                augmented_matrix[i] = [iv - lv * rv for rv, iv in zip(augmented_matrix[r], augmented_matrix[i])]
            lead = lead + 1
    END FUNCTION

```

```

FUNCTION back_substitution(augmented_matrix, rows, cols):
    # Initialize the solution vector
    solution = [0 for _ in range(rows)]
    # Perform back substitution
    FOR i FROM rows-1 DOWNTO 0:
        solution[i] = augmented_matrix[i][cols-1]
        FOR j FROM i+1 TO cols-2:
            solution[i] = solution[i] - augmented_matrix[i][j] * solution[j]
    RETURN solution
END FUNCTION

```

```

FUNCTION swap_rows(matrix, row1, row2):
    temp = matrix[row1]
    matrix[row1] = matrix[row2]
    matrix[row2] = temp
END FUNCTION
...

```

****Explanation:****

1. Augment the coefficient matrix A with the constant matrix B .
2. Perform Gaussian elimination to reduce the augmented matrix to row echelon form.
3. Back-substitute to find the solution vector X .
4. Return the solution vector X .

Review Problems

****Q1:**** Fill in the missing parts of the pseudocode to yield a meaningful algebraic operation

****Pseudocode:****

```
```python
FUNCTION matrix_op1(A, B):
 rows = number_of_rows(A)
 cols = number_of_columns(A)
 result = create_matrix(rows, cols, 0)

 FOR i FROM 0 TO rows-1:
 FOR j FROM 0 TO cols-1:
 result[i][j] = A[i][j] + ---

 RETURN result
END FUNCTION
```
```

****Q2:**** Write the pseudocode to get useful derivable from a given a matrix by fill in the mi

****Pseudocode:****

```
```python
FUNCTION matrix_op2(A):
 rows = number_of_rows(A)
 cols = number_of_columns(A)
 result = create_matrix(cols, rows, 0)

 FOR i FROM 0 TO rows-1:
 FOR j FROM 0 TO cols-1:
 result[j][i] = A[i][--]

 RETURN result
END FUNCTION
```
```

Transition from Pseudocode to Python Programming

In this course, our initial approach to understanding and solving linear algebra problems has

However, to fully leverage the power of computational tools and prepare for real-world appli

1. ****Practical Implementation:**** Python provides numerous libraries and tools, such as NumPy
2. ****Hands-On Experience:**** Moving to Python programming gives students hands-on experience :

3. ****Industry Relevance:**** Python is extensively used in industry for data analysis, machine
4. ****Integration with Other Tools:**** Python's compatibility with various tools and platforms
5. ****Enhanced Learning:**** Implementing algorithms in Python helps reinforce theoretical concepts

By transitioning to Python programming, we not only achieve our course objectives but also equip

Python Fundamentals

Python Programming Overview

Python is a high-level, interpreted programming language that was created by Guido van Rossum

Variables

In Python, variables are used to store data that can be used and manipulated throughout a program

****Basic Input/Output Functions****

Python provides built-in functions for basic input and output operations. The `print()` function

Output with `print()` function

>Example 1

```
```python
Printing text
print("Hello, World!")

Printing multiple values
x = 5
y = 10
print("The value of x is:", x, "and the value of y is:", y)
```
```

>Example 2

```
```python
Assigning values to variables
a = 10
```

```
b = 20.5
name = "Alice"
```

```
Printing the values
print("Values Stored in the Variables:")
print(a)
print(b)
print(name)
'''
```

\*Input with `input()` Function:\*

```
```python
# Taking input from the user
name = input("Enter usr name: ")
print("Hello, " + name + "!")

# Taking numerical input
age = int(input("Enter usr age: "))
print("us are", age, "years old.")
'''
```

:::{.callout-note}

The `print()` function in Python, defined in the built-in `__builtin__` module, is used to d

:::

****Combining Variables and Input/Output****

us can combine variables and input/output functions to create interactive programs.

>Example:

```
```python
Program to calculate the sum of two numbers
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))

Calculate sum
sum = num1 + num2

Display the result
print("The sum of", num1, "and", num2, "is", sum)
'''
```

### ### Python Programming Style

#### #### Indentation

Python uses indentation to define the blocks of code. Proper indentation is crucial as it affects the execution of the code.

```
```python
if a > b:
    print("a is greater than b")
else:
    print("b is greater than or equal to a")
```
```

#### #### Comments

Use comments to explain user code. Comments begin with the `#` symbol and extend to the end of the line.

```
```python
# This is a comment
a = 10 # This is an inline comment
```
```

#### #### Variable Naming

Use meaningful variable names to make user code more understandable. Variable names should be descriptive and use snake case.

```
```python
student_name = "John"
total_score = 95
```
```

#### #### Consistent Style

Follow the `PEP 8` style guide for Python code to maintain consistency and readability. Use consistent indentation and naming conventions.

```
```python

def calculate_sum(x, y):
    return x + y

result = calculate_sum(5, 3)
print(result)
```
```

### ## Basic Datatypes in Python

In Python, a datatype is a classification that specifies which type of value a variable can hold.

### ### Numeric Types

Numeric types represent data that consists of numbers. Python has three distinct numeric types.

1. **\*\*Integers (`int`)\*\*:**
  - Whole numbers, positive or negative, without decimals.
  - Example: `a = 10`, `b = -5`.
2. **\*\*Floating Point Numbers (`float`)\*\*:**
  - Numbers that contain a decimal point.
  - Example: `pi = 3.14`, `temperature = -7.5`.
3. **\*\*Complex Numbers (`complex`)\*\*:**
  - Numbers with a real and an imaginary part.
  - Example: `z = 3 + 4j`.

```
```python
# Examples of numeric types
a = 10          # Integer
pi = 3.14       # Float
z = 3 + 4j      # Complex
```
```

### ### Sequence Types

Sequence types are used to store multiple items in a single variable. Python has several sequence types.

#### #### String Type

Strings in Python are sequences of characters enclosed in quotes. They are used to handle and manipulate text.

##### **\*\*Characteristics of Strings\*\***

- **\*Ordered\***: Characters in a string have a defined order.
- **\*Immutable\***: Strings cannot be modified after they are created.
- **\*Heterogeneous\***: Strings can include any combination of letters, numbers, and symbols.

##### **\*\*Creating Strings\*\***

Strings can be created using single quotes, double quotes, or triple quotes for multiline strings.

>Example:



```

```python
# Creating strings with different types of quotes
single_quoted = 'Hello, World!'
double_quoted = "Hello, World!"
multiline_string = """This is a
multiline string"""
```

Accessing String Characters

```

Characters in a string are accessed using their index, with the first character having an index of 0.

>Example:

```

```python
# Accessing characters in a string
first_char = single_quoted[0] # Output: 'H'
last_char = single_quoted[-1] # Output: '!'
```

```

**Common String Methods**

Python provides various methods for string manipulation:

1. `.upper()`: Converts all characters to uppercase.
2. `.lower()`: Converts all characters to lowercase.
3. `.strip()`: Removes leading and trailing whitespace.
4. `.replace(old, new)`: Replaces occurrences of a substring with another substring.
5. `.split(separator)`: Splits the string into a list based on a separator.

>Example:

```

```python
# Using string methods
text = "  hello, world!  "
uppercase_text = text.upper() # Result: "  HELLO, WORLD!  "
stripped_text = text.strip() # Result: "hello, world!"
replaced_text = text.replace("world", "Python") # Result: "  hello, Python!  "
words = text.split(",") # Result: ['hello', ' world!  ']
```

```

#### List Type

Lists are one of the most versatile and commonly used sequence types in Python. They allow for

**\*\*Characteristics of Lists\*\***

- **\*Ordered\***: The items in a list have a defined order, which will not change unless explicitly changed.
- **\*Mutable\***: The content of a list can be changed after its creation (i.e., items can be added or removed).
- **\*Dynamic\***: Lists can grow or shrink in size as items are added or removed.
- **\*Heterogeneous\***: Items in a list can be of different data types (e.g., integers, strings, etc.).

**\*\*Creating Lists\*\***

Lists are created by placing comma-separated values inside square brackets.

>Example:

```
```python
# Creating a list of fruits
fruits = ["apple", "banana", "cherry"]

# Creating a mixed list
mixed_list = [1, "Hello", 3.14]
```
```

**\*\*Accessing List Items\*\***

List items are accessed using their index, with the first item having an index of 0.

>Example:

```
```python
# Accessing the first item
first_fruit = fruits[0] # Output: "apple"

# Accessing the last item
last_fruit = fruits[-1] # Output: "cherry"
```
```

**\*\*Modifying Lists\*\***

Lists can be modified by changing the value of specific items, adding new items, or removing

>Example:

```

```python
# Changing the value of an item
fruits[1] = "blueberry" # fruits is now ["apple", "blueberry", "cherry"]

# Adding a new item
fruits.append("orange") # fruits is now ["apple", "blueberry", "cherry", "orange"]

# Removing an item
fruits.remove("blueberry") # fruits is now ["apple", "cherry", "orange"]
```

```

### **\*\*List Methods\*\***

`Python` provides several built-in methods to work with lists:

1. `append(item)`: Adds an item to the end of the list.
2. `insert(index, item)`: Inserts an item at a specified index.
3. `remove(item)`: Removes the first occurrence of an item.
4. `pop(index)`: Removes and returns the item at the specified index.
5. `sort()`: Sorts the list in ascending order.
6. `reverse()`: Reverses the order of the list.

>Example:

```

```python
# Using list methods
numbers = [5, 2, 9, 1]

numbers.append(4) # numbers is now [5, 2, 9, 1, 4]
numbers.sort()   # numbers is now [1, 2, 4, 5, 9]
numbers.reverse() # numbers is now [9, 5, 4, 2, 1]
first_number = numbers.pop(0) # first_number is 9, numbers is now [5, 4, 2, 1]
```

```

### **#### Tuple Type**

Tuples are a built-in sequence type in Python that is used to store an ordered collection of

### **\*\*Characteristics of Tuples\*\***

- **\*Ordered\***: Tuples maintain the order of items, which is consistent throughout their lifetime.
- **\*Immutable\***: Once a tuple is created, its contents cannot be modified. This includes adding or removing items.
- **\*Fixed Size\***: The size of a tuple is fixed; it cannot grow or shrink after creation.

- **\*Heterogeneous\***: Tuples can contain items of different data types, such as integers, strings,

### **\*\*Creating Tuples\*\***

Tuples are created by placing comma-separated values inside parentheses. Single-element tuples

>Example:

```
```python
# Creating a tuple with multiple items
coordinates = (10, 20, 30)

# Creating a single-element tuple
single_element_tuple = (5,)

# Creating a tuple with mixed data types
mixed_tuple = (1, "Hello", 3.14)
```
```

### **\*\*Accessing Tuple Items\*\***

Tuple items are accessed using their index, with the first item having an index of 0. Negative

>Example:

```
```python
# Accessing the first item
x = coordinates[0] # Output: 10

# Accessing the last item
z = coordinates[-1] # Output: 30
```
```

### **\*\*Modifying Tuples\*\***

Since tuples are immutable, their contents cannot be modified. However, we can create new tuples

>Example:

```
```python
# Combining tuples
new_coordinates = coordinates + (40, 50) # Result: (10, 20, 30, 40, 50)
```
```

```
Slicing tuples
sub_tuple = coordinates[1:3] # Result: (20, 30)
```

```
...
```

### **\*\*Tuple Methods\*\***

Tuples have a limited set of built-in methods compared to lists:

1. ``count(item)``: Returns the number of occurrences of the specified item.
2. ``index(item)``: Returns the index of the first occurrence of the specified item.

>Example:

```
```python
# Using tuple methods
numbers = (1, 2, 3, 1, 2, 1)

# Counting occurrences of an item
count_1 = numbers.count(1) # Result: 3

# Finding the index of an item
index_2 = numbers.index(2) # Result: 1
```
```

### **### Mapping Types**

Mapping types in Python are used to store data in key-value pairs. Unlike sequences, mappings

#### **#### Dictionary (`dict`)**

The primary mapping type in Python is the ``dict``. Dictionaries store data as key-value pairs

#### **\*\*Characteristics of Dictionaries\*\***

- **\*Unordered\***: The order of items is not guaranteed and may vary.
- **\*Mutable\***: we can add, remove, and change items after creation.
- **\*Keys\***: Must be unique and immutable (e.g., strings, numbers, tuples).
- **\*Values\***: Can be of any data type and can be duplicated.

#### **\*\*Creating Dictionaries\*\***

Dictionaries are created using curly braces ``{}`` with key-value pairs separated by colons ``:`

>Example:

```
```python
# Creating a dictionary
student = {
    "name": "Alice",
    "age": 21,
    "major": "Computer Science"
}
```
```

**\*\*Accessing and Modifying Dictionary Items\*\***

Items in a dictionary are accessed using their keys. us can also modify, add, or remove items

>Example:

```
```python
# Accessing a value
name = student["name"]  # Output: "Alice"

# Modifying a value
student["age"] = 22  # Updates the age to 22

# Adding a new key-value pair
student["graduation_year"] = 2024

# Removing a key-value pair
del student["major"]
```
```

**\*\*Dictionary Methods\*\***

Python provides several built-in methods to work with dictionaries:

1. ``keys()``: Returns a view object of all keys.
2. ``values()``: Returns a view object of all values.
3. ``items()``: Returns a view object of all key-value pairs.
4. ``get(key, default)``: Returns the value for the specified key, or a default value if the key is not found.
5. ``pop(key, default)``: Removes and returns the value for the specified key, or a default value if the key is not found.

>Example:

```
```python
```

```

# Using dictionary methods
keys = student.keys()      # Result: dict_keys(['name', 'age', 'graduation_year'])
values = student.values()  # Result: dict_values(['Alice', 22, 2024])
items = student.items()    # Result: dict_items([('name', 'Alice'), ('age', 22), ('graduation_year', 2024)])
name = student.get("name") # Result: "Alice"
age = student.pop("age")   # Result: 22
...

```

Set Types

Sets are a built-in data type in Python used to store unique, unordered collections of items

****Characteristics of Sets****

- ***Unordered*** : The items in a set do not have a specific order and may change.
- ***Mutable*** : us can add or remove items from a set after its creation.
- ***Unique*** : Sets do not allow duplicate items; all items must be unique.
- ***Unindexed*** : Sets do not support indexing or slicing.

****Creating Sets****

Sets are created using curly braces `{}` with comma-separated values, or using the `set()` function

>Example:

```

```python
Creating a set using curly braces
fruits = {"apple", "banana", "cherry"}

Creating a set using the set() function
numbers = set([1, 2, 3, 4, 5])
...

```

#### **\*\*Accessing and Modifying Set Items\*\***

While us cannot access individual items by index, us can check for membership and perform operations

>Example:

```

```python
# Checking membership
has_apple = "apple" in fruits # Output: True

# Adding an item

```

```
fruits.add("orange")
```

```
# Removing an item
```

```
fruits.remove("banana") # Raises KeyError if item is not present
```

```
...
```

```
**Set Operations**
```

Sets support various mathematical set operations, such as ``union``, ``intersection``, and ``diff``.

>Example:

```
```python
```

```
Union of two sets
```

```
set1 = {1, 2, 3}
```

```
set2 = {3, 4, 5}
```

```
union = set1 | set2 # Result: {1, 2, 3, 4, 5}
```

```
Intersection of two sets
```

```
intersection = set1 & set2 # Result: {3}
```

```
Difference between two sets
```

```
difference = set1 - set2 # Result: {1, 2}
```

```
Symmetric difference (items in either set, but not in both)
```

```
symmetric_difference = set1 ^ set2 # Result: {1, 2, 4, 5}
```

```
...
```

```
Set Methods
```

Python provides several built-in methods for set operations:

1. ``add(item)``: Adds an item to the set.

2. ``remove(item)``: Removes an item from the set; raises `KeyError` if item is not present.

3. ``discard(item)``: Removes an item from the set if present; does not raise an error if item

4. ``pop()``: Removes and returns an arbitrary item from the set.

5. ``clear()``: Removes all items from the set.

>Example:

```
```python
```

```
# Using set methods
```

```
set1 = {1, 2, 3}
```

```
set1.add(4) # set1 is now {1, 2, 3, 4}
```



```

set1.remove(2)      # set1 is now {1, 3, 4}
set1.discard(5)     # No error, set1 remains {1, 3, 4}
item = set1.pop()   # Removes and returns an arbitrary item, e.g., 1
set1.clear()        # set1 is now an empty set {}
...

```

Frozen Sets

Frozen sets are a built-in data type in Python that are similar to sets but are immutable. On

Characteristics of Frozen Sets

- **Unordered** : The items in a frozen set do not have a specific order and may change.
- **Immutable** : Unlike regular sets, frozen sets cannot be altered after creation. No items
- **Unique** : Like sets, frozen sets do not allow duplicate items; all items must be unique.
- **Unindexed** : Frozen sets do not support indexing or slicing.

Creating Frozen Sets

Frozen sets are created using the `frozenset()` function, which takes an iterable as an argument.

>Example:

```

```python
Creating a frozen set
numbers = frozenset([1, 2, 3, 4, 5])

Creating a frozen set from a set
fruits = frozenset({"apple", "banana", "cherry"})
...

```

#### \*\*Accessing and Modifying Frozen Set Items\*\*

Frozen sets do not support modification operations such as adding or removing items. However

>Example:

```

```python
# Checking membership
has_apple = "apple" in fruits # Output: True

# Since frozenset is immutable, we cannot use add() or remove() methods
...

```

Set Operations with Frozen Sets

Frozen sets support various mathematical set operations similar to regular sets, such as union.

>Example:

```
```python
Union of two frozen sets
set1 = frozenset([1, 2, 3])
set2 = frozenset([3, 4, 5])
union = set1 | set2 # Result: frozenset({1, 2, 3, 4, 5})

Intersection of two frozen sets
intersection = set1 & set2 # Result: frozenset({3})

Difference between two frozen sets
difference = set1 - set2 # Result: frozenset({1, 2})

Symmetric difference (items in either set, but not in both)
symmetric_difference = set1 ^ set2 # Result: frozenset({1, 2, 4, 5})
```

**Frozen Set Methods**
```

Frozen sets have a subset of the methods available to regular sets. The available methods include:

1. `copy()` : Returns a shallow copy of the frozen set.
2. `difference(other)` : Returns a new frozen set with elements in the original frozen set but not in `other`.
3. `intersection(other)` : Returns a new frozen set with elements common to both frozen sets.
4. `union(other)` : Returns a new frozen set with elements from both frozen sets.
5. `symmetric_difference(other)` : Returns a new frozen set with elements in either frozen set but not in both.

>Example:

```
```python
Using frozen set methods
set1 = frozenset([1, 2, 3])
set2 = frozenset([3, 4, 5])

Getting the difference
difference = set1.difference(set2) # Result: frozenset({1, 2})

Getting the intersection
intersection = set1.intersection(set2) # Result: frozenset({3})
```

```
Getting the union
union = set1.union(set2) # Result: frozenset({1, 2, 3, 4, 5})

Getting the symmetric difference
symmetric_difference = set1.symmetric_difference(set2) # Result: frozenset({1, 2, 4, 5})
...

Control Structures in Python
```

Control structures in Python allow us to control the flow of execution in our programs. They

### ### Conditional Statements

Conditional statements are used to execute code based on certain conditions. The primary con

>Syntax:

```
```python
if condition:
    # Code block to execute if condition is True
elif another_condition:
    # Code block to execute if another_condition is True
else:
    # Code block to execute if none of the above conditions are True
...
```
```

>Example: Program to classify a person based on his/her age.

```
```python
age = 20

if age < 18:
    print("us are a minor.")
elif age < 65:
    print("us are an adult.")
else:
    print("us are a senior citizen.")
...
```
```

### ### Looping Statements

Looping statements are used to repeat a block of code multiple times. Python supports for lo

#### #### For Loop

The `for` loop iterates over a sequence (like a list, tuple, or string) and executes a block

>Syntax:

```
```python
for item in sequence:
    # Code block to execute for each item
...

```

>Example: Program to print names of fruits saved in a list.

```
```python
Iterating over a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
 print(fruit)
...

```

#### While Loop

The `while` loop repeatedly executes a block of code as long as a specified condition is True

>Syntax:

```
```python
while condition:
    # Code block to execute while condition is True
...

```

>Example: Print all counting numbers less than 5.

```
```python
Counting from 0 to 4
count = 0
while count < 5:
 print(count)
 count += 1
...

```

### Control Flow Statements

Control flow statements alter the flow of execution within loops and conditionals.

#### #### Break Statement

The ``break`` statement exits the current loop, regardless of the loop's condition.

>Example: Program to exit from the printing of whole numbers less than 10, while trigger 5.

```
```python
for i in range(10):
    if i == 5:
        break
    print(i)
# Output: 0 1 2 3 4
```
```

#### #### Continue Statement

The ``continue`` statement skips the rest of the code inside the current loop iteration and proceeds to the next iteration.

>Example: Program to print all the whole numbers in the range 5 except 2.

```
```python
for i in range(5):
    if i == 2:
        continue
    print(i)
# Output: 0 1 3 4
```
```

#### #### Pass Statement

The ``pass`` statement is a placeholder that does nothing and is used when a statement is syntactically required but no action is needed.

>Example: Program to print all the whole numbers in the range 5 except 3.

```
```python
for i in range(5):
    if i == 3:
        pass # Placeholder for future code
    else:
        print(i)
# Output: 0 1 2 4
```
```

::: {.callout-caution collapse="true"}

## ## Cautions When Using Control Flow Structures

Control flow structures are essential in Python programming for directing the flow of execution.

### \*\*Infinite Loops\*\*

- **\*\*Issue\*\***: A ``while`` loop with a condition that never becomes ``False`` can lead to an infinite loop.
- **\*\*Caution\*\***: Always ensure that the condition in a ``while`` loop will eventually become ``False``.

### \*\*Example\*\*

```
```python
# Infinite loop example
count = 0
while count < 5:
    print(count)
    # Missing count increment, causing an infinite loop
...
:::
```

Functions in Python Programming

Functions are a fundamental concept in Python programming that enable code reuse, modularity, and organization.

What is a Function

A function is a named block of code designed to perform a specific task. Functions can take arguments and return results.

Defining a Function

In Python, functions are defined using the ``def`` keyword, followed by the function name, parameters in parentheses, and a colon.

>Syntax:

```
```python
def function_name(parameters):
 # Code block
 return result
...

```

>Example:

```
```python
def greet(name):
    """

```

```

    Returns a greeting message for the given name.
    """
    return f"Hello, {name}!"
...

#### Relevance of functions in Programming
1. *Code Reusability* : Functions allow us to define a piece of code once and reuse it in multiple places.
2. *Modularity* : Functions break down complex problems into smaller, manageable pieces. Each function handles a specific task.
3. *Abstraction* : Functions enable us to abstract away the implementation details. We can use a function without knowing its internal workings.
4. *Testing and Debugging* : Functions allow us to test individual components of our code separately, making it easier to find and fix errors.
5. *Library Creation* : Functions are the building blocks of libraries and modules. By organizing code into functions, we can create reusable libraries for others to use.

>Example: Creating a Simple Library

**Stage 1:** Define Functions in a Module

```python
my_library.py

def add(a, b):
 """
 Returns the sum of two numbers.
 """
 return a + b

def multiply(a, b):
 """
 Returns the product of two numbers.
 """
 return a * b
...

Stage 2: Use the Library in Another Program

```python
# main.py

import my_library

result_sum = my_library.add(5, 3)

```

```
result_product = my_library.multiply(5, 3)
```

```
print(f"Sum: {result_sum}")
print(f"Product: {result_product}")
```
```

## Object-Oriented Programming (OOP) in Python

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to design and

### Key Concepts of OOP

1. **Classes and Objects**

- **Class**: A class is a blueprint for creating objects. It defines a set of attributes and
- **Object**: An object is an instance of a class. It is a specific realization of the class

#### Example

```
```python
# Defining a class
class Dog:
    def __init__(self, name, age):
        self.name = name # Attribute
        self.age = age    # Attribute

    def bark(self):
        return "Woof!"    # Method

# Creating an object of the class
my_dog = Dog(name="Buddy", age=3)

# Accessing attributes and methods
print(my_dog.name) # Output: Buddy
print(my_dog.age)  # Output: 3
print(my_dog.bark()) # Output: Woof!
```
```

2. **Encapsulation**

Encapsulation is the concept of bundling data (attributes) and methods (functions) that oper

>Example: Controll the access to member variables using encapsulation.

```
```python
```



```

class Account:
    def __init__(self, balance):
        self.__balance = balance # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def get_balance(self):
        return self.__balance

# Creating an object of the class
my_account = Account(balance=1000)
my_account.deposit(500)

print(my_account.get_balance()) # Output: 1500
# print(my_account.__balance) # This will raise an AttributeError
...

```

3. ****Inheritance****

Inheritance is a mechanism in which a new class (child or derived class) inherits attributes

>Example: Demonstrating usage of attributes of base class in the derived classes.

```

...python
# Base class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return "Some sound"

# Derived class
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name) # Calling the constructor of the base class
        self.breed = breed

    def speak(self):
        return "Woof!"

# Another derived class

```

```

class Cat(Animal):
    def __init__(self, name, color):
        super().__init__(name) # Calling the constructor of the base class
        self.color = color

    def speak(self):
        return "Meow!"

# Creating objects of the derived classes
dog = Dog(name="Buddy", breed="Golden Retriever")
cat = Cat(name="Whiskers", color="Gray")

print(f"{dog.name} is a {dog.breed} and says {dog.speak()}") # Output: Buddy is a Golden Retriever
print(f"{cat.name} is a {cat.color} cat and says {cat.speak()}") # Output: Whiskers is a Gray cat

```

4. ****Polymorphism****

Polymorphism allows objects of different classes to be treated as objects of a common superclass.

>Example:

```

```python
class Bird:
 def fly(self):
 return "Flies in the sky"

class Penguin(Bird):
 def fly(self):
 return "Cannot fly, swims instead"

Creating objects of different classes
bird = Bird()
penguin = Penguin()

print(bird.fly()) # Output: Flies in the sky
print(penguin.fly()) # Output: Cannot fly, swims instead

```

#### **## Working with Files in Python**

File handling is an essential part of programming that allows us to work with data stored in files.

#### **\*\*Opening a File\*\***

In Python, we use the `open()` function to open a file. This function returns a file object,

>Syntax:

```
```python
file_object = open(file_path, mode)
```
```

Where,

- `file_path` : Path to the file (can be a relative or absolute path).
- `mode` : Specifies the file access mode (e.g., 'r' for reading, 'w' for writing, 'a' for append).

>Example:

```
```python
# Opening a file in read mode
file = open('example.txt', 'r')
```

Reading from a File
```

Once a file is opened, we can read its contents using various methods. Common methods include

- `read()` : Reads the entire file content.
- `readline()` : Reads a single line from the file.
- `readlines()` : Reads all the lines into a list.

>Example:

```
```python
# Reading the entire file
file_content = file.read()
print(file_content)

# Reading a single line
file.seek(0) # Move cursor to the start of the file
line = file.readline()
print(line)

# Reading all lines
file.seek(0)
lines = file.readlines()
print(lines)
```
```

```
...
```

### **\*\*Writing to a File\*\***

To write data to a file, we need to open the file in write ('w') or append ('a') mode. When c

>Example:

```
```python
# Opening a file in write mode
file = open('example.txt', 'w')

# Writing data to the file
file.write("Hello, World!\n")
file.write("Python file handling example.")

# Closing the file
file.close()
```
```

### **\*\*Closing a File\*\***

It is important to close a file after performing operations to ensure that all changes are s

>Example:

```
```python
f_1 = open('example.txt', 'w') # open the file example.txt to f_1
f_1.close() # close the file with handler 'f_1'
```
```

### **\*\*Using Context Managers\*\***

Context managers provide a convenient way to handle file operations, automatically managing

>Example:

```
```python
# Using context manager to open and write to a file
with open('example.txt', 'w') as file:
    file.write("This is written using a context manager.")
```

From Theory to Practice
```

In this section, we transition from theoretical concepts to practical applications by exploring

### ### Applications of Matrix Operations in Digital Image Processing

Matrix operations play a pivotal role in digital image processing, enabling a wide range of

#### #### Matrix Addition in Image Blending

Matrix addition is a fundamental operation in image processing, particularly useful in the

##### **\*\*Concept\*\***

When working with grayscale images, each image can be represented as a matrix where each element

>Example:

Consider two 2x2 grayscale images represented as matrices:

```
```python
image1= [[100, 150],[200, 250]]
image2=[[50, 100],[100, 150]]
```
```

To blend these images, we add the corresponding pixel values as:

```
```python
blended_image[i][j] = image1[i][j] + image2[i][j]
```
```

Ensure that the resulting pixel values do not exceed the maximum value allowed (255 for 8-bit)

##### **\*\*Python Implementation of image blending\*\***

Below is the Python code for blending two images using matrix addition:

```
::: {.cell execution_count=1}
``` {.python .cell-code}
def matrix_addition(image1, image2):
    rows = len(image1)
    cols = len(image1[0])
```

```

    blended_image = [[0] * cols for _ in range(rows)]

    for i in range(rows):
        for j in range(cols):
            blended_pixel = image1[i][j] + image2[i][j]
            blended_image[i][j] = min(blended_pixel, 255) # Clip to 255

    return blended_image

# Example matrices (images)
image1 = [[100, 150], [200, 250]]
image2 = [[50, 100], [100, 150]]

blended_image = matrix_addition(image1, image2)
print("Blended Image:")
for row in blended_image:
    print(row)
...

::: {.cell-output .cell-output-stdout}
...
Blended Image:
[150, 250]
[255, 255]
...

:::
:::

```

Image blending is a powerful technique with numerous real-time applications. It is widely used in various fields such as computer vision, image processing, and digital image processing.

```

::: {.callout-tip title="Image Blending as Basic Arithmetic with Libraries"}
In upcoming chapters, we will explore how specific libraries for image handling simplify the
adding two objects. Using these libraries, such as PIL (Python Imaging Library) or OpenCV, a
:::

```

Let's summarize a few more matrix operations and its uses in digital image processing tasks :

Matrix Subtraction in Image Sharpening

Matrix subtraction is a fundamental operation in image processing, essential for techniques like image sharpening, edge detection, and background removal.

****Concept****

In grayscale images, each pixel value represents the intensity of light at that point. Image

>Example:

Consider a grayscale image represented as a matrix:

```
```python
original_image [[100, 150, 200],[150, 200, 250],[200, 250, 100]]
```
```

To sharpen the image, we subtract a blurred version (smoothed image) from the original. This

```
```python
sharpened_image[i][j] = original_image[i][j] - blurred_image[i][j]
```
```

****Python Implementation****

Below is a simplified Python example of image sharpening using matrix subtraction:

```
```python
Original image matrix (grayscale values)
original_image = [
 [100, 150, 200],
 [150, 200, 250],
 [200, 250, 100]
]

Function to apply Gaussian blur (for demonstration, simplified as average smoothing)
def apply_blur(image):
 blurred_image = []
 for i in range(len(image)):
 row = []
 for j in range(len(image[0])):
 neighbors = []
 for dx in [-1, 0, 1]:
 for dy in [-1, 0, 1]:
 ni, nj = i + dx, j + dy
 if 0 <= ni < len(image) and 0 <= nj < len(image[0]):
 neighbors.append(image[ni][nj])
 blurred_value = sum(neighbors) // len(neighbors)
 row.append(blurred_value)
 blurred_image.append(row)
 return blurred_image
```
```

```

        return blurred_image

# Function for matrix subtraction (image sharpening)
def image_sharpening(original_image, blurred_image):
    sharpened_image = []
    for i in range(len(original_image)):
        row = []
        for j in range(len(original_image[0])):
            sharpened_value = original_image[i][j] - blurred_image[i][j]
            row.append(sharpened_value)
        sharpened_image.append(row)
    return sharpened_image

# Apply blur to simulate smoothed image
blurred_image = apply_blur(original_image)

# Perform matrix subtraction for image sharpening
sharpened_image = image_sharpening(original_image, blurred_image)

# Print the sharpened image
print("Sharpened Image:")
for row in sharpened_image:
    print(row)

...

```

Matrix Multiplication in Image Filtering (Convolution)

Matrix multiplication, specifically convolution in the context of image processing, is a function

****Concept****

In grayscale images, each pixel value represents the intensity of light at that point. Convolution

>Example:

Consider a grayscale image represented as a matrix:

```

```python
original_image= [[100, 150, 200, 250],
 [150, 200, 250, 300],
 [200, 250, 300, 350],
 [250, 300, 350, 400]]
...

```



To perform smoothing (averaging) using a simple kernel:

```
```python
[[1/9, 1/9, 1/9],
 [1/9, 1/9, 1/9],
 [1/9, 1/9, 1/9]]
```
```

The kernel is applied over the image using convolution:

```
```python
smoothed_image[i][j] = sum(original_image[ii][jj] * kernel[k][l] for all (ii, jj) in neighbors)
```
```

**\*\*Python Implementation\*\***

Here's a simplified Python example demonstrating convolution for image smoothing without external libraries:

```
```python
# Original image matrix (grayscale values)
original_image = [
    [100, 150, 200, 250],
    [150, 200, 250, 300],
    [200, 250, 300, 350],
    [250, 300, 350, 400]
]

# Define a simple kernel/filter for smoothing (averaging)
kernel = [
    [1/9, 1/9, 1/9],
    [1/9, 1/9, 1/9],
    [1/9, 1/9, 1/9]
]

# Function for applying convolution (image filtering)
def apply_convolution(image, kernel):
    height = len(image)
    width = len(image[0])
    ksize = len(kernel)
    kcenter = ksize // 2 # Center of the kernel

    # Initialize result image
    filtered_image = [[0]*width for _ in range(height)]
```
```

```

Perform convolution
for i in range(height):
 for j in range(width):
 sum = 0.0
 for k in range(ksize):
 for l in range(ksize):
 ii = i + k - kcenter
 jj = j + l - kcenter
 if ii >= 0 and ii < height and jj >= 0 and jj < width:
 sum += image[ii][jj] * kernel[k][l]
 filtered_image[i][j] = int(sum)

return filtered_image

Apply convolution to simulate smoothed image (averaging filter)
smoothed_image = apply_convolution(original_image, kernel)

Print the smoothed image
print("Smoothed Image:")
for row in smoothed_image:
 print(row)
...

```

#### Determinant: Image Transformation

**\*\*Concept\*\***

The determinant of a transformation matrix helps understand how transformations like scaling

>Example:

Here, we compute the determinant of a scaling matrix to understand how the scaling affects t

```

```python
def calculate_determinant(matrix):
    a, b = matrix[0]
    c, d = matrix[1]
    return a * d - b * c

# Example transformation matrix (scaling)
transformation_matrix = [[2, 0], [0, 2]]
determinant = calculate_determinant(transformation_matrix)
print(f"Determinant of the transformation matrix: {determinant}")

```

```
...
```

This value indicates how the transformation scales the image area.

Rank: Image Rank and Data Compression

****Concept****

The rank of a matrix indicates the number of linearly independent rows or columns. In image c

>Example:

Here, we compute the rank of a matrix representing an image. A lower rank might indicate that

```
```python
```

```
def matrix_rank(matrix):
 def is_zero_row(row):
 return all(value == 0 for value in row)

 def row_echelon_form(matrix):
 A = [row[:] for row in matrix]
 m = len(A)
 n = len(A[0])
 rank = 0
 for i in range(min(m, n)):
 if A[i][i] != 0:
 for j in range(i + 1, m):
 factor = A[j][i] / A[i][i]
 for k in range(i, n):
 A[j][k] -= factor * A[i][k]
 rank += 1
 return rank

 return row_echelon_form(matrix)

Example matrix (image)
image_matrix = [[1, 2], [3, 4]]
rank = matrix_rank(image_matrix)
print(f"Rank of the image matrix: {rank}")
```
```

Matrix Operations Using `Python` Libraries

Introduction

In this section, we will explore the computational aspects of basic matrix algebra using Pytl

Introduction to SymPy

`SymPy` is a powerful Python library designed for symbolic mathematics. It provides tools for

Key Matrix Functions in SymPy

- **Matrix Addition**: Adds two matrices element-wise.
- **Matrix Subtraction**: Subtracts one matrix from another element-wise.
- **Matrix Multiplication**: Multiplies two matrices using the dot product.
- **Matrix Power**: Raises a matrix to a given power using matrix multiplication.

>>>Example 1: Matrix Addition<<<

Pseudocode

```
```python
FUNCTION matrix_add():
 # Define matrices A and B
 A = [[1, 2], [3, 4]]
 B = [[5, 6], [7, 8]]

 # Check if matrices A and B have the same dimensions
 if dimensions_of(A) != dimensions_of(B):
 raise ValueError("Matrices must have the same dimensions")

 # Initialize result matrix with zeros
 result = [[0 for _ in range(len(A[0]))] for _ in range(len(A))]

 # Add corresponding elements from A and B
 for i in range(len(A)):
 for j in range(len(A[0])):
 result[i][j] = A[i][j] + B[i][j]

Return the result matrix
 return result
ENDFUNCTION
```
```

Python implementation of the above pseudocode is given below:

```
::: {.cell execution_count=2}
``` {.python .cell-code}
import sympy as sy
```

```

sy.init_printing()
Define matrices A and B
A = sy.Matrix([[1, 2], [3, 4]])
B = sy.Matrix([[5, 6], [7, 8]])

Add matrices
C = A + B

Print the result in symbolic form
print("Matrix Addition Result:")
display(C)

Convert to LaTeX code for documentation or presentation
#latex_code = sy.latex(C)
#print("LaTeX Code for Addition Result:")
#print(latex_code)
...

::: {.cell-output .cell-output-stdout}
...
Matrix Addition Result:
...
:::

::: {.cell-output .cell-output-display}
{fig-pos='H'}
:::
:::

>*Example 2: Matrix Subtraction*

Pseudocode:

```python
# Define matrices A and B
A = [[5, 6], [7, 8]]
B = [[1, 2], [3, 4]]

# Check if matrices A and B have the same dimensions
if dimensions_of(A) != dimensions_of(B):
    raise ValueError("Matrices must have the same dimensions")

```

```

# Initialize result matrix with zeros
result = [[0 for _ in range(len(A[0]))] for _ in range(len(A))]

# Subtract corresponding elements from A and B
for i in range(len(A)):
    for j in range(len(A[0])):
        result[i][j] = A[i][j] - B[i][j]

# Return the result matrix
return result

```

Python implementation of the above pseudocode is given below:

```

::: {.cell execution_count=3}
``` {.python .cell-code}
from sympy import Matrix
Define matrices A and B
A = Matrix([[5, 6], [7, 8]])
B = Matrix([[1, 2], [3, 4]])

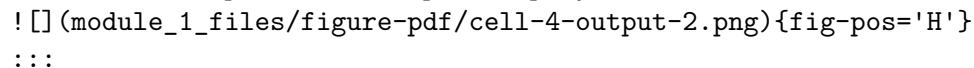
Subtract matrices
C = A - B

Print the result in symbolic form
print("Matrix Subtraction Result:")
display(C)

Convert to LaTeX code for documentation or presentation
latex_code = sy.latex(C)
print("LaTeX Code for Subtraction Result:")
print(latex_code)
```

::: {.cell-output .cell-output-stdout}
```
Matrix Subtraction Result:
```

:::

::: {.cell-output .cell-output-display}

:::

```

```

::: {.cell-output .cell-output-stdout}
...

```

LaTeX Code for Subtraction Result:

```

\left[\begin{matrix}4 & 4\end{matrix}\right] - \left[\begin{matrix}4 & 4\end{matrix}\right]
...

```

```

:::
:::

```

>**Example 3: Matrix Multiplication**

***Pseudocode:**

```

```python
Define matrices A and B
A = [[1, 2], [3, 4]]
B = [[5, 6], [7, 8]]

Check if the number of columns in A equals the number of rows in B
if len(A[0]) != len(B):
 raise ValueError("Number of columns in A must equal number of rows in B")

Initialize result matrix with zeros
result = [[0 for _ in range(len(B[0]))] for _ in range(len(A))]

Multiply matrices A and B
for i in range(len(A)):
 for j in range(len(B[0])):
 for k in range(len(B)):
 result[i][j] += A[i][k] * B[k][j]

Return the result matrix
return result

...

```

Python implementation of the above pseudocode is given below:

```

::: {.cell execution_count=4}
``` {.python .cell-code}
# Define matrices A and B
A = Matrix([[1, 2], [3, 4]])
B = Matrix([[5, 6], [7, 8]])

```



```

if len(A[0]) != len(B):
    raise ValueError("Number of columns in A must equal number of rows in B")

# Initialize result matrix with zeros
result = [[0 for _ in range(len(B[0]))] for _ in range(len(A))]

# Multiply matrices A and B
for i in range(len(A)):
    for j in range(len(B[0])):
        for k in range(len(B)):
            result[i][j] += A[i][k] * B[k][j]

# Return the result matrix
return result

```

Python code for implementing the above pseudocode is shown below:

```

::: {.cell execution_count=5}
``` {.python .cell-code}
Define matrices A and B
A = Matrix([[1, 2], [3, 4]])
B = Matrix([[5, 6], [7, 8]])

Multiply matrices
M = A * B

Print the result in symbolic form
print("Matrix Multiplication Result:")
display(M)

Convert to LaTeX code for documentation or presentation
latex_code = sy.latex(M)
print("LaTeX Code for Multiplication Result:")
print(latex_code)
```

::: {.cell-output .cell-output-stdout}
```
Matrix Multiplication Result:
```

:::

::: {.cell-output .cell-output-display}

```

```
{fig-pos='H'}
:::
```

```
::: {.cell-output .cell-output-stdout}
```
```

LaTeX Code for Multiplication Result:

```
\left[\begin{matrix}19 & 22\\43 & 50\end{matrix}\right]
```
```

```
:::
:::
```

> **Example 4: Matrix Power**

Pseudocode:

```
```python
Define matrix A and power n
A = [[1, 2], [3, 4]]
n = 2

Initialize result matrix as identity matrix
result = identity_matrix_of(len(A))

Compute A raised to the power of n
for _ in range(n):
 result = matrix_multiply(result, A)

Return the result matrix
return result

```
```

Python implementation of the above pseudocode is shown below:

```
::: {.cell execution_count=6}
``` {.python .cell-code}
Define matrix A
A = Matrix([[1, 2], [3, 4]])

Compute matrix A raised to the power of 2
n = 2
C = A**n
```

```

Print the result in symbolic form
print("Matrix Power Result:")
display(C)

Convert to LaTeX code for documentation or presentation
latex_code = sy.latex(C)
print("LaTeX Code for Power Result:")
print(latex_code)
```

::: {.cell-output .cell-output-stdout}
```
Matrix Power Result:
```

:::

::: {.cell-output .cell-output-display}
{fig-pos='H'}
:::

::: {.cell-output .cell-output-stdout}
```
LaTeX Code for Power Result:
\left[\begin{matrix}7 & 10\\15 & 22\end{matrix}\right]
```

:::
:::

#### Introduction to PIL for Image Manipulation

The `PIL` (Python Imaging Library), now known as `Pillow`, provides essential tools for open.

Matrix operations have significant applications in digital image processing. These operations

1. **Matrix Addition: Image Blending**

Matrix addition can be used to blend two images by adding their pixel values. This process ca

Example 1: Simple Image Blending

::: {.cell execution_count=7}
```python .cell-code}

```

```

import numpy as np
from PIL import Image
import urllib.request
urllib.request.urlretrieve('http://lenna.org/len_top.jpg',"input.jpg")
img1 = Image.open("input.jpg") #loading first image

urllib.request.urlretrieve('https://www.keralatourism.org/images/destination/large/thekkekud.

img2 = Image.open("input2.jpg")# loading second image

Resize second image to match the size of the first image
img2 = img2.resize(img1.size)
Convert images to numpy arrays
arr1 = np.array(img1)
arr2 = np.array(img2)

Add the images
blended_arr = arr1 + arr2

Clip the values to be in the valid range [0, 255]
blended_arr = np.clip(blended_arr, 0, 255).astype(np.uint8)

Convert back to image
blended_img = Image.fromarray(blended_arr)

Save or display the blended image
#blended_img.save('blended_image.jpg')
#blended_img.show()
#blended_img #display the blended image
...
:::

```

The input and output images are shown below:

```

::: {.cell execution_count=8}
``` {.python .cell-code}
img1
...

::: {.cell-output .cell-output-display execution_count=8}
{fig-pos='H'}
:::

```

```

:::

::: {.cell execution_count=9}
``` {.python .cell-code}
img2
```

::: {.cell-output .cell-output-display execution_count=9}
{fig-pos='H'}
:::
:::

::: {.cell execution_count=10}
``` {.python .cell-code}
blended_img
```

::: {.cell-output .cell-output-display execution_count=10}
{fig-pos='H'}
:::
:::

> **Example 2: Weighted Image Blending**

::: {.cell execution_count=11}
``` {.python .cell-code}
Blend with weights
alpha = 0.7
blended_arr = alpha * arr1 + (1 - alpha) * arr2

Clip the values to be in the valid range [0, 255]
blended_arr = np.clip(blended_arr, 0, 255).astype(np.uint8)

Convert back to image
blended_img = Image.fromarray(blended_arr)

Save or display the weighted blended image
#blended_img.save('weighted_blended_image.jpg')
#blended_img.show()
blended_img

```

```
...
```

```
::: {.cell-output .cell-output-display execution_count=11}
{fig-pos='H'}
:::
:::
```

#### #### Matrix Subtraction: Image Sharpening

Matrix subtraction can be used to sharpen images by subtracting a blurred version of the image.

> \*\*Example 1: Sharpening by Subtracting Blurred Image\*\*

```
::: {.cell execution_count=12}
``` {.python .cell-code}  
from PIL import Image, ImageFilter  
# Convert image to grayscale for simplicity  
img_gray = img1.convert('L')  
arr = np.array(img_gray)  
  
# Apply Gaussian blur  
blurred_img = img_gray.filter(ImageFilter.GaussianBlur(radius=5))  
blurred_arr = np.array(blurred_img)  
  
# Sharpen the image by subtracting blurred image  
sharpened_arr = arr - blurred_arr  
  
# Clip the values to be in the valid range [0, 255]  
sharpened_arr = np.clip(sharpened_arr, 0, 255).astype(np.uint8)  
  
# Convert back to image  
sharpened_img = Image.fromarray(sharpened_arr)  
  
# Save or display the sharpened image  
#sharpened_img.save('sharpened_image.jpg')  
#sharpened_img.show()  
sharpened_img  
```
```

```
::: {.cell-output .cell-output-display execution_count=12}
{fig-pos='H'}
:::
```

```
:::
```

#### #### Matrix Multiplication: Image Filtering (Convolution)

Matrix multiplication is used in image filtering to apply convolution kernels for various effects.

> \*\*Example 1: Applying a Convolution Filter\*\*

```
::: {.cell execution_count=13}
``` {.python .cell-code}
# Define a simple convolution kernel (e.g., edge detection)
kernel = np.array([
    [1, 0, -1],
    [1, 0, -1],
    [1, 0, -1]
])

# Convert the image to grayscale for simplicity
img_gray = img1.convert('L')
arr = np.array(img_gray)

# Apply convolution
filtered_arr = np.zeros_like(arr)
for i in range(1, arr.shape[0] - 1):
    for j in range(1, arr.shape[1] - 1):
        region = arr[i-1:i+2, j-1:j+2]
        filtered_arr[i, j] = np.sum(region * kernel)

# Convert back to image
filtered_img = Image.fromarray(np.clip(filtered_arr, 0, 255).astype(np.uint8))

# Save or display the filtered image
#filtered_img.save('filtered_image.jpg')
#filtered_img.show()
filtered_img
```
```

```
::: {.cell-output .cell-output-display execution_count=13}
![] (module_1_files/figure-pdf/cell-14-output-1.png){fig-pos='H'}
:::
:::
```

>\*\*Example 2: Applying a Gaussian Blur Filter\*\*

```
::: {.cell execution_count=14}
``` {.python .cell-code}
# Define a Gaussian blur filter
blurred_img = img1.filter(ImageFilter.GaussianBlur(radius=5))

# Save or display the blurred image
#blurred_img.save('blurred_image.jpg')
#blurred_img.show()
blurred_img
```

::: {.cell-output .cell-output-display execution_count=14}
{fig-pos='H'}
:::
:::
```

#### Solving Systems of Equations and Applications

\*\*Introduction\*\*

Solving systems of linear equations is crucial in various image processing tasks, such as im

>\*\*Example 1: Solving a System of Equations\*\*

Consider the system:

```
$$\begin{cases}
2x + 3y = 13 \\
4x - y = 7
\end{cases}$$
```

>\*\*Python Implementation\*\*

```
::: {.cell execution_count=15}
``` {.python .cell-code}
from sympy import Matrix
# Define the coefficient matrix and constant matrix
A = Matrix([[2, 3], [4, -1]])
B = Matrix([13, 7])
```



```
# Solve the system of equations
solution = A.solve_least_squares(B)
# Print the solution
print("Solution to the System of Equations:")
print(solution)
```

```

```
::: {.cell-output .cell-output-stdout}
```

```

```
Solution to the System of Equations:
Matrix([[17/7], [19/7]])
```

```

```
:::
```

```
:::
```

## ## Conclusion

In this chapter, we transitioned from understanding fundamental matrix operations to applying

We further explored the application of these matrix operations to real-world image processing

By integrating theoretical understanding with practical implementation, this chapter reinforces

```
`<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6Ii4ifQ== -->`{=html}
```

```
````{=html}
```

```
<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6Ii4iLCJib29rSXRlbVR5cGUiOiJjaGFwdGVyIiwiaWYm9va
```
```

## # Transforming Linear Algebra to Computational Language

```
````````{.quarto-title-block template='C:\Program Files\Quarto\share\projects\book\pandoc\tit
---
```

```
title: Transforming Linear Algebra to Computational Language
```

```
execute:
```

```
  enabled: true
```

```
jupyter: python3
```

```
---
```

Introduction

In the first module, we established a solid foundation in matrix algebra by exploring pseudocode and implementing fundamental matrix operations using `Python`. We practiced key concepts such as matrix addition, subtraction, multiplication, and determinants through practical examples in image processing, leveraging the `SymPy` library for symbolic computation.

As we begin the second module, “**Transforming Linear Algebra to Computational Language**,” our focus will shift towards applying these concepts with greater depth and actionable insight. This module is designed to bridge the theoretical knowledge from matrix algebra with practical computational applications. You will learn to interpret and utilize matrix operations, solve systems of equations, and analyze the rank of matrices within a variety of real-world contexts.

A new concept we will introduce is the **Rank-Nullity Theorem**, which provides a fundamental relationship between the rank of a matrix and the dimensions of its null space. This theorem is crucial for understanding the solution spaces of linear systems and the properties of linear transformations. By applying this theorem, you will be able to gain deeper insights into the structure of solutions and the behavior of matrix transformations.

This transition will not only reinforce your understanding of linear algebra but also enhance your ability to apply these concepts effectively in computational settings. Through engaging examples and practical exercises, you will gain valuable experience in transforming abstract mathematical principles into tangible solutions, setting a strong groundwork for advanced computational techniques.

Relearning of Terms and Operations in Linear Algebra

In this section, we will revisit fundamental matrix operations such as addition, subtraction, scaling, and more through practical examples. Our goal is to transform theoretical linear algebra into modern computational applications. We will demonstrate these concepts using `Python`, focusing on practical and industrial applications.

Matrix Addition and Subtraction in Data Analysis

Matrix addition and subtraction are fundamental operations that help in combining datasets and analyzing differences.

Simple Example: Combining Quarterly Sales Data

We begin with quarterly sales data from different regions and combine them to get the total sales. The sales data is given in Table 2. A `ar` plot of the total sales is shown in Fig 1.

Table 2: Quarterly Sales Data

| Region | Q1 | Q2 | Q3 | Q4 |
|--------|------|------|------|------|
| A | 2500 | 2800 | 3100 | 2900 |
| B | 1500 | 1600 | 1700 | 1800 |

From Scratch Python Implementation:

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

# Quarterly sales data
sales_region_a = np.array([2500, 2800, 3100, 2900])
sales_region_b = np.array([1500, 1600, 1700, 1800])

# Combine sales data
total_sales = sales_region_a + sales_region_b

# Visualization
quarters = ['Q1', 'Q2', 'Q3', 'Q4']
plt.bar(quarters, total_sales, color='skyblue')
plt.xlabel('Quarter')
plt.ylabel('Total Sales')
plt.title('Combined Quarterly Sales Data for Regions A and B')
plt.show()
```

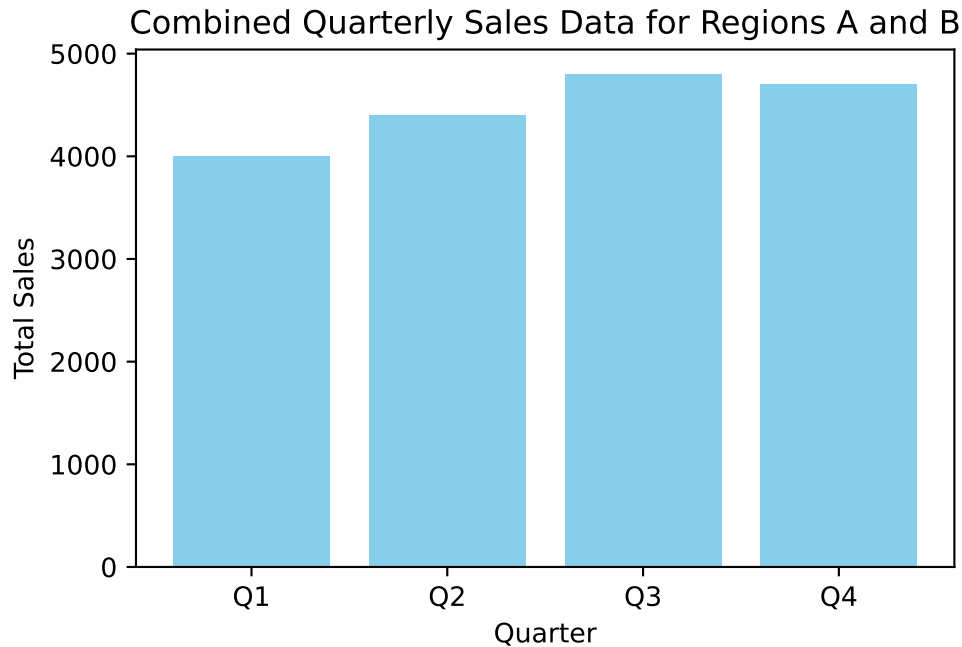


Figure 1: Computing Total Sales using Numpy aggregation method

In the above Python code, we have performed the aggregation operation with the NumPy method. Same can be done in a more data analysis style using pandas inorder to handle tabular data meaningfully. In this approach, quarterly sales data of each region is stored as DataFrames (like an excel sheet). Then we combine these two DataFrames into one. After that create a new row with index 'Total' and populate this row with sum of quarterly sales in Region A and Region B. Finally a bar plot is created using this 'Total' sales. Advantage of this approach is that we don't need the matplotlib library to create visualizations!. The EDA using this approach is shown in Fig 2.

```
import pandas as pd
import matplotlib.pyplot as plt

# DataFrames for quarterly sales data
df_a = pd.DataFrame({'Q1': [2500], 'Q2': [2800], 'Q3': [3100], 'Q4': [2900]},
                    ↪ index=['Region A'])
df_b = pd.DataFrame({'Q1': [1500], 'Q2': [1600], 'Q3': [1700], 'Q4': [1800]},
                    ↪ index=['Region B'])

# Combine data
df_combined = df_a.add(df_b, fill_value=0)
```

```
df_combined.loc["Total"] = df_combined.sum(axis=0)
# Visualization
df_combined.loc["Total"].plot(kind='bar', color=['green'])
plt.xlabel('Quarter')
plt.ylabel('Total Sales')
plt.title('Combined Quarterly Sales Data for Regions A and B')
plt.show()
```

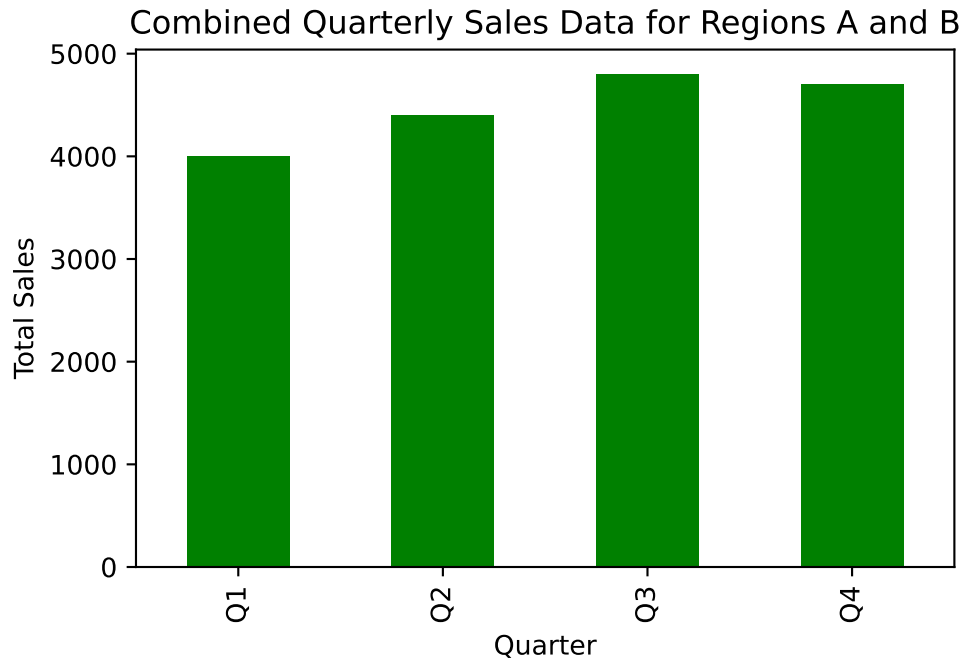


Figure 2: Computation of Total Sales using Pandas method

We can extend this in to more advanced examples. Irrespective to the size of the data, for representation and aggregation tasks matrix models are best options and are used in industry as a standard. Let us consider an advanced example to analyse difference in stock prices. For this example we are using a simulated data. The python code for this simulation process is shown in Fig 3.

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

# Simulated observed and predicted stock prices
observed_prices = np.random.uniform(100, 200, size=(100, 5))
```

```

predicted_prices = np.random.uniform(95, 210, size=(100, 5))

# Calculate the difference matrix
price_differences = observed_prices - predicted_prices

# Visualization
plt.imshow(price_differences, cmap='coolwarm', aspect='auto')
plt.colorbar()
plt.title('Stock Price Differences')
plt.xlabel('Stock Index')
plt.ylabel('Day Index')
plt.show()

```

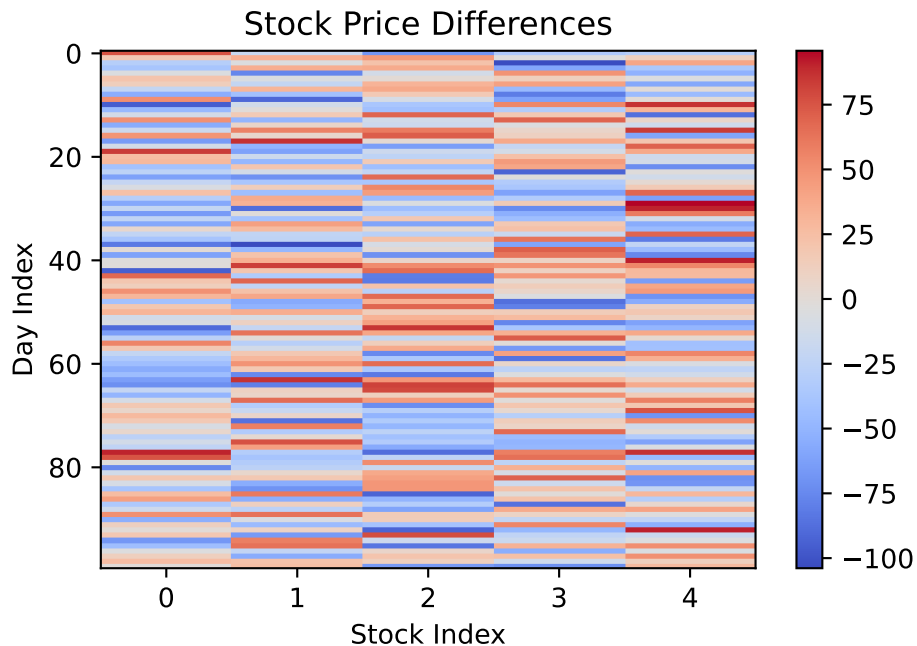


Figure 3: Demonstration of Stock Price simulated from a Uniform Distribution

Another important matrix operation relevant to data analytics and Machine Learning application is scaling. This is considered as a statistical tool to make various features (attributes) in to same scale so as to avoid unnecessary misleading impact in data analysis and its interpretation. In Machine Learning context, this pre-processing stage is inevitable so as to make the model relevant and usable.

Simple Example: Normalizing Employee Performance Data

Table 3: Employee Performance Data

| Employee | Metric A | Metric B |
|----------|----------|----------|
| X | 80 | 700 |
| Y | 90 | 800 |
| Z | 100 | 900 |
| A | 110 | 1000 |
| B | 120 | 1100 |

Using simple python code we can simulate the model for **min-max** scaling. The formula for **min-max** scaling is:

$$\text{min}_m\text{ax}(X) = \frac{X - \min(X)}{\max(X) - \min(X)}$$

For example, while applying the **min-max** scaling in the first value of Metric A, the scaled value is

$$\text{min}_m\text{ax}(80) \frac{80 - 80}{120 - 80} = 0$$

Similarly

$$\text{min}_m\text{ax}(100) \frac{100 - 80}{120 - 80} = 0.5$$

When we apply this formula to Metric A and Metric B, the scaled output from Table 3 will be as follows:

Table 4: Employee Performance Data

| Employee | Metric A | Metric B |
|----------|----------|----------|
| X | 0.00 | 0.00 |
| Y | 0.25 | 0.25 |
| Z | 0.50 | 0.50 |
| A | 0.75 | 0.75 |
| B | 1.00 | 1.00 |

It is interesting to look into the scaled data! In the original table (Table 3) it is looked like Metric B is superior. But from the scaled table (Table 4), it is clear that both the Metrics are representing same relative information. This will help us to identify the redundancy in measure and so skip any one of the Metric before analysis!.

The same can be achieved through a matrix operation. The **Python** implementation of this scaling process is shown in Fig 4.

```

import numpy as np
import matplotlib.pyplot as plt

# Employee performance data with varying scales
data = np.array([[80, 700], [90, 800], [100, 900], [110, 1000], [120, 1100]])

# Manual scaling
min_vals = np.min(data, axis=0)
max_vals = np.max(data, axis=0)
scaled_data = (data - min_vals) / (max_vals - min_vals)

# Visualization
plt.figure(figsize=(8, 5))
plt.subplot(1, 2, 1)
plt.imshow(data, cmap='viridis')
plt.title('Original Data')
plt.colorbar()

plt.subplot(1, 2, 2)
plt.imshow(scaled_data, cmap='viridis')
plt.title('Scaled Data')
plt.colorbar()

plt.show()

```

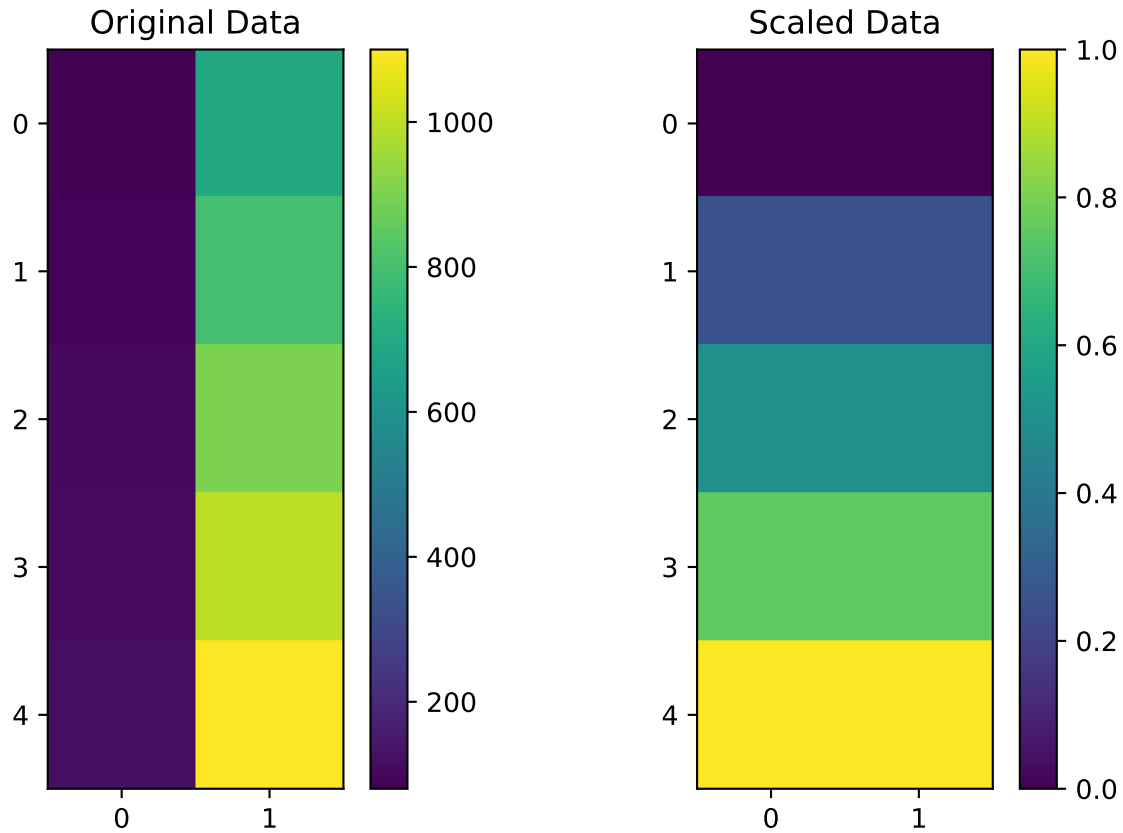



Figure 4: Total sales using **pandas** method

From the first sub plot, it is clear that there is a significant difference in the distributions (Metric A and Metric B values). But the second sub plot shows that both the distributions have same pattern and the values ranges between 0 and 1. In short the visualization is more appealing and self explanatory in this case.

i Note

The **min-max** scaling method will confine the feature values (attributes) into the range $[0, 1]$. So in effect all the features are scaled proportionally to the data spectrum.

Similarly, we can use the **standard scaling** (transformation to normal distribution) using the transformation $\frac{x - \bar{x}}{\sigma}$. Scaling table is given as a practice task to the reader. The python code for this operation is shown in Fig 5.

```

# Standard scaling from scratch
def standard_scaling(data):
    mean = np.mean(data, axis=0)
    std = np.std(data, axis=0)
    scaled_data = (data - mean) / std
    return scaled_data

# Apply standard scaling
scaled_data_scratch = standard_scaling(data)

print("Standard Scaled Data (from scratch):\n", scaled_data_scratch)

# Visualization
plt.figure(figsize=(6, 5))
plt.subplot(1, 2, 1)
plt.imshow(data, cmap='viridis')
plt.title('Original Data')
plt.colorbar()

plt.subplot(1, 2, 2)
plt.imshow(scaled_data_scratch, cmap='viridis')
plt.title('Scaled Data')
plt.colorbar()

plt.show()

```

```

Standard Scaled Data (from scratch):
[[-1.41421356 -1.41421356]
 [-0.70710678 -0.70710678]
 [ 0.          0.         ]
 [ 0.70710678  0.70710678]
 [ 1.41421356  1.41421356]]

```

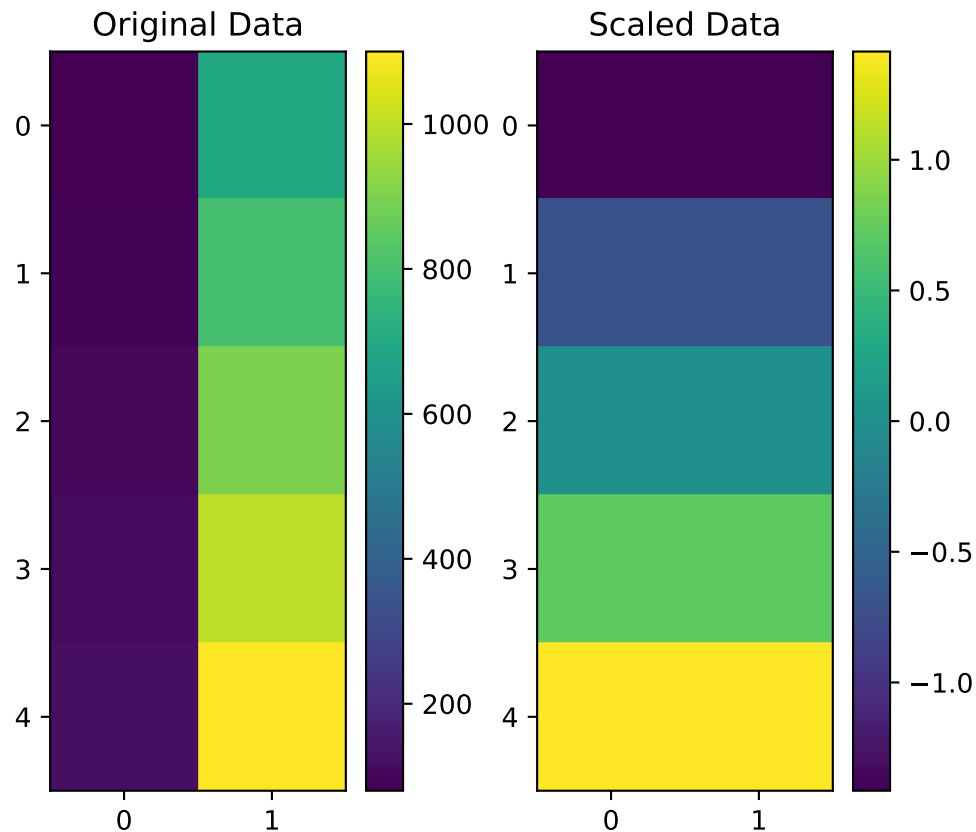


Figure 5: Min-max scaling using basic python

To understand the effect of standard scaling, let us consider Fig 6. This plot create the frequency distribution of the data as a histogram along with the density function. From the first sub-plot, it is clear that the distribution has multiple modes (peaks). When we apply the standard scaling, the distribution become un-modal(only one peek). This is demonstrated in the second sub-plot.

```
# Standard scaling from scratch
import seaborn as sns
# Create plots
plt.figure(figsize=(6, 5))

# Plot for original data
plt.subplot(1, 2, 1)
sns.histplot(data, kde=True, bins=10, palette="viridis")
plt.title('Original Data Distribution')
```

```
plt.xlabel('Value')
plt.ylabel('Frequency')

# Plot for standard scaled data
plt.subplot(1, 2, 2)
sns.histplot(scaled_data_scratch, kde=True, bins=10, palette="viridis")
plt.title('Standard Scaled Data Distribution')
plt.xlabel('Value')
plt.ylabel('Frequency')

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

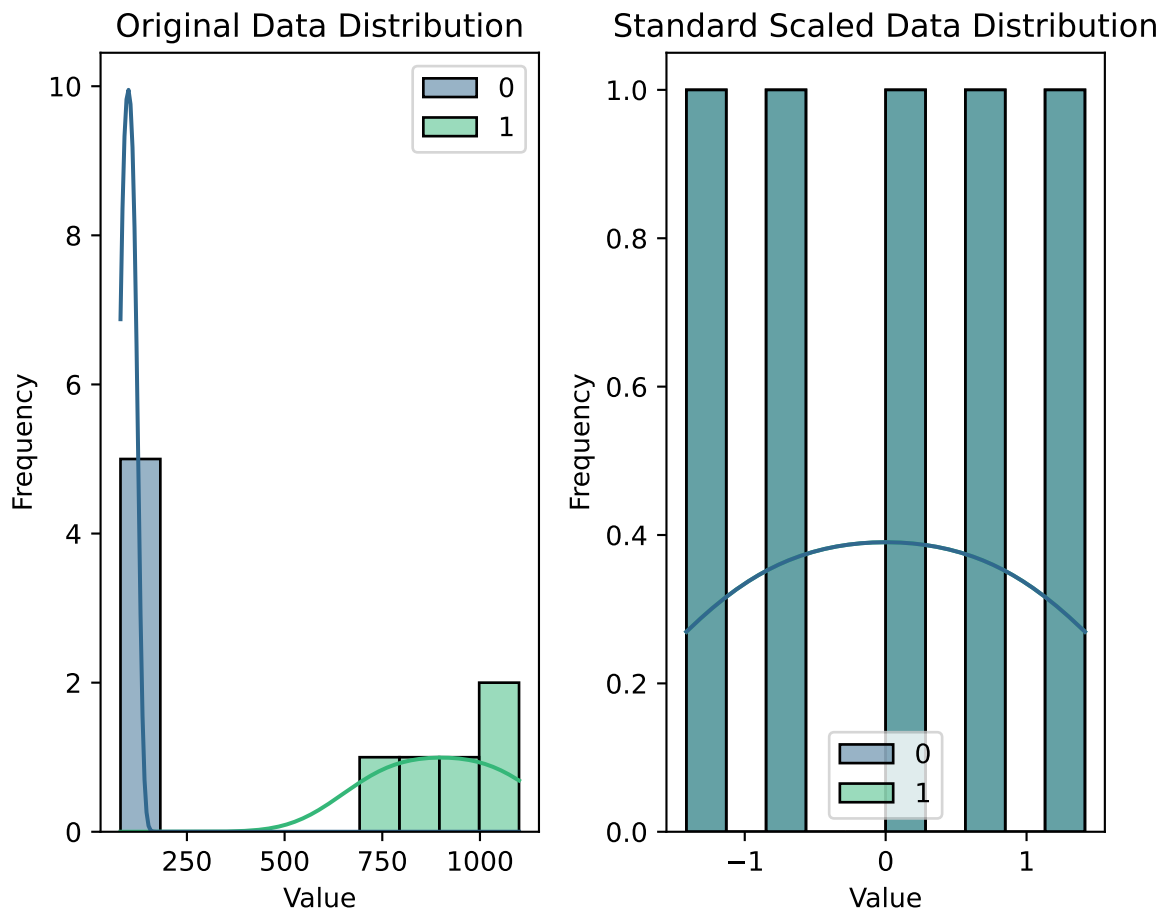


Figure 6: Impact of standard scaling on the distribution

A scatter plot showing the compare the impact of scaling on the given distribution is shown in Fig 7.

```
# Plot original and scaled data
plt.figure(figsize=(6, 5))

# Original Data
plt.subplot(1, 3, 1)
plt.scatter(data[:, 0], data[:, 1], color='blue')
plt.title('Original Data')
plt.xlabel('Metric A')
plt.ylabel('Metric B')

# Standard Scaled Data
plt.subplot(1, 3, 2)
plt.scatter(scaled_data_scratch[:, 0], scaled_data_scratch[:, 1],
    ↪ color='green')
plt.title('Standard Scaled Data')
plt.xlabel('Metric A (Standard Scaled)')
plt.ylabel('Metric B (Standard Scaled)')

# Min-Max Scaled Data
plt.subplot(1, 3, 3)
plt.scatter(scaled_data[:, 0], scaled_data[:, 1], color='red')
plt.title('Min-Max Scaled Data')
plt.xlabel('Metric A (Min-Max Scaled)')
plt.ylabel('Metric B (Min-Max Scaled)')

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

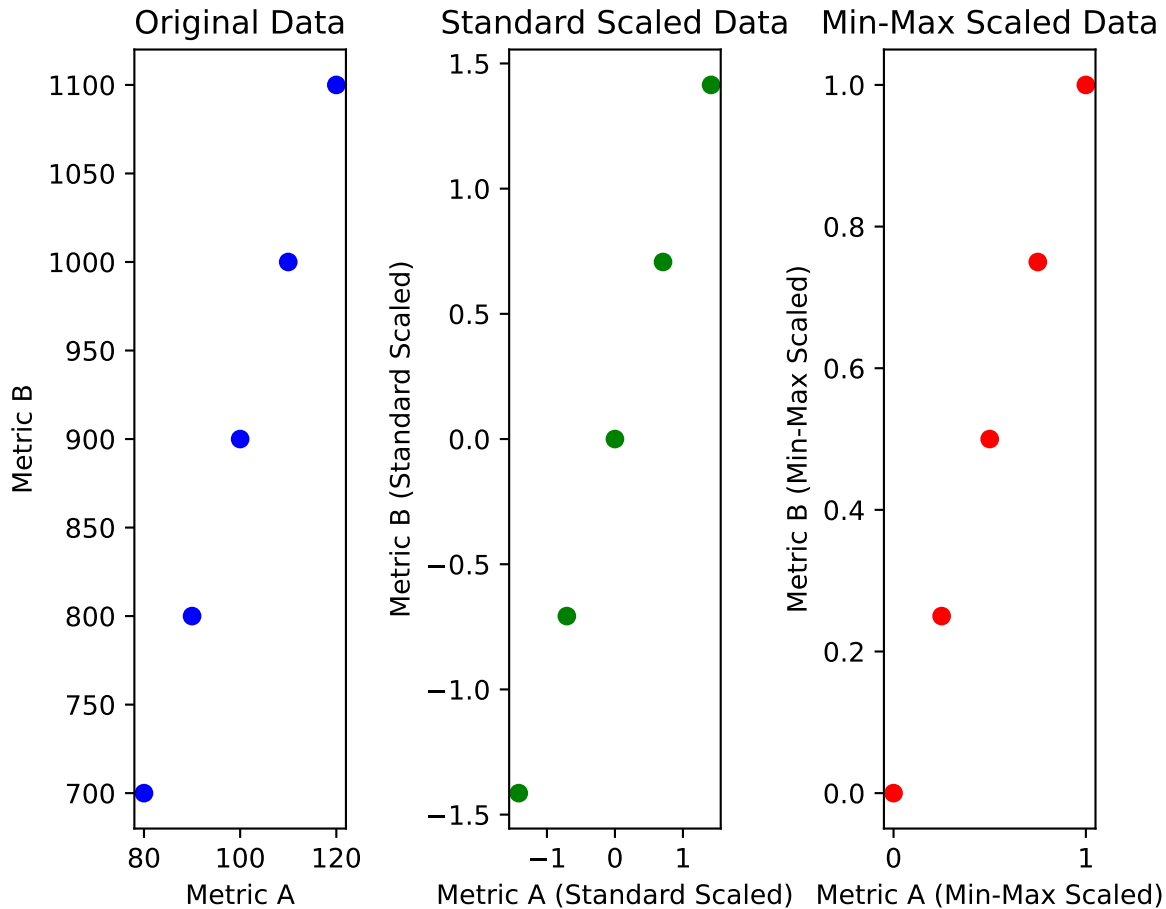


Figure 7: Comparison of impact of scaling on the distribution

From the Fig 7, it is clear that the scaling does not affect the pattern of the data, instead it just scale the distribution proportionally!

We can use the `scikit-learn` library for do the same thing in a very simple handy approach. The `python` code for this job is shown below.

```
from sklearn.preprocessing import MinMaxScaler

# Min-max scaling using sklearn
scaler = MinMaxScaler()
min_max_scaled_data_sklearn = scaler.fit_transform(data)

print("Min-Max Scaled Data (using sklearn):\n", min_max_scaled_data_sklearn)
```

Min-Max Scaled Data (using sklearn):

```
[[0.  0.  ]
 [0.25 0.25]
 [0.5  0.5  ]
 [0.75 0.75]
 [1.   1.   ]]
```

```
from sklearn.preprocessing import StandardScaler

# Standard scaling using sklearn
scaler = StandardScaler()
scaled_data_sklearn = scaler.fit_transform(data)

print("Standard Scaled Data (using sklearn):\n", scaled_data_sklearn)
```

Standard Scaled Data (using sklearn):

```
[[-1.41421356 -1.41421356]
 [-0.70710678 -0.70710678]
 [ 0.          0.         ]
 [ 0.70710678  0.70710678]
 [ 1.41421356  1.41421356]]
```

A scatter plot showing the impact on scaling is shown in Fig 8. This plot compare the mmin-max and standard-scaling.

```
# Plot original and scaled data
plt.figure(figsize=(6, 5))

# Original Data
plt.subplot(1, 3, 1)
plt.scatter(data[:, 0], data[:, 1], color='blue')
plt.title('Original Data')
plt.xlabel('Metric A')
plt.ylabel('Metric B')

# Standard Scaled Data
plt.subplot(1, 3, 2)
plt.scatter(scaled_data_sklearn[:, 0], scaled_data_sklearn[:, 1],
            color='green')
plt.title('Standard Scaled Data')
plt.xlabel('Metric A (Standard Scaled)')
```

```
plt.ylabel('Metric B (Standard Scaled)')

# Min-Max Scaled Data
plt.subplot(1, 3, 3)
plt.scatter(min_max_scaled_data_sklearn[:, 0], min_max_scaled_data_sklearn[:,
↪ 1], color='red')
plt.title('Min-Max Scaled Data')
plt.xlabel('Metric A (Min-Max Scaled)')
plt.ylabel('Metric B (Min-Max Scaled)')

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

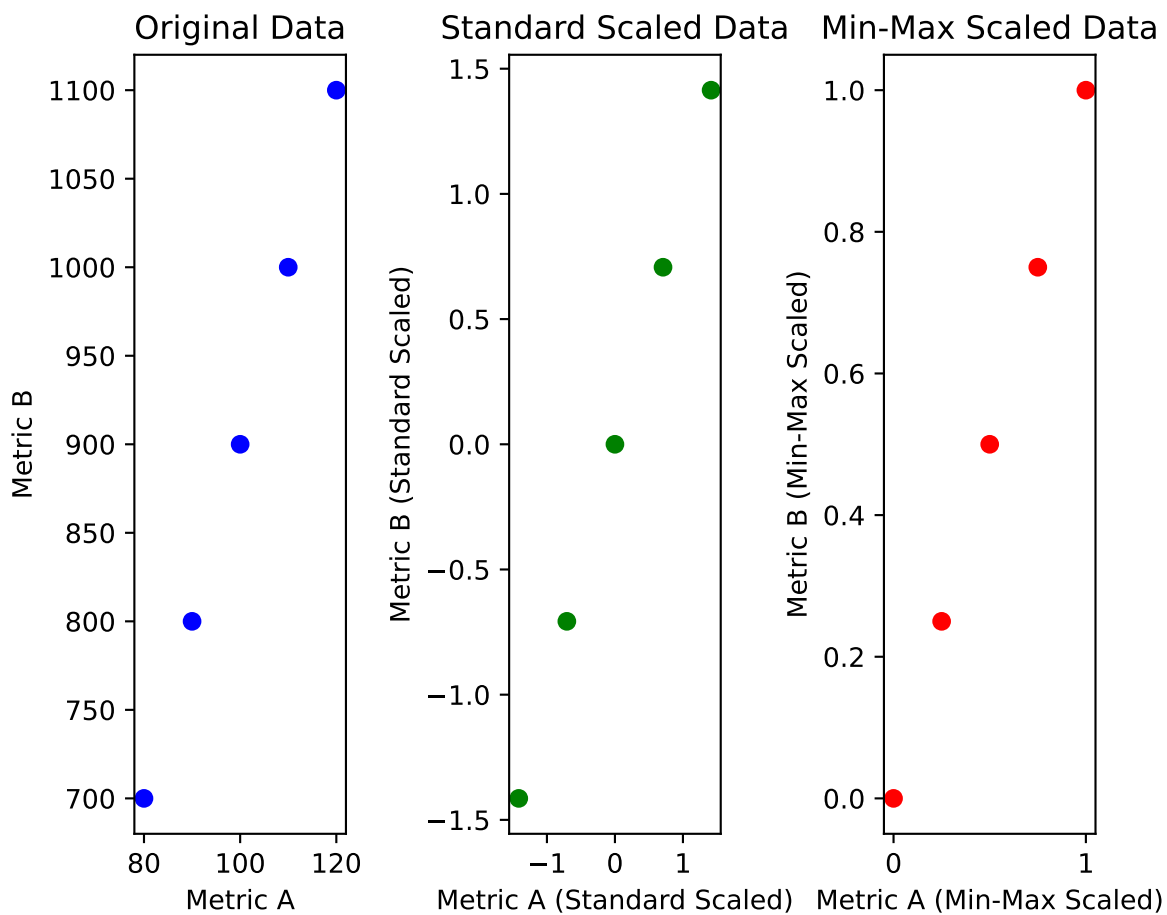


Figure 8: Comparison of Min-max and standard scalings with original data

More on Matrix Product and its Applications

In the first module of our course, we introduced matrix products as scalar projections, focusing on how matrices interact through basic operations. In this section, we will expand on this by exploring different types of matrix products that have practical importance in various fields. One such product is the *Hadamard product*, which is particularly useful in applications ranging from image processing to neural networks and statistical analysis. We will cover the definition, properties, and examples of the Hadamard product, and then delve into practical applications with simulated data.

Hadamard Product

The Hadamard product (or element-wise product) of two matrices is a binary operation that combines two matrices of the same dimensions to produce another matrix of the same dimensions, where each element is the product of corresponding elements in the original matrices.

! Definition (Hadamard Product):

For two matrices A and B of the same dimension $m \times n$, the Hadamard product $A \circ B$ is defined as:

$$(A \circ B)_{ij} = A_{ij} \cdot B_{ij}$$

where \cdot denotes element-wise multiplication.

i Properties of Hadamard Product

1. **Commutativity:**

$$A \circ B = B \circ A$$

2. **Associativity:**

$$(A \circ B) \circ C = A \circ (B \circ C)$$

3. **Distributivity:**

$$A \circ (B + C) = (A \circ B) + (A \circ C)$$

Some simple examples to demonstrate the Hadamard product is given below.

Example 1: Basic Hadamard Product

Given matrices:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

The Hadamard product $A \circ B$ is:

$$A \circ B = \begin{pmatrix} 1 \cdot 5 & 2 \cdot 6 \\ 3 \cdot 7 & 4 \cdot 8 \end{pmatrix} = \begin{pmatrix} 5 & 12 \\ 21 & 32 \end{pmatrix}$$

Example 2: Hadamard Product with Larger Matrices

Given matrices:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \quad B = \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix}$$

The Hadamard product $A \circ B$ is:

$$A \circ B = \begin{pmatrix} 1 \cdot 9 & 2 \cdot 8 & 3 \cdot 7 \\ 4 \cdot 6 & 5 \cdot 5 & 6 \cdot 4 \\ 7 \cdot 3 & 8 \cdot 2 & 9 \cdot 1 \end{pmatrix} = \begin{pmatrix} 9 & 16 & 21 \\ 24 & 25 & 24 \\ 21 & 16 & 9 \end{pmatrix}$$

In the following code chunks the computational process of Hadamard product is implemented in **Python**. Here both the from the scratch and use of external module versions are included.

1. Compute Hadamard Product from Scratch (without Libraries)

Here's how you can compute the Hadamard product manually:

```
# Define matrices A and B
A = [[1, 2, 3], [4, 5, 6]]
B = [[7, 8, 9], [10, 11, 12]]

# Function to compute Hadamard product
def hadamard_product(A, B):
    # Get the number of rows and columns
    num_rows = len(A)
    num_cols = len(A[0])

    # Initialize the result matrix
    result = [[0]*num_cols for _ in range(num_rows)]

    # Compute the Hadamard product
    for i in range(num_rows):
        for j in range(num_cols):
            result[i][j] = A[i][j] * B[i][j]
```

```

        return result

# Compute Hadamard product
hadamard_product_result = hadamard_product(A, B)

# Display result
print("Hadamard Product (From Scratch):")
for row in hadamard_product_result:
    print(row)

```

Hadamard Product (From Scratch):
 [7, 16, 27]
 [40, 55, 72]

2. Compute Hadamard Product Using SymPy

Here's how to compute the Hadamard product using SymPy:

```

import sympy as sp

# Define matrices A and B
A = sp.Matrix([[1, 2, 3], [4, 5, 6]])
B = sp.Matrix([[7, 8, 9], [10, 11, 12]])

# Compute Hadamard product using SymPy
Hadamard_product_sympy = A.multiply_elementwise(B)

# Display result
print("Hadamard Product (Using SymPy):")
print(Hadamard_product_sympy)

```

Hadamard Product (Using SymPy):
 Matrix([[7, 16, 27], [40, 55, 72]])

Practical Applications

Application 1: Image Masking

The Hadamard product can be used for image masking. Here's how you can apply a mask to an image and visualize it as shown in Fig 9.

```

import matplotlib.pyplot as plt
import numpy as np

# Simulated large image (2D array) using NumPy
image = np.random.rand(100, 100)

# Simulated mask (binary matrix) using NumPy
mask = np.random.randint(0, 2, size=(100, 100))

# Compute Hadamard product
masked_image = image * mask

# Plot original image and masked image
fig, ax = plt.subplots(1, 2, figsize=(12, 5))
ax[0].imshow(image, cmap='gray')
ax[0].set_title('Original Image')
ax[1].imshow(masked_image, cmap='gray')
ax[1].set_title('Masked Image')
plt.show()

```

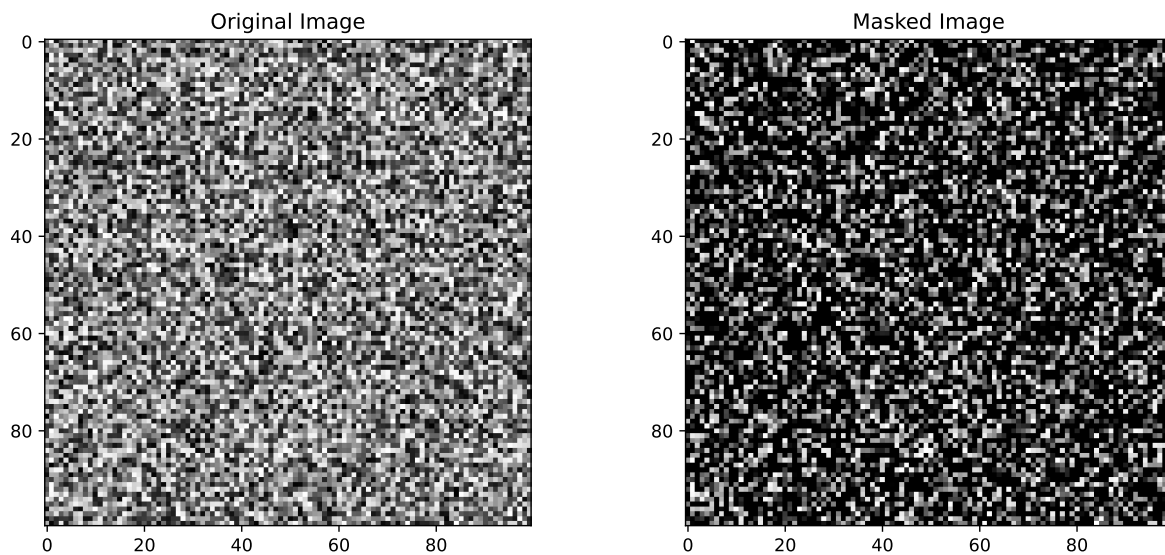


Figure 9: Demonstration of Masking in DIP using Hadamard Product

Application 2: Element-wise Scaling in Neural Networks

The Hadamard product can be used for dropout¹ in neural networks. A simple simulated example is given below.

```
# Simulated large activations (2D array) using NumPy
activations = np.random.rand(100, 100)

# Simulated dropout mask (binary matrix) using NumPy
dropout_mask = np.random.randint(0, 2, size=(100, 100))

# Apply dropout
dropped_activations = activations * dropout_mask

# Display results
print("Original Activations:")
print(activations)
print("\nDropout Mask:")
print(dropout_mask)
print("\nDropped Activations:")
print(dropped_activations)
```

Original Activations:

```
[[1.50241810e-01 4.33985567e-01 8.06799811e-01 ... 3.17691660e-02
 2.65296312e-01 1.95520882e-01]
 [3.95437889e-01 2.63829859e-01 5.16712381e-01 ... 4.27303912e-01
 8.20997228e-01 4.15547921e-01]
 [5.98635672e-01 5.22905270e-01 8.08365101e-01 ... 7.49319840e-01
 3.02072551e-01 4.00749514e-01]
 ...
 [5.51947232e-01 7.93479508e-01 3.29339876e-01 ... 6.07145901e-01
 7.96351298e-01 3.83801103e-01]
 [8.97921944e-01 5.98465485e-01 2.37418723e-02 ... 1.44208520e-01
 6.93063855e-01 3.02194139e-01]
 [1.10657022e-01 4.26475997e-01 2.83044675e-02 ... 5.35278273e-04
 5.20851155e-01 2.01807675e-01]]
```

Dropout Mask:

```
[[1 0 0 ... 0 1 0]
 [0 1 0 ... 0 0 1]
 [0 1 0 ... 0 0 0]
 ...]
```

¹A regularization techniques in Deep learning. This approach deactivate some selected neurons to control model over-fitting

```
[1 1 0 ... 1 1 1]
[1 0 0 ... 1 0 0]
[0 0 1 ... 1 1 0]]
```

Dropped Activations:

```
[[1.50241810e-01 0.00000000e+00 0.00000000e+00 ... 0.00000000e+00
 2.65296312e-01 0.00000000e+00]
[0.00000000e+00 2.63829859e-01 0.00000000e+00 ... 0.00000000e+00
 0.00000000e+00 4.15547921e-01]
[0.00000000e+00 5.22905270e-01 0.00000000e+00 ... 0.00000000e+00
 0.00000000e+00 0.00000000e+00]
...
[5.51947232e-01 7.93479508e-01 0.00000000e+00 ... 6.07145901e-01
 7.96351298e-01 3.83801103e-01]
[8.97921944e-01 0.00000000e+00 0.00000000e+00 ... 1.44208520e-01
 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 2.83044675e-02 ... 5.35278273e-04
 5.20851155e-01 0.00000000e+00]]
```

Application 3: Statistical Data Analysis

In statistics, the Hadamard product can be applied to scale covariance matrices. Here's how we can compute the covariance matrix using matrix operations and apply scaling. Following Python code demonstrate this.

```
import sympy as sp
import numpy as np

# Simulated large dataset (2D array) using NumPy
data = np.random.rand(100, 10)

# Compute the mean of each column
mean = np.mean(data, axis=0)

# Center the data
centered_data = data - mean

# Compute the covariance matrix using matrix product operation
cov_matrix = (centered_data.T @ centered_data) / (centered_data.shape[0] - 1)
cov_matrix_sympy = sp.Matrix(cov_matrix)

# Simulated scaling factors (2D array) using SymPy Matrix
scaling_factors = sp.Matrix(np.random.rand(10, 10))
```

```
# Compute Hadamard product
scaled_cov_matrix = cov_matrix_sympy.multiply(scaling_factors)

# Display results
print("Covariance Matrix:")
print(cov_matrix_sympy)
print("\nScaling Factors:")
print(scaling_factors)
print("\nScaled Covariance Matrix:")
print(scaled_cov_matrix)
```

Covariance Matrix:

Matrix([[0.0829729792128583, -0.00130926119215640, -0.00449358582009456, -0.0085411236596643,

Scaling Factors:

Matrix([[0.529376470070988, 0.563200130818408, 0.318460183719583, 0.797238283210820, 0.21628,

Scaled Covariance Matrix:

Matrix([[0.0392621161636811, 0.0379768336785228, 0.0163507664815586, 0.0591680411074049, 0.0,

Practice Problems

Problem 1: Basic Hadamard Product

Given matrices:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Find the Hadamard product $C = A \circ B$.

Solution:

$$C = \begin{bmatrix} 1 \cdot 5 & 2 \cdot 6 \\ 3 \cdot 7 & 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix}$$

Problem 2: Hadamard Product with Identity Matrix

Given matrices:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Find the Hadamard product $C = A \circ I$.

Solution:

$$C = \begin{bmatrix} 1 \cdot 1 & 2 \cdot 0 & 3 \cdot 0 \\ 4 \cdot 0 & 5 \cdot 1 & 6 \cdot 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \end{bmatrix}$$

Problem 3: Hadamard Product with Zero Matrix

Given matrices:

$$A = \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix}$$

$$Z = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Find the Hadamard product $C = A \circ Z$.

Solution:

$$C = \begin{bmatrix} 3 \cdot 0 & 4 \cdot 0 \\ 5 \cdot 0 & 6 \cdot 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Problem 4: Hadamard Product of Two Identity Matrices

Given identity matrices:

$$I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Find the Hadamard product $C = I_2 \circ I_3$ (extend I_2 to match dimensions of I_3).

Solution:

Extend I_2 to I_3 :

$$I_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 \cdot 1 & 0 \cdot 0 & 0 \cdot 0 \\ 0 \cdot 0 & 1 \cdot 1 & 0 \cdot 0 \\ 0 \cdot 0 & 0 \cdot 0 & 0 \cdot 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Problem 5: Hadamard Product with Random Matrices

Given random matrices:

$$A = \begin{bmatrix} 2 & 3 \\ 1 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 5 \\ 6 & 2 \end{bmatrix}$$

Find the Hadamard product $C = A \circ B$.

Solution:

$$C = \begin{bmatrix} 2 \cdot 0 & 3 \cdot 5 \\ 1 \cdot 6 & 4 \cdot 2 \end{bmatrix} = \begin{bmatrix} 0 & 15 \\ 6 & 8 \end{bmatrix}$$

Problem 6: Hadamard Product of 3x3 Matrices

Given matrices:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

$$B = \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

Find the Hadamard product $C = A \circ B$.

Solution:

$$C = \begin{bmatrix} 1 \cdot 9 & 2 \cdot 8 & 3 \cdot 7 \\ 4 \cdot 6 & 5 \cdot 5 & 6 \cdot 4 \\ 7 \cdot 3 & 8 \cdot 2 & 9 \cdot 1 \end{bmatrix} = \begin{bmatrix} 9 & 16 & 21 \\ 24 & 25 & 24 \\ 21 & 16 & 9 \end{bmatrix}$$

Problem 7: Hadamard Product of Column Vectors

Given column vectors:

$$u = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

$$v = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$$

Find the Hadamard product $w = u \circ v$.

Solution:

$$w = \begin{bmatrix} 2 \cdot 5 \\ 3 \cdot 6 \end{bmatrix} = \begin{bmatrix} 10 \\ 18 \end{bmatrix}$$

Problem 8: Hadamard Product with Non-Square Matrices

Given matrices:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

$$B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \end{bmatrix}$$

Find the Hadamard product $C = A \circ B$ (extend B to match dimensions of A).

Solution:

Extend B to match dimensions of A :

$$B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 7 & 8 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 \cdot 7 & 2 \cdot 8 \\ 3 \cdot 9 & 4 \cdot 10 \\ 5 \cdot 7 & 6 \cdot 8 \end{bmatrix} = \begin{bmatrix} 7 & 16 \\ 27 & 40 \\ 35 & 48 \end{bmatrix}$$

Problem 9: Hadamard Product in Image Processing

Given matrices representing image pixel values:

$$A = \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.5 & 1.5 \\ 2.0 & 0.5 \end{bmatrix}$$

Find the Hadamard product $C = A \circ B$.

Solution:

$$C = \begin{bmatrix} 10 \cdot 0.5 & 20 \cdot 1.5 \\ 30 \cdot 2.0 & 40 \cdot 0.5 \end{bmatrix} = \begin{bmatrix} 5 & 30 \\ 60 & 20 \end{bmatrix}$$

Problem 10: Hadamard Product in Statistical Data

Given matrices representing two sets of statistical data:

$$A = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{bmatrix}$$
$$B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Find the Hadamard product $C = A \circ B$.

Solution:

$$C = \begin{bmatrix} 5 \cdot 1 & 6 \cdot 2 & 7 \cdot 3 \\ 8 \cdot 4 & 9 \cdot 5 & 10 \cdot 6 \end{bmatrix} = \begin{bmatrix} 5 & 12 & 21 \\ 32 & 45 & 60 \end{bmatrix}$$

Inner Product of Matrices

The inner product of two matrices is a generalized extension of the dot product, where each matrix is treated as a vector in a high-dimensional space. For two matrices A and B of the same dimension $m \times n$, the inner product is defined as the sum of the element-wise products of the matrices.

! Definition (Inner product)

For two matrices A and B of dimension $m \times n$, the inner product $\langle A, B \rangle$ is given by:

$$\langle A, B \rangle = \sum_{i=1}^m \sum_{j=1}^n A_{ij} \cdot B_{ij}$$

where \cdot denotes element-wise multiplication.

! Properties

1. **Commutativity:**

$$\langle A, B \rangle = \langle B, A \rangle$$

2. **Linearity:**

$$\langle A + C, B \rangle = \langle A, B \rangle + \langle C, B \rangle$$

3. **Positive Definiteness:**

$$\langle A, A \rangle \geq 0$$

with equality if and only if A is a zero matrix.

Some simple examples showing the mathematical process of calculating the inner product is given bellow.

Example 1: Basic Inner Product

Given matrices:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

The inner product $\langle A, B \rangle$ is:

$$\langle A, B \rangle = 1 \cdot 5 + 2 \cdot 6 + 3 \cdot 7 + 4 \cdot 8 = 5 + 12 + 21 + 32 = 70$$

Example 2: Inner Product with Larger Matrices

Given matrices:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \quad B = \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix}$$

The inner product $\langle A, B \rangle$ is calculated as:

$$\begin{aligned} \langle A, B \rangle &= 1 \cdot 9 + 2 \cdot 8 + 3 \cdot 7 + 4 \cdot 6 + 5 \cdot 5 + 6 \cdot 4 + 7 \cdot 3 + 8 \cdot 2 + 9 \cdot 1 \\ &= 9 + 16 + 21 + 24 + 25 + 24 + 21 + 16 + 9 \\ &= 175 \end{aligned}$$

Practice Problems

Problem 1: Inner Product of 2x2 Matrices

Given matrices:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Solution:

$$\begin{aligned}
\langle A, B \rangle &= \sum_{i,j} A_{ij} B_{ij} \\
&= 1 \cdot 5 + 2 \cdot 6 + 3 \cdot 7 + 4 \cdot 8 \\
&= 5 + 12 + 21 + 32 \\
&= 70
\end{aligned}$$

Problem 2: Inner Product of 3x3 Matrices

Given matrices:

$$A = \begin{bmatrix} 1 & 0 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

$$B = \begin{bmatrix} 8 & 7 & 6 \\ 5 & 4 & 3 \\ 2 & 1 & 0 \end{bmatrix}$$

Solution:

$$\begin{aligned}
\langle A, B \rangle &= \sum_{i,j} A_{ij} B_{ij} \\
&= 1 \cdot 8 + 0 \cdot 7 + 2 \cdot 6 + \\
&\quad 3 \cdot 5 + 4 \cdot 4 + 5 \cdot 3 + \\
&\quad 6 \cdot 2 + 7 \cdot 1 + 8 \cdot 0 \\
&= 8 + 0 + 12 + 15 + 16 + 15 + 12 + 7 + 0 \\
&= 85
\end{aligned}$$

Problem 3: Inner Product of Diagonal Matrices

Given diagonal matrices:

$$A = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 6 & 0 \\ 0 & 0 & 7 \end{bmatrix}$$

Solution:

$$\begin{aligned} \langle A, B \rangle &= \sum_{i,j} A_{ij} B_{ij} \\ &= 2 \cdot 5 + 0 \cdot 0 + 0 \cdot 0 + \\ &\quad 0 \cdot 0 + 3 \cdot 6 + 0 \cdot 0 + \\ &\quad 0 \cdot 0 + 0 \cdot 0 + 4 \cdot 7 \\ &= 10 + 0 + 0 + 0 + 18 + 0 + 0 + 0 + 28 \\ &= 56 \end{aligned}$$

Problem 4: Inner Product of Column Vectors

Given column vectors:

$$u = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$v = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}$$

Solution:

$$\begin{aligned} \langle u, v \rangle &= \sum_i u_i v_i \\ &= 1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 \\ &= 4 + 10 + 18 \\ &= 32 \end{aligned}$$

Problem 5: Inner Product with Random Matrices

Given matrices:

$$A = \begin{bmatrix} 3 & 2 \\ 1 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 7 \\ 8 & 6 \end{bmatrix}$$

Solution:

$$\begin{aligned} \langle A, B \rangle &= \sum_{i,j} A_{ij} B_{ij} \\ &= 3 \cdot 5 + 2 \cdot 7 + \\ &\quad 1 \cdot 8 + 4 \cdot 6 \\ &= 15 + 14 + 8 + 24 \\ &= 61 \end{aligned}$$

Problem 6: Inner Product of 2x3 and 3x2 Matrices

Given matrices:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}$$

Solution:

$$\begin{aligned} \langle A, B \rangle &= \sum_{i,j} A_{ij} B_{ij} \\ &= 1 \cdot 7 + 2 \cdot 8 + 3 \cdot 11 + \\ &\quad 4 \cdot 9 + 5 \cdot 10 + 6 \cdot 12 \\ &= 7 + 16 + 33 + 36 + 50 + 72 \\ &= 214 \end{aligned}$$

Problem 7: Inner Product with Transpose Operation

Given matrices:

$$A = \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$$

$$B = \begin{bmatrix} 6 & 7 \\ 8 & 9 \end{bmatrix}$$

Solution:

$$\begin{aligned} \langle A, B \rangle &= \sum_{i,j} A_{ij} B_{ij} \\ &= 2 \cdot 6 + 3 \cdot 7 + \\ &\quad 4 \cdot 8 + 5 \cdot 9 \\ &= 12 + 21 + 32 + 45 \\ &= 110 \end{aligned}$$

Problem 8: Inner Product of Symmetric Matrices

Given symmetric matrices:

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 5 \\ 5 & 6 \end{bmatrix}$$

Solution:

$$\begin{aligned} \langle A, B \rangle &= \sum_{i,j} A_{ij} B_{ij} \\ &= 1 \cdot 4 + 2 \cdot 5 + \\ &\quad 2 \cdot 5 + 3 \cdot 6 \\ &= 4 + 10 + 10 + 18 \\ &= 42 \end{aligned}$$

Problem 9: Inner Product with Complex Matrices

Given matrices:

$$A = \begin{bmatrix} 1+i & 2-i \\ 3+i & 4-i \end{bmatrix}$$

$$B = \begin{bmatrix} 5-i & 6+i \\ 7-i & 8+i \end{bmatrix}$$

Solution:

$$\begin{aligned} \langle A, B \rangle &= \sum_{i,j} \operatorname{Re}(A_{ij} \overline{B_{ij}}) \\ &= (1+i) \cdot (5+i) + (2-i) \cdot (6-i) + \\ &\quad (3+i) \cdot (7+i) + (4-i) \cdot (8+i) \\ &= (5+i+5i-i^2) + (12-i-6i+i^2) + \\ &\quad (21+i+7i-i^2) + (32+i-8i-i^2) \\ &= 5+5+12-6+21+32-2 \\ &= 62 \end{aligned}$$

Problem 10: Inner Product of 4x4 Matrices

Given matrices:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$B = \begin{bmatrix} 16 & 15 & 14 & 13 \\ 12 & 11 & 10 & 9 \\ 8 & 7 & 6 & 5 \\ 4 & 3 & 2 & 1 \end{bmatrix}$$

Solution:

$$\begin{aligned}
\langle A, B \rangle &= \sum_{i,j} A_{ij} B_{ij} \\
&= 1 \cdot 16 + 2 \cdot 15 + 3 \cdot 14 + 4 \cdot 13 + \\
&\quad 5 \cdot 12 + 6 \cdot 11 + 7 \cdot 10 + 8 \cdot 9 + \\
&\quad 9 \cdot 8 + 10 \cdot 7 + 11 \cdot 6 + 12 \cdot 5 + \\
&\quad 13 \cdot 4 + 14 \cdot 3 + 15 \cdot 2 + 16 \cdot 1 \\
&= 16 + 30 + 42 + 52 + 60 + 66 + 70 + 72 + \\
&\quad 72 + 70 + 66 + 60 + 52 + 42 + 30 + 16 \\
&= 696
\end{aligned}$$

Now let's look into the computational part of *inner product*.

1. Compute Inner Product from Scratch (without Libraries)

Here's how you can compute the inner product from the scratch:

```

# Define matrices A and B
A = [[1, 2, 3], [4, 5, 6]]
B = [[7, 8, 9], [10, 11, 12]]

# Function to compute inner product
def inner_product(A, B):
    # Get the number of rows and columns
    num_rows = len(A)
    num_cols = len(A[0])

    # Initialize the result
    result = 0

    # Compute the inner product
    for i in range(num_rows):
        for j in range(num_cols):
            result += A[i][j] * B[i][j]

    return result

# Compute inner product

```

```

inner_product_result = inner_product(A, B)

# Display result
print("Inner Product (From Scratch):")
print(inner_product_result)

```

Inner Product (From Scratch):
217

2. Compute Inner Product Using NumPy

Here's how to compute the inner product using Numpy:

```

import numpy as np
# Define matrices A and B
A = np.array([[1, 2, 3], [4, 5, 6]])
B = np.array([[7, 8, 9], [10, 11, 12]])
# calculating innerproduct
inner_product = (A*B).sum() # calculate element-wise product, then column sum

print("Inner Product (Using numpy):")
print(inner_product)

```

Inner Product (Using numpy):
217

The same operation can be done using SymPy functions as follows.

```

import sympy as sp
import numpy as np
# Define matrices A and B
A = sp.Matrix([[1, 2, 3], [4, 5, 6]])
B = sp.Matrix([[7, 8, 9], [10, 11, 12]])

# Compute element-wise product
elementwise_product = A.multiply_elementwise(B)

# Calculate sum of each column
inner_product_sympy = np.sum(elementwise_product)

```

```
# Display result
print("Inner Product (Using SymPy):")
print(inner_product_sympy)
```

```
Inner Product (Using SymPy):
217
```

A vector dot product (in Physics) can be calculated using SymPy `.dot()` function as shown below.

Let $A = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$ and $B = \begin{pmatrix} 4 & 5 & 6 \end{pmatrix}$, then the dot product, $A \cdot B$ is computed as:

```
import sympy as sp
A=sp.Matrix([1,2,3])
B=sp.Matrix([4,5,6])
display(A.dot(B)) # calculate dot product of A and B
```

32

A word of caution

In SymPy, `sp.Matrix([1,2,3])` create a column vector. But `np.array([1,2,3])` creates a row vector. So be careful while applying matrix/ dot product operations on these objects.

The same dot product using numpy object can be done as follows:

```
import numpy as np
A=np.array([1,2,3])
B=np.array([4,5,6])
display(A.dot(B.T))# dot() stands for dot product B.T represents the
↪ transpose of B
```

```
np.int64(32)
```

Practical Applications

Application 1: Signal Processing

In signal processing, the inner product can be used to measure the similarity between two signals. Here the most popular measure of similarity is the **cosine** similarity. This measure is defined as:

$$\cos \theta = \frac{A \cdot B}{||A|| ||B||}$$

Now consider two digital signals are given. It's cosine similarity measure can be calculated with a simulated data as shown below.

```
import numpy as np

# Simulated large signals (1D array) using NumPy
signal1 = np.sin(np.random.rand(1000))
signal2 = np.cos(np.random.rand(1000))

# Compute inner product
inner_product_signal = np.dot(signal1, signal2)
# cosine_sim=np.dot(signal1,signal2)/(np.linalg.norm(signal1)*np.linalg.norm(
    ↪ signal2))
# Display result
cosine_sim=inner_product_signal/(np.sqrt(np.dot(signal1,signal1))*np.sqrt(np
    ↪ .dot(signal2,signal2)))
print("Inner Product (Using numpy):")
print(inner_product_signal)
print("Similarity of signals:")
print(cosine_sim)
```

```
Inner Product (Using numpy):
376.2416744359054
Similarity of signals:
0.8656766327701758
```

Application 2: Machine Learning - Feature Similarity

In machine learning, the inner product is used to calculate the similarity between feature vectors.

```
import numpy as np

# Simulated feature vectors (2D array) using NumPy
features1 = np.random.rand(100, 10)
features2 = np.random.rand(100, 10)

# Compute inner product for each feature vector
```

```

inner_products = np.einsum('ij,ij->i', features1, features2) # use Einstein's
↪ sum

# Display results
print("Inner Products of Feature Vectors:")
display(inner_products)

```

Inner Products of Feature Vectors:

```

array([1.86618745, 2.51274116, 3.22480512, 1.75063304, 2.71721823,
       1.98680212, 2.49529215, 2.71034517, 2.13720204, 2.11147807,
       1.96657719, 3.5413832 , 1.79540191, 3.70295428, 3.65000013,
       2.73129036, 2.73390532, 2.72873231, 2.16813032, 2.70008207,
       2.5691309 , 3.8267985 , 2.92428261, 3.70546606, 1.76740125,
       1.87750167, 2.6071815 , 2.84333663, 2.18774745, 1.12079554,
       1.57466902, 1.64387872, 2.90078842, 1.99736669, 2.89345103,
       3.27215634, 1.15500698, 3.4420285 , 2.66379682, 1.97189201,
       2.69580422, 3.66770217, 2.17603749, 2.64082386, 1.84436202,
       2.52202546, 3.69840123, 2.65554438, 2.56514562, 3.30404738,
       2.9149493 , 3.20316329, 1.67700272, 2.56251505, 2.13654397,
       2.58640145, 2.0852157 , 1.729585 , 2.00190409, 2.36285034,
       1.89257731, 2.76695102, 2.22617247, 3.38664877, 2.44943569,
       3.23633599, 2.78761947, 2.42658745, 1.9870498 , 2.14063901,
       1.8996319 , 1.78367225, 2.55715996, 2.31152679, 2.77771334,
       1.62328576, 1.7009172 , 3.26510025, 3.36515712, 2.73481074,
       2.34406789, 1.77924725, 3.06492737, 1.87634451, 2.40285279,
       2.26284099, 2.40946304, 1.95833024, 3.13335721, 3.12926964,
       2.5930494 , 1.01589811, 1.54176088, 1.8962135 , 1.9040875 ,
       2.16756144, 1.2903262 , 2.26065421, 4.28424505, 1.82733956])

```

Application 3: Covariance Matrix in Statistics

The inner product can be used to compute covariance matrices for statistical data analysis. If X is a given distribution and $x = X - \bar{X}$. Then the covariance of X can be calculated as $cov(X) = \frac{1}{n-1}(x \cdot x^T)$ ². The python code a simulated data is shown below.

²Remember that the covariance of X is defined as $Cov(X) = \frac{\sum (X - \bar{X})^2}{n-1}$

```

import sympy as sp
import numpy as np

# Simulated large dataset (2D array) using NumPy
data = np.random.rand(100, 10)

# Compute the mean of each column
mean = np.mean(data, axis=0)

# Center the data
centered_data = data - mean

# Compute the covariance matrix using matrix product operation
cov_matrix = (centered_data.T @ centered_data) / (centered_data.shape[0] - 1)
cov_matrix_sympy = sp.Matrix(cov_matrix)

# Display results
print("Covariance Matrix:")
display(cov_matrix_sympy)

```

Covariance Matrix:

| | | | | |
|-----------------------|-----------------------------------|-----------------------|----------------------|-----------------------|
| 0.0688323944026016 | 0.0153740605980509 | −0.000140933353214011 | 0.00920489541395922 | −0.000140933353214011 |
| 0.0153740605980509 | 0.0905182214346242 | 0.000712744651254393 | −0.00436319787429453 | −0.000712744651254393 |
| −0.000140933353214011 | 0.000712744651254393 | 0.0585555577321913 | −0.00452162693037121 | 0.0585555577321913 |
| 0.00920489541395922 | −0.00436319787429453 | −0.00452162693037121 | 0.0862201676624409 | 0.00920489541395922 |
| −0.00387734773004558 | −0.000825052380333573 | 0.00126191421665622 | 0.0141142358813661 | −0.00387734773004558 |
| −0.00571190548041544 | 0.00786460769477729 | 0.00217942019679709 | −0.0044274235853324 | −0.00571190548041544 |
| −0.0049428689590279 | $-7.22532226030962 \cdot 10^{-5}$ | 0.000750926068297655 | 0.00328828339659587 | −0.0049428689590279 |
| −0.00463812193721968 | −0.00028578801387221 | −0.00674883183942774 | −0.00724495179921262 | −0.00463812193721968 |
| 0.0105897102719774 | 0.00456567668280999 | 0.00612207233520953 | 0.00795687838697261 | 0.0105897102719774 |
| 0.000992217456020462 | −0.000925756164661477 | −0.0137231192102392 | −0.00242695079854651 | 0.000992217456020462 |

These examples demonstrate the use of inner product and dot product in various applications.

Outer Product

The outer product of two vectors results in a matrix, and it is a way to combine these vectors into a higher-dimensional representation.

i Definition (Outer Product)

For two vectors \mathbf{u} and \mathbf{v} of dimensions m and n respectively, the outer product $\mathbf{u} \otimes \mathbf{v}$ is an $m \times n$ matrix defined as:

$$(\mathbf{u} \otimes \mathbf{v})_{ij} = u_i \cdot v_j$$

where \cdot denotes the outer product operation. In matrix notation, for two column vectors u, v ,

$$u \otimes v = uv^T$$

i Properties

1. **Linearity:**

$$(\mathbf{u} + \mathbf{w}) \otimes \mathbf{v} = (\mathbf{u} \otimes \mathbf{v}) + (\mathbf{w} \otimes \mathbf{v})$$

2. **Distributivity:**

$$\mathbf{u} \otimes (\mathbf{v} + \mathbf{w}) = (\mathbf{u} \otimes \mathbf{v}) + (\mathbf{u} \otimes \mathbf{w})$$

3. **Associativity:**

$$(\mathbf{u} \otimes \mathbf{v}) \otimes \mathbf{w} = \mathbf{u} \otimes (\mathbf{v} \otimes \mathbf{w})$$

Some simple examples of outer product is given below.

Example 1: Basic Outer Product

Given vectors:

$$\mathbf{u} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} 3 \\ 4 \\ 5 \end{pmatrix}$$

The outer product $\mathbf{u} \otimes \mathbf{v}$ is:

$$\mathbf{u} \otimes \mathbf{v} = \begin{pmatrix} 1 \cdot 3 & 1 \cdot 4 & 1 \cdot 5 \\ 2 \cdot 3 & 2 \cdot 4 & 2 \cdot 5 \end{pmatrix} = \begin{pmatrix} 3 & 4 & 5 \\ 6 & 8 & 10 \end{pmatrix}$$

Example 2: Outer Product with Larger Vectors

Given vectors:

$$\mathbf{u} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} 4 \\ 5 \end{pmatrix}$$

The outer product $\mathbf{u} \otimes \mathbf{v}$ is:

$$\mathbf{u} \otimes \mathbf{v} = \begin{pmatrix} 1 \cdot 4 & 1 \cdot 5 \\ 2 \cdot 4 & 2 \cdot 5 \\ 3 \cdot 4 & 3 \cdot 5 \end{pmatrix} = \begin{pmatrix} 4 & 5 \\ 8 & 10 \\ 12 & 15 \end{pmatrix}$$

Practice Problems

Find the outer product of **A** and **B** where **A** and **B** are given as follows:

Problem 1:

Find the outer product of:

$$A = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$B = [3 \quad 4]$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 1 \\ 2 \end{bmatrix} \otimes [3 \quad 4] \\ &= \begin{bmatrix} 1 \cdot 3 & 1 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} \\ &= \begin{bmatrix} 3 & 4 \\ 6 & 8 \end{bmatrix} \end{aligned}$$

Problem 2:

Find the outer product of:

$$A = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$B = [4 \quad 5 \quad 6]$$

Solution:

$$\begin{aligned}
 A \otimes B &= \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \otimes \begin{bmatrix} 4 & 5 & 6 \end{bmatrix} \\
 &= \begin{bmatrix} 1 \cdot 4 & 1 \cdot 5 & 1 \cdot 6 \\ 2 \cdot 4 & 2 \cdot 5 & 2 \cdot 6 \\ 3 \cdot 4 & 3 \cdot 5 & 3 \cdot 6 \end{bmatrix} \\
 &= \begin{bmatrix} 4 & 5 & 6 \\ 8 & 10 & 12 \\ 12 & 15 & 18 \end{bmatrix}
 \end{aligned}$$

Problem 3:

Find the outer product of:

$$\begin{aligned}
 A &= \begin{bmatrix} 1 & 2 \end{bmatrix} \\
 B &= \begin{bmatrix} 3 \\ 4 \end{bmatrix}
 \end{aligned}$$

Solution:

$$\begin{aligned}
 A \otimes B &= \begin{bmatrix} 1 & 2 \end{bmatrix} \otimes \begin{bmatrix} 3 \\ 4 \end{bmatrix} \\
 &= \begin{bmatrix} 1 \cdot 3 & 1 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} \\
 &= \begin{bmatrix} 3 & 4 \\ 6 & 8 \end{bmatrix}
 \end{aligned}$$

Problem 4:

Find the outer product of:

$$\begin{aligned}
 A &= \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\
 B &= \begin{bmatrix} 1 & -1 \end{bmatrix}
 \end{aligned}$$

Solution:

$$\begin{aligned}
 A \otimes B &= \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & -1 \end{bmatrix} \\
 &= \begin{bmatrix} 0 \cdot 1 & 0 \cdot -1 \\ 1 \cdot 1 & 1 \cdot -1 \end{bmatrix} \\
 &= \begin{bmatrix} 0 & 0 \\ 1 & -1 \end{bmatrix}
 \end{aligned}$$

Problem 5:

Find the outer product of:

$$\begin{aligned}
 A &= \begin{bmatrix} 2 \\ 3 \end{bmatrix} \\
 B &= \begin{bmatrix} 5 & -2 \end{bmatrix}
 \end{aligned}$$

Solution:

$$\begin{aligned}
 A \otimes B &= \begin{bmatrix} 2 \\ 3 \end{bmatrix} \otimes \begin{bmatrix} 5 & -2 \end{bmatrix} \\
 &= \begin{bmatrix} 2 \cdot 5 & 2 \cdot -2 \\ 3 \cdot 5 & 3 \cdot -2 \end{bmatrix} \\
 &= \begin{bmatrix} 10 & -4 \\ 15 & -6 \end{bmatrix}
 \end{aligned}$$

Problem 6:

Find the outer product of:

$$\begin{aligned}
 A &= \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \\
 B &= \begin{bmatrix} 2 & -1 & 0 \end{bmatrix}
 \end{aligned}$$

Solution:

$$\begin{aligned}
A \otimes B &= \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 2 & -1 & 0 \end{bmatrix} \\
&= \begin{bmatrix} 1 \cdot 2 & 1 \cdot (-1) & 1 \cdot 0 \\ 0 \cdot 2 & 0 \cdot (-1) & 0 \cdot 0 \\ 1 \cdot 2 & 1 \cdot (-1) & 1 \cdot 0 \end{bmatrix} \\
&= \begin{bmatrix} 2 & -1 & 0 \\ 0 & 0 & 0 \\ 2 & -1 & 0 \end{bmatrix}
\end{aligned}$$

Problem 7:

Find the outer product of:

$$\begin{aligned}
A &= \begin{bmatrix} 1 \\ -1 \end{bmatrix} \\
B &= \begin{bmatrix} 2 & 0 \\ 3 & -1 \end{bmatrix}
\end{aligned}$$

Solution:

$$\begin{aligned}
A \otimes B &= \begin{bmatrix} 1 \\ -1 \end{bmatrix} \otimes \begin{bmatrix} 2 & 0 \\ 3 & -1 \end{bmatrix} \\
&= \begin{bmatrix} 2 & 3 & 0 & -1 \\ -2 & -3 & 0 & 1 \end{bmatrix}
\end{aligned}$$

Problem 8:

Find the outer product of:

$$\begin{aligned}
A &= \begin{bmatrix} 3 \\ 4 \end{bmatrix} \\
B &= \begin{bmatrix} 1 & -2 & 3 \end{bmatrix}
\end{aligned}$$

Solution:

$$\begin{aligned}
 A \otimes B &= \begin{bmatrix} 3 \\ 4 \end{bmatrix} \otimes \begin{bmatrix} 1 & -2 & 3 \end{bmatrix} \\
 &= \begin{bmatrix} 3 \cdot 1 & 3 \cdot -2 & 3 \cdot 3 \\ 4 \cdot 1 & 4 \cdot -2 & 4 \cdot 3 \end{bmatrix} \\
 &= \begin{bmatrix} 3 & -6 & 9 \\ 4 & -8 & 12 \end{bmatrix}
 \end{aligned}$$

Problem 9:

Find the outer product of:

$$\begin{aligned}
 A &= \begin{bmatrix} 2 \\ 3 \\ -1 \end{bmatrix} \\
 B &= \begin{bmatrix} 4 & -2 \end{bmatrix}
 \end{aligned}$$

Solution:

$$\begin{aligned}
 A \otimes B &= \begin{bmatrix} 2 \\ 3 \\ -1 \end{bmatrix} \otimes \begin{bmatrix} 4 & -2 \end{bmatrix} \\
 &= \begin{bmatrix} 2 \cdot 4 & 2 \cdot -2 \\ 3 \cdot 4 & 3 \cdot -2 \\ -1 \cdot 4 & -1 \cdot -2 \end{bmatrix} \\
 &= \begin{bmatrix} 8 & -4 \\ 12 & -6 \\ -4 & 2 \end{bmatrix}
 \end{aligned}$$

Problem 10:

Find the outer product of:

$$\begin{aligned}
 A &= \begin{bmatrix} 0 \\ 5 \end{bmatrix} \\
 B &= \begin{bmatrix} 3 & 1 \end{bmatrix}
 \end{aligned}$$

Solution:

$$\begin{aligned}
 A \otimes B &= \begin{bmatrix} 0 \\ 5 \end{bmatrix} \otimes \begin{bmatrix} 3 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} 0 \cdot 3 & 0 \cdot 1 \\ 5 \cdot 3 & 5 \cdot 1 \end{bmatrix} \\
 &= \begin{bmatrix} 0 & 0 \\ 15 & 5 \end{bmatrix}
 \end{aligned}$$

1. Compute Outer Product of Vectors from Scratch (without Libraries)

Here's how you can compute the outer product manually:

```
# Define vectors u and v
u = [1, 2]
v = [3, 4, 5]

# Function to compute outer product
def outer_product(u, v):
    # Initialize the result
    result = [[a * b for b in v] for a in u]
    return result

# Compute outer product
outer_product_result = outer_product(u, v)

# Display result
print("Outer Product of Vectors (From Scratch):")
for row in outer_product_result:
    print(row)
```

```
Outer Product of Vectors (From Scratch):
[3, 4, 5]
[6, 8, 10]
```

2. Compute Outer Product of Vectors Using SymPy

Here's how to compute the outer product using SymPy:

```
import sympy as sp

# Define vectors u and v
u = sp.Matrix([1, 2])
v = sp.Matrix([3, 4, 5])

# Compute outer product using SymPy
outer_product_sympy = u * v.T

# Display result
print("Outer Product of Vectors (Using SymPy):")
display(outer_product_sympy)
```

Outer Product of Vectors (Using SymPy):

$$\begin{bmatrix} 3 & 4 & 5 \\ 6 & 8 & 10 \end{bmatrix}$$

Outer Product of Matrices

The outer product of two matrices extends the concept from vectors to higher-dimensional tensors. For two matrices A and B , the outer product results in a higher-dimensional tensor and is generally expressed as block matrices.

i Definition (Outer Product of Matrices)

For two matrices A of dimension $m \times p$ and B of dimension $q \times n$, the outer product $A \otimes B$ results in a tensor of dimension $m \times q \times p \times n$. The entries of the tensor are given by:

$$(A \otimes B)_{ijkl} = A_{ij} \cdot B_{kl}$$

where \cdot denotes the outer product operation.

i Properties

1. **Linearity:**

$$(A + C) \otimes B = (A \otimes B) + (C \otimes B)$$

2. **Distributivity:**

$$A \otimes (B + D) = (A \otimes B) + (A \otimes D)$$

3. **Associativity:**

$$(A \otimes B) \otimes C = A \otimes (B \otimes C)$$

Here are some simple examples to demonstrate the mathematical procedure to find outer product of matrices.

Example 1: Basic Outer Product of Matrices

Given matrices:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 5 \\ 6 \end{pmatrix}$$

The outer product $A \otimes B$ is:

$$A \otimes B = \begin{pmatrix} 1 \cdot 5 & 1 \cdot 6 \\ 2 \cdot 5 & 2 \cdot 6 \\ 3 \cdot 5 & 3 \cdot 6 \\ 4 \cdot 5 & 4 \cdot 6 \end{pmatrix} = \begin{pmatrix} 5 & 6 \\ 10 & 12 \\ 15 & 18 \\ 20 & 24 \end{pmatrix}$$

Example 2: Outer Product with Larger Matrices

Given matrices:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}, \quad B = \begin{pmatrix} 7 \\ 8 \end{pmatrix}$$

The outer product $A \otimes B$ is:

$$A \otimes B = \begin{pmatrix} 1 \cdot 7 & 1 \cdot 8 \\ 2 \cdot 7 & 2 \cdot 8 \\ 3 \cdot 7 & 3 \cdot 8 \\ 4 \cdot 7 & 4 \cdot 8 \\ 5 \cdot 7 & 5 \cdot 8 \\ 6 \cdot 7 & 6 \cdot 8 \end{pmatrix} = \begin{pmatrix} 7 & 8 \\ 14 & 16 \\ 21 & 24 \\ 28 & 32 \\ 35 & 40 \\ 42 & 48 \end{pmatrix}$$

Example 3: Compute the outer product of the following vectors $\mathbf{u} = [0, 1, 2]$ and $\mathbf{v} = [2, 3, 4]$.

To find the outer product, we calculate each element (i, j) as the product of the (i) -th element of \mathbf{u} and the (j) -th element of \mathbf{v} . Mathematically:

$$\mathbf{u} \otimes \mathbf{v} = \begin{bmatrix} 0 \cdot 2 & 0 \cdot 3 & 0 \cdot 4 \\ 1 \cdot 2 & 1 \cdot 3 & 1 \cdot 4 \\ 2 \cdot 2 & 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 2 & 3 & 4 \\ 4 & 6 & 8 \end{bmatrix}$$

1. Compute Outer Product of Matrices from Scratch (without Libraries)

Here's how you can compute the outer product manually:

```
# Define matrices A and B
A = [[1, 2], [3, 4]]
B = [[5], [6]]

# Function to compute outer product
def outer_product_matrices(A, B):
    m = len(A)
    p = len(A[0])
    q = len(B)
    n = len(B[0])
    result = [[0] * (n * p) for _ in range(m * q)]

    for i in range(m):
        for j in range(p):
            for k in range(q):
                for l in range(n):
                    result[i*q + k][j*n + l] = A[i][j] * B[k][l]

    return result

# Compute outer product
outer_product_result_matrices = outer_product_matrices(A, B)

# Display result
print("Outer Product of Matrices (From Scratch):")
for row in outer_product_result_matrices:
    print(row)
```

Outer Product of Matrices (From Scratch):

```
[5, 10]
[6, 12]
[15, 20]
[18, 24]
```

Here is the Python code to compute the outer product of these vectors using the NumPy function `.outer()`:

```
import numpy as np

# Define vectors
u = np.array([[1,2],[3,4]])
v = np.array([[5],[4]])

# Compute outer product
outer_product = np.outer(u, v)

print("Outer Product of u and v:")
display(outer_product)
```

Outer Product of u and v:

```
array([[ 5,  4],
       [10,  8],
       [15, 12],
       [20, 16]])
```

Example 3: Real-world Application in Recommendation Systems

In recommendation systems, the outer product can represent user-item interactions. A simple context is here. Let the user preferences of items is given as $u = [4, 3, 5]$ and the item scores is given by $v = [2, 5, 4]$. Now the recommendation score can be calculated as the outer product of these two vectors. Calculation of this score is shown below. The outer product $\mathbf{u} \otimes \mathbf{v}$ is calculated as follows:

$$\mathbf{u} \otimes \mathbf{v} = \begin{bmatrix} 4 \cdot 2 & 4 \cdot 5 & 4 \cdot 4 \\ 3 \cdot 2 & 3 \cdot 5 & 3 \cdot 4 \\ 5 \cdot 2 & 5 \cdot 5 & 5 \cdot 4 \end{bmatrix} = \begin{bmatrix} 8 & 20 & 16 \\ 6 & 15 & 12 \\ 10 & 25 & 20 \end{bmatrix}$$

The python code for this task is given below.

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

# Define the user and product ratings vectors
user_ratings = np.array([4, 3, 5])
product_ratings = np.array([2, 5, 4])

# Compute the outer product
```

```

predicted_ratings = np.outer(user_ratings, product_ratings)

# Print the predicted ratings matrix
print("Predicted Ratings Matrix:")
display(predicted_ratings)

# Plot the result
plt.imshow(predicted_ratings, cmap='coolwarm', interpolation='nearest')
plt.colorbar()
plt.title('Predicted Ratings Matrix (Recommendation System)')
plt.xlabel('Product Ratings')
plt.ylabel('User Ratings')
plt.xticks(ticks=np.arange(len(product_ratings)), labels=product_ratings)
plt.yticks(ticks=np.arange(len(user_ratings)), labels=user_ratings)
plt.show()

```

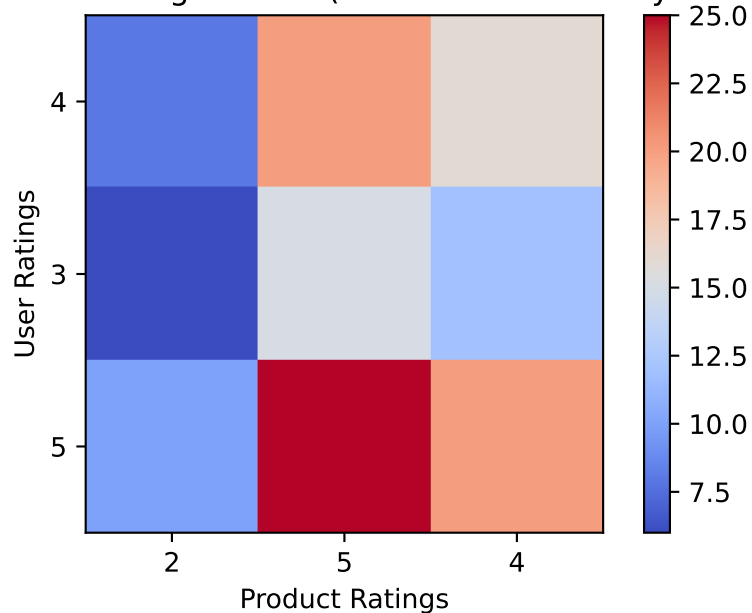
Predicted Ratings Matrix:

```

array([[ 8, 20, 16],
       [ 6, 15, 12],
       [10, 25, 20]])

```

Predicted Ratings Matrix (Recommendation System)



i Additional Properties & Definitions

1. Definition and Properties

Given two vectors:

- $\mathbf{u} \in \mathbb{R}^m$
- $\mathbf{v} \in \mathbb{R}^n$

The outer product $\mathbf{u} \otimes \mathbf{v}$ results in an $m \times n$ matrix where each element (i, j) of the matrix is calculated as:

$$(\mathbf{u} \otimes \mathbf{v})_{ij} = u_i \cdot v_j$$

2. Non-Symmetry

The outer product is generally not symmetric. For vectors \mathbf{u} and \mathbf{v} , the matrix $\mathbf{u} \otimes \mathbf{v}$ is not necessarily equal to $\mathbf{v} \otimes \mathbf{u}$:

$$\mathbf{u} \otimes \mathbf{v} \neq \mathbf{v} \otimes \mathbf{u}$$

3. Rank of the Outer Product

The rank of the outer product matrix $\mathbf{u} \otimes \mathbf{v}$ is always 1, provided neither \mathbf{u} nor \mathbf{v} is a zero vector. This is because the matrix can be expressed as a single rank-1 matrix.

4. Distributive Property

The outer product is distributive over vector addition. For vectors $\mathbf{u}_1, \mathbf{u}_2 \in \mathbb{R}^m$ and $\mathbf{v} \in \mathbb{R}^n$:

$$(\mathbf{u}_1 + \mathbf{u}_2) \otimes \mathbf{v} = (\mathbf{u}_1 \otimes \mathbf{v}) + (\mathbf{u}_2 \otimes \mathbf{v})$$

5. Associativity with Scalar Multiplication

The outer product is associative with scalar multiplication. For a scalar α and vectors $\mathbf{u} \in \mathbb{R}^m$ and $\mathbf{v} \in \mathbb{R}^n$:

$$\alpha(\mathbf{u} \otimes \mathbf{v}) = (\alpha\mathbf{u}) \otimes \mathbf{v} = \mathbf{u} \otimes (\alpha\mathbf{v})$$

6. Matrix Trace

The trace of the outer product of two vectors is given by:

$$\text{tr}(\mathbf{u} \otimes \mathbf{v}) = (\mathbf{u}^T \mathbf{v}) = (\mathbf{v}^T \mathbf{u})$$

Here, tr denotes the trace of a matrix, which is the sum of its diagonal elements.

7. Matrix Norm

The Frobenius norm of the outer product matrix can be expressed in terms of the norms of the original vectors:

$$\|\mathbf{u} \otimes \mathbf{v}\|_F = \|\mathbf{u}\|_2 \cdot \|\mathbf{v}\|_2$$

where $\|\cdot\|_2$ denotes the Euclidean norm.

Example Calculation in Python

Here's how to compute and visualize the outer product properties using Python:

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

# Define vectors
u = np.array([1, 2, 3])
v = np.array([4, 5])

# Compute outer product
outer_product = np.outer(u, v)

# Display results
print("Outer Product Matrix:")
print(outer_product)

# Compute and display rank
rank = np.linalg.matrix_rank(outer_product)
print(f"Rank of Outer Product Matrix: {rank}")

# Compute Frobenius norm
frobenius_norm = np.linalg.norm(outer_product, 'fro')
print(f"Frobenius Norm: {frobenius_norm}")

# Plot the result
plt.imshow(outer_product, cmap='viridis', interpolation='nearest')
plt.colorbar()
plt.title('Outer Product Matrix')
plt.xlabel('Vector v')
plt.ylabel('Vector u')
plt.xticks(ticks=np.arange(len(v)), labels=v)
plt.yticks(ticks=np.arange(len(u)), labels=u)
plt.show()
```

Outer Product Matrix:

```
[[ 4  5]
 [ 8 10]
 [12 15]]
```

Rank of Outer Product Matrix: 1

Frobenius Norm: 23.958297101421877

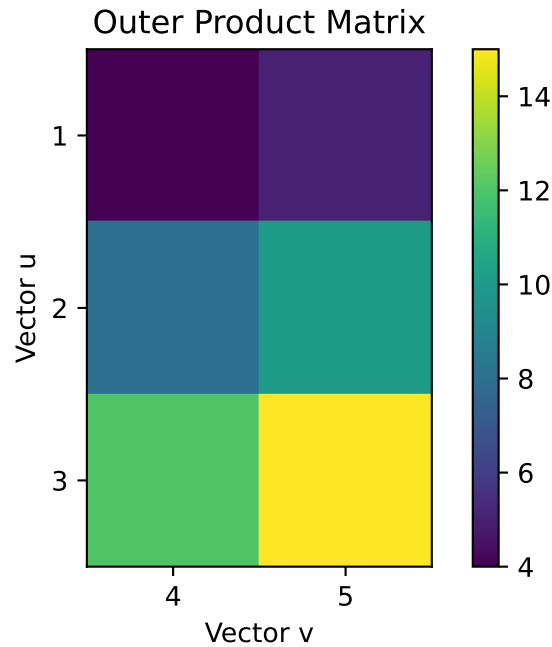


Figure 10: Demonstration of Outer Product and its Properties

Kronecker Product

In mathematics, the Kronecker product, sometimes denoted by \otimes , is an operation on two matrices of arbitrary size resulting in a *block matrix*. It is a specialization of the tensor product (which is denoted by the same symbol) from vectors to matrices and gives the matrix of the tensor product linear map with respect to a standard choice of basis. The Kronecker product is to be distinguished from the usual matrix multiplication, which is an entirely different operation. The Kronecker product is also sometimes called *matrix direct product*.

i Note

If A is an $m \times n$ matrix and B is a $p \times q$ matrix, then the Kronecker product $A \otimes B$ is the $pm \times qn$ block matrix defined as: Each a_{ij} of A is replaced by the matrix $a_{ij}B$. Symbolically this will result in a block matrix defined by:

$$A \otimes B = A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ a_{21}B & a_{22}B & \cdots & a_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{bmatrix}$$

i Properties of the Kronecker Product

1. Associativity

The Kronecker product is associative. For matrices $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{p \times q}$, and $C \in \mathbb{R}^{r \times s}$:

$$(A \otimes B) \otimes C = A \otimes (B \otimes C)$$

2. Distributivity Over Addition

The Kronecker product distributes over matrix addition. For matrices $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{p \times q}$, and $C \in \mathbb{R}^{p \times q}$:

$$A \otimes (B + C) = (A \otimes B) + (A \otimes C)$$

3. Mixed Product Property

The Kronecker product satisfies the mixed product property with the matrix product. For matrices $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{p \times q}$, $C \in \mathbb{R}^{r \times s}$, and $D \in \mathbb{R}^{r \times s}$:

$$(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$$

4. Transpose

The transpose of the Kronecker product is given by:

$$(A \otimes B)^T = A^T \otimes B^T$$

5. Norm

The Frobenius norm of the Kronecker product can be computed as:

$$\|A \otimes B\|_F = \|A\|_F \cdot \|B\|_F$$

where $\|\cdot\|_F$ denotes the Frobenius norm.

💡 Frobenius Norm

The Frobenius norm, also known as the Euclidean norm for matrices, is a measure of a matrix's magnitude. It is defined as the square root of the sum of the absolute squares of its elements. Mathematically, for a matrix A with elements a_{ij} , the Frobenius norm is given by:

$$\|A\|_F = \sqrt{\sum_{i,j} |a_{ij}|^2}$$

Example 1: Calculation of Frobenius Norm

Consider the matrix A :

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

To compute the Frobenius norm:

$$\|A\|_F = \sqrt{1^2 + 2^2 + 3^2 + 4^2} = \sqrt{1 + 4 + 9 + 16} = \sqrt{30} \approx 5.48$$

Example 2: Frobenius Norm of a Sparse Matrix

Consider the sparse matrix B :

$$B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

To compute the Frobenius norm:

$$\|B\|_F = \sqrt{0^2 + 0^2 + 0^2 + 5^2 + 0^2 + 0^2} = \sqrt{25} = 5$$

Example 3: Frobenius Norm in a Large Matrix

Consider the matrix C of size 3×3 :

$$C = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

To compute the Frobenius norm:

$$\begin{aligned}\|C\|_F &= \sqrt{1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2 + 9^2} \\ &= \sqrt{1 + 4 + 9 + 16 + 25 + 36 + 49 + 64 + 81} \\ &= \sqrt{285} \\ &\approx 16.88\end{aligned}$$

Applications of the Frobenius Norm

- *Application 1: Image Compression:* In image processing, the Frobenius norm can measure the difference between the original and compressed images, indicating how well the compression has preserved the original image quality.
- *Application 2: Matrix Factorization:* In numerical analysis, Frobenius norm is used to evaluate the error in matrix approximations, such as in Singular Value Decomposition (SVD). A lower Frobenius norm of the error indicates a better approximation.
- *Application 3: Error Measurement in Numerical Solutions:* In solving systems of linear equations, the Frobenius norm can be used to measure the error between the true solution and the computed solution, providing insight into the accuracy of numerical methods.

The `linalg` sub module of NumPy library can be used to calculate various norms. Basically norm is the generalized form of Euclidean distance.

```
import numpy as np

# Example 1: Simple Matrix
A = np.array([[1, 2], [3, 4]])
frobenius_norm_A = np.linalg.norm(A, 'fro')
print(f"Frobenius Norm of A: {frobenius_norm_A:.2f}")

# Example 2: Sparse Matrix
B = np.array([[0, 0, 0], [0, 5, 0], [0, 0, 0]])
frobenius_norm_B = np.linalg.norm(B, 'fro')
print(f"Frobenius Norm of B: {frobenius_norm_B:.2f}")

# Example 3: Large Matrix
C = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
frobenius_norm_C = np.linalg.norm(C, 'fro')
print(f"Frobenius Norm of C: {frobenius_norm_C:.2f}")
```

Frobenius Norm of A: 5.48
 Frobenius Norm of B: 5.00
 Frobenius Norm of C: 16.88

Frobenius norm of Kronecker product

Let us consider two matrices,

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

and

$$B = \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix}$$

The Kronecker product $C = A \otimes B$ is:

$$C = \begin{bmatrix} 1 \cdot B & 2 \cdot B \\ 3 \cdot B & 4 \cdot B \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} & \begin{bmatrix} 0 \cdot 2 & 5 \cdot 2 \\ 6 \cdot 2 & 7 \cdot 2 \end{bmatrix} \\ \begin{bmatrix} 0 \cdot 3 & 5 \cdot 3 \\ 6 \cdot 3 & 7 \cdot 3 \end{bmatrix} & \begin{bmatrix} 0 \cdot 4 & 5 \cdot 4 \\ 6 \cdot 4 & 7 \cdot 4 \end{bmatrix} \end{bmatrix}$$

This expands to:

$$C = \begin{bmatrix} 0 & 5 & 0 & 10 \\ 6 & 7 & 12 & 14 \\ 0 & 15 & 0 & 20 \\ 18 & 21 & 24 & 28 \end{bmatrix}$$

Computing the Frobenius Norm

To compute the Frobenius norm of C :

$$\|C\|_F = \sqrt{\sum_{i=1}^4 \sum_{j=1}^4 |c_{ij}|^2}$$

$$\|C\|_F = \sqrt{0^2 + 5^2 + 0^2 + 10^2 + 6^2 + 7^2 + 12^2 + 14^2 + 0^2 + 15^2 + 0^2 + 20^2 + 18^2 + 21^2 + 24^2 + 28^2}$$

$$\|C\|_F = \sqrt{0 + 25 + 0 + 100 + 36 + 49 + 144 + 196 + 0 + 225 + 0 + 400 + 324 + 441 + 576 + 784}$$

$$\|C\|_F = \sqrt{2896}$$

$$\|C\|_F \approx 53.87$$

Practice Problems

Find the Kronecker product of A and B where A and B are given as follows:

Problem 1:

Find the Kronecker product of:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & 2 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\ 3 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & 4 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 & 0 & 2 \\ 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 4 \\ 3 & 0 & 4 & 0 \end{bmatrix} \end{aligned}$$

Problem 2:

Find the Kronecker product of:

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} \\ &= \begin{bmatrix} 1 \cdot \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} & 0 \cdot \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} \\ 0 \cdot \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} & 1 \cdot \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} \end{bmatrix} \\ &= \begin{bmatrix} 2 & 3 & 0 & 0 \\ 4 & 5 & 0 & 0 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 4 & 5 \end{bmatrix} \end{aligned}$$

Problem 3:

Find the Kronecker product of:

$$A = \begin{bmatrix} 1 & 2 \end{bmatrix}$$

$$B = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 1 & 2 \end{bmatrix} \otimes \begin{bmatrix} 3 \\ 4 \end{bmatrix} \\ &= \begin{bmatrix} 1 \cdot \begin{bmatrix} 3 \\ 4 \end{bmatrix} & 2 \cdot \begin{bmatrix} 3 \\ 4 \end{bmatrix} \end{bmatrix} \\ &= \begin{bmatrix} 3 & 6 \\ 4 & 8 \end{bmatrix} \end{aligned}$$

Problem 4:

Find the Kronecker product of:

$$A = \begin{bmatrix} 0 & 1 \end{bmatrix}$$
$$B = \begin{bmatrix} 1 & -1 \\ 2 & 0 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & -1 \\ 2 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 0 \cdot \begin{bmatrix} 1 & -1 \\ 2 & 0 \end{bmatrix} & 1 \cdot \begin{bmatrix} 1 & -1 \\ 2 & 0 \end{bmatrix} \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 & 1 & -1 \\ 0 & 0 & 2 & 0 \end{bmatrix} \end{aligned}$$

Problem 5:

Find the Kronecker product of:

$$A = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$
$$B = \begin{bmatrix} 4 & -2 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 2 \\ 3 \end{bmatrix} \otimes \begin{bmatrix} 4 & -2 \end{bmatrix} \\ &= \begin{bmatrix} 2 \cdot \begin{bmatrix} 4 & -2 \end{bmatrix} \\ 3 \cdot \begin{bmatrix} 4 & -2 \end{bmatrix} \end{bmatrix} \\ &= \begin{bmatrix} 8 & -4 \\ 12 & -6 \end{bmatrix} \end{aligned}$$

Problem 6:

Find the Kronecker product of:

$$A = \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & -1 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\ 0 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & 2 \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 \end{bmatrix} \end{aligned}$$

Problem 7:

Find the Kronecker product of:

$$A = [2]$$

$$B = \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= [2] \otimes \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix} \\ &= 2 \cdot \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix} \\ &= \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix} \end{aligned}$$

Problem 8:

Find the Kronecker product of:

$$A = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & 1 \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \end{aligned}$$

Problem 9:

Find the Kronecker product of:

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 \cdot \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} & 0 \cdot \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \\ 0 \cdot \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} & 1 \cdot \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \end{aligned}$$

Problem 10:

Find the Kronecker product of:

$$A = \begin{bmatrix} 2 & -1 \\ 3 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 5 \\ -2 & 3 \end{bmatrix}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 2 & -1 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 0 & 5 \\ -2 & 3 \end{bmatrix} \\ &= \begin{bmatrix} 2 \cdot \begin{bmatrix} 0 & 5 \\ -2 & 3 \end{bmatrix} & -1 \cdot \begin{bmatrix} 0 & 5 \\ -2 & 3 \end{bmatrix} \\ 3 \cdot \begin{bmatrix} 0 & 5 \\ -2 & 3 \end{bmatrix} & 4 \cdot \begin{bmatrix} 0 & 5 \\ -2 & 3 \end{bmatrix} \end{bmatrix} \\ &= \begin{bmatrix} 0 & 10 & 0 & -5 \\ -4 & 6 & 2 & -3 \\ 0 & 15 & 0 & 20 \\ -6 & 9 & -8 & 12 \end{bmatrix} \end{aligned}$$

Connection Between Outer Product and Kronecker Product**1. Conceptual Connection:**

- The **outer product** is a special case of the **Kronecker product**. Specifically, if \mathbf{A} is a column vector and \mathbf{B} is a row vector, then \mathbf{A} is a $m \times 1$ matrix and \mathbf{B} is a $1 \times n$ matrix. The Kronecker product of these two matrices will yield the same result as the outer product of these vectors.
- For matrices \mathbf{A} and \mathbf{B} , the Kronecker product involves taking the outer product of each element of \mathbf{A} with the entire matrix \mathbf{B} .

2. Mathematical Formulation:

- Let $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$. Then:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} \end{bmatrix}$$

- If $\mathbf{A} = \mathbf{u}\mathbf{v}^T$ where \mathbf{u} is a column vector and \mathbf{v}^T is a row vector, then the Kronecker product of \mathbf{u} and \mathbf{v}^T yields the same result as the outer product $\mathbf{u} \otimes \mathbf{v}$.

Note

Summary

- The **outer product** is a specific case of the **Kronecker product** where one of the matrices is a vector (either row or column).
- The **Kronecker product** generalizes the outer product to matrices and is more versatile in applications involving tensor products and higher-dimensional constructs.

Matrix Multiplication as Kronecker Product

Given matrices \mathbf{A} and \mathbf{B} , where: - \mathbf{A} is an $m \times n$ matrix - \mathbf{B} is an $n \times p$ matrix

The product $\mathbf{C} = \mathbf{AB}$ can be expressed using Kronecker products as:

$$\mathbf{C} = \sum_{k=1}^n (\mathbf{A}_{:,k} \otimes \mathbf{B}_{k,:})$$

where: - $\mathbf{A}_{:,k}$ denotes the k -th column of matrix \mathbf{A} - $\mathbf{B}_{k,:}$ denotes the k -th row of matrix \mathbf{B}

Example:

Let:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

and:

$$\mathbf{B} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

To find $\mathbf{C} = \mathbf{AB}$ using Kronecker products:

1. Compute the Kronecker Product of Columns of \mathbf{A} and Rows of \mathbf{B} :

- For column $\mathbf{A}_{:,1} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$ and row $\mathbf{B}_{1,:} = \begin{bmatrix} 0 & 1 \end{bmatrix}$:

$$\mathbf{A}_{:,1} \otimes \mathbf{B}_{1,:} = \begin{bmatrix} 0 & 1 \\ 0 & 3 \end{bmatrix}$$

- For column $\mathbf{A}_{:,2} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$ and row $\mathbf{B}_{2,:} = [1 \ 0]$:

$$\mathbf{A}_{:,2} \otimes \mathbf{B}_{2,:} = \begin{bmatrix} 2 & 0 \\ 4 & 0 \end{bmatrix}$$

2. Sum the Kronecker Products:

$$\mathbf{C} = \begin{bmatrix} 0 & 1 \\ 0 & 3 \end{bmatrix} + \begin{bmatrix} 2 & 0 \\ 4 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 4 & 3 \end{bmatrix}$$

In the previous block we have discussed the Frobenius norm and its applications. Now came back to the discussions on the Kronecker product. The Kronecker product is particularly useful in scenarios where interactions between different types of data need to be modeled comprehensively. In recommendation systems, it allows us to integrate user preferences with item relationships to improve recommendation accuracy.

In addition to recommendation systems, Kronecker products are used in various fields such as:

- Signal Processing: For modeling multi-dimensional signals.
- Machine Learning: In building features for complex models.
- Communication Systems: For modeling network interactions.

By understanding the Kronecker product and its applications, we can extend it to solve complex problems and enhance systems across different domains. To understand the practical use of Kronecker product in a Machine Learning scenario let us consider the following problem statement and its solution.

Problem statement

In the realm of recommendation systems, predicting user preferences for various product categories based on past interactions is a common challenge. Suppose we have data on user preferences for different products and categories. We can use this data to recommend the best products for each user by employing mathematical tools such as the Kronecker product. The User Preference and Category relationships are given in Table 5 and Table 6 .

Table 5: User Preference

| User/Item | Electronics | Clothing | Books |
|-----------|-------------|----------|-------|
| User 1 | 5 | 3 | 4 |
| User 2 | 2 | 4 | 5 |
| User 3 | 3 | 4 | 4 |

Table 6: Category Relationships

| Category/Feature | Feature 1 | Feature 2 | Feature 3 |
|------------------|-----------|-----------|-----------|
| Electronics | 1 | 0 | 0 |
| Clothing | 0 | 1 | 1 |
| Books | 0 | 1 | 1 |

Predict user preferences for different product categories using the Kronecker product matrix.

Solution Procedure

1. *Compute the Kronecker Product:* Calculate the Kronecker product of matrices U and C to obtain matrix K .

To model the problem, we use the Kronecker product of the user preference matrix U and the category relationships matrix C . This product allows us to predict the user's rating for each category by combining their preferences with the category features.

Formulating Matrices

User Preference Matrix (U): - Dimension: 3×3 (3 users, 3 items) - from the User preference data, we can create the User Preference Matrix as follows:

$$U = \begin{pmatrix} 5 & 3 & 4 \\ 2 & 4 & 5 \\ 3 & 4 & 4 \end{pmatrix}$$

Category Relationships Matrix (C): - Dimension: 3×3 (3 categories) - from the Category Relationships data, we can create the Category Relationship Matrix as follows:

$$C = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

Kronecker Product Calculation

The Kronecker product K of U and C is calculated as follows:

1. Matrix Dimensions:

- U is 3×3 (3 users, 3 items).
- C is 3×3 (3 categories, 3 features).

2. Calculate Kronecker Product:

- For each element u_{ij} in U , multiply by the entire matrix C .

The Kronecker product K is computed as:

$$K = U \otimes C$$

Explicitly, the Kronecker product K is:

$$K = \begin{pmatrix} 5 \cdot C & 3 \cdot C & 4 \cdot C \\ 2 \cdot C & 4 \cdot C & 5 \cdot C \\ 3 \cdot C & 4 \cdot C & 4 \cdot C \end{pmatrix}$$

As an example the blocks in first row are:

$$5 \cdot C = \begin{pmatrix} 5 & 0 & 0 \\ 0 & 5 & 5 \\ 0 & 5 & 5 \end{pmatrix}, \quad 3 \cdot C = \begin{pmatrix} 3 & 0 & 0 \\ 0 & 3 & 3 \\ 0 & 3 & 3 \end{pmatrix}, \quad 4 \cdot C = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 4 & 4 \\ 0 & 4 & 4 \end{pmatrix}$$

Combining these blocks:

$$K = \begin{pmatrix} 5 & 0 & 0 & 3 & 0 & 0 & 4 & 0 & 0 \\ 0 & 5 & 5 & 0 & 3 & 3 & 0 & 4 & 4 \\ 0 & 5 & 5 & 0 & 3 & 3 & 0 & 4 & 4 \\ 2 & 0 & 0 & 4 & 0 & 0 & 5 & 0 & 0 \\ 0 & 2 & 2 & 0 & 4 & 4 & 0 & 5 & 5 \\ 0 & 2 & 2 & 0 & 4 & 4 & 0 & 5 & 5 \\ 3 & 0 & 0 & 4 & 0 & 0 & 4 & 0 & 0 \\ 0 & 3 & 3 & 0 & 4 & 4 & 0 & 4 & 4 \\ 0 & 3 & 3 & 0 & 4 & 4 & 0 & 4 & 4 \end{pmatrix}$$

- 2. Interpret the Kronecker Product Matrix:** The resulting matrix K represents all possible combinations of user preferences and category features.

3. **Predict Ratings:** For each user, use matrix K to predict the rating for each category by summing up the values in the corresponding rows.
4. **Generate Recommendations:** Identify the top categories with the highest predicted ratings for each user.

The python code to solve this problem computationally is given below.

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

# Define the matrices
U = np.array([[5, 3, 4],
              [2, 4, 5],
              [3, 4, 4]])

C = np.array([[1, 0, 0],
              [0, 1, 1],
              [0, 1, 1]])

# Compute the Kronecker product
K = np.kron(U, C)

# Create a DataFrame to visualize the Kronecker product matrix
df_K = pd.DataFrame(K,
                    columns=['Electronics_F1', 'Electronics_F2',
↪   'Electronics_F3',
                            'Clothing_F1', 'Clothing_F2', 'Clothing_F3',
                            'Books_F1', 'Books_F2', 'Books_F3'],
                    index=['User 1 Electronics', 'User 1 Clothing', 'User 1
↪   Books',
                            'User 2 Electronics', 'User 2 Clothing', 'User 2
↪   Books',
                            'User 3 Electronics', 'User 3 Clothing', 'User 3
↪   Books'])

# Print the Kronecker product matrix
print("Kronecker Product Matrix (K):\n", df_K)

# Predict ratings and create recommendations
def recommend(user_index, top_n=3):
    """ Recommend top_n categories for a given user based on Kronecker
↪   product matrix. """
```

```

user_ratings = K[user_index * len(C):(user_index + 1) * len(C), :]
predicted_ratings = np.sum(user_ratings, axis=0)
recommendations = np.argsort(predicted_ratings)[::-1][:top_n]
return recommendations

# Recommendations for User 1
user_index = 0 # User 1
top_n = 3
recommendations = recommend(user_index, top_n)

print(f"\nTop {top_n} recommendations for User {user_index + 1}:")
for rec in recommendations:
    print(df_K.columns[rec])

```

Kronecker Product Matrix (K):

| | Electronics_F1 | Electronics_F2 | Electronics_F3 | \ |
|--------------------|----------------|----------------|----------------|---|
| User 1 Electronics | 5 | 0 | 0 | |
| User 1 Clothing | 0 | 5 | 5 | |
| User 1 Books | 0 | 5 | 5 | |
| User 2 Electronics | 2 | 0 | 0 | |
| User 2 Clothing | 0 | 2 | 2 | |
| User 2 Books | 0 | 2 | 2 | |
| User 3 Electronics | 3 | 0 | 0 | |
| User 3 Clothing | 0 | 3 | 3 | |
| User 3 Books | 0 | 3 | 3 | |

| | Clothing_F1 | Clothing_F2 | Clothing_F3 | Books_F1 | Books_F2 | \ |
|--------------------|-------------|-------------|-------------|----------|----------|---|
| User 1 Electronics | 3 | 0 | 0 | 4 | 0 | |
| User 1 Clothing | 0 | 3 | 3 | 0 | 4 | |
| User 1 Books | 0 | 3 | 3 | 0 | 4 | |
| User 2 Electronics | 4 | 0 | 0 | 5 | 0 | |
| User 2 Clothing | 0 | 4 | 4 | 0 | 5 | |
| User 2 Books | 0 | 4 | 4 | 0 | 5 | |
| User 3 Electronics | 4 | 0 | 0 | 4 | 0 | |
| User 3 Clothing | 0 | 4 | 4 | 0 | 4 | |
| User 3 Books | 0 | 4 | 4 | 0 | 4 | |

| | Books_F3 |
|--------------------|----------|
| User 1 Electronics | 0 |
| User 1 Clothing | 4 |
| User 1 Books | 4 |

| | |
|--------------------|---|
| User 2 Electronics | 0 |
| User 2 Clothing | 5 |
| User 2 Books | 5 |
| User 3 Electronics | 0 |
| User 3 Clothing | 4 |
| User 3 Books | 4 |

Top 3 recommendations for User 1:

Electronics_F2

Electronics_F3

Books_F3

A simple visualization of this recommendation system is shown in Fig 11.

```
# Visualization
def plot_recommendations(user_index):
    """ Plot the predicted ratings for each category for a given user. """
    user_ratings = K[user_index * len(C):(user_index + 1) * len(C), :]
    predicted_ratings = np.sum(user_ratings, axis=0)
    categories = df_K.columns
    plt.figure(figsize=(6, 5))
    plt.bar(categories, predicted_ratings)
    plt.xlabel('Categories')
    plt.ylabel('Predicted Ratings')
    plt.title(f'Predicted Ratings for User {user_index + 1}')
    plt.xticks(rotation=45)
    plt.show()

# Plot recommendations for User 1
plot_recommendations(user_index)
```

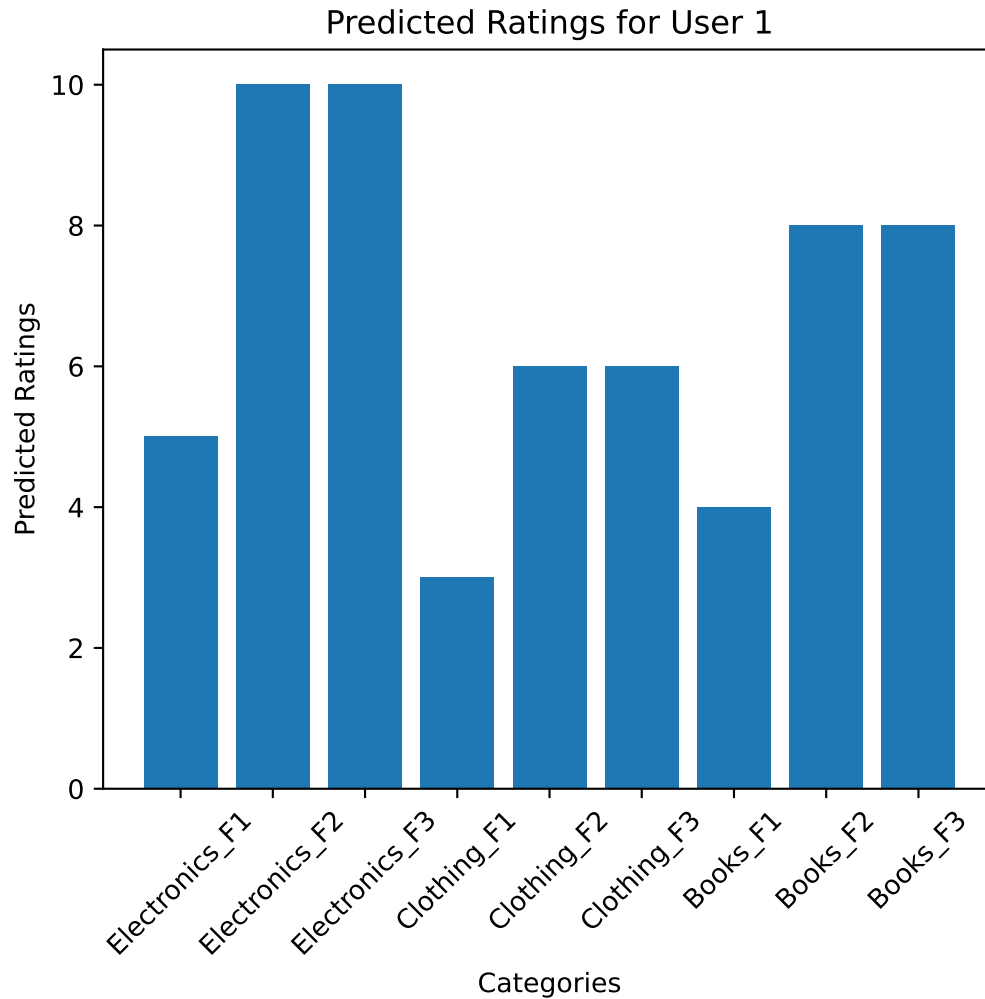


Figure 11: EDA for the Recommendation System

This micro project illustrate one of the popular use of Kronecker product on ML application.

Matrix Measures of Practical Importance

Matrix measures, such as rank and determinant, play crucial roles in linear algebra. While both rank and determinant provide valuable insights into the properties of a matrix, they serve different purposes. Understanding their roles and applications is essential for solving complex problems in computer science, engineering, and applied mathematics.

Determinant

Determinant of a 2×2 matrix $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ is defined as $|A| = ad - bc$. Determinant of higher order square matrices can be found using the Laplace method or Sarrus method.

The determinant of a matrix provides information about the matrix's invertibility and scaling factor for volume transformation. Specifically:

1. *Invertibility*: A matrix is invertible if and only if its determinant is non-zero.
2. *Volume Scaling*: The absolute value of the determinant gives the scaling factor by which the matrix transforms volume.
3. *Parallelism*: If the determinant of a matrix composed of vectors is zero, the vectors are linearly dependent, meaning they are parallel or redundant.
4. *Redundancy*: A zero determinant indicates that the vectors span a space of lower dimension than the number of vectors, showing redundancy.

! Least Possible Values of Determinant

1. *Least Positive Determinant*: For a 1×1 matrix, the smallest non-zero determinant is any positive value, typically ϵ , where ϵ is a small positive number.
2. *Least Non-Zero Determinant*: For higher-dimensional matrices, the smallest non-zero determinant is a non-zero value that represents the smallest area or volume spanned by the matrix's rows or columns. For example a 2×2 matrix with determinant ϵ could be:

$$B = \begin{pmatrix} \epsilon & 0 \\ 0 & \epsilon \end{pmatrix}$$

Here, ϵ is a small positive number, indicating a very small but *non-zero* area.

Now let's look into the most important matrix measure for advanced application in Linear Algebra.

As we know the matrix is basically a representation tool that make things abstract- remove unnecessary details. Then the matrix itself can be represented in many ways. This is the real story telling with this most promising mathematical structure. Consider a context of collecting feedback about a product in three aspects- cost, quality and practicality. For simplicity in calculation, we consider responses from 3 customers only. The data is shown in Table 7.

Table 7: User rating of a consumer product

| User | Cost | Quality | Practicality |
|--------|------|---------|--------------|
| User-1 | 1 | 4 | 5 |

| User | Cost | Quality | Practicality |
|--------|------|---------|--------------|
| User-2 | 3 | 2 | 5 |
| User-3 | 2 | 1 | 3 |

It's perfect and nice looking. But both mathematics and a computer can't handle this table as it is. So we create an abstract representation of this data- the rating matrix. Using the traditional approach, let's represent this rating data as:

$$A = \begin{bmatrix} 1 & 4 & 5 \\ 3 & 2 & 5 \\ 2 & 1 & 3 \end{bmatrix}$$

Now both the column names and row indices were removed and the data is transformed into the abstract form. This representation has both advantages and disadvantages. Be positive! So we are focused only in the advantages.

Just consider the product. Its sales fully based on its features. So the product sales perspective will be represented in terms of the features- cost, quality and practicality. These features are columns of our rating matrix. Definitely people will have different rating for these features. Keeping all these in mind let's introduce the concept of *linear combination*. This leads to a new matrix product as shown below.

$$\begin{aligned} Ax &= \begin{bmatrix} 1 & 4 & 5 \\ 3 & 2 & 5 \\ 2 & 1 & 3 \end{bmatrix} x \\ &= \begin{bmatrix} 1 & 4 & 5 \\ 3 & 2 & 5 \\ 2 & 1 & 3 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix} x_1 + \begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix} x_2 + \begin{bmatrix} 5 \\ 5 \\ 3 \end{bmatrix} x_3 \end{aligned}$$

As the number of users increases, the product sales perspective become more informative. In short the span of the features define the feature space of the product. In real cases, a manufacture wants to know what are the features really influence the customers. This new matrix product will help the manufactures to identify that features!

So we are going to define this new matrix product as the feature space, that will provide more insights to this context as:

$$A = CR$$

Where C is the column space and R is the row reduced Echelon form of A . But the product is not the usual scalar projection, Instead the weight of linear combination of elements in the column space.

Let's formally illustrate this in our example. From the first observation itself, it is clear that last column is just the sum of first and second columns (That is in our context the feature 'practicality' is just depends on 'cost' and 'quality'. meaningful?). So only first columns are independent and so spans the column space.

$$C = \begin{bmatrix} 1 & 4 \\ 3 & 2 \\ 2 & 1 \end{bmatrix}$$

Now look into the matrix R . Applying elementary row transformations, A will transformed into:

$$R = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Hence we can form a decomposition for the given rating matrix, A as:

$$\begin{aligned} A &= CR \\ &= \begin{bmatrix} 1 & 4 \\ 3 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \end{aligned}$$

This decomposition says that there are only two independent features (columns) and the third feature (column) is the sum of first two features (columns).

! Interpretation of the R matrix

Each column in the R matrix represents the weights for linear combination of vectors in the column space to get that column in A . In this example, third column of R is $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$. This means that third column of A will be $1 \times C_1 + 1 \times C_2$ of the column space, C !

This first matrix decomposition donate a new type of matrix product (outer product) and a new measure- the number of independent columns and number of independent rows. This count is called the *rank* of the matrix A . In the case of features, if the rank of the column space

is less than the number of features then definitely a less number of feature set will perfectly represent the data. This will help us to reduce the dimension of the dataset and there by reducing computational complexities in data analysis and machine Learning jobs.

In the above discussion, we consider only the columns of A . Now we will mention the row space. It is the set of all linearly independent rows of A . For any matrix A , both the row space and column space are of same rank. This correspondance is a helpful result in many practical applications.

Now we consider a stable equation, $Ax = 0$. With the usual notation of dot product, it implies that x is orthogonal to A . Set of all those independent vectors which are orthogonal to A constitute a new space of interest. It is called the *null space* of A . If A represents a linear transformation, then the null space will be populated by those non-zero vectors which are *nullified* by the transformation A . As a summary of this discussion, the row space and null space of a matrix A creates an orthogonal system. Considering the relationship between A and A^T , it is clear that row space of A is same as the column space of A^T and vice verse are. So we can restate the orthogonality as: ‘the null space of A is orthogonal to the column space of A^T ’ and ‘the null space of A^T is orthogonal to the column space of A ’. Mathematically this property can be represents as follows.

i Note

$$\begin{aligned}\mathcal{N}(A) &\perp \mathcal{C}(A^T) \\ \mathcal{N}(A^T) &\perp \mathcal{C}(A)\end{aligned}$$

In the given example, solving $Ax = 0$ we get $x = [1 \quad 1 \quad -1]^T$.

So the rank of $\mathcal{N}(A) = 1$. Already we have rank of $A = 2$. This leads to an interesting result:

$$\text{Rank}(A) + \text{Rank}(\mathcal{N}(A)) = 3$$

This observation can be framed as a theorem.

Rank Nullity Theorem

The rank-nullity theorem is a fundamental theorem in linear algebra that is important for understanding the connections between mathematical operations in engineering, physics, and computer science. It states that the sum of the rank and nullity of a matrix equals the number of columns in the matrix. The rank is the maximum number of linearly independent columns, and the nullity is the dimension of the nullspace.

Theorem 0.1 (Rank Nullity Theorem). *The Rank-Nullity Theorem states that for any $m \times n$ matrix A , the following relationship holds:*

$$\text{Rank}(A) + \text{Nullity}(A) = n$$

where: - **Rank** of A is the dimension of the column space of A , which is also equal to the dimension of the row space of A . - **Nullity** of A is the dimension of the null space of A , which is the solution space to the homogeneous system $A\mathbf{x} = \mathbf{0}$.

Steps to Formulate for Matrix A

1. **Find the Rank of A :** The rank of a matrix is the maximum number of linearly independent columns (or rows). It can be determined by transforming A into its row echelon form or reduced row echelon form (RREF).
2. **Find the Nullity of A :** The nullity is the dimension of the solution space of $A\mathbf{x} = \mathbf{0}$. This can be found by solving the homogeneous system and counting the number of free variables.
3. **Apply the Rank-Nullity Theorem:** Use the rank-nullity theorem to verify the relationship.

Example 1: Calculate the rank and nullity of $A = \begin{bmatrix} 1 & 4 & 5 \\ 3 & 2 & 5 \\ 2 & 1 & 3 \end{bmatrix}$ and verify the rank nullity theorem.

1. **Row Echelon Form:**

Perform Gaussian elimination on A :

$$A = \begin{bmatrix} 1 & 4 & 5 \\ 3 & 2 & 5 \\ 2 & 1 & 3 \end{bmatrix}$$

Perform row operations to get it to row echelon form:

- Subtract 3 times row 1 from row 2:

$$\begin{bmatrix} 1 & 4 & 5 \\ 0 & -10 & -10 \\ 2 & 1 & 3 \end{bmatrix}$$

- Subtract 2 times row 1 from row 3:

$$\begin{bmatrix} 1 & 4 & 5 \\ 0 & -10 & -10 \\ 0 & -7 & -7 \end{bmatrix}$$

- Add $\frac{7}{10}$ times row 2 to row 3:

$$\begin{bmatrix} 1 & 4 & 5 \\ 0 & -10 & -10 \\ 0 & 0 & 0 \end{bmatrix}$$

The matrix is now in row echelon form.

Rank is the number of non-zero rows, which is 2.

2. **Find the Nullity:** The matrix A has 3 columns. The number of free variables in the solution of $A\mathbf{x} = \mathbf{0}$ is $3 - \text{Rank}$.

So,

$$\text{Nullity}(A) = 3 - 2 = 1$$

3. **Apply the Rank-Nullity Theorem:**

$$\text{Rank}(A) + \text{Nullity}(A) = 2 + 1 = 3$$

This matches the number of columns of A , confirming the theorem.

Fundamental Subspaces

In section (**note-ortho?**), we have seen that for any matrix A , there is two pairs of inter-related orthogonal spaces. This leads to the concept of Fundamental sub spaces.

Matrices are not just arrays of numbers; they can represent linear transformations too. A linear transformation maps vectors from one vector space to another while preserving vector addition and scalar multiplication. The matrix A can be viewed as a representation of a linear transformation T from \mathbb{R}^n to \mathbb{R}^m where:

$$T(\mathbf{x}) = A\mathbf{x}$$

In this context:

- The column space of A represents the range of T , which is the set of all possible outputs.

- The null space of A represents the kernel of T , which is the set of vectors that are mapped to the zero vector.

The Four Fundamental Subspaces

Understanding the four fundamental subspaces helps in analyzing the properties of a linear transformation. These subspaces are:

Definition 0.1 (Four Fundamental Subspaces). Let $T : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ be a linear transformation and A represents the matrix of transformation. The four fundamental subspaces are defined as:

1. **Column Space (Range)**: The set of all possible outputs of the transformation. For matrix A , this is the span of its columns. It represents the image of \mathbb{R}^n under T .
2. **Null Space (Kernel)**: The set of all vectors that are mapped to the zero vector by the transformation. For matrix A , this is the solution space of $A\mathbf{x} = \mathbf{0}$.
3. **Row Space**: The span of the rows of A . This space is crucial because it helps in understanding the rank of A . The dimension of the row space is equal to the rank of A , which represents the maximum number of linearly independent rows.
4. **Left Null Space**: The set of all vectors \mathbf{y} such that $A^T\mathbf{y} = \mathbf{0}$. It provides insight into the orthogonal complement of the row space.

This idea is depicted as a ‘Big picture of the four sub spaces of a matrix’ in the Strang’s text book on Linear algebra for every one (Strang 2020). This ‘Big Picture’ is shown in Fig- 12.

A video session from Strang’s session is here:

<https://youtu.be/rwLOdfc4dw?si=DsJb8KJTf05hHc76>

Practice Problems

Problem 1: Express the vector $(1, -2, 5)$ as a linear combination of the vectors $(1, 1, 1)$, $(1, 2, 3)$ and $(2, -1, 1)$.

Problem 2: Show that the feature vector $(2, -5, 3)$ is not linearly associated with the features $(1, -3, 2)$, $(2, -4, -1)$ and $(1, -5, 7)$.

Problem 3: Show that the feature vectors $(1, 1, 1)$, $(1, 2, 3)$ and $(2, -1, 1)$ are non-redundant.

Problem 4: Prove that the features $(1, -1, 1)$, $(0, 1, 2)$ and $(3, 0, -1)$ form basis for the feature space.

Problem 5: Check whether the vectors $(1, 2, 1)$, $(2, 1, 4)$ and $(4, 5, 6)$ form a basis for \mathbb{R}^3 .

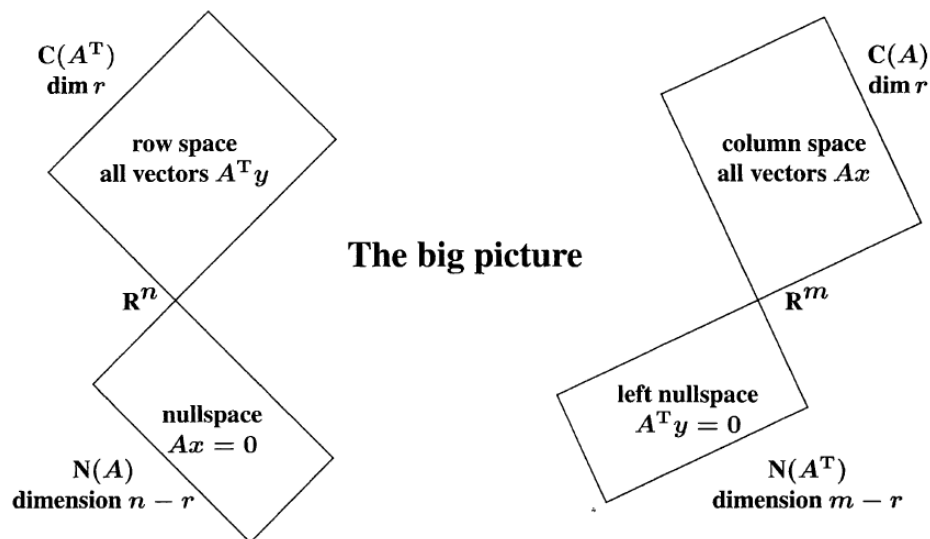


Figure 12: The Big Picture of Fundamental Subspaces

Problem 6: Find the four fundamental subspaces of the feature space created by $(1, 2, 1)$, $(2, 1, 4)$ and $(4, 5, 6)$.

Problem 7: Find the four fundamental subspaces and its dimensions of the matrix $\begin{bmatrix} 1 & 2 & 4 \\ 2 & 1 & 5 \\ 1 & 4 & 6 \end{bmatrix}$.

Problem 8: Express $A = \begin{bmatrix} 1 & 2 & -1 \\ 3 & 1 & -1 \\ 2 & -1 & 0 \end{bmatrix}$ as the Kronecker product of the column space and the row space in the form $A = C \otimes R$.

Problem 9: Find the four fundamental subspaces of $A = \begin{bmatrix} 1 & 2 & 0 & 2 & 5 \\ -2 & -5 & 1 & -1 & -8 \\ 0 & -3 & 3 & 4 & 1 \\ 3 & 6 & 0 & -7 & 2 \end{bmatrix}$.

Problem 10: Find the four fundamental subspaces of $A = \begin{bmatrix} -1 & 2 & -1 & 5 & 6 \\ 4 & -4 & -4 & -12 & -8 \\ 2 & 0 & -6 & -2 & 4 \\ -3 & 1 & 7 & -2 & 12 \end{bmatrix}$.

Problem 11: Express $A = \begin{bmatrix} 2 & 3 & -1 & -1 \\ 1 & -1 & -2 & -4 \\ 3 & 1 & 3 & -2 \\ 6 & 3 & 0 & -7 \end{bmatrix}$ in $A = C \otimes R$, where C is the column space and R is the row space of A .

Problem 12: Express $A = \begin{bmatrix} 0 & 1 & -3 & -1 \\ 1 & 0 & 1 & 1 \\ 3 & 1 & 0 & 2 \\ 1 & 1 & -2 & 0 \end{bmatrix}$ in $A = C \otimes R$, where C is the column space and R is the row space of A .

Problem 13: Show that the feature vectors $(2, 3, 0)$, $(1, 2, 0)$ and $(8, 13, 0)$ are redundant and hence find the relationship between them.

Problem 14: Show that the feature vectors $(1, 2, 1)$, $(4, 1, 2)$, $(-3, 8, 1)$ and $(6, 5, 4)$ are redundant and hence find the relationship between them.

Problem 15: Show that the feature vectors $(1, 2, -1, 0)$, $(1, 3, 1, 2)$, $(4, 2, 1, 0)$ and $(6, 1, 0, 1)$ are redundant and hence find the relationship between them.

! Important

Three Parts of the *Fundamental theorem* The fundamental theorem of linear algebra relates all four of the fundamental subspaces in a number of different ways. There are main parts to the theorem:

Part 1:(Rank nullity theorem) The column and row spaces of an $m \times n$ matrix A both have dimension r , the rank of the matrix. The nullspace has dimension $n - r$, and the left nullspace has dimension $m - r$.

Part 2:(Orthogonal subspaces) The nullspace and row space are orthogonal. The left nullspace and the column space are also orthogonal.

Part 3:(Matrix decomposition) The final part of the fundamental theorem of linear algebra constructs an orthonormal basis, and demonstrates a singular value decomposition: any matrix M can be written in the form $M = U\Sigma V^T$, where $U_{m \times m}$ and $V_{n \times n}$ are unitary matrices, $\Sigma_{m \times n}$ matrix with nonnegative values on the diagonal.

This part of the fundamental theorem allows one to immediately find a basis of the subspace in question. This can be summarized in the following table.

| Subspace | Subspace
of | Symbol | Dimension | Basis |
|-----------------------|----------------|------------------|-----------|-----------------------------|
| Column space | \mathbb{R}^m | $\text{im}(A)$ | r | First r columns of U |
| Nullspace
(kernel) | \mathbb{R}^n | $\text{ker}(A)$ | $n - r$ | Last $n - r$ columns of V |
| Row space | \mathbb{R}^n | $\text{im}(A^T)$ | r | First r columns of V |

| | | | | |
|----------------------------|----------------|-------------|---------|-----------------------------|
| Left nullspace
(kernel) | \mathbb{R}^m | $\ker(A^T)$ | $m - r$ | Last $m - r$ columns of U |
|----------------------------|----------------|-------------|---------|-----------------------------|

Computational methods to find all the four fundamental subspaces of a matrix

There are different approaches to find the four fundamental subspaces of a matrix using **Python**. Simplest method is just convert our mathematical procedure into **Python** functions and call them to find respective spaces. This method is illustrated below.

```
# importing numpy library for numerical computation
import numpy as np
# define the function create the row-reduced Echelon form of given matrix
def row_echelon_form(A):
    """Convert matrix A to its row echelon form."""
    A = A.astype(float)
    rows, cols = A.shape
    for i in range(min(rows, cols)):
        # Pivot: find the maximum element in the current column
        max_row = np.argmax(np.abs(A[i:, i])) + i
        if A[max_row, i] == 0:
            continue # Skip if the column is zero
        # Swap the current row with the max_row
        A[[i, max_row]] = A[[max_row, i]]
        # Eliminate entries below the pivot
        for j in range(i + 1, rows):
            factor = A[j, i] / A[i, i]
            A[j, i:] -= factor * A[i, i:]
    return A

# define function to generate null space from the row-reduced echelon form
def null_space_of_matrix(A, rtol=1e-5):
    """Compute the null space of a matrix A using row reduction."""
    A_reduced = row_echelon_form(A)
    rows, cols = A_reduced.shape
    # Identify pivot columns
    pivots = []
    for i in range(rows):
        for j in range(cols):
            if np.abs(A_reduced[i, j]) > rtol:
                pivots.append(j)
```

```

        break
    free_vars = set(range(cols)) - set(pivots)

    null_space = []
    for free_var in free_vars:
        null_vector = np.zeros(cols)
        null_vector[free_var] = 1
        for pivot, row in zip(pivots, A_reduced[:len(pivots)]):
            null_vector[pivot] = -row[free_var]
        null_space.append(null_vector)

    return np.array(null_space).T

# define the function to generate the row-space of A

def row_space_of_matrix(A):
    """Compute the row space of a matrix A using row reduction."""
    A_reduced = row_echelon_form(A)
    # The non-zero rows of the reduced matrix form the row space
    non_zero_rows = A_reduced[~np.all(A_reduced == 0, axis=1)]
    return non_zero_rows

# define the function to generate the column space of A

def column_space_of_matrix(A):
    """Compute the column space of a matrix A using row reduction."""
    A_reduced = row_echelon_form(A)
    rows, cols = A_reduced.shape
    # Identify pivot columns
    pivots = []
    for i in range(rows):
        for j in range(cols):
            if np.abs(A_reduced[i, j]) > 1e-5:
                pivots.append(j)
                break
    column_space = A[:, pivots]
    return column_space

```

Examples:

1. Find all the fundamental subspaces of $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$.

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

print("Matrix A:")
print(A)

# Null Space
null_space_A = null_space_of_matrix(A)
print("\nNull Space of A:")
print(null_space_A)

# Row Space
row_space_A = row_space_of_matrix(A)
print("\nRow Space of A:")
print(row_space_A)

# Column Space
column_space_A = column_space_of_matrix(A)
print("\nColumn Space of A:")
print(column_space_A)
```

```
Matrix A:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```
Null Space of A:
[[-9.          ]
 [-1.71428571]
 [ 1.          ]]
```

```
Row Space of A:
[[7.00000000e+00 8.00000000e+00 9.00000000e+00]
 [0.00000000e+00 8.57142857e-01 1.71428571e+00]
 [0.00000000e+00 5.55111512e-17 1.11022302e-16]]
```

Column Space of A:

$\begin{bmatrix} 1 & 2 \\ 4 & 5 \\ 7 & 8 \end{bmatrix}$

Rank and Solution of System of Linear Equations

In linear algebra, the rank of a matrix is a crucial concept for understanding the structure of a system of linear equations. It provides insight into the solutions of these systems, helping us determine the number of independent equations and the nature of the solution space.

Definition 0.2 (Rank and System Consistency). The rank of a matrix A is defined as the maximum number of linearly independent rows or columns. When solving a system of linear equations represented by $A\mathbf{x} = \mathbf{b}$, where A is an $m \times n$ matrix and \mathbf{b} is a vector, the rank of A plays a crucial role in determining the solution's existence and uniqueness.

Consistency of the System

1. **Consistent System:** A system of linear equations is consistent if there exists at least one solution. This occurs if the rank of the coefficient matrix A is equal to the rank of the augmented matrix $[A|\mathbf{b}]$. Mathematically, this can be expressed as:

$$\text{rank}(A) = \text{rank}([A|\mathbf{b}])$$

If this condition is met, the system has solutions. The solutions can be:

- **Unique** if the rank equals the number of variables.
- **Infinitely many** if the rank is less than the number of variables.

2. **Inconsistent System:** A system is inconsistent if there are no solutions. This occurs when:

$$\text{rank}(A) \neq \text{rank}([A|\mathbf{b}])$$

In this case, the equations represent parallel or conflicting constraints that cannot be satisfied simultaneously.

Use of Null space in creation of general solution from particular solution

If the system $AX = b$ has many solutions, then the general solution of the system can be found using a particular solution and the elements in the null space of the coefficient matrix A as

$$X = x_p + tX_N$$

where X is the general solution and t is a free variable (parameter) and $X_N \in N(A)$.

Computational method to solve system of linear equations.

If for a system $AX = b$, $\det(A) \neq 0$, then the system has a unique solution and can be found by `solve()` function from NumPy. If the system is consistent and many solutions, then computationally we will generate the general solution using the $N(A)$. A detailed Python code is given below.

```
import numpy as np

def check_consistency(A, b):
    """
    Check the consistency of a linear system  $Ax = b$  and return the solution
    ↪ if consistent.

    Parameters:
    A (numpy.ndarray): Coefficient matrix.
    b (numpy.ndarray): Right-hand side vector.

    Returns:
    tuple: A tuple with consistency status, particular solution (if
    ↪ consistent), and null space (if infinite solutions).
    """
    A = np.array(A)
    b = np.array(b)

    # Augment the matrix A with vector b
    augmented_matrix = np.column_stack((A, b))

    # Compute ranks
    rank_A = np.linalg.matrix_rank(A)
    rank_augmented = np.linalg.matrix_rank(augmented_matrix)

    # Check for consistency
    if rank_A == rank_augmented:
        if rank_A == A.shape[1]:
            # Unique solution
            solution = np.linalg.solve(A, b)
            return "Consistent and has a unique solution", solution, None
        else:
            # Infinite solutions
```

```

        # Infinitely many solutions
        particular_solution = np.linalg.lstsq(A, b, rcond=None)[0]
        null_space = null_space_of_matrix(A)
        return "Consistent but has infinitely many solutions",
            ↪ particular_solution, null_space
    else:
        return "Inconsistent system (no solution)", None, None

def null_space_of_matrix(A):
    """
    Compute the null space of matrix A, which gives the set of solutions to
    ↪ Ax = 0.

    Parameters:
    A (numpy.ndarray): Coefficient matrix.

    Returns:
    numpy.ndarray: Basis for the null space of A.
    """
    u, s, vh = np.linalg.svd(A)
    null_mask = (s <= 1e-10) # Singular values near zero
    null_space = np.compress(null_mask, vh, axis=0)
    return null_space.T

```

Example 1: Solve

$$\begin{aligned}
 2x - y + z &= 1 \\
 x + 2y &= 3 \\
 3x + 2y + z &= 4
 \end{aligned}$$

```

# Example usage 1: System with a unique solution
A1 = np.array([[2, -1, 1], [1, 0, 2], [3, 2, 1]])
b1 = np.array([1, 3, 4])

status1, solution1, null_space1 = check_consistency(A1, b1)
print("Example 1 - Status:", status1)

if solution1 is not None:
    print("Solution:", solution1)
if null_space1 is not None:
    print("Null Space:", null_space1)

```

Example 1 - Status: Consistent and has a unique solution
Solution: [0.27272727 0.90909091 1.36363636]

Example 2: Solve the system of equations,

$$\begin{aligned}x + 2y + z &= 3 \\ 2x + 4y + 2z &= 6 \\ x + y + z &= 2\end{aligned}$$

```
# Example usage 2: System with infinitely many solutions
A2 = np.array([[1, 2, 1], [2, 4, 2], [1, 1, 1]])
b2 = np.array([3, 6, 2])

status2, solution2, null_space2 = check_consistency(A2, b2)
print("\nExample 2 - Status:", status2)

if solution2 is not None:
    print("Particular Solution:", solution2)
if null_space2 is not None:
    print("Null Space (Basis for infinite solutions):", null_space2)
```

Example 2 - Status: Consistent but has infinitely many solutions
Particular Solution: [0.5 1. 0.5]
Null Space (Basis for infinite solutions): [[7.07106781e-01]
[1.11022302e-16]
[-7.07106781e-01]]

1 Python Libraries for Computational Linear Algebra

In the first two modules, we gained a foundational understanding of **Python** programming and the basics of linear algebra, including fundamental subspaces such as row space, column space, and null space, both theoretically and through Python implementations. These essential concepts provided the groundwork for solving linear algebra problems manually and computationally. Now, as we move into Module 3, the focus shifts toward leveraging advanced **Python** libraries to handle more complex and large-scale computations in linear algebra efficiently.

This module introduces the powerful computational tools available in **Python**, such as **NumPy**, **SymPy**, **SciPy**, and **Matplotlib**. These libraries are designed to enhance the ability to perform both numerical and symbolic operations on matrices, vectors, and systems of equations. With **NumPy**'s high-performance array operations, **SymPy**'s symbolic computation abilities, and **SciPy**'s extensive collection of scientific routines, students will be able to compute solutions for real-world problems with ease. The module also incorporates visualization techniques through **Matplotlib**, allowing students to graphically represent mathematical solutions, interpret data, and communicate their findings effectively. This module empowers students to move beyond manual calculations and explore advanced problem-solving strategies computationally.

1.1 Introduction to NumPy

In this section, we will introduce **NumPy**, the core library for scientific computing in Python. NumPy provides support for arrays, matrices, and a host of mathematical functions to operate on these structures. This is particularly useful for linear algebra computations, making it an essential tool in computational mathematics. The library also serves as the foundation for many other Python libraries like SciPy, Pandas, and Matplotlib.

1.1.1 Purpose of Using NumPy

The primary purpose of NumPy is to enable efficient numerical computations involving large datasets, vectors, and matrices. With NumPy, one can perform mathematical operations on arrays and matrices in a way that is highly optimized for performance, both in terms of memory and computational efficiency (Harris et al. 2020).

Some key advantages of using NumPy include:

- **Efficient handling of large datasets:** Arrays in NumPy are optimized for performance and consume less memory compared to native Python lists.
- **Matrix operations:** NumPy provides built-in functions for basic matrix operations, allowing one to perform tasks like matrix multiplication, transpose, and inversion easily.
- **Linear algebra:** It includes functions for solving systems of equations, finding eigenvalues and eigenvectors, computing matrix factorizations, and more.

1.2 Basic Operations in NumPy

This section will present several examples of using NumPy array manipulation to access data and subarrays, and to split, reshape, and join the arrays. While the types of operations shown here may seem a bit dry and pedantic, they comprise the building blocks of many other examples used throughout the book. Get to know them well!

We'll cover a few categories of basic array manipulations here:

- *Attributes of arrays:* Determining the size, shape, memory consumption, and data types of arrays
- *Indexing of arrays:* Getting and setting the value of individual array elements
- *Slicing of arrays:* Getting and setting smaller subarrays within a larger array
- *Reshaping of arrays:* Changing the shape of a given array
- *Joining and splitting of arrays:* Combining multiple arrays into one, and splitting one array into many

Loading numpy to a python programme

Syntax

```
import numpy as "name of instance"
```

eg: `import numpy as np`

1.2.0.1 Array Creation

At the core of NumPy is the **ndarray** object, which represents arrays and matrices. Here's how to create arrays using NumPy:

```
import numpy as np

# Creating a 1D array
arr = np.array([1, 2, 3, 4, 5])

# Creating a 2D matrix
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print("1D Array: \n", arr)
print("2D Matrix: \n", matrix)
```

1.2.1 Define different types of numpy arrays

As the first step to understand different types of arrays in NumPy let us consider the following examples.

1.2.1.1 1D Array (Vector)

In NumPy, a one-dimensional (1D) array is similar to a list or vector in mathematics. It consists of a single row or column of numbers, making it an ideal structure for storing sequences of values.

```
import numpy as np

# Creating a 1D array
arr = np.array([1, 2, 3, 4])
print(arr)
```

```
[1 2 3 4]
```

Here, `np.array()` is used to create a 1D array (or vector) containing the values `[1, 2, 3, 4]`. The array represents a single sequence of numbers, and it is the basic structure of NumPy.

i Use:

A 1D array can represent many things, such as a vector in linear algebra, a list of numbers, or a single dimension of data in a machine learning model.

1.2.1.2 2D Array (Matrix)

A two-dimensional (2D) array is equivalent to a matrix in mathematics. It consists of rows and columns and is often used to store tabular data or perform matrix operations.

```
from IPython.display import display, HTML
# Creating a 2D array (Matrix)
matrix = np.array([[1, 2, 3], [4, 5, 6]])
display(matrix)
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

In this example, the 2D array (or matrix) is created using `np.array()` by providing a list of lists, where each list represents a row in the matrix. The result is a matrix with two rows and three columns.

i Use:

Matrices are fundamental structures in linear algebra. They can represent anything from transformation matrices in graphics to coefficients in systems of linear equations.

1.2.1.3 Zero Arrays

Zero arrays are used to initialize matrices or arrays with all elements set to zero. This can be useful when creating placeholder arrays where the values will be computed or updated later.

```
# Creating an array of zeros
zero_matrix = np.zeros((3, 3))
print(zero_matrix)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

The `np.zeros()` function creates an array filled with zeros. In this example, we create a 3x3 matrix with all elements set to zero.

i Use:

Zero arrays are commonly used in algorithms that require the allocation of memory for arrays that will be updated later.

1.2.1.4 Identity Matrix

An identity matrix is a square matrix with ones on the diagonal and zeros elsewhere. It plays a crucial role in linear algebra, especially in solving systems of linear equations and matrix factorizations.

```
# Creating an identity matrix
identity_matrix = np.eye(3)
print(identity_matrix)
```

```
[[1.  0.  0.]
 [0.  1.  0.]
 [0.  0.  1.]]
```

The `np.eye(n)` function creates an identity matrix with the specified size. In this case, we create a 3x3 identity matrix, where all diagonal elements are 1, and off-diagonal elements are 0.

1.2.1.5 Arange Function

The `np.arange()` function is used to create an array with evenly spaced values within a given range. It's similar to Python's built-in `range()` function but returns a NumPy array instead of a list.

```
# Creating an array using arange
arr = np.arange(1, 10, 2)
print(arr)
```

```
[1 3 5 7 9]
```

Here, `np.arange(1, 10, 2)` generates an array of numbers starting at 1, ending before 10, with a step size of 2. The result is `[1, 3, 5, 7, 9]`.

i Use:

This function is useful when creating arrays for loops, data generation, or defining sequences for analysis.

1.2.1.6 Linspace Function

The `np.linspace()` function generates an array of evenly spaced values between a specified start and end, with the number of intervals defined by the user.

```
# Creating an array using linspace
arr = np.linspace(0, 1, 5)
print(arr)
```

```
[0.    0.25 0.5   0.75 1.   ]
```

`np.linspace(0, 1, 5)` creates an array with 5 evenly spaced values between 0 and 1, including both endpoints. The result is `[0. , 0.25, 0.5 , 0.75, 1.]`. :::{.callout-note} ### Use: `linspace()` is often used when you need a specific number of evenly spaced points within a range, such as for plotting functions or simulating data. :::

1.2.1.7 Reshaping Arrays

The `reshape()` function changes the shape of an existing array without changing its data. It's useful when you need to convert an array to a different shape for computations or visualizations.

```
# Reshaping an array
arr = np.arange(1, 10)
reshaped_arr = arr.reshape(3, 3)
print(reshaped_arr)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

In this example, a 1D array with 9 elements is reshaped into a 3x3 matrix using the `reshape()` method. The data remains the same but is now structured in a 2D form.

i Use:

Reshaping is critical in linear algebra and machine learning when working with input data of different dimensions.

1.2.1.8 Random Arrays

NumPy's random module is used to generate arrays with random values. These arrays are useful in simulations, testing algorithms, and initializing variables in machine learning.

```
# Creating a random array
random_arr = np.random.rand(3, 3)
print(random_arr)
```

```
[[0.92165029 0.03973442 0.70572319]
 [0.34062442 0.85814191 0.87849212]
 [0.41109116 0.37151661 0.26540892]]
```

`np.random.rand(3, 3)` creates a 3x3 matrix with random values between 0 and 1. The `rand()` function generates random floats in the range $[0, 1)$.

i Use:

Random arrays are commonly used for initializing weights in machine learning algorithms, simulating stochastic processes, or for testing purposes.

! Syntax

- **One-Dimensional Array:** `np.array([list of values])`
- **Two-Dimensional Array:** `np.array([[list of values], [list of values]])`
- **Zero Array:** `np.zeros(shape)`
 - `shape` is a tuple representing the dimensions (e.g., `(3, 3)` for a 3x3 matrix).
- **Identity Matrix:** `np.eye(n)`
 - `n` is the size of the matrix.
- **Arrange Function:** `np.arange(start, stop, step)`

- `start` is the starting value, `stop` is the end value (exclusive), and `step` is the increment.
- **Linspace Function:** `np.linspace(start, stop, num)`
 - `start` and `stop` define the range, and `num` is the number of evenly spaced values.
- **Reshaping Arrays:** `np.reshape(array, new_shape)`
 - `array` is the existing array, and `new_shape` is the desired shape (e.g., (3, 4)).
- **Random Arrays without Using rand:** `np.random.randint(low, high, size)`
 - `low` and `high` define the range of values, and `size` defines the shape of the array.

1.2.2 Review Questions

Q1: What is the purpose of using `np.array()` in NumPy?

Ans: `np.array()` is used to create arrays in NumPy, which can be 1D, 2D, or multi-dimensional arrays.

Q2: How do you create a 2D array in NumPy?

Ans: A 2D array can be created using `np.array([[list of values], [list of values]])`.

Q3: What is the difference between `np.zeros()` and `np.eye()`?

Ans: `np.zeros()` creates an array filled with zeros of a specified shape, while `np.eye()` creates an identity matrix of size `n`.

Q4: What is the syntax to create an evenly spaced array using `np.linspace()`?

Ans: The syntax is `np.linspace(start, stop, num)`, where `num` specifies the number of evenly spaced points between `start` and `stop`.

Q5: How can you reshape an array in NumPy?

Ans: Arrays can be reshaped using `np.reshape(array, new_shape)`, where `new_shape` is the desired shape for the array.

Q6: How do you create a random integer array in a specific range using NumPy?

Ans: You can use `np.random.randint(low, high, size)` to generate a random array with integers between `low` and `high`, and `size` defines the shape of the array.

Q7: What does the function `np.arange(start, stop, step)` do?

Ans: It generates an array of values from `start` to `stop` (exclusive) with a step size of `step`.

Q8: What is array broadcasting in NumPy?

Ans: Array broadcasting allows NumPy to perform element-wise operations on arrays of different shapes by automatically expanding the smaller array to match the shape of the larger array.

Q9: How do you generate a zero matrix of size 4x4 in NumPy?

Ans: A zero matrix of size 4x4 can be generated using `np.zeros((4, 4))`.

Q10: What is the difference between `np.arange()` and `np.linspace()`?

Ans: `np.arange()` generates values with a specified step size, while `np.linspace()` generates evenly spaced values over a specified range and includes the endpoint.

1.2.3 Tensors in NumPy

A tensor is a generalized concept of matrices and vectors. In mathematical terms, tensors are multi-dimensional arrays, and their dimensionality (or rank) is what differentiates them from simpler structures like scalars (rank 0), vectors (rank 1), and matrices (rank 2). A tensor with three dimensions or more is often referred to as a higher-order tensor.

In practical terms, tensors can be seen as multi-dimensional arrays where each element is addressed by multiple indices. Tensors play a significant role in machine learning and deep learning frameworks, where operations on these multi-dimensional data structures are common.

1.2.3.1 Types of Tensors:

1. *Scalar (0-D Tensor)*: A single number.

Example: 5 Rank: 0 Shape: ()

2. *Vector (1-D Tensor)*: An array of numbers.

Example: [1, 2, 3] Rank: 1 Shape: (3)

3. *Matrix (2-D Tensor)*: A 2D array (rows and columns).

Example: [[1, 2, 3], [4, 5, 6]] Rank: 2 Shape: (2, 3)

4. *3-D Tensor*: An array of matrices.

Example: [[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]] Rank: 3 Shape: (2, 2, 3)

5. *N-D Tensor*: A tensor with N dimensions, where $N > 3$.

Example: A 4-D tensor could represent data with shape (n_samples, n_channels, height, width) in image processing.

1.2.3.2 Creating Tensors Using NumPy

In NumPy, tensors are represented as multi-dimensional arrays. You can create tensors in a way similar to how you create arrays, but you extend the dimensions to represent higher-order tensors.

Creating a 1D Tensor (Vector)

A 1D tensor is simply a vector. You can create one using `np.array()`:

```
import numpy as np
vector = np.array([1, 2, 3])
print(vector)
```

```
[1 2 3]
```

Creating a 2D Tensor (Matrix)

A 2D tensor is a matrix:

```
matrix = np.array([[1, 2, 3], [4, 5, 6]])
print(matrix)
```

```
[[1 2 3]
 [4 5 6]]
```

Creating a 3D Tensor

To create a 3D tensor (a stack of matrices):

```
tensor_3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(tensor_3d)
```

```
[[[ 1  2  3]
 [ 4  5  6]]

 [[ 7  8  9]
 [10 11 12]]]
```

Creating a 4D Tensor

In applications like deep learning, a 4D tensor is often used to represent a batch of images, where the dimensions could be (batch_size, channels, height, width):

```
tensor_4d = np.random.randint(10, size=(2, 3, 4, 5)) # 2 batches, 3
↳ channels, 4x5 images
print(tensor_4d)
```

```
[[[0 6 0 1 6]
   [0 9 5 7 6]
   [7 5 2 2 1]
   [6 6 1 9 9]]
```

```
 [[3 5 2 1 1]
   [6 7 3 6 0]
   [7 4 9 6 5]
   [0 8 6 9 2]]
```

```
 [[0 8 6 7 6]
   [2 9 7 7 5]
   [4 2 1 1 0]
   [7 7 0 2 3]]]
```

```
[[[8 9 8 5 7]
   [0 8 1 7 3]
   [0 9 4 4 4]
   [5 9 2 3 7]]
```

```
 [[4 1 8 2 2]
   [4 8 3 2 3]
   [4 1 2 9 6]
   [0 3 7 2 3]]
```

```
 [[0 7 9 4 5]
   [6 0 1 7 6]
   [8 2 5 2 4]
   [4 8 3 9 4]]]]
```

i General Syntax for Creating Tensors Using NumPy

```
np.array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)
```

- **object**: An array-like object (nested lists) that you want to convert to a tensor.
- **dtype**: The desired data type for the tensor elements.
- **copy**: Whether to copy the data (default True).
- **order**: Row-major (C) or column-major (F) order.
- **ndmin**: Specifies the minimum number of dimensions for the tensor.

In the next section we will discuss the various attributes of the NumPy array.

1.2.3.3 Attributes of arrays

Each array has attributes **ndim** (the number of dimensions), **shape** (the size of each dimension), and **size** (the total size of the array):

To illustrate this attributes, consider the following arrays:

```
#np.random.seed(0) # seed for reproducibility

x1 = np.random.randint(10, size=6) # One-dimensional array
x2 = np.random.randint(10, size=(3, 4)) # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5)) # Three-dimensional array
```

The array attributes of x_3 is shown below.

```
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

Another useful attribute are the **dtype** which return the data type of the array , **itemsize**, which lists the size (in bytes) of each array element, and **nbytes**, which lists the total size (in bytes) of the array:

```
print("dtype:", x3.dtype)
print("itemsize:", x3.itemsize, "bytes")
print("nbytes:", x3.nbytes, "bytes")
```

```
dtype: int32
itemsize: 4 bytes
nbytes: 240 bytes
```

1.2.4 Array Indexing: Accessing Single Elements

If you are familiar with Python's standard list indexing, indexing in NumPy will feel quite familiar. In a one-dimensional array, the i^{th} value (counting from zero) can be accessed by specifying the desired index in square brackets, just as with Python lists:

To demonstrate indexing, let us consider the one dimensional array:

```
x1=np.array([8, 5, 4, 7,4,1])
```

The fourth element of **x1** can be accessed as

```
print(x1[3])
```

7

Now the second element from the end of the the array **x1** can be accessed as:

```
print(x1[-2])
```

4

1.2.4.1 Accessing elements in multi-dimensional arrays

In a multi-dimensional array, items can be accessed using a comma-separated tuple of indices. An example is shown below.

```
x2=np.array([[3, 3, 9, 2],
             [5, 2, 3, 5],
             [7, 2, 7, 1]])
print(x2)# list the 2-D array
```

```
[[3 3 9 2]
 [5 2 3 5]
 [7 2 7 1]]
```

Now print the third element in the first row, we will use the following code.

```
x2[0, 2] ## access the element in first row and thrid column
```

```
np.int64(9)
```

```
x2[2, -1] ## access the element in the 3rd row and last column
```

```
np.int64(1)
```

1.2.4.2 Modification of array elements

Values can also be modified using any of the above index notation. An example is shown below.

```
x2[2, -1]=20 ## replace the 3rd row last column element of x2 by 20
print(x2)
```

```
[[ 3  3  9  2]
 [ 5  2  3  5]
 [ 7  2  7 20]]
```

Homogeneity of data in NumPy arrays

Keep in mind that, unlike Python lists, NumPy arrays have a fixed type. This means, for example, that if you attempt to insert a floating-point value to an integer array, the value will be silently truncated. Don't be caught unaware by this behavior!

1.2.4.3 Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (:) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array `x`, use this:

```
x[start:stop:step]
```

If any of these are unspecified, they default to the values `start=0`, `stop=size of dimension`, `step=1`.

We'll take a look at accessing sub-arrays in one dimension and in multiple dimensions.

1. One-dimensional subarrays

```
x = np.arange(0,10)
x
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
x[1:6] # first five elements
```

```
array([1, 2, 3, 4, 5])
```

```
x[5:] # elements after index 5
```

```
array([5, 6, 7, 8, 9])
```

```
x[4:7] # middle sub-array
```

```
array([4, 5, 6])
```

```
x[::2] # every other element with step 2 (alternate elements)
```

```
array([0, 2, 4, 6, 8])
```

2. Multi-dimensional subarrays (slicing)

Multi-dimensional slices work in the same way, with multiple slices separated by commas. For example:

```
# creating a two dimensional array
x2=np.array([[1,2,3],[3,4,5],[5,6,7]])
print(x2)
```

```
[[1 2 3]
 [3 4 5]
 [5 6 7]]
```

```
# selecting first 3 rows and first two columns from x2
print(x2[:3,:2])
```

```
[[1 2]
 [3 4]
 [5 6]]
```

```
print(x2[:3:2,:3:2]) # slice alternate elements in first three rows and first
↪ three columns
```

```
[[1 3]
 [5 7]]
```

Accessing array rows and columns

One commonly needed routine is accessing of single rows or columns of an array. This can be done by combining indexing and slicing, using an empty slice marked by a single colon (:)

For example *all the elements* in first column can be accessed as:

```
print(x2[:, 0]) # first column of x2
```

```
[1 3 5]
```


1.2.4.4 Creating copies of arrays

Despite the nice features of array views, it is sometimes useful to instead explicitly copy the data within an array or a subarray. This can be most easily done with the `copy()` method.

This concept can be illustrated through an example. Consider the array `x2` previously defined:

```
print(x2)
```

```
[[1 2 3]
 [3 4 5]
 [5 6 7]]
```

Now take a copy of a slice of `x2` as follows.

```
# create a copy of subarray and store it with the new name
x2_sub_copy = x2[:2, :2].copy()
print(x2_sub_copy)
```

```
[[1 2]
 [3 4]]
```

Now the changes happen in the copy will not affect the original array. For example, replace one element in the copy slice and check how it is reflected in both arrays.

```
x2_sub_copy[0, 0] = 42
print(x2_sub_copy)
```

```
[[42  2]
 [ 3  4]]
```

```
print(x2)
```

```
[[1 2 3]
 [3 4 5]
 [5 6 7]]
```

1.2.4.5 More on reshaping

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the `reshape` method. There are various approaches in reshaping of arrays. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, you can do the following:

```
np.arange(1, 10)
```

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

```
grid = np.arange(1, 10).reshape((9, 1))  
print(grid)
```

```
[[1]  
 [2]  
 [3]  
 [4]  
 [5]  
 [6]  
 [7]  
 [8]  
 [9]]
```

Note

Note that for this to work, the size of the initial array must match the size of the reshaped array. Where possible, the reshape method will use a no-copy view of the initial array, but with non-contiguous memory buffers this is not always the case.

Another common reshaping pattern is the conversion of a one-dimensional array into a two-dimensional row or column matrix. This can be done with the reshape method, or more easily done by making use of the `newaxis` keyword within a slice operation:

More Examples

```
x = np.array([1, 2, 3])  
print(x)
```

```
[1 2 3]
```

Now check the dimension of the array created.

```
x.shape
```

```
(3,)
```

Reshaping the array as a matrix.

```
# row vector via reshape
x1=x.reshape((1, 3))
x1.shape
```

```
(1, 3)
```

We can achieve the same using the `newaxis` function as shown below.

```
# row vector via newaxis
print(x[np.newaxis, :])
```

```
[[1 2 3]]
```

Some other similar operations are here.

```
# column vector via reshape
x.reshape((3, 1))
```

```
array([[1],
       [2],
       [3]])
```

```
# column vector via newaxis
x[:, np.newaxis]
```

```
array([[1],
       [2],
       [3]])
```

1.2.5 Array Concatenation and Splitting

All of the preceding routines worked on single arrays. It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays. We'll take a look at those operations here.

1.2.5.1 Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished using the routines `np.concatenate`, `np.vstack`, and `np.hstack`. `np.concatenate` takes a tuple or list of arrays as its first argument, as we can see here:

```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

```
array([1, 2, 3, 3, 2, 1])
```

Another example is shown here:

```
np.concatenate([y, y, y])
```

```
array([3, 2, 1, 3, 2, 1, 3, 2, 1])
```

It can also be used for two-dimensional arrays:

```
grid1 = np.array([[1, 2, 3],
                  [4, 5, 6]])
grid2=np.array([[5,5,5],[7,7,7]])
# concatenate along the first axis
nm=np.concatenate([grid1, grid2],axis=0)
nm.shape
print(nm)
```

```
[[1 2 3]
 [4 5 6]
 [5 5 5]
 [7 7 7]]
```

Row-wise concatenation is shown below.

```
# concatenate along the second axis (horizontal) (zero-indexed)
np.concatenate([grid1, grid2], axis=1)
```

```
array([[1, 2, 3, 5, 5, 5],
       [4, 5, 6, 7, 7, 7]])
```

For working with arrays of mixed dimensions, it can be clearer to use the `np.vstack` (vertical stack) and `np.hstack` (horizontal stack) functions:

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

# vertically stack the arrays
grid
```

```
array([[9, 8, 7],
       [6, 5, 4]])
```

Now the new vector `x` has the same number of columns of `grid`. So we can only vertically stack it `grid`. For this the numpy function `vstack` will be used as follows.

```
grid2=np.vstack([grid,x])
print(grid2)
```

```
[[9 8 7]
 [6 5 4]
 [1 2 3]]
```

Similarly the horizontal stacking can be shown as follows.

```
# horizontally stack the arrays
y = np.array([[99],
              [99], [3]])
np.hstack([grid2, y])
```

```
array([[ 9,  8,  7, 99],
       [ 6,  5,  4, 99],
       [ 1,  2,  3,  3]])
```

1.2.5.2 Splitting of arrays

The opposite of concatenation is splitting, which is implemented by the functions `np.split`, `np.hsplit`, and `np.vsplit`. For each of these, we can pass a list of indices giving the split points:

Let's begin with one dimensional arrays. First we split this array at specified locations and save it into sub arrays.

```
x = [1, 2, 3, 99, 99, 3, 2, 1]
```

Now split the list into two sub lists at index 2

```
x1,x2=np.split(x,[2])
```

Now see the sub-arrays:

```
print("the first array is:", x1)
print("the second array is:", x2)
```

```
the first array is: [1 2]
the second array is: [ 3 99 99  3  2  1]
```

More sub arrays can be created by passing the splitting locations as a list as follows.

```
x1,x2,x3=np.split(x,[2,4])
print(x1,"\n",x2,'\n',x3)
```

```
[1 2]
[ 3 99]
[99  3  2  1]
```

Note

Notice that N split-points, leads to $N + 1$ subarrays. The related functions `np.hsplit` and `np.vsplit` are similar:

Now use the `vsplit` and `hsplit` functions on multi dimensional arrays.

```
grid = np.arange(16).reshape((4, 4))
grid
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

```
# vsplit
upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
```

```
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

```
#hsplit
left, right = np.hsplit(grid, [2])
print("Left array:\n",left,"\n Right array:\n",right)
```

```
Left array:
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
Right array:
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

1.2.6 Review Questions

Short Answer Questions (SAQ)

Q1: What is the main purpose of the NumPy library in Python?

Ans: The main purpose of NumPy is to provide support for large, multi-dimensional arrays

and matrices, along with a collection of mathematical functions to perform operations on these arrays efficiently.

Q2: How can a 1D array be created in NumPy?

Ans: A 1D array can be created using `np.array()` function, like:

```
np.array([1, 2, 3])
```

Q3: How do you access the shape of a NumPy array?

Ans: You can access the shape of a NumPy array using the `.shape` attribute. For example, `array.shape` gives the dimensions of the array.

Q4: What does the `np.reshape()` function do?

Ans: The `np.reshape()` function reshapes an array to a new shape without changing its data.

Q5: Explain the difference between `vstack()` and `hstack()` in NumPy.

Ans: `vstack()` vertically stacks arrays (along rows), while `hstack()` horizontally stacks arrays (along columns).

Q6: How does NumPy handle array slicing?

Ans: Array slicing in NumPy is done by specifying the start, stop, and step index like `array[start:stop:step]`, which returns a portion of the array.

Q7: What is the difference between the `np.zeros()` and `np.ones()` functions?

Ans: `np.zeros()` creates an array filled with zeros, while `np.ones()` creates an array filled with ones.

Q8: What is array broadcasting in NumPy?

Ans: Broadcasting in NumPy allows arrays of different shapes to be used in arithmetic operations by stretching the smaller array to match the shape of the larger array.

Q9: How can you stack arrays along a new axis in NumPy? **Ans:** You can use `np.stack()` to join arrays along a new axis.

Q10: How do you generate a random integer array using NumPy?

Ans: You can generate a random integer array using `np.random.randint(low, high, size)`.

**Long
An-
swer
Ques-
tions
(LAQ)**

title:
Lin-
ear
Alge-
bra
for
Ad-
vanced
Ap-
plica-
tions
exe-
cute:
en-
abled:
true
jupyter:
python3

““““

1.3 Introduction

Matrix decomposition plays a pivotal role in computational linear algebra, forming the backbone of numerous modern applications in fields such as data science, machine learning, computer vision, and signal processing. The core idea behind matrix decomposition is to break down complex matrices into simpler, structured components that allow for more efficient computation. Techniques such as LU, QR, Singular Value Decomposition (SVD), and Eigenvalue decompositions not only reduce computational complexity but also provide deep insights into the geometry and structure of data. These methods are essential in solving systems of linear equations, performing dimensionality reduction, and extracting meaningful features from data. For instance, LU decomposition is widely used to solve large linear systems, while QR decomposition plays a key role in solving least squares problems—a fundamental task in machine learning models.

In emerging fields like big data analytics and artificial intelligence, matrix decomposition techniques are indispensable for processing and analyzing high-dimensional datasets. SVD and Principal Component Analysis (PCA), for example, are extensively used for data compression and noise reduction, making machine learning algorithms more efficient by reducing the number of variables while retaining key information. Additionally, sparse matrix decompositions allow for the handling of enormous datasets where most entries are zero, optimizing memory usage and computation time. As data science and machine learning continue to evolve, mastering these matrix decomposition techniques provides not only a computational advantage but also deeper insights into the structure and relationships within data, enhancing the performance of algorithms in real-world applications.

1.4 LU Decomposition

LU decomposition is a powerful tool in linear algebra that elegantly unravels the complexity of solving systems of linear equations. At its core, LU decomposition expresses a matrix A as the product of two distinct matrices: L (a lower triangular matrix with ones on the diagonal) and U (an upper triangular matrix). This decomposition transforms the problem of solving $Ax = b$ into a two-step process: first, solving $Ly = b$ for y , followed by $Ux = y$ for x . This systematic approach not only simplifies computations but also provides insightful perspectives on the relationships between the equations involved.

The magic of LU decomposition lies in its utilization of elementary transformations—operations that allow us to manipulate the rows of a matrix to achieve a row-reduced echelon form. These transformations include row swaps, scaling, and adding multiples of one row to another. By applying these operations, we can gradually transform the original matrix A into the upper triangular matrix U , while simultaneously capturing the essence of these transformations in the lower triangular matrix L . This interplay of L and U not only enhances computational efficiency but also unveils the deeper structural relationships within the matrix.

Moreover, the beauty of matrix multiplication shines through in LU decomposition. The product $A = LU$ showcases how two simpler matrices can combine to reconstruct a more complex one, demonstrating the power of linear combinations in solving equations. As we delve into LU decomposition, we embark on a journey that highlights the synergy between algebraic manipulation and geometric interpretation, empowering us to tackle intricate problems with grace and precision. Given a square matrix A , the LU decomposition expresses A as a product of a lower triangular matrix L and an upper triangular matrix U :

$$A = LU$$

Where: - L is a lower triangular matrix with 1's on the diagonal and other elements like $l_{21}, l_{31}, l_{32}, \dots$, - U is an upper triangular matrix with elements $u_{11}, u_{12}, u_{13}, u_{22}, u_{23}, u_{33}, \dots$

1.4.1 Step-by-Step Procedure

Let's assume A is a 3×3 matrix for simplicity:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

We need to find matrices L and U , where:

$$\begin{aligned} \bullet \quad L &= \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix} \\ \bullet \quad U &= \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix} \end{aligned}$$

The product of L and U gives:

$$LU = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix} = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ l_{21}u_{11} & l_{21}u_{12} + u_{22} & l_{21}u_{13} + u_{23} \\ l_{31}u_{11} & l_{31}u_{12} + l_{32}u_{22} & l_{31}u_{13} + l_{32}u_{23} + u_{33} \end{pmatrix}$$

By equating this with A , we can set up a system of equations to solve for l_{ij} and u_{ij} .

Step 1: Solve for u_{11}, u_{12}, u_{13}

From the first row of $A = LU$, we have:

$$u_{11} = a_{11}$$

$$u_{12} = a_{12}$$

$$u_{13} = a_{13}$$

Step 2: Solve for l_{21} and u_{22}, u_{23}

From the second row, we get:

$$l_{21}u_{11} = a_{21} \quad \Rightarrow \quad l_{21} = \frac{a_{21}}{u_{11}}$$

$$l_{21}u_{12} + u_{22} = a_{22} \quad \Rightarrow \quad u_{22} = a_{22} - l_{21}u_{12}$$

$$l_{21}u_{13} + u_{23} = a_{23} \quad \Rightarrow \quad u_{23} = a_{23} - l_{21}u_{13}$$

Step 3: Solve for l_{31}, l_{32} and u_{33}

From the third row, we get:

$$l_{31}u_{11} = a_{31} \Rightarrow l_{31} = \frac{a_{31}}{u_{11}}$$

$$l_{31}u_{12} + l_{32}u_{22} = a_{32} \Rightarrow l_{32} = \frac{a_{32} - l_{31}u_{12}}{u_{22}}$$

$$l_{31}u_{13} + l_{32}u_{23} + u_{33} = a_{33} \Rightarrow u_{33} = a_{33} - l_{31}u_{13} - l_{32}u_{23}$$

Final Result

Thus, the LU decomposition is given by the matrices: - $L = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix}$ - $U = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}$

Where: - $u_{11} = a_{11}, u_{12} = a_{12}, u_{13} = a_{13} - l_{21}u_{11}, u_{22} = a_{22} - l_{21}u_{12}, u_{23} = a_{23} - l_{21}u_{13} - l_{31}u_{11}, l_{32} = \frac{a_{32} - l_{31}u_{12}}{u_{22}}, u_{33} = a_{33} - l_{31}u_{13} - l_{32}u_{23}$

1.4.2 Example

Let's decompose the following matrix:

$$A = \begin{pmatrix} 4 & 3 & 2 \\ 6 & 3 & 1 \\ 2 & 1 & 3 \end{pmatrix}$$

Following the steps outlined above:

- $u_{11} = 4, u_{12} = 3, u_{13} = 2$
- $l_{21} = \frac{6}{4} = 1.5$, so:
 - $u_{22} = 3 - 1.5 \times 3 = -1.5$
 - $u_{23} = 1 - 1.5 \times 2 = -2$
- $l_{31} = \frac{2}{4} = 0.5$, so:
 - $l_{32} = \frac{1 - 0.5 \times 3}{-1.5} = 0.67$
 - $u_{33} = 3 - 0.5 \times 2 - 0.67 \times (-2) = 2.67$

Thus, the decomposition is: - $L = \begin{pmatrix} 1 & 0 & 0 \\ 1.5 & 1 & 0 \\ 0.5 & 0.67 & 1 \end{pmatrix}$ - $U = \begin{pmatrix} 4 & 3 & 2 \\ 0 & -1.5 & -2 \\ 0 & 0 & 2.67 \end{pmatrix}$

1.4.3 Python Implementation

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

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

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

# Print the results
print("L = \n", L)
print("U = \n", U)
```

```
L =
[[1.  0.  0. ]
 [0.66666667 1.  0. ]
 [0.33333333 0.  1. ]]
U =
[[6.  3.  1. ]
 [0.  1.  1.33333333]
 [0.  0.  2.66666667]]
```

Note

Since there are many row transformations that reduce a given matrix into row echelon form. So the LU decomposition is not unique.

1.4.4 LU Decomposition Practice Problems with Solutions

Problem 1: Decompose the matrix

$$A = \begin{pmatrix} 4 & 3 \\ 6 & 3 \end{pmatrix}$$

into the product of a lower triangular matrix L and an upper triangular matrix U .

Solution:

Let

$$L = \begin{pmatrix} 1 & 0 \\ l_{21} & 1 \end{pmatrix}, \quad U = \begin{pmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{pmatrix}.$$

We have:

1. From the first row: $u_{11} = 4$ and $u_{12} = 3$.
2. From the second row: $6 = l_{21} \cdot 4$ gives $l_{21} = \frac{6}{4} = 1.5$.
3. Finally, $3 = 1.5 \cdot 3 + u_{22}$ gives $u_{22} = 3 - 4.5 = -1.5$.

Thus, we have:

$$L = \begin{pmatrix} 1 & 0 \\ 1.5 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 4 & 3 \\ 0 & -1.5 \end{pmatrix}.$$

Problem 2: Given the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 5 & 8 \\ 4 & 5 & 6 \end{pmatrix},$$

perform LU decomposition to find matrices L and U .

Solution:

Let

$$L = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix}, \quad U = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}.$$

We have:

1. From Row 1: $u_{11} = 1, u_{12} = 2, u_{13} = 3$.
2. From Row 2: $2 = l_{21} \cdot 1$ gives $l_{21} = 2$.
 - For Row 2: $5 = l_{21} \cdot 2 + u_{22}$ gives $5 = 4 + u_{22} \Rightarrow u_{22} = 1$.
 - $8 = l_{21} \cdot 3 + u_{23} \Rightarrow 8 = 6 + u_{23} \Rightarrow u_{23} = 2$.
3. From Row 3: $4 = l_{31} \cdot 1 \Rightarrow l_{31} = 4$.
 - $5 = l_{31} \cdot 2 + l_{32} \cdot 1 \Rightarrow 5 = 8 + l_{32} \Rightarrow l_{32} = -3$.
 - Finally, $6 = l_{31} \cdot 3 + l_{32} \cdot 2 + u_{33} \Rightarrow 6 = 12 - 6 + u_{33} \Rightarrow u_{33} = 0$.

Thus,

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & -3 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{pmatrix}.$$

Problem 3: Perform LU decomposition of the matrix

$$A = \begin{pmatrix} 2 & 1 & 1 \\ 4 & -6 & 0 \\ -2 & 7 & 2 \end{pmatrix},$$

and verify the decomposition by checking $A = LU$.

Solution:

Let

$$L = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix}, \quad U = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}.$$

We have:

1. From Row 1: $u_{11} = 2, u_{12} = 1, u_{13} = 1$.
2. From Row 2: $4 = l_{21} \cdot 2 \Rightarrow l_{21} = 2$.
 - $-6 = 2 \cdot 1 + u_{22} \Rightarrow u_{22} = -8$.
 - $0 = 2 \cdot 1 + u_{23} \Rightarrow u_{23} = -2$.
3. From Row 3: $-2 = l_{31} \cdot 2 \Rightarrow l_{31} = -1$.
 - $7 = -1 \cdot 1 + l_{32} \cdot -8 \Rightarrow 7 = -1 - 8l_{32} \Rightarrow l_{32} = -1$.
 - Finally, $2 = -1 \cdot 1 + -1 \cdot -2 + u_{33} \Rightarrow 2 = 1 + u_{33} \Rightarrow u_{33} = 1$.

Thus,

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -1 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 2 & 1 & 1 \\ 0 & -8 & -2 \\ 0 & 0 & 1 \end{pmatrix}.$$

Problem 4: For the matrix

$$A = \begin{pmatrix} 3 & 1 & 6 \\ 2 & 1 & 1 \\ 1 & 2 & 2 \end{pmatrix},$$

find the LU decomposition and use it to solve the system $Ax = b$ where $b = \begin{pmatrix} 9 \\ 5 \\ 4 \end{pmatrix}$.

Solution:

Let

$$L = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix}, \quad U = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}.$$

We have:

1. From Row 1: $u_{11} = 3, u_{12} = 1, u_{13} = 6$.
2. From Row 2: $2 = l_{21} \cdot 3 \Rightarrow l_{21} = \frac{2}{3}$.
 - $1 = \frac{2}{3} \cdot 1 + u_{22} \Rightarrow 1 = \frac{2}{3} + u_{22} \Rightarrow u_{22} = \frac{1}{3}$.
 - $1 = \frac{2}{3} \cdot 6 + u_{23} \Rightarrow 1 = 4 + u_{23} \Rightarrow u_{23} = -3$.
3. From Row 3: $1 = l_{31} \cdot 3 \Rightarrow l_{31} = \frac{1}{3}$.
 - $2 = \frac{1}{3} \cdot 1 + l_{32} \cdot \frac{1}{3} \Rightarrow 2 = \frac{1}{3} + \frac{1}{3}l_{32} \Rightarrow l_{32} = 6$.
 - Finally, $2 = \frac{1}{3} \cdot 6 + 6 \cdot -3 + u_{33} \Rightarrow 2 = 2 - 18 + u_{33} \Rightarrow u_{33} = 18$.

Thus,

$$L = \begin{pmatrix} 1 & 0 & 0 \\ \frac{2}{3} & 1 & 0 \\ \frac{1}{3} & 6 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 3 & 1 & 6 \\ 0 & \frac{1}{3} & -3 \\ 0 & 0 & 18 \end{pmatrix}.$$

Now, to solve $Ax = b$, we first solve $Ly = b$:

$$\begin{pmatrix} 1 & 0 & 0 \\ \frac{2}{3} & 1 & 0 \\ \frac{1}{3} & 6 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 9 \\ 5 \\ 4 \end{pmatrix}$$

Solving this gives: 1. $y_1 = 9$ 2. $\frac{2}{3} \cdot 9 + y_2 = 5 \Rightarrow 6 + y_2 = 5 \Rightarrow y_2 = -1$ 3. $\frac{1}{3} \cdot 9 + 6 \cdot -1 + y_3 = 4 \Rightarrow 3 - 6 + y_3 = 4 \Rightarrow y_3 = 7$

Next, solve $Ux = y$:

$$\begin{pmatrix} 3 & 1 & 6 \\ 0 & \frac{1}{3} & -3 \\ 0 & 0 & 18 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 9 \\ -1 \\ 7 \end{pmatrix}$$

1. From Row 3: $18x_3 = 7 \Rightarrow x_3 = \frac{7}{18}$
2. From Row 2: $\frac{1}{3}x_2 - 3x_3 = -1 \Rightarrow \frac{1}{3}x_2 - \frac{21}{18} = -1 \Rightarrow \frac{1}{3}x_2 = -\frac{18}{18} + \frac{21}{18} = \frac{3}{18} \Rightarrow x_2 = \frac{1}{3}$
3. From Row 1: $3x_1 + x_2 + 6x_3 = 9 \Rightarrow 3x_1 + \frac{1}{3} + \frac{42}{18} = 9 \Rightarrow 3x_1 + \frac{1}{3} + \frac{7}{3} = 9 \Rightarrow 3x_1 = 9 - \frac{8}{3} = \frac{27-8}{3} = \frac{19}{3} \Rightarrow x_1 = \frac{19}{9}$

Thus, the solution to $Ax = b$ is

$$x = \begin{pmatrix} \frac{19}{9} \\ \frac{1}{3} \\ \frac{7}{18} \end{pmatrix}.$$

1.5 LU Decomposition Practice Problems

Problem 1: Decompose the matrix

$$A = \begin{pmatrix} 4 & 3 \\ 6 & 3 \end{pmatrix}$$

into the product of a lower triangular matrix L and an upper triangular matrix U .

Problem 2: Given the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 5 & 8 \\ 4 & 5 & 6 \end{pmatrix},$$

perform LU decomposition to find matrices L and U .

Problem 3: Perform LU decomposition of the matrix

$$A = \begin{pmatrix} 2 & 1 & 1 \\ 4 & -6 & 0 \\ -2 & 7 & 2 \end{pmatrix},$$

and verify the decomposition by checking $A = LU$.

Problem 4: For the matrix

$$A = \begin{pmatrix} 3 & 1 & 6 \\ 2 & 1 & 1 \\ 1 & 2 & 2 \end{pmatrix},$$

find the LU decomposition and use it to solve the system $Ax = b$ where $b = \begin{pmatrix} 9 \\ 5 \\ 4 \end{pmatrix}$.

Problem 5: Decompose the matrix

$$A = \begin{pmatrix} 1 & 3 & 1 \\ 2 & 6 & 1 \\ 1 & 1 & 4 \end{pmatrix}$$

into L and U , and solve the system $Ax = \begin{pmatrix} 5 \\ 9 \\ 6 \end{pmatrix}$.

Problem 6: Given the matrix

$$A = \begin{pmatrix} 7 & 3 \\ 2 & 5 \end{pmatrix},$$

perform LU decomposition and use the result to solve $Ax = b$ for $b = \begin{pmatrix} 10 \\ 7 \end{pmatrix}$.

Problem 7: Find the LU decomposition of the matrix

$$A = \begin{pmatrix} 2 & -1 & 1 \\ -2 & 2 & -1 \\ 4 & -1 & 3 \end{pmatrix},$$

and use it to solve $Ax = b$ where $b = \begin{pmatrix} 1 \\ -1 \\ 7 \end{pmatrix}$.

Problem 8: Perform LU decomposition of the matrix

$$A = \begin{pmatrix} 5 & 2 & 1 \\ 10 & 4 & 3 \\ 15 & 8 & 6 \end{pmatrix}.$$

Problem 9: Use LU decomposition to find the solution to the system $Ax = b$ where

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 3 & 5 \\ 4 & 6 & 8 \end{pmatrix}, \quad b = \begin{pmatrix} 6 \\ 15 \\ 30 \end{pmatrix}.$$

Problem 10: Decompose the matrix

$$A = \begin{pmatrix} 6 & -2 & 2 \\ 12 & -8 & 6 \\ -6 & 3 & -3 \end{pmatrix}$$

into L and U , and verify that $A = LU$.

1.6 Matrix Approach to Create LU Decomposition

LU decomposition can be performed using *elementary matrix operations*. In this method, we iteratively apply elementary matrices to reduce the given matrix A into an upper triangular matrix U , while keeping track of the transformations to form the lower triangular matrix L .

The LU decomposition can be written as:

$$A = LU$$

where: - L is the product of the inverses of the elementary matrices. - U is the upper triangular matrix obtained after applying the row operations.

Example: LU Decomposition of a 3x3 Matrix

Given the matrix:

$$A = \begin{pmatrix} 2 & 1 & 1 \\ 4 & -6 & 0 \\ -2 & 7 & 2 \end{pmatrix}$$

We will decompose A into L and U using elementary row operations.

Step 1: Applying Elementary Matrices

We want to perform row operations to reduce A into upper triangular form.

Step 1.1: Eliminate the a_{21} entry (below the pivot in column 1)

To eliminate the 4 in position a_{21} , perform the operation:

$$R_2 \rightarrow R_2 - 2R_1$$

This corresponds to multiplying A by the elementary matrix:

$$E_1 = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

After this row operation, the matrix becomes:

$$E_1 A = \begin{pmatrix} 2 & 1 & 1 \\ 0 & -8 & -2 \\ -2 & 7 & 2 \end{pmatrix}$$

Step 1.2: Eliminate the a_{31} entry

To eliminate the -2 in position a_{31} , perform the operation:

$$R_3 \rightarrow R_3 + R_1$$

This corresponds to multiplying the matrix by another elementary matrix:

$$E_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

Now, the matrix becomes:

$$E_2 E_1 A = \begin{pmatrix} 2 & 1 & 1 \\ 0 & -8 & -2 \\ 0 & 8 & 3 \end{pmatrix}$$

Step 1.3: Eliminate the a_{32} entry

Finally, to eliminate the 8 in position a_{32} , perform the operation:

$$R_3 \rightarrow R_3 + R_2$$

This corresponds to multiplying the matrix by the third elementary matrix:

$$E_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

After applying this operation, the matrix becomes:

$$E_3 E_2 E_1 A = \begin{pmatrix} 2 & 1 & 1 \\ 0 & -8 & -2 \\ 0 & 0 & 1 \end{pmatrix}$$

This is the upper triangular matrix U .

Step 2: Construct the Lower Triangular Matrix L

The lower triangular matrix L is formed by taking the inverses of the elementary matrices E_1, E_2, E_3 . Each inverse corresponds to the inverse of the row operations we applied.

- E_1^{-1} corresponds to adding back $2R_1$ to R_2 , so:

$$E_1^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- E_2^{-1} corresponds to subtracting R_1 from R_3 , so:

$$E_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix}$$

- E_3^{-1} corresponds to subtracting R_2 from R_3 , so:

$$E_3^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix}$$

Now, the lower triangular matrix L is obtained by multiplying these inverses in reverse order:

$$L = E_3^{-1}E_2^{-1}E_1^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -1 & 1 \end{pmatrix}$$

Thus, the LU decomposition of A is:

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -1 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 2 & 1 & 1 \\ 0 & -8 & -2 \\ 0 & 0 & 1 \end{pmatrix}$$

Verification

Now, we check if $A = LU$.

Multiply L and U :

```
import numpy as np

L = np.array([[1, 0, 0],
              [2, 1, 0],
              [-1, -1, 1]])

U = np.array([[2, 1, 1],
              [0, -8, -2],
              [0, 0, 1]])

A = L @ U
A
```

```
array([[ 2,  1,  1],
       [ 4, -6,  0],
       [-2,  7,  2]])
```

2 Spectral Decomposition

2.1 Background

Imagine encountering a low-resolution image of a familiar scene. The human brain excels at recognizing familiar objects by relying on essential features, often extracting the most significant details while discarding the less important information. This cognitive process mirrors the power of eigenvalue decomposition, where eigenvectors represent the “nectar” of a matrix, capturing its most important characteristics.

As an example, try to identify this image. If you can do it, then your brain know this place!

Reconstructed Image with LA



Before proceeding further just compare the size of its' original clean image and the low-quality image shown in Figure

```
Original image size: 985.69 KB  
Reconstructed image size: 1.12 KB
```

The reconstructed image is just 0.2% of the original in size! This is the core principle of optimizing image storage of CCTV system. This resizing can be done and execute with optimal scaling with the help of Linear Algebra. This module mainly focuses on such engineering applications.

2.2 Introduction

Spectral decomposition, also known as eigenvalue decomposition, is a powerful tool in computational linear algebra that breaks down a matrix into its eigenvalues and eigenvectors. This technique allows matrices to be represented in terms of their fundamental components, making it easier to analyze and manipulate them. It is especially useful for symmetric matrices, which are common in various applications. Spectral decomposition facilitates solving systems of equations, optimizing functions, and performing transformations in a simplified, structured manner, as it allows operations to be performed on the eigenvalues, which often leads to more efficient computations.

The importance of spectral decomposition extends across a wide range of fields, including computer science, engineering, and data science. In machine learning, for instance, it forms the backbone of algorithms like Principal Component Analysis (PCA), which is used for dimensionality reduction. It also plays a vital role in numerical stability when dealing with large matrices and is central to many optimization problems, such as those found in machine learning and physics. Spectral decomposition not only provides a deeper understanding of the properties of matrices but also offers practical benefits in improving the efficiency and accuracy of numerical algorithms.

2.3 Spectral Decomposition: Detailed Concepts

2.3.1 Eigenvalues and Eigenvectors

The core idea behind spectral decomposition is that it expresses a matrix in terms of its eigenvalues and eigenvectors. For a square matrix $A \in \mathbb{R}^{n \times n}$, an eigenvalue $\lambda \in \mathbb{R}$ and an eigenvector $v \in \mathbb{R}^n$ satisfy the following equation:

$$Av = \lambda v$$

This implies that when the matrix A acts on the vector v , it only scales the vector by λ , but does not change its direction. The eigenvector v represents the direction of this scaling, while the eigenvalue λ represents the magnitude of the scaling.

i Properties of Eigen values

- If λ is an eigenvalue of A , then it satisfies the characteristic polynomial:

$$p(\lambda) = \det(A - \lambda I) = 0.$$

- The sum of the eigenvalues (counted with algebraic multiplicity) is equal to the trace of the matrix:

$$\sum_{i=1}^n \lambda_i = \text{trace}(A).$$

- The product of the eigenvalues (counted with algebraic multiplicity) is equal to the determinant of the matrix:

$$\prod_{i=1}^n \lambda_i = \det(A).$$

- If A is symmetric, then:
 - All eigenvalues λ are real.
 - If λ_i and λ_j are distinct eigenvalues, then their corresponding eigenvectors \mathbf{v}_i and \mathbf{v}_j satisfy:

$$\mathbf{v}_i^T \mathbf{v}_j = 0.$$

- If A is a scalar multiple of k , then:

$$\lambda_i \text{ of } kA = k \cdot \lambda_i \text{ of } A.$$

- If A is invertible, then:

$$\lambda_i \text{ of } A^{-1} = \frac{1}{\lambda_i \text{ of } A}.$$

- If A and B are similar, then:

$$B = P^{-1}AP \implies \lambda_i \text{ of } B = \lambda_i \text{ of } A.$$

- If λ is an eigenvalue, it has:
 - **Algebraic Multiplicity:** The number of times λ appears as a root of $p(\lambda)$.
 - **Geometric Multiplicity:** The dimension of the eigenspace $E_\lambda = \{\mathbf{v} : A\mathbf{v} = \lambda\mathbf{v}\}$.
- If A is symmetric and all eigenvalues λ are positive, then A is positive definite:

$$\lambda_i > 0 \implies A \text{ is positive definite.}$$

- A square matrix A has an eigenvalue $\lambda = 0$ if and only if A is singular:

$$\det(A) = 0 \iff \lambda = 0.$$

! Eigen Vectors

Eigen vectors are the non-trivial solutions of $\det(A - \lambda I) = 0$ for distinct λ .

i Properties of Eigen vectors

- If \mathbf{v} is an eigenvector of a square matrix A corresponding to the eigenvalue λ , then:

$$A\mathbf{v} = \lambda\mathbf{v}.$$

- Eigenvectors corresponding to distinct eigenvalues are linearly independent. If λ_1 and λ_2 are distinct eigenvalues of A , with corresponding eigenvectors \mathbf{v}_1 and \mathbf{v}_2 , then:

$$c_1\mathbf{v}_1 + c_2\mathbf{v}_2 = \mathbf{0} \implies c_1 = 0 \text{ and } c_2 = 0.$$

- If \mathbf{v} is an eigenvector corresponding to the eigenvalue λ , then any non-zero scalar multiple of \mathbf{v} is also an eigenvector corresponding to λ :

If \mathbf{v} is an eigenvector, then $c\mathbf{v}$ is an eigenvector for any non-zero scalar c .

- The eigenspace E_λ associated with an eigenvalue λ is defined as:

$$E_\lambda = \{\mathbf{v} : A\mathbf{v} = \lambda\mathbf{v}\} = \text{Null}(A - \lambda I).$$

- The dimension of the eigenspace E_λ is equal to the geometric multiplicity of the eigenvalue λ .
- If A is a symmetric matrix, then eigenvectors corresponding to distinct eigenvalues are orthogonal:

$$\mathbf{v}_i^T \mathbf{v}_j = 0 \text{ for distinct eigenvalues } \lambda_i \text{ and } \lambda_j.$$

- For any square matrix A , if $\lambda = 0$ is an eigenvalue, the eigenvectors corresponding to this eigenvalue form the null space of A :

$$E_0 = \{\mathbf{v} : A\mathbf{v} = \mathbf{0}\} = \text{Null}(A).$$

- If A is invertible, then A has no eigenvalue equal to zero, meaning all eigenvectors correspond to non-zero eigenvalues.
- For A as a scalar multiple of k :

$$A\mathbf{v} = k\lambda\mathbf{v} \text{ for eigenvalue } \lambda.$$

2.3.2 Eigenvalue Decomposition (Spectral Decomposition)

For matrices that are diagonalizable (including symmetric matrices), spectral decomposition expresses the matrix as a combination of its eigenvalues and eigenvectors. Specifically, for a matrix A , spectral decomposition is represented as:

$$A = V\Lambda V^{-1}$$

where: - V is the matrix of eigenvectors of A , - Λ is a diagonal matrix of eigenvalues of A , - V^{-1} is the inverse of the matrix of eigenvectors (if V is invertible).

For symmetric matrices A , the decomposition becomes simpler:

$$A = Q\Lambda Q^T$$

Here, Q is an orthogonal matrix of eigenvectors (i.e., $Q^T Q = I$), and Λ is a diagonal matrix of eigenvalues.

2.3.3 Geometric Interpretation

Eigenvalues and eigenvectors provide insights into the geometry of linear transformations represented by matrices. Eigenvectors represent directions that remain invariant under the transformation, while eigenvalues indicate how these directions are stretched or compressed.

For example, in the case of a transformation matrix that scales or rotates data points, eigenvalues show the magnitude of scaling along the principal axes (directions defined by eigenvectors).

2.3.4 Importance of Diagonalization

The key advantage of spectral decomposition is that it simplifies matrix operations. When a matrix is diagonalized as $A = Q\Lambda Q^\top$, any function of the matrix A (such as powers, exponentials, or inverses) can be easily computed by operating on the diagonal matrix Λ . For example:

$$A^k = Q\Lambda^k Q^\top$$

Since Λ is diagonal, raising Λ to any power k is straightforward, involving only raising each eigenvalue to the power k .

2.3.5 Properties of Symmetric Matrices

Spectral decomposition applies particularly well to symmetric matrices, which satisfy $A = A^\top$. Symmetric matrices have the following key properties:

- **Real eigenvalues:** The eigenvalues of a symmetric matrix are always real numbers.
- **Orthogonal eigenvectors:** The eigenvectors corresponding to distinct eigenvalues of a symmetric matrix are orthogonal to each other.
- **Diagonalizability:** Every symmetric matrix can be diagonalized by an orthogonal matrix.

These properties make symmetric matrices highly desirable in computational applications.

2.4 Mathematical Requirements for Spectral Decomposition

2.4.1 Determining Eigenvalues and Eigenvectors

The eigenvalues of a matrix A are the solutions to the characteristic equation:

$$\det(A - \lambda I) = 0$$

Here, I is the identity matrix, and λ represents the eigenvalues. Solving this polynomial equation provides the eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_n$. Once the eigenvalues are determined, the eigenvectors can be computed by solving the equation $(A - \lambda I)v = 0$ for each eigenvalue.

2.4.2 Characteristic Polynomial of 2×2 Matrices

For a 2×2 matrix:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

the characteristic polynomial is derived from the determinant of $A - \lambda I$, where I is the identity matrix:

$$\det(A - \lambda I) = 0$$

This leads to:

$$\det \begin{pmatrix} a - \lambda & b \\ c & d - \lambda \end{pmatrix} = (a - \lambda)(d - \lambda) - bc = 0$$

Short-cut Method: The characteristic polynomial can be simplified to:

$$\lambda^2 - (a + d)\lambda + (ad - bc) = 0$$

This polynomial can be solved using the quadratic formula:

$$\lambda = \frac{(a + d) \pm \sqrt{(a + d)^2 - 4(ad - bc)}}{2}$$

! Shortcut to write Characteristic polynomial of a 2×2 matrix

If $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, then the characteristic polynomial is

$$\lambda^2 - (\text{Trace}(A))\lambda + \det(A) = 0$$

Eigen vectors can be found by using the formula:

$$EV(\lambda = \lambda_1) = \begin{bmatrix} \lambda_1 - d \\ c \end{bmatrix}$$

2.4.3 Problems

Example 1: Find Eigenvalues and Eigenvectors of the matrix,

$$A = \begin{pmatrix} 3 & 2 \\ 4 & 1 \end{pmatrix}$$

Solution:

The characteristic equation is given by

$$\det(A - \lambda I) = 0$$

$$\begin{aligned} \lambda^2 - 4\lambda - 5 &= 0 \\ (\lambda - 5)(\lambda + 1) &= 0 \end{aligned}$$

Hence the eigen values are $\lambda_1 = 5$, $\lambda_2 = -1$.

So the eigen vectors are:

$$\begin{aligned} EV(\lambda = \lambda_1) &= \begin{bmatrix} \lambda_1 - d \\ c \end{bmatrix} \\ \therefore EV(\lambda = 5) &= \begin{bmatrix} 4 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ \therefore EV(\lambda = -1) &= \begin{bmatrix} -2 \\ 4 \end{bmatrix} = \begin{bmatrix} -1 \\ 2 \end{bmatrix} \end{aligned}$$

Problem 2: Calculate the eigenvalues and eigenvectors of the matrix: $A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$

Solution:

To find the eigenvalues and eigenvectors of a 2×2 matrix, we can use the shortcut formula for the characteristic polynomial:

$$\lambda^2 - \text{trace}(A)\lambda + \det(A) = 0,$$

where A is the matrix. Let's apply this to the matrix

$$A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}.$$

First, we calculate the trace and determinant of A :

- The trace is the sum of the diagonal elements:

$$\text{trace}(A) = 2 + 2 = 4.$$

- The determinant is calculated as follows:

$$\det(A) = (2)(2) - (1)(1) = 4 - 1 = 3.$$

Next, substituting the trace and determinant into the characteristic polynomial gives:

$$\lambda^2 - (4)\lambda + 3 = 0,$$

which simplifies to:

$$\lambda^2 - 4\lambda + 3 = 0.$$

We can factor this quadratic equation:

$$(\lambda - 1)(\lambda - 3) = 0.$$

Setting each factor to zero gives the eigenvalues:

$$\lambda_1 = 1, \quad \lambda_2 = 3.$$

To find the eigenvectors corresponding to each eigenvalue, we use the shortcut for the eigenvector of a 2×2 matrix $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$:

$$EV(\lambda) = \begin{pmatrix} \lambda - d \\ c \end{pmatrix}.$$

For the eigenvalue $\lambda_1 = 1$:

$$EV(1) = \begin{pmatrix} 1-2 \\ 1 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \end{pmatrix}.$$

This eigenvector can be simplified (up to a scalar multiple) to:

$$\mathbf{v}_1 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}.$$

For the eigenvalue $\lambda_2 = 3$:

$$EV(3) = \begin{pmatrix} 3-2 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

This eigenvector is already in a simple form:

$$\mathbf{v}_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Problem 3: For the matrix: $A = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$, find the eigenvalues and eigenvectors.

Solution:

We are given the matrix

$$A = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

and we aim to find its eigenvalues using the characteristic polynomial.

The shortcut formula for the characteristic polynomial of a 3×3 matrix is given by:

$$\lambda^3 - \text{tr}(A)\lambda^2 + (\text{sum of principal minors of } A)\lambda - \det(A) = 0.$$

The trace of a matrix is the sum of its diagonal elements. For matrix A , we have:

$$\text{tr}(A) = 1 + 1 + 1 = 3.$$

The principal minors are the determinants of the 2×2 submatrices obtained by deleting one row and one column of A .

The first minor is obtained by deleting the third row and third column:

$$\det \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix} = (1)(1) - (2)(0) = 1.$$

The second minor is obtained by deleting the second row and second column:

$$\det \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} = (1)(1) - (1)(1) = 0.$$

The third minor is obtained by deleting the first row and first column:

$$\det \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = (1)(1) - (0)(0) = 1.$$

Thus, the sum of the principal minors is:

$$1 + 0 + 1 = 2.$$

The determinant of A can be calculated using cofactor expansion along the first row:

$$\begin{aligned} \det(A) &= 1 \cdot \det \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - 2 \cdot \det \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} + 1 \cdot \det \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \\ &= 1 \cdot (1) - 2 \cdot (0) + 1 \cdot (-1) = 1 - 0 - 1 = 0. \end{aligned}$$

Now, we substitute these values into the characteristic polynomial formula:

$$\begin{aligned} \lambda^3 - \text{tr}(A)\lambda^2 + (\text{sum of principal minors})\lambda - \det(A) &= 0 \\ \lambda^3 - 3\lambda^2 + 2\lambda - 0 &= 0. \end{aligned}$$

We now solve the equation:

$$\lambda^3 - 3\lambda^2 + 2\lambda = 0.$$

Factoring out λ and apply factor theorem, we get:

$$\begin{aligned} \lambda(\lambda^2 - 3\lambda + 2) &= 0 \\ \lambda(\lambda - 2)(\lambda - 1) &= 0 \end{aligned}$$

This gives one eigenvalue:

$$\lambda_1 = 0; \quad \lambda_2 = 2; \quad \lambda_3 = 1$$

Now we find the eigenvectors corresponding to each eigenvalue.

For $\lambda_1 = 0$, solve $(A - 0I)\mathbf{v} = 0$:

$$\begin{pmatrix} 1 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

This gives the system:

$$x + 2y + z = 0, \quad y = 0, \quad x + z = 0.$$

Thus, $x = -z$, and the eigenvector is:

$$\mathbf{v}_1 = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}.$$

For $\lambda_2 = 2$, solve $(A - 2I)\mathbf{v} = 0$:

$$\begin{pmatrix} -1 & 2 & 1 \\ 0 & -1 & 0 \\ 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

This gives the system:

$$-x + 2y + z = 0, \quad -y = 0, \quad x - z = 0.$$

Thus, $x = z$, and the eigenvector is:

$$\mathbf{v}_2 = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}.$$

For $\lambda_3 = 1$, solve $(A - I)\mathbf{v} = 0$:

$$\begin{pmatrix} 0 & 2 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

This gives the system:

$$2y + z = 0, \quad x = 0.$$

Thus, $z = -2y$, and the eigenvector is:

$$\mathbf{v}_3 = \begin{pmatrix} 0 \\ 1 \\ -2 \end{pmatrix}.$$

Problem 3: If $A = \begin{bmatrix} 1 & 2 & 4 \\ 0 & 3 & 4 \\ 1 & -1 & -1 \end{bmatrix}$, compute the eigen values and eigen vectors and left eigen vectors of A .

Solution:

We are given the matrix

$$A = \begin{pmatrix} 1 & 2 & 4 \\ 0 & 3 & 4 \\ 1 & -1 & -1 \end{pmatrix}$$

and need to find its eigenvalues and eigenvectors.

The characteristic polynomial for a 3×3 matrix is given by:

$$\lambda^3 - \text{tr}(A)\lambda^2 + (\text{sum of principal minors})\lambda - \det(A) = 0.$$

The trace is the sum of the diagonal elements:

$$\text{tr}(A) = 1 + 3 + (-1) = 3.$$

We now compute the 2×2 principal minors:

- Minor by removing the third row and third column:

$$\det \begin{pmatrix} 1 & 2 \\ 0 & 3 \end{pmatrix} = (1)(3) - (2)(0) = 3.$$

- Minor by removing the second row and second column:

$$\det \begin{pmatrix} 1 & 4 \\ 1 & -1 \end{pmatrix} = (1)(-1) - (4)(1) = -1 - 4 = -5.$$

- Minor by removing the first row and first column:

$$\det \begin{pmatrix} 3 & 4 \\ -1 & -1 \end{pmatrix} = (3)(-1) - (4)(-1) = -3 + 4 = 1.$$

Thus, the sum of the principal minors is:

$$3 + (-5) + 1 = -1.$$

We calculate the determinant of A by cofactor expansion along the first row:

$$\det(A) = 1 \cdot \det \begin{pmatrix} 3 & 4 \\ -1 & -1 \end{pmatrix} - 2 \cdot \det \begin{pmatrix} 0 & 4 \\ 1 & -1 \end{pmatrix} + 4 \cdot \det \begin{pmatrix} 0 & 3 \\ 1 & -1 \end{pmatrix}.$$

The 2×2 determinants are:

$$\det \begin{pmatrix} 3 & 4 \\ -1 & -1 \end{pmatrix} = -3 + 4 = 1, \quad \det \begin{pmatrix} 0 & 4 \\ 1 & -1 \end{pmatrix} = -4,$$

$$\det \begin{pmatrix} 0 & 3 \\ 1 & -1 \end{pmatrix} = -3.$$

Thus:

$$\det(A) = 1 \cdot 1 - 2 \cdot (-4) + 4 \cdot (-3) = 1 + 8 - 12 = -3.$$

Substituting into the characteristic polynomial:

$$\lambda^3 - \operatorname{tr}(A)\lambda^2 + (\text{sum of principal minors})\lambda - \det(A) = 0,$$

we get:

$$\lambda^3 - 3\lambda^2 - \lambda + 3 = 0.$$

We now solve the cubic equation:

$$\begin{aligned} \lambda^3 - 3\lambda^2 - \lambda + 3 &= 0. \\ (\lambda - 1)(\lambda + 1)(\lambda - 3) &= 0 \end{aligned}$$

$$\lambda_1 = 1, \quad \lambda_2 = -1, \quad \lambda_3 = 3.$$

To find the eigenvector corresponding to $\lambda_1 = 3$, solve $(A - 3I)\mathbf{v} = 0$:

$$A - 3I = \begin{pmatrix} 1 & 2 & 4 \\ 0 & 3 & 4 \\ 1 & -1 & -1 \end{pmatrix} - 3 \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} -2 & 2 & 4 \\ 0 & 0 & 4 \\ 1 & -1 & -4 \end{pmatrix}.$$

Solving this system gives the eigenvector:

$$\mathbf{v}_1 = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}.$$

For $\lambda_2 = -1$, solve $(A + I)\mathbf{v} = 0$:

$$A + I = \begin{pmatrix} 1 & 2 & 4 \\ 0 & 3 & 4 \\ 1 & -1 & -1 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 2 & 4 \\ 0 & 4 & 4 \\ 1 & -1 & 0 \end{pmatrix}.$$

Note that the third row is depending on first and second rows. So by finding the cross product of first two rows,

$$\mathbf{v}_2 = \begin{pmatrix} -1 \\ -1 \\ 1 \end{pmatrix}.$$

For $\lambda_3 = 1$, solve $(A - I)\mathbf{v} = 0$:

$$A - I = \begin{pmatrix} 1 & 2 & 4 \\ 0 & 3 & 4 \\ 1 & -1 & -1 \end{pmatrix} - \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 2 & 4 \\ 0 & 2 & 4 \\ 1 & -1 & -2 \end{pmatrix}.$$

Note that the second row is same as first row. So by finding the cross product of first and third rows,

$$\mathbf{v}_3 = \begin{pmatrix} 0 \\ -2 \\ 1 \end{pmatrix}.$$

Thus, the eigenvalues of the matrix are:

$$\lambda_1 = 3, \quad \lambda_2 = -1, \quad \lambda_3 = 1$$

with corresponding eigenvectors $\mathbf{v}_1 = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$, $\mathbf{v}_2 = \begin{pmatrix} -1 \\ -1 \\ 1 \end{pmatrix}$, and $\mathbf{v}_3 = \begin{pmatrix} 0 \\ -2 \\ 1 \end{pmatrix}$.

Left eigen vectors of the matrix A are eigen vectors of A^T .

$$\text{Here } A^T = \begin{bmatrix} 1 & 0 & 1 \\ 2 & 3 & -1 \\ 4 & 4 & -1 \end{bmatrix}.$$

Since A and A^T have same eigen values, it is enough to find corresponding eigen vectors. When

$$\lambda = 3, \text{ the coefficient matrix of } (A - \lambda I)X = 0 \text{ reduced into } \begin{bmatrix} -2 & 0 & 1 \\ 2 & 0 & -1 \\ 4 & 4 & -4 \end{bmatrix}$$

Here the only independent rows are first and last. So the eigen vector can be found as the

$$\text{cross product of these two rows. } \therefore \mathbf{v}_1 = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}.$$

$$\text{When } \lambda = -1, \text{ the coefficient matrix of } (A - \lambda I)X = 0 \text{ reduced into } \begin{bmatrix} 2 & 0 & 1 \\ 2 & 4 & -1 \\ 4 & 4 & 0 \end{bmatrix}$$

Here the only independent rows are first and second. So the eigen vector can be found as the cross product of these two rows. $\therefore v_2 = \begin{bmatrix} -1 \\ 1 \\ 2 \end{bmatrix}$. When $\lambda = 1$, the coefficient matrix of

$$(A - \lambda I)X = 0 \text{ reduced into } \begin{bmatrix} 0 & 0 & 1 \\ 2 & 2 & -1 \\ 4 & 4 & -2 \end{bmatrix}$$

Here the only independent rows are first and second. So the eigen vector can be found as the cross product of these two rows. $\therefore v_2 = \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}$.

2.4.4 Python code to find eigen values and eigen vectors

1. Find eigen values and eigen vectors of $A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$.

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

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

# Find eigenvalues
eigenvalues, _ = np.linalg.eig(A)

# Define identity matrix I
I = np.eye(A.shape[0])

# Iterate over eigenvalues to find corresponding eigenvectors
for i, eigenvalue in enumerate(eigenvalues):
    # Compute A - lambda * I
    A_lambda_I = A - eigenvalue * I

    # Find the null space (which gives the eigenvector)
    eig_vector = null_space(A_lambda_I)

    print(f"Eigenvalue {i+1}: {eigenvalue}")
    print(f"Eigenvector {i+1}: \n{eig_vector}\n")
```

Eigenvalue 1: 3.0

Eigenvector 1:

```
[0.70710678]  
[0.70710678]]
```

Eigenvalue 2: 1.0

Eigenvector 2:

```
[-0.70710678]  
[ 0.70710678]]
```

Same can be done using direct approach. Code for this task is given below.

```
import numpy as np  
  
# Define matrix A  
A = np.array([[2, 1],  
              [1, 2]])  
  
# Find eigenvalues and eigenvectors  
eigenvalues, eigenvectors = np.linalg.eig(A)  
  
# Display the results  
print("Eigenvalues:", eigenvalues)  
print("Eigenvectors:\n", eigenvectors)
```

Eigenvalues: [3. 1.]

Eigenvectors:

```
[[ 0.70710678 -0.70710678]  
 [ 0.70710678  0.70710678]]
```

2.4.5 Diagonalization of Symmetric Matrices

For a symmetric matrix A , the process of diagonalization can be summarized as follows:

1. **Compute eigenvalues:** Solve the characteristic equation $\det(A - \lambda I) = 0$ to find the eigenvalues.
2. **Find eigenvectors:** For each eigenvalue λ_i , solve $(A - \lambda_i I)v_i = 0$ to find the corresponding eigenvector v_i .
3. **Form the eigenvector matrix:** Arrange the eigenvectors into a matrix Q , with each eigenvector as a column.

4. **Form the diagonal matrix of eigenvalues:** Construct Λ by placing the eigenvalues along the diagonal of the matrix.

Thus, the matrix can be expressed as $A = Q\Lambda Q^\top$.

1. Diagonalize the matrix $A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$.

Python code for this task is given below.

```
import numpy as np

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

# Step 1: Find eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)

# Step 2: Construct the diagonal matrix D (eigenvalues)
D = np.diag(eigenvalues)

# Step 3: Construct the matrix P (eigenvectors)
P = eigenvectors

# Step 4: Calculate the inverse of P
P_inv = np.linalg.inv(P)

# Verify the diagonalization: A = P D P_inv
A_reconstructed = P @ D @ P_inv

print("Matrix A:")
print(A)

print("\nEigenvalues (Diagonal matrix D):")
print(D)

print("\nEigenvectors (Matrix P):")
print(P)

print("\nInverse of P:")
print(P_inv)
```

```
print("\nReconstructed matrix A (P D P^(-1)):")
print(A_reconstructed)
```

Matrix A:

```
[[2 1]
 [1 2]]
```

Eigenvalues (Diagonal matrix D):

```
[[3. 0.]
 [0. 1.]]
```

Eigenvectors (Matrix P):

```
[[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]
```

Inverse of P:

```
[[ 0.70710678  0.70710678]
 [-0.70710678  0.70710678]]
```

Reconstructed matrix A (P D P⁻¹):

```
[[2. 1.]
 [1. 2.]]
```

2.4.6 Matrix Functions and Spectral Theorem

Once a matrix is diagonalized, various matrix functions become easier to compute. For a function $f(A)$, such as the exponential of a matrix or any power, the function can be applied to the diagonal matrix of eigenvalues:

$$f(A) = Qf(\Lambda)Q^T$$

where $f(\Lambda)$ is the function applied element-wise to the eigenvalues in the diagonal matrix Λ .

3 QR Decomposition

The **QR decomposition** of a matrix is a factorization technique that expresses a matrix A as the product of an orthogonal matrix Q and an upper triangular matrix R :

$$A = QR$$

where: - Q is an orthogonal matrix ($Q^T Q = I$), meaning its columns are orthonormal vectors.
- R is an upper triangular matrix.

Properties

1. **Orthogonality:** The columns of Q are orthonormal, which implies that $Q^T = Q^{-1}$.
2. **Uniqueness:** The QR decomposition is unique if the columns of A are linearly independent.

Relation with Other Decompositions

1. **LU Decomposition:**

- **LU decomposition** factors a matrix A into a lower triangular matrix L and an upper triangular matrix U :

$$A = LU$$

- Unlike QR, LU decomposition does not require the columns of A to be orthogonal.
- QR decomposition is often used when A is not square or when numerical stability is a concern, as it can be computed using Gram-Schmidt or Householder reflections.

2. **Cholesky Decomposition (CR):**

- The **Cholesky decomposition** is a specific case of LU decomposition applicable to symmetric positive-definite matrices:

$$A = LL^T$$

- It is more efficient than LU decomposition for suitable matrices but does not provide orthogonality like QR.

3. Spectral Decomposition:

- The **spectral decomposition** expresses a symmetric matrix A in terms of its eigenvalues and eigenvectors:

$$A = Q\Lambda Q^T$$

- While QR decomposition provides an orthogonal basis for any matrix, spectral decomposition is specifically used for symmetric matrices, providing insights into the matrix's properties through its eigenvalues and eigenvectors.

4. Singular Value Decomposition (SVD):

- The **SVD** decomposes a matrix A into three matrices:

$$A = U\Sigma V^T$$

- U and V are orthogonal matrices, and Σ is a diagonal matrix of singular values.
- SVD is more general than QR and is particularly useful in applications involving rank-deficient matrices, dimensionality reduction, and noise reduction.

3.0.1 Practical Uses of QR Decomposition

1. Solving Linear Systems:

- QR decomposition is used to solve linear systems of equations, especially over-determined systems where there are more equations than unknowns. The least squares solution can be efficiently obtained via QR.

2. Eigenvalue Problems:

- QR algorithms are often used in iterative methods for finding eigenvalues and eigenvectors of matrices, especially for large matrices.

3. Numerical Stability:

- QR decomposition is numerically stable, making it suitable for computations involving floating-point arithmetic, particularly when dealing with ill-conditioned matrices.

4. Computer Graphics:

- In computer graphics, QR decomposition can be used in perspective projection, where 3D points are projected onto a 2D plane, often needing orthogonal transformations.

5. Signal Processing:

- In signal processing, QR decomposition is utilized in adaptive filtering algorithms and for solving problems related to estimation theory.

3.0.2 Python method for QR decomposition

1. Find the QR decomposition of the matrix, $A = \begin{bmatrix} 12 & -51 & 4 \\ 6 & 167 & -68 \\ -4 & 24 & -41 \end{bmatrix}$. Verify the decomposition using the reconstruction.

```
import numpy as np

# Define A
A = np.array([[12, -51, 4],
              [6, 167, -68],
              [-4, 24, -41]])

# Perform QR decomposition
Q, R = np.linalg.qr(A)

# Display the results
print("Matrix A:")
print(A)

print("\nOrthogonal Matrix Q:")
print(Q)

print("\nUpper Triangular Matrix R:")
print(R)

# Verify the decomposition
print("\nVerification (Q @ R):")
print(np.dot(Q, R))

# Check if Q is orthogonal (Q^T @ Q should be the identity matrix)
print("\nQ^T @ Q (should be identity):")
print(np.dot(Q.T, Q))
```

```
Matrix A:
[[ 12 -51  4]
 [  6 167 -68]
 [ -4  24 -41]]
```

```
[ -4  24 -41]]
```

Orthogonal Matrix Q:

```
[[ -0.85714286  0.39428571  0.33142857]
 [ -0.42857143 -0.90285714 -0.03428571]
 [  0.28571429 -0.17142857  0.94285714]]
```

Upper Triangular Matrix R:

```
[[ -14.  -21.   14.]
 [   0. -175.   70.]
 [   0.    0.  -35.]]
```

Verification (Q @ R):

```
[[ 12. -51.   4.]
 [  6. 167. -68.]
 [ -4.  24. -41.]]
```

$Q^T @ Q$ (should be identity):

```
[[ 1.00000000e+00 -5.04131884e-17 -3.39864191e-17]
 [-5.04131884e-17  1.00000000e+00  2.30881074e-17]
 [-3.39864191e-17  2.30881074e-17  1.00000000e+00]]
```

3.1 Overdetermined Systems

An **overdetermined system** of linear equations is a system in which there are more equations than unknowns. Mathematically, if we have a matrix A of size $m \times n$ where $m > n$, the system can be represented as:

$$A\mathbf{x} = \mathbf{b}$$

where: - A is the coefficient matrix, - \mathbf{x} is the vector of unknowns (with size n), - \mathbf{b} is the vector of constants (with size m).

3.1.1 Example of an Overdetermined System

Consider the following system of equations:

$$\begin{aligned}2x_1 + 3x_2 &= 5 \\4x_1 + 6x_2 &= 10 \\1x_1 + 2x_2 &= 3\end{aligned}$$

Here, we have three equations but only two unknowns (x_1 and x_2). This system is overdetermined.

3.1.2 Challenges in Solving Overdetermined Systems

1. **No Exact Solutions:** In most cases, an overdetermined system does not have an exact solution because the equations may be inconsistent. For example, if one equation contradicts another, no value of \mathbf{x} can satisfy all equations simultaneously.
2. **Finding Best Approximation:** When the system is consistent, it may still be that no single solution satisfies all equations perfectly. Therefore, the goal is often to find an approximate solution that minimizes the error.

3.1.3 Why We Need QR Decomposition

QR decomposition is particularly useful for solving overdetermined systems for the following reasons:

1. **Least Squares Solution:**
 - The primary goal in solving an overdetermined system is to find the least squares solution, which minimizes the sum of the squared residuals (the differences between the left and right sides of the equations). QR decomposition allows us to efficiently compute this solution.
2. **Orthogonality:**
 - The QR decomposition expresses the matrix A as the product of an orthogonal matrix Q and an upper triangular matrix R :

$$A = QR$$

- The orthogonality of Q ensures numerical stability and helps in reducing the problem to solving triangular systems.
3. **Stability:**

- QR decomposition is more stable than other methods, such as Gaussian elimination, especially when dealing with ill-conditioned matrices. This is crucial in applications where precision is important.

4. Computational Efficiency:

- The process of obtaining the QR decomposition can be performed using efficient algorithms, such as Gram-Schmidt orthogonalization or Householder reflections, which makes it suitable for large systems.

3.1.4 Solving an Overdetermined System using QR Decomposition

Given an overdetermined system represented as $A\mathbf{x} = \mathbf{b}$, the steps to find the least squares solution using QR decomposition are as follows:

1. Compute QR Decomposition:

- Decompose the matrix A into Q and R .

2. Formulate the Normal Equations:

- The least squares solution can be found from the equation:

$$R\mathbf{x} = Q^T\mathbf{b}$$

3. Solve the Triangular System:

- Solve for \mathbf{x} using back substitution, as R is an upper triangular matrix.

Python code for solving the above system of equations is given below.

```
import numpy as np

# Define the coefficient matrix A and the constant vector b
A = np.array([[2, 3],
              [4, 6],
              [1, 2]])

b = np.array([5, 10, 3])

# Perform QR decomposition
Q, R = np.linalg.qr(A)

# Calculate the least squares solution
# Solve the equation R * x = Q^T * b
```

```

Q_b = np.dot(Q.T, b)
x = np.linalg.solve(R, Q_b)

print("The least squares solution is:")
print(x)

```

The least squares solution is:
[1. 1.]

3.1.5 Problems

Problem 1: Simple Overdetermined System

Problem Statement:

Solve the overdetermined system given by the equations:

$$\begin{aligned}
 2x + 3y &= 5 \\
 4x + 6y &= 10 \\
 1x + 2y &= 2
 \end{aligned}$$

```

# Problem 1: Simple Overdetermined System

import numpy as np

# Define the coefficient matrix A and the vector b
A1 = np.array([[2, 3],
               [4, 6],
               [1, 2]])

b1 = np.array([5, 10, 2])

# QR decomposition
Q1, R1 = np.linalg.qr(A1)
x_qr1 = np.linalg.solve(R1, Q1.T @ b1)

# Ordinary Least Squares solution using np.linalg.lstsq
x_ols1, residuals1, rank1, s1 = np.linalg.lstsq(A1, b1, rcond=None)

```

```
print("Problem 1 - QR Decomposition Solution:", x_qr1)
print("Problem 1 - Ordinary Least Squares Solution:", x_ols1)
```

Problem 1 - QR Decomposition Solution: [4. -1.]
 Problem 1 - Ordinary Least Squares Solution: [4. -1.]

Problem 2: Overdetermined System with No Exact Solution

Problem Statement:

Solve the following system:

$$x + 2y = 3$$

$$2x + 4y = 6$$

$$3x + 1y = 5$$

```
# Problem 2: Overdetermined System with No Exact Solution

# Define the coefficient matrix A and the vector b
A2 = np.array([[1, 2],
               [2, 4],
               [3, 1]])

b2 = np.array([3, 6, 5])

# QR decomposition
Q2, R2 = np.linalg.qr(A2)
x_qr2 = np.linalg.solve(R2, Q2.T @ b2)

# Ordinary Least Squares solution using np.linalg.lstsq
x_ols2, residuals2, rank2, s2 = np.linalg.lstsq(A2, b2, rcond=None)

print("Problem 2 - QR Decomposition Solution:", x_qr2)
print("Problem 2 - Ordinary Least Squares Solution:", x_ols2)
```

Problem 2 - QR Decomposition Solution: [1.4 0.8]
 Problem 2 - Ordinary Least Squares Solution: [1.4 0.8]

Problem 3: Overdetermined System with Random Data

Problem Statement:

Generate a random overdetermined system and solve it:

$$Ax = b$$

Where (A) is a random (6 × 3) matrix and (b) is generated accordingly.

```
# Problem 3: Overdetermined System with Random Data

# Generate a random overdetermined system
np.random.seed(0) # For reproducibility
A3 = np.random.rand(6, 3)
x_true = np.array([1, 2, 3]) # True solution
b3 = A3 @ x_true + np.random.normal(0, 0.1, 6) # Adding some noise

# QR decomposition
Q3, R3 = np.linalg.qr(A3)
x_qr3 = np.linalg.solve(R3, Q3.T @ b3)

# Ordinary Least Squares solution using np.linalg.lstsq
x_ols3, residuals3, rank3, s3 = np.linalg.lstsq(A3, b3, rcond=None)

print("Problem 3 - QR Decomposition Solution:", x_qr3)
print("Problem 3 - Ordinary Least Squares Solution:", x_ols3)
```

Problem 3 - QR Decomposition Solution: [0.94791379 2.10331498 2.98999875]

Problem 3 - Ordinary Least Squares Solution: [0.94791379 2.10331498 2.98999875]

Problem 4: Real-World Data Fitting

Problem Statement:

Fit a linear model to the following data points:

$$(1, 2), (2, 3), (3, 5), (4, 7), (5, 11)$$

```
# Problem 4: Real-World Data Fitting

# Data points
x_data = np.array([1, 2, 3, 4, 5])
y_data = np.array([2, 3, 5, 7, 11])
```

```

# Create the design matrix A
A4 = np.vstack([x_data, np.ones(len(x_data))]).T # Add intercept

# QR decomposition
Q4, R4 = np.linalg.qr(A4)
x_qr4 = np.linalg.solve(R4, Q4.T @ y_data)

# Ordinary Least Squares solution using np.linalg.lstsq
x_ols4, residuals4, rank4, s4 = np.linalg.lstsq(A4, y_data, rcond=None)

print("Problem 4 - QR Decomposition Solution:", x_qr4)
print("Problem 4 - Ordinary Least Squares Solution:", x_ols4)

```

Problem 4 - QR Decomposition Solution: [2.2 -1.]
 Problem 4 - Ordinary Least Squares Solution: [2.2 -1.]

Problem 5: Polynomial Fit (Higher Degree)

Problem Statement:

Fit a quadratic polynomial to the following data points:

(1, 1), (2, 4), (3, 9), (4, 16), (5, 25)

```

# Problem 5: Polynomial Fit (Higher Degree)

# Data points for polynomial fitting
x_data_poly = np.array([1, 2, 3, 4, 5])
y_data_poly = np.array([1, 4, 9, 16, 25])

# Create the design matrix for a quadratic polynomial
A5 = np.vstack([x_data_poly**2, x_data_poly, np.ones(len(x_data_poly))]).T

# QR decomposition
Q5, R5 = np.linalg.qr(A5)
x_qr5 = np.linalg.solve(R5, Q5.T @ y_data_poly)

# Ordinary Least Squares solution using np.linalg.lstsq
x_ols5, residuals5, rank5, s5 = np.linalg.lstsq(A5, y_data_poly, rcond=None)

```

```
print("Problem 5 - QR Decomposition Solution:", x_qr5)
print("Problem 5 - Ordinary Least Squares Solution:", x_ols5)
```

Problem 5 - QR Decomposition Solution: [1.00000000e+00 -2.73316113e-15 3.80986098e-15]

Problem 5 - Ordinary Least Squares Solution: [1.00000000e+00 -6.77505297e-15 1.06049542e-15]

References

- Harris, Charles R., K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, et al. 2020. “Array Programming with NumPy.” *Nature* 585 (7825): 357–62. <https://doi.org/10.1038/s41586-020-2649-2>.
- Strang, Gilbert. 2020. *Linear Algebra for Everyone*. SIAM.