

Computational Linear Algebra

Siju Swamy

2024-07-16

Table of contents

Preface	7
Introduction	8
Introduction to Computational Linear Algebra	8
Course Themes	8
Relevance and Impact	8
1 Python for Linear Algebra	10
1.1 Pseudocode: the new language for algorithm design	10
1.1.1 Matrix Sum	10
1.1.2 Matrix Difference	11
1.1.3 Matrix Product	12
1.1.4 Determinant	14
1.1.5 Rank of a Matrix	17
1.1.6 Practice Problems	19
1.1.7 Solving a System of Equations	20
1.1.8 Review Problems	22
1.2 Transition from Pseudocode to Python Programming	23
1.3 Python Fundamentals	24
1.3.1 Python Programming Overview	24
1.3.2 Variables	25
1.3.3 Python Programming Style	26
1.4 Basic Datatypes in Python	27
1.4.1 Numeric Types	28
1.4.2 Sequence Types	28
1.4.3 Mapping Types	33
1.4.4 Set Types	35
1.5 Control Structures in Python	39
1.5.1 Conditional Statements	39
1.5.2 Looping Statements	40
1.5.3 Control Flow Statements	41
1.6 Functions in Python Programming	42
1.7 Object-Oriented Programming (OOP) in Python	44
1.7.1 Key Concepts of OOP	45
1.8 Working with Files in Python	48

1.9	From Theory to Practice	50
1.9.1	Applications of Matrix Operations in Digital Image Processing	50
1.10	Matrix Operations Using Python Libraries	58
1.10.1	Introduction	58
1.10.2	Introduction to SymPy	58
1.11	Module review	72
1.12	Conclusion	73
2	Transforming Linear Algebra to Computational Language	75
2.1	Introduction	75
2.2	Relearning of Terms and Operations in Linear Algebra	75
2.2.1	Matrix Addition and Subtraction in Data Analysis	76
2.2.2	More on Matrix Product and its Applications	90
2.2.3	Matrix Measures of Practical Importance	145
2.2.4	Rank Nullity Theorem	149
2.2.5	Fundamental Subspaces	151
2.3	Module review	161
3	Python Libraries for Computational Linear Algebra	165
3.1	Introduction to NumPy	165
3.1.1	Purpose of Using NumPy	165
3.2	Basic Operations in NumPy	166
3.2.1	Define different types of numpy arrays	167
3.2.2	Review Questions	172
3.2.3	Tensors in NumPy	173
3.2.4	Array Indexing: Accessing Single Elements	177
3.2.5	Array Concatenation and Splitting	184
3.2.6	Review Questions	187
3.3	Some important NumPy function for Linear Algrbra	192
3.3.1	Linear Regression using NumPy	198
3.3.2	Some interesting handy matrix operations using numpy arrays	198
3.4	Basics of SciPy Library for Computational Linear Algebra	201
3.4.1	Basic Matrix operations	201
3.5	Sparse Matrices	204
3.5.1	Sparse Matrix operations in SciPy	204
3.6	Visualization Libraries	208
3.6.1	Matplotlib: A Comprehensive Data Visualization Library in Python	209
3.6.2	How to Display Your Plots?	211
3.6.3	Plotting from a Jupyter Notebook	211
3.6.4	Histograms, Binnings, and Density	226
3.6.5	Working with datafiles	229
3.6.6	Data Visualization With Seaborn	236
3.6.7	Histograms With Seaborn	237

3.6.8	Scatter Plots With Seaborn	238
3.7	Module review	241
4	Linear Algebra for Advanced Applications	245
4.1	Introduction	245
4.2	LU Decomposition	245
4.2.1	Step-by-Step Procedure	246
4.2.2	Example	248
4.2.3	Python Implementation	248
4.2.4	LU Decomposition Practice Problems with Solutions	249
4.3	LU Decomposition Practice Problems	252
4.4	Matrix Approach to Create LU Decomposition	254
5	Spectral Decomposition	257
5.1	Background	257
5.2	Introduction	258
5.3	Spectral Decomposition: Detailed Concepts	258
5.3.1	Eigenvalues and Eigenvectors	258
5.3.2	Eigenvalue Decomposition (Spectral Decomposition)	261
5.3.3	Geometric Interpretation	262
5.3.4	Importance of Diagonalization	262
5.3.5	Properties of Symmetric Matrices	262
5.4	Mathematical Requirements for Spectral Decomposition	263
5.4.1	Determining Eigenvalues and Eigenvectors	263
5.4.2	Characteristic Polynomial of 2×2 Matrices	263
5.4.3	Problems	264
5.4.4	Python code to find eigen values and eigen vectors	272
5.4.5	Diagonalization of Symmetric Matrices	273
5.4.6	Matrix Functions and Spectral Theorem	275
6	QR Decomposition	276
6.0.1	Practical Uses of QR Decomposition	277
6.0.2	Python method for DR decomposition	278
6.1	Overdetermined Systems	279
6.1.1	Example of an Overdetermined System	279
6.1.2	Challenges in Solving Overdetermined Systems	280
6.1.3	Why We Need QR Decomposition	280
6.1.4	Solving an Overdetermined System using QR Decomposition	281
6.1.5	Problems	282
6.2	Module review	286
7	Practical Uses Cases	289
7.1	Singular Value Decomposition (SVD) – An Intuitive and Mathematical Approach	289

7.2	The SVD Theorem	289
7.3	Intuition Behind SVD	290
7.4	Spectral Decomposition vs. SVD	290
7.4.1	Comparison:	290
7.5	Steps to Find U , Σ , and V^T	290
7.6	Example	291
7.6.1	Step 1: Compute $A^T A$	291
7.6.2	Step 2: Find V from the eigenvectors of $A^T A$	292
7.6.3	Step 3: Construct Σ	292
7.6.4	Step 4: Find U from the eigenvectors of AA^T	292
7.6.5	Step 5: Final SVD	293
7.7	Reconstructing Matrix A Using SVD	294
7.7.1	Breakdown of Terms:	294
7.7.2	Example:	295
7.8	Singular Value Decomposition in Image Processing	296
7.8.1	Image Compression	296
7.8.2	Noise Reduction	296
7.8.3	Image Reconstruction	296
7.8.4	Facial Recognition	296
7.8.5	Image Segmentation	297
7.8.6	Color Image Processing	297
7.8.7	Pattern Recognition	297
7.8.8	Example	297
7.9	Takeaway	311
7.10	Principal Component Analysis	312
7.11	Principal Component Analysis as a Special Case of SVD	312
7.12	Problem Setting for PCA	312
7.13	Covariance Matrix	312
7.14	Eigenvalue Decomposition	313
7.15	Singular Value Decomposition (SVD)	313
7.16	Applications of PCA	314
7.17	Image Compression using PCA	314
7.17.1	Sample problems	314
7.17.2	Step 1: Center the Data	314
7.17.3	Step 2: Calculate the Covariance Matrix	315
7.17.4	Step 3: Calculate Eigenvalues and Eigenvectors	316
7.17.5	Python Implementation	317
7.18	Principal Component Analysis (PCA) for Image Reconstruction	318
7.19	Micro Projects	322
7.20	Sample questions for Lab Work	322
7.20.1	Question 1: Spectral Decomposition of a Symmetric Matrix	323
7.20.2	Question 2: Diagonalization of a Matrix	323
7.20.3	Question 3: Eigenvalues and Eigenvectors of a Square Matrix	323

7.20.4 Question 4: Orthogonal Matrix Decomposition	323
7.20.5 Question 5: Singular Value Decomposition (SVD)	324
7.20.6 Question 6: Rank-1 Approximation Using SVD	324
7.20.7 Question 7: Matrix Compression Using SVD	324
7.20.8 Question 8: SVD Data Reconstruction	324
7.20.9 Question 9: Principal Component Analysis (PCA)	325
7.20.10 Question 10: Dimensionality Reduction with PCA	325
7.21 Hint to Solutions.	325
7.22 Module review	330
References	333

Preface

Welcome to the course on **Computational Linear Algebra**. This course is designed to provide a practical perspective on linear algebra, bridging the gap between mathematical theory and real-world applications. As we delve into the intricacies of linear algebra, our focus will be on equipping you with the skills to effectively utilize these concepts in the design, development, and manipulation of data-driven processes applicable to Computer Science and Engineering.

Throughout this course, you will explore linear algebra not just as a set of abstract mathematical principles, but as a powerful tool for solving complex problems and optimizing processes. The curriculum integrates robust mathematical theory with hands-on implementation, enabling you to apply linear algebra techniques in practical scenarios.

From understanding fundamental operations to applying advanced concepts in data-driven contexts, this course aims to build a strong foundation that supports both theoretical knowledge and practical expertise. Whether you're tackling computational challenges or developing innovative solutions, the skills and insights gained here will be invaluable in your academic and professional endeavors Knuth (1984).

We look forward to guiding you through this journey of blending theory with practice and helping you harness the full potential of linear algebra in your work.

Introduction

Introduction to Computational Linear Algebra

Welcome to the Computational Linear Algebra course, a pivotal component of our Computational Mathematics for Engineering minor program. This course is meticulously designed to connect theoretical linear algebra concepts with their practical applications in Artificial Intelligence (AI) and Data Science.

Course Themes

1. Practical Application Proficiency

- Our primary focus is on seamlessly translating theoretical concepts into practical solutions for real-world challenges.
- Develop robust problem-solving skills applicable to AI, Data Science, and advanced engineering scenarios.

2. Mathematical Expertise for Data Insights

- Gain in-depth proficiency in computational linear algebra, covering essential topics like matrix operations, eigendecomposition, and singular value decomposition.
- Leverage linear algebra techniques to derive meaningful insights and make informed decisions in data science applications.

3. Hands-On Learning

- Engage in immersive, project-based learning experiences with a strong emphasis on Python implementation.
- Apply linear algebra principles to practical problems, including linear regression, principal component analysis (PCA), and neural networks.

Relevance and Impact

In today's technology-driven landscape, linear algebra forms the backbone of many critical algorithms and applications in AI and Data Science. This course will not only enhance your

analytical and computational skills but also prepare you to address complex engineering problems with confidence.

By the end of this course, you will have acquired a comprehensive understanding of the role of linear algebra in computational mathematics and its practical applications. This knowledge will equip you with the tools necessary to excel in the rapidly evolving tech industry.

Let us start this educational journey together, where theoretical knowledge meets practical application, and explore the fascinating and impactful world of Computational Linear Algebra.

1 Python for Linear Algebra

1.1 Pseudocode: the new language for algorithm design

Pseudocode is a way to describe algorithms in a structured but plain language. It helps in planning the logic without worrying about the syntax of a specific programming language. In this module, we'll use Python-flavored pseudocode to describe various matrix operations.



Caution

There are varieties of approaches in writing pseudocode. Students can adopt any of the standard approach to write pseudocode.

1.1.1 Matrix Sum

Mathematical Procedure:

To add two matrices A and B , both matrices must have the same dimensions. The sum C of two matrices A and B is calculated element-wise:

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

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:

```

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 .

1.1.2 Matrix Difference

Mathematical Procedure:

To subtract matrix B from matrix A , both matrices must have the same dimensions. The difference C of two matrices A and B is calculated element-wise:

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

Example:

Let A and B be two 2×2 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:

```

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:

```

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 .

1.1.3 Matrix Product

Mathematical Procedure:

To find the product of two matrices A and B , the number of columns in A must be equal to the number of rows in B . The element $C[i][j]$ in the product matrix C is computed as:

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

Example:

Let A be a 2×3 matrix and B be a 3×2 matrix:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}$$

The product C is:

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

Pseudocode:

```
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:

```

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 of i the row of A to j th column of B for each element.
4. Return the resulting matrix C .

1.1.4 Determinant

Mathematical Procedure:

To find the determinant of a square matrix A , we can use the Laplace expansion, which involves breaking the matrix down into smaller submatrices. For a 2×2 matrix, the determinant is calculated as:

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

For larger matrices, the determinant is calculated recursively.

Example:

Let A be a 2×2 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:

```

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:

```

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 sub_matrix
    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 2×2 , 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.

1.1.5 Rank of a Matrix

Mathematical Procedure:

The rank of a matrix A is the maximum number of linearly independent rows or columns in A . This can be found using Gaussian elimination to transform the matrix into its row echelon form (REF) and then counting the number of non-zero rows.

Example:

Let A be a 3×3 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:

```
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:

```

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 out the elements below the diagonal.
4. If the diagonal element is zero, try to swap with a lower row that has a non-zero element in the same column.
5. If no such row is found, decrement the rank.
6. Return the resulting rank of the matrix.

1.1.6 Practice Problems

Find the rank of the following matrices.

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

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

1.1.7 Solving a System of Equations

Mathematical Procedure:

To solve a system of linear equations represented as $A\mathbf{x} = \mathbf{b}$, where A is the coefficient matrix, \mathbf{x} is the vector of variables, and \mathbf{b} is the constant vector, we can use Gaussian elimination to transform the augmented matrix $[A|\mathbf{b}]$ into its row echelon form (REF) and then perform back substitution to find the solution vector \mathbf{x} Strang (2022).

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:

$$[A|\mathbf{b}] = \left[\begin{array}{ccc|c} 1 & 2 & 3 & 9 \\ 4 & 5 & 6 & 24 \\ 7 & 8 & 9 & 39 \end{array} \right]$$

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

$$\text{REF}(A) = \left[\begin{array}{ccc|c} 1 & 2 & 3 & 9 \\ 0 & -3 & -6 & -12 \\ 0 & 0 & 0 & 0 \end{array} \right]$$

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

$$\begin{cases} z = 1 \\ y = 0 \\ x = 3 \end{cases}$$

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

Pseudocode:

```
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 .

1.1.8 Review Problems

Q1: Fill in the missing parts of the pseudocode to yield a meaningful algebraic operation on of two matrices.

Pseudocode:

```

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 missing part.

Pseudocode:

```

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

```

1.2 Transition from Pseudocode to Python Programming

In this course, our initial approach to understanding and solving linear algebra problems has been through pseudocode. Pseudocode allows us to focus on the logical steps and algorithms without getting bogged down by the syntax of a specific programming language. This method helps us build a strong foundation in the computational aspects of linear algebra.

However, to fully leverage the power of computational tools and prepare for real-world applications, it is essential to implement these algorithms in a practical programming language. Python is a highly versatile and widely-used language in the fields of data science, artificial intelligence, and engineering. By transitioning from pseudocode to Python, we align with the following course objectives:

1. **Practical Implementation:** Python provides numerous libraries and tools, such as NumPy and SciPy, which are specifically designed for numerical computations and linear algebra. Implementing our algorithms in Python allows us to perform complex calculations efficiently and accurately.
2. **Hands-On Experience:** Moving to Python programming gives students hands-on experience in coding, debugging, and optimizing algorithms. This practical experience is crucial for developing the skills required in modern computational tasks.
3. **Industry Relevance:** Python is extensively used in industry for data analysis, machine learning, and scientific research. Familiarity with Python and its libraries ensures that students are well-prepared for internships, research projects, and future careers in these fields.
4. **Integration with Other Tools:** Python's compatibility with various tools and platforms allows for seamless integration into larger projects and workflows. This integration is vital for tackling real-world problems that often require multi-disciplinary approaches.
5. **Enhanced Learning:** Implementing algorithms in Python helps reinforce theoretical concepts by providing immediate feedback through code execution and results visualization. This iterative learning process deepens understanding and retention of the material.

By transitioning to Python programming, we not only achieve our course objectives but also equip students with valuable skills that are directly applicable to their academic and professional pursuits.

1.3 Python Fundamentals

1.3.1 Python Programming Overview

Python is a high-level, interpreted programming language that was created by Guido van Rossum and first released in 1991. Its design philosophy emphasizes code readability and simplicity, making it an excellent choice for both beginners and experienced developers. Over the years, Python has undergone significant development and improvement, with major releases adding new features and optimizations. The language's versatility and ease of use have made it popular in various domains, including web development, data science, artificial intelligence, scientific computing, automation, and more. Python's extensive standard library and active community contribute to its widespread adoption, making it one of the most popular programming languages in the world today.

1.3.2 Variables

In Python, variables are used to store data that can be used and manipulated throughout a program. Variables do not need explicit declaration to reserve memory space. The declaration happens automatically when a value is assigned to a variable.

Basic Input/Output Functions

Python provides built-in functions for basic input and output operations. The `print()` function is used to display output, while the `input()` function is used to take input from the user.

Output with `print()` function

Example 1

```
# 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

```
# 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:

```
# 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.")
```

Note

The `print()` function in Python, defined in the built-in `__builtin__` module, is used to display output on the screen, providing a simple way to output text and variable values to the console.

Combining Variables and Input/Output

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

Example:

```
# 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)
```

1.3.3 Python Programming Style

1.3.3.1 Indentation

Python uses indentation to define the blocks of code. Proper indentation is crucial as it affects the program's flow. Use 4 spaces per indentation level.

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

1.3.3.2 Comments

Use comments to explain user code. Comments begin with the # symbol and extend to the end of the line. Write comments that are clear and concise. See the example:

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

1.3.3.3 Variable Naming

Use meaningful variable names to make user code more understandable. Variable names should be in lowercase with words separated by underscores.

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

1.3.3.4 Consistent Style

Follow the PEP 8 style guide for Python code to maintain consistency and readability. Use blank lines to separate different sections of user code. See the following example of function definition:

```
def calculate_sum(x, y):  
    return x + y  
  
result = calculate_sum(5, 3)  
print(result)
```

1.4 Basic Datatypes in Python

In Python, a datatype is a classification that specifies which type of value a variable can hold. Understanding datatypes is essential as it helps in performing appropriate operations on variables. Python supports various built-in datatypes, which can be categorized into several groups.

1.4.1 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.`

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

1.4.2 Sequence Types

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

1.4.2.1 String Type

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

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:

```
# 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. Negative indexing can be used to access characters from the end.

Example:

```
# 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:

```
# 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!', '']
```

1.4.2.2 List Type

Lists are one of the most versatile and commonly used sequence types in Python. They allow for the storage and manipulation of ordered collections of items.

Characteristics of Lists

- *Ordered*: The items in a list have a defined order, which will not change unless explicitly modified.
- *Mutable*: The content of a list can be changed after its creation (i.e., items can be added, removed, or modified).
- *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, floats).

Creating Lists

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

Example:

```
# 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:

```
# 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 existing items.

Example:

```
# 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:

```
# 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]
```

1.4.2.3 Tuple Type

Tuples are a built-in sequence type in Python that is used to store an ordered collection of items. Unlike lists, tuples are immutable, which means their contents cannot be changed after creation.

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, removing, or changing 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, and floats.

Creating Tuples

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

Example:

```
# 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 indexing can be used to access items from the end.

Example:

```
# 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 by combining or slicing existing ones.

Example:

```

# 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:

```

# 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

```

1.4.3 Mapping Types

Mapping types in Python are used to store data in key-value pairs. Unlike sequences, mappings do not maintain an order and are designed for quick lookups of data.

1.4.3.1 Dictionary (`dict`)

The primary mapping type in Python is the `dict`. Dictionaries store data as key-value pairs, where each key must be unique, and keys are used to access their corresponding values.

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:

```
# 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:

```
# 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:

```

# 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

```

1.4.4 Set Types

Sets are a built-in data type in Python used to store unique, unordered collections of items. They are particularly useful for operations involving membership tests, set operations, and removing duplicates.

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:

```

# 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 like adding or removing items.

Example:

```

# 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 `difference`.

Example:

```

# 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 is not found.
4. `pop()`: Removes and returns an arbitrary item from the set.
5. `clear()`: Removes all items from the set.

Example:

```

# 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 {}

```

1.4.4.1 ## Frozen Sets

Frozen sets are a built-in data type in Python that are similar to sets but are immutable. Once created, a frozen set cannot be modified, making it suitable for use as a key in dictionaries or as elements of other sets.

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 can be added or removed.
- *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:

```

# 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, we can perform membership tests and other set operations.

Example:

```

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

# Since frozenset is immutable, us 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, intersection, and difference. These operations return new frozen sets and do not modify the original ones.

Example:

```

# 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:

```

# 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})

```

1.5 Control Structures in Python

Control structures in Python allow us to control the flow of execution in our programs. They help manage decision-making, looping, and the execution of code blocks based on certain conditions. Python provides several key control structures: `if` statements, `for` loops, `while` loops, and control flow statements like `break`, `continue`, and `pass`.

1.5.1 Conditional Statements

Conditional statements are used to execute code based on certain conditions. The primary conditional statement in Python is the `if` statement, which can be combined with `elif` and `else` to handle multiple conditions.

Syntax:

```

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.

```
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.")
```

1.5.2 Looping Statements

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

1.5.2.1 For Loop

The **for** loop iterates over a sequence (like a list, tuple, or string) and executes a block of code for each item in the sequence.

Syntax:

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

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

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

1.5.2.2 While Loop

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

Syntax:

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

Example: Print all counting numbers less than 5.

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

1.5.3 Control Flow Statements

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

1.5.3.1 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.

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

1.5.3.2 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.

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

1.5.3.3 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.

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

🔥 Cautions When Using Control Flow Structures

Control flow structures are essential in Python programming for directing the flow of execution. However, improper use of these structures can lead to errors, inefficiencies, and unintended behaviors. Here are some cautions to keep in mind:

Infinite Loops

- **Issue:** A `while` loop with a condition that never becomes `False` can lead to an infinite loop, which will cause the program to hang or become unresponsive.
- **Caution:** Always ensure that the condition in a `while` loop will eventually become `False`, and include logic within the loop to modify the condition.

Example:

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

1.6 Functions in Python Programming

Functions are a fundamental concept in Python programming that enable code reuse, modularity, and organization. They allow us to encapsulate a block of code that performs a specific task, which can be executed whenever needed. Functions are essential for writing clean, maintainable, and scalable code, making them a cornerstone of effective programming practices.

What is a Function?

A function is a named block of code designed to perform a specific task. Functions can take inputs, called parameters or arguments, and can return outputs, which are the results of the computation or task performed by the function. By defining functions, we can write code once and reuse it multiple times, which enhances both efficiency and readability.

Defining a Function

In Python, functions are defined using the `def` keyword, followed by the function name, parentheses containing any parameters, and a colon. The function body, which contains the code to be executed, is indented below the function definition.

Syntax:

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

Example:

```
def greet(name):
    """
    Returns a greeting message for the given name.
    """
    return f"Hello, {name}!"
```

1.6.0.1 Relevance of functions in Programming

1. *Code Reusability* : Functions allow us to define a piece of code once and reuse it in multiple places. This reduces redundancy and helps maintain consistency across our codebase.
2. *Modularity* : Functions break down complex problems into smaller, manageable pieces. Each function can be focused on a specific task, making it easier to understand and maintain the code.
3. *Abstraction* : Functions enable us to abstract away the implementation details. We can use a function without needing to know its internal workings, which simplifies the code we write and enhances readability.
4. *Testing and Debugging* : Functions allow us to test individual components of our code separately. This isolation helps in identifying and fixing bugs more efficiently.

5. *Library Creation* : Functions are the building blocks of libraries and modules. By organizing related functions into libraries, we can create reusable components that can be shared and utilized across different projects.

Example: Creating a Simple Library

Stage 1: Define Functions in a Module

```
# 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

```
# 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}")
```

1.7 Object-Oriented Programming (OOP) in Python

Object-Oriented Programming (OOP) is a programming paradigm that uses “objects” to design and implement software. It emphasizes the organization of code into classes and objects, allowing for the encapsulation of data and functionality. OOP promotes code reusability, scalability, and maintainability through key principles such as encapsulation, inheritance, and polymorphism.

1.7.1 Key Concepts of OOP

1. Classes and Objects

- **Class:** A class is a blueprint for creating objects. It defines a set of attributes and methods that the created objects will have. A class can be thought of as a template or prototype for objects.
- **Object:** An object is an instance of a class. It is a specific realization of the class with actual values for its attributes.

1.7.1.1 Example

```
# 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 operate on the data into a single unit, or class. It restricts direct access to some of the object's components and can help protect the internal state of the object from unintended modifications.

Example: Controll the access to member variables using encapsulation.

```

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 and methods from an existing class (parent or base class). It allows for code reuse and the creation of a hierarchy of classes.

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

```

# 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 and says Woof!
print(f"{cat.name} is a {cat.color} cat and says {cat.speak()}") # Output:
    ↳ Whiskers is a Gray cat and says Meow!

```

4. Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to be used for different data types. In Python, polymorphism is often achieved through method overriding, where a method in a derived class has the same name as a method in the base class but implements different functionality.

Example:

```

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

```

1.8 Working with Files in Python

File handling is an essential part of programming that allows us to work with data stored in files. Python provides built-in functions and methods to create, read, write, and manage files efficiently. This section will cover basic file operations, including opening, reading, writing, and closing files.

Opening a File

In Python, we use the `open()` function to open a file. This function returns a file object, which provides methods and attributes to interact with the file. The `open()` function requires at least one argument: the path to the file. We can also specify the mode in which the file should be opened.

Syntax:

```
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 appending).

Example:

```
# 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()`, `readline()`, and `readlines()`.

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

Example:

```
# 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 opened in `write` mode, the file is truncated (i.e., existing content is deleted). When opened in `append` mode, new data is added to the end of the file.

Example:

```
# 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 saved and resources are released. We can close a file using the `close()` method of the file object.

Example:

```
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 file opening and closing. We can use the `with` statement to ensure that a file is properly closed after its block of code is executed.

Example:

```
# 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.")
```

1.9 From Theory to Practice

In this section, we transition from theoretical concepts to practical applications by exploring how fundamental matrix operations can be used in the field of image processing. By leveraging the knowledge gained from understanding matrix addition, subtraction, multiplication, and other operations, we can tackle real-world problems such as image blending, sharpening, filtering, and transformations. This hands-on approach not only reinforces the theoretical principles but also demonstrates their utility in processing and enhancing digital images. Through practical examples and coding exercises, you'll see how these mathematical operations are essential tools in modern image manipulation and analysis.

1.9.1 Applications of Matrix Operations in Digital Image Processing

Matrix operations play a pivotal role in digital image processing, enabling a wide range of techniques for manipulating and enhancing images. By leveraging fundamental matrix operations such as addition, subtraction, multiplication, and transformations, we can perform essential tasks like image blending, filtering, edge detection, and geometric transformations. These operations provide the mathematical foundation for various algorithms used in image analysis, compression, and reconstruction. Understanding and applying matrix operations is crucial for developing efficient and effective image processing solutions, making it an indispensable skill in fields like computer vision, graphics, and multimedia applications.

1.9.1.1 Matrix Addition in Image Blending

Matrix addition is a fundamental operation in image processing, particularly useful in the technique of image blending. Image blending involves combining two images to produce a single image that integrates the features of both original images. This technique is commonly

used in applications such as image overlay, transition effects in videos, and creating composite images.

Concept

When working with grayscale images, each image can be represented as a matrix where each element corresponds to the intensity of a pixel. By adding corresponding elements (pixels) of two matrices (images), we can blend the images together. The resultant matrix represents the blended image, where each pixel is the sum of the corresponding pixels in the original images.

Example:

Consider two 2x2 grayscale images represented as matrices:

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

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

```
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 images).

Python Implementation of image blending

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

```
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)
```

Blended Image:

[150, 250]

[255, 255]

Image blending is a powerful technique with numerous real-time applications. It is widely used in creating smooth transitions in video editing, overlaying images in augmented reality, and producing composite images in photography and graphic design. By understanding and applying matrix operations, we can develop efficient algorithms that seamlessly integrate multiple images, enhancing the overall visual experience. The practical implementation of matrix addition in image blending underscores the importance of mathematical foundations in achieving sophisticated image processing tasks in real-world applications.

Image Blending as Basic Arithmetic with Libraries

In upcoming chapters, we will explore how specific libraries for image handling simplify the process of image blending to a basic arithmetic operation—adding two objects. Using these libraries, such as PIL (Python Imaging Library) or OpenCV, allows us to leverage efficient built-in functions that streamline tasks like resizing, matrix operations, and pixel manipulation.

Let's summarize a few more matrix operations and its uses in digital image processing tasks in the following sections.

1.9.1.2 Matrix Subtraction in Image Sharpening

Matrix subtraction is a fundamental operation in image processing, essential for techniques like image sharpening. Image sharpening enhances the clarity and detail of an image by increasing the contrast along edges and boundaries.

Concept

In grayscale images, each pixel value represents the intensity of light at that point. Image sharpening involves subtracting a smoothed version of the image from the original. This process accentuates edges and fine details, making them more prominent.

Example:

Consider a grayscale image represented as a matrix:

```
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 enhances edges and fine details:

```
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:

```
# 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

# 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)

```

1.9.1.3 Matrix Multiplication in Image Filtering (Convolution)

Matrix multiplication, specifically convolution in the context of image processing, is a fundamental operation used for various tasks such as smoothing, sharpening, edge detection, and more. Convolution involves applying a small matrix, known as a kernel or filter, to an image matrix. This process modifies the pixel values of the image based on the values in the kernel, effectively filtering the image.

Concept In grayscale images, each pixel value represents the intensity of light at that point. Convolution applies a kernel matrix over the image matrix to compute a weighted sum of neighborhood pixels. This weighted sum determines the new value of each pixel in the resulting filtered image.

Example:

Consider a grayscale image represented as a matrix:

```

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:

```
[[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:

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

Python Implementation

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

```
# 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)

```

1.9.1.4 Determinant: Image Transformation

Concept The determinant of a transformation matrix helps understand how transformations like scaling affect an image. A transformation matrix determines how an image is scaled, rotated, or sheared.

Example:

Here, we compute the determinant of a scaling matrix to understand how the scaling affects the image area.

```

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.

1.9.1.5 Rank: Image Rank and Data Compression

Concept The rank of a matrix indicates the number of linearly independent rows or columns. In image compression, matrix rank helps approximate an image with fewer data.

Example:

Here, we compute the rank of a matrix representing an image. A lower rank might indicate that the image can be approximated with fewer data.

```
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}")
```

1.10 Matrix Operations Using Python Libraries

1.10.1 Introduction

In this section, we will explore the computational aspects of basic matrix algebra using Python. We will utilize the `Sympy` library for symbolic mathematics, which allows us to perform matrix operations and convert results into `LaTeX` format. Additionally, the `Pillow` (PIL) library will be used for image manipulation to demonstrate practical applications of these matrix operations in digital image processing. By the end of this section, you'll understand how to implement matrix operations and apply them to real-world problems such as image blending, sharpening, filtering, and solving systems of equations.

1.10.2 Introduction to SymPy

`Sympy` is a powerful Python library designed for symbolic mathematics. It provides tools for algebraic operations, equation solving, and matrix handling in a symbolic form. This makes it ideal for educational purposes and theoretical work where exact results are needed.

1.10.2.1 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

```
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:

```

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)

```

Matrix Addition Result:

$$\begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

Example 2: Matrix Subtraction

Pseudocode:

```

# 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:

```

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)

```

Matrix Subtraction Result:

$$\begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}$$

LaTeX Code for Subtraction Result:

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

Example 3: Matrix Multiplication

Pseudocode:

```
# 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:

```
# 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)

```

Matrix Multiplication Result:

$$\begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

LaTeX Code for Multiplication Result:
 $\left[\begin{matrix} 19 & 22 \\ 43 & 50 \end{matrix} \right]$

Example 3: Matrix Multiplication

Pseudocode:

```

# 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 code for implementing the above pseudocode is shown below:

```

# 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)

```

Matrix Multiplication Result:

$$\begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

LaTeX Code for Multiplication Result:

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

Example 4: Matrix Power

Pseudocode:

```

# 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:

```
# 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)
```

Matrix Power Result:

$$\begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$$

LaTeX Code for Power Result:

```
\left[\begin{matrix} 7 & 10 \\ 15 & 22 \end{matrix}\right]
```

1.10.2.2 Introduction to PIL for Image Manipulation

The PIL (Python Imaging Library), now known as Pillow, provides essential tools for opening, manipulating, and saving various image file formats. In this session, we will use Pillow to perform image operations such as resizing and blending to demonstrate the practical applications of these matrix operations in digital image processing.

Matrix operations have significant applications in digital image processing. These operations can manipulate images in various ways, from blending to filtering. Below we will discuss how matrix addition, subtraction, and multiplication are used in real-time image processing tasks.

1. Matrix Addition: Image Blending

Matrix addition can be used to blend two images by adding their pixel values. This process can be straightforward or involve weighted blending.

Example 1: Simple Image Blending

```
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/thekekudi_cave_temple_in_pathanamthitta20131205062431_315_1.jpg' \
    ↴ , "input2.jpg")

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:

```
img1
```



img2



blended_img



Example 2: Weighted Image Blending

```
# 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
```



1.10.2.3 Matrix Subtraction: Image Sharpening

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

Example 1: Sharpening by Subtracting Blurred Image

```
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
```



1.10.2.4 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

```
# 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

```



Example 2: Applying a Gaussian Blur Filter

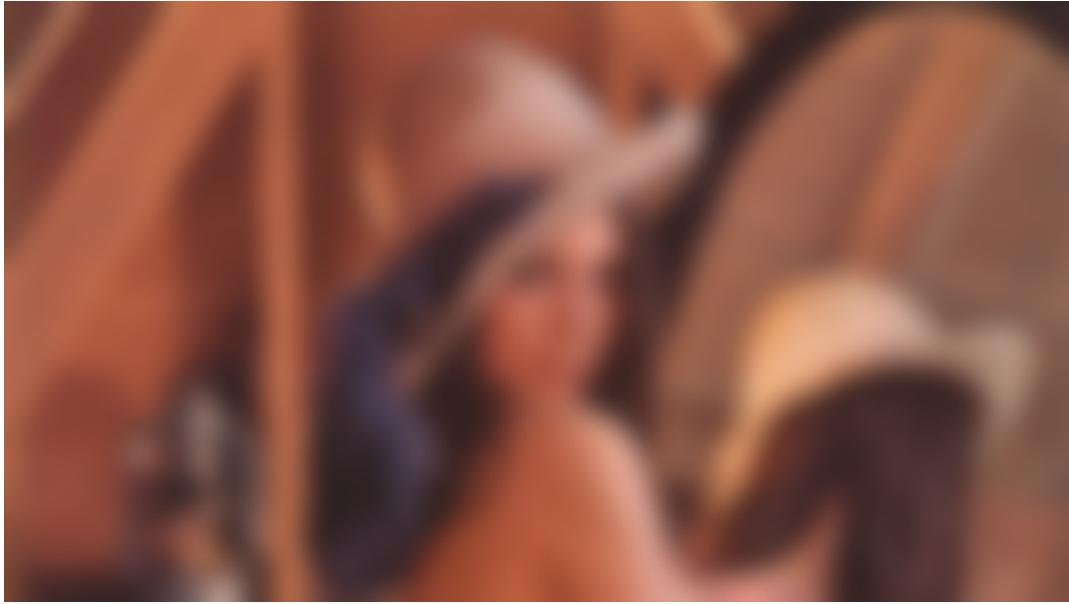
```

# 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
```



1.10.2.5 Solving Systems of Equations and Applications

Introduction

Solving systems of linear equations is crucial in various image processing tasks, such as image transformation, camera calibration, and object detection. In this section, we will demonstrate how to solve systems of linear equations using Python and explore practical applications in image processing.

Example 1: Solving a System of Equations

Consider the system:

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

Python Implementation

```

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)

```

Solution to the System of Equations:
 $\text{Matrix}([[17/7], [19/7]])$

1.11 Module review

1. Write Pseudocode for Matrix Addition.

- **Hint:** Use nested loops to iterate over rows and columns and sum elements.
- **Pseudocode:**

```

Initialize C as an m x n zero matrix
For i = 1 to m:
    For j = 1 to n:
        C[i][j] = A[i][j] + B[i][j]
End
Return C

```

2. Blend Grayscale Images A and B Using Weighted Blending ($\alpha = 0.70$).
- **Hint:** Use $C[i][j] = \alpha A[i][j] + (1 - \alpha)B[i][j]$.
3. Reduce a Matrix Into Row-Reduced Echelon Form and Find Its Row and Column Spaces.
- **Hint:** Use Gaussian elimination to achieve RREF.
4. Write Pseudocode for Row-Reduced Echelon Form.
- **Hint:** Use row operations to make pivots 1 and eliminate non-zero elements in pivot columns.
- **Pseudocode:**

```

For each row i in A:
    Normalize pivot to 1
    For each other row j:
        Subtract multiple of row i to make column i of row j zero
End

```

5. Differentiate Between Array and Dictionary in Python With Examples.

- **Hint:**
 - **Array:** Indexed collection of elements.
 - **Dictionary:** Key-value pair mapping.
- **Example:**

```

array = [1, 2, 3, 4]
dictionary = {"key1": "value1", "key2": "value2"}

```

6. Write Pseudocode to Transpose a Matrix.

- **Hint:** Swap rows with columns.

7. Perform Scalar Multiplication of a Matrix With a Scalar k.

- **Hint:** Multiply each element of A by k .

8. Create a Random Grayscale Image and Perform Pixel-wise Addition With Another Image.

- **Hint:** Use NumPy to generate and manipulate matrices.

9. Extract the Diagonal Elements of a Matrix and Find Their Sum.

- **Hint:** Diagonal elements are $A[i][i]$ for all i .

10. Discuss NumPy Arrays in Linear Algebra and How They Differ From Python Lists.

- **Hint:** NumPy arrays support efficient numerical operations, unlike general-purpose lists.

1.12 Conclusion

In this chapter, we transitioned from understanding fundamental matrix operations to applying them in practical scenarios, specifically in the realm of image processing. We began by covering essential matrix operations such as addition, subtraction, multiplication, and determinant calculations, providing both pseudocode and detailed explanations. This foundational knowledge was then translated into Python code, demonstrating how to perform these operations computationally.

We further explored the application of these matrix operations to real-world image processing tasks. By applying techniques such as image blending, sharpening, filtering, and transformation, we illustrated how theoretical concepts can be used to manipulate and enhance digital images effectively. These practical examples highlighted the significance of matrix operations in solving complex image processing challenges.

By integrating theoretical understanding with practical implementation, this chapter reinforced how matrix operations form the backbone of many image processing techniques. This blend of theory and practice equips you with essential skills for tackling advanced problems and developing innovative solutions in the field of image processing and beyond.

2 Transforming Linear Algebra to Computational Language

2.1 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.

2.2 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.

2.2.1 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 2.1. A bar plot of the total sales is shown in Fig 2.1.

Table 2.1: 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()
```

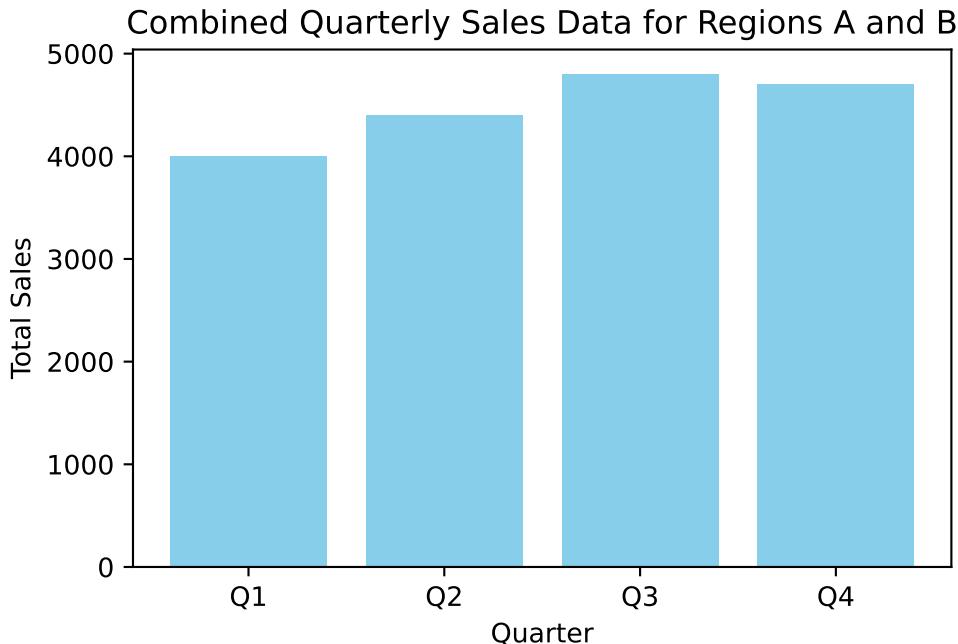


Figure 2.1: Computing Total Sales using Numpy aggregation method

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 2.2.

```

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()

```

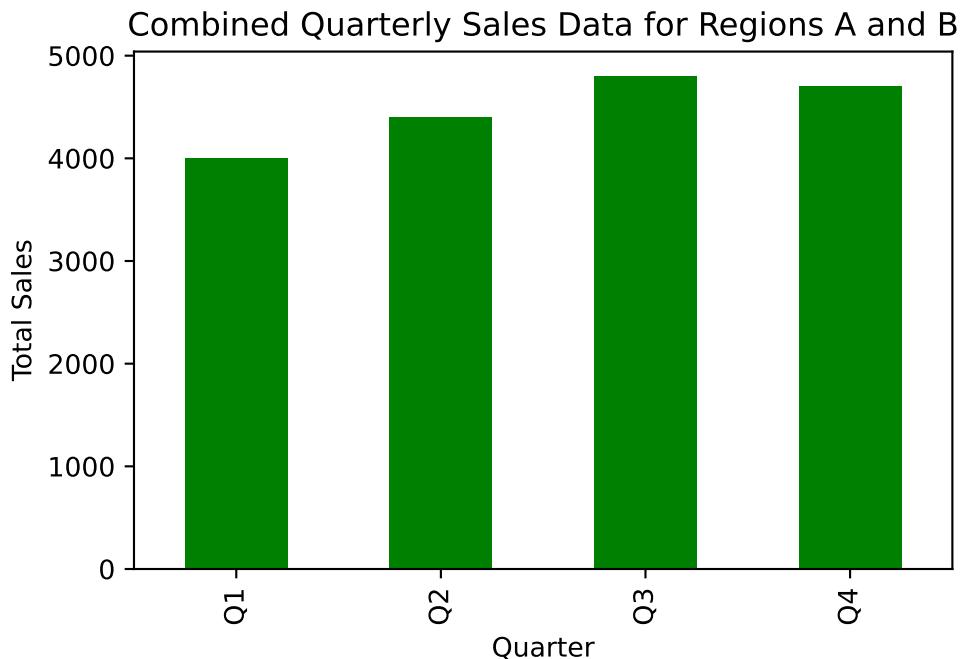


Figure 2.2: Computation of Total Sales using Pandas method

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 2.3.

```

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 2.3: 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 2.2: 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:

$$\min_{\max}(X) = \frac{X - \min(X)}{\max(X) - \min(X)}$$

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

$$\min_{\max}(80) \frac{80 - 80}{120 - 80} = 0$$

Similarly

$$\min_{\max}(100) \frac{100 - 80}{120 - 80} = 0.5$$

When we apply this formula to Metric A and Metric B, the scaled output from Table 2.2 will be as follows:

Table 2.3: 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 orginal table (Table 2.2) it is looked like Metric B is superior. But from the scaled table (Table 2.3), 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 2.4.

```
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()
```

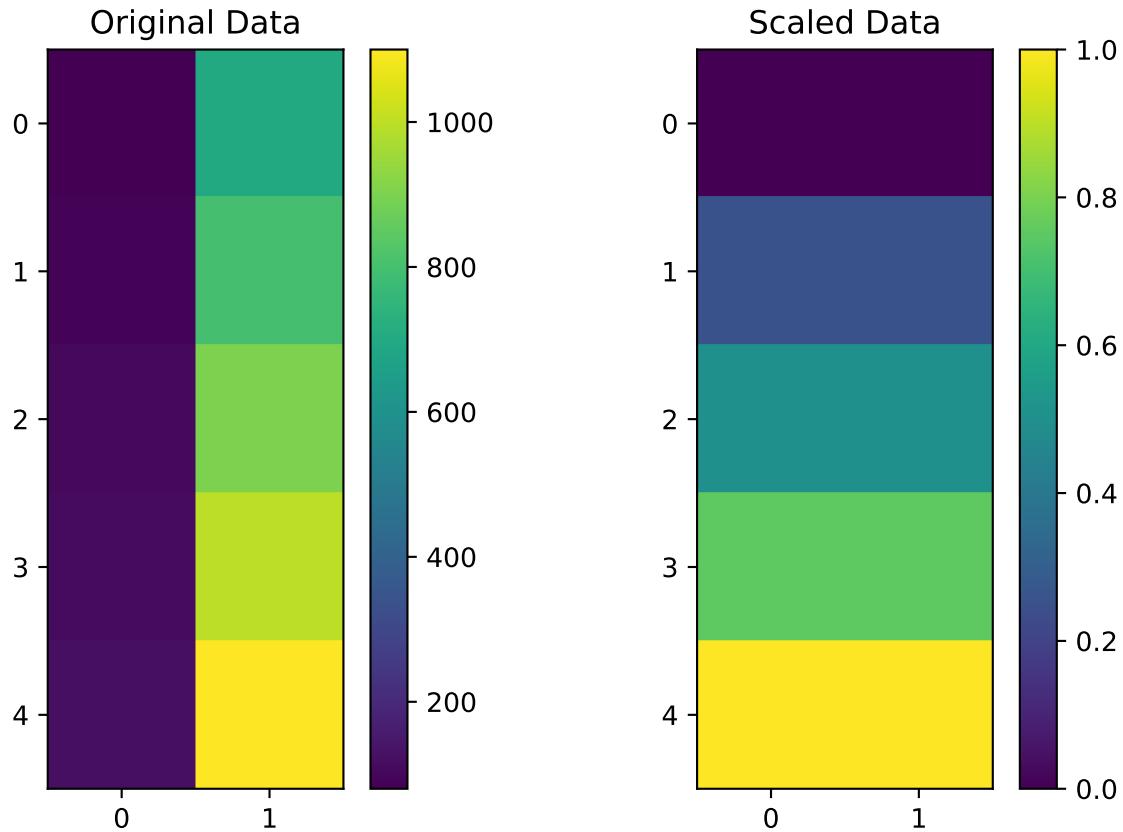


Figure 2.4: Total sales using `pandas` method

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 2.5.

```

# 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]]

```

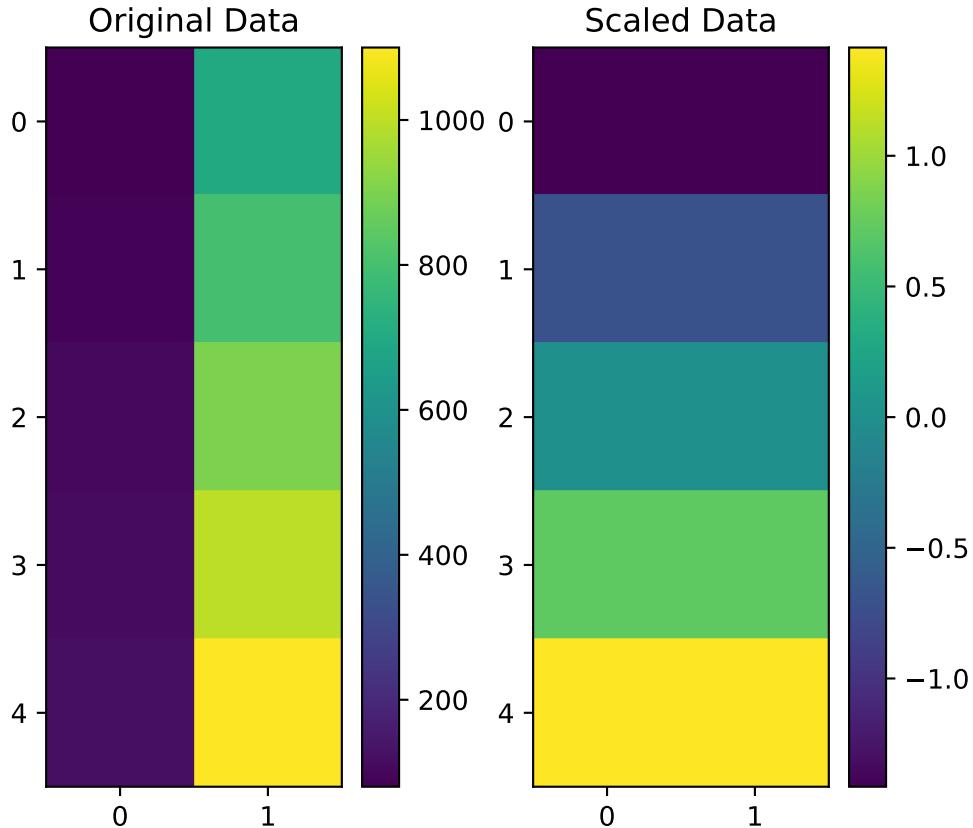


Figure 2.5: Min-max scaling using basic python

To understand the effect of standard scaling, let us consider Fig 2.6. 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()

```

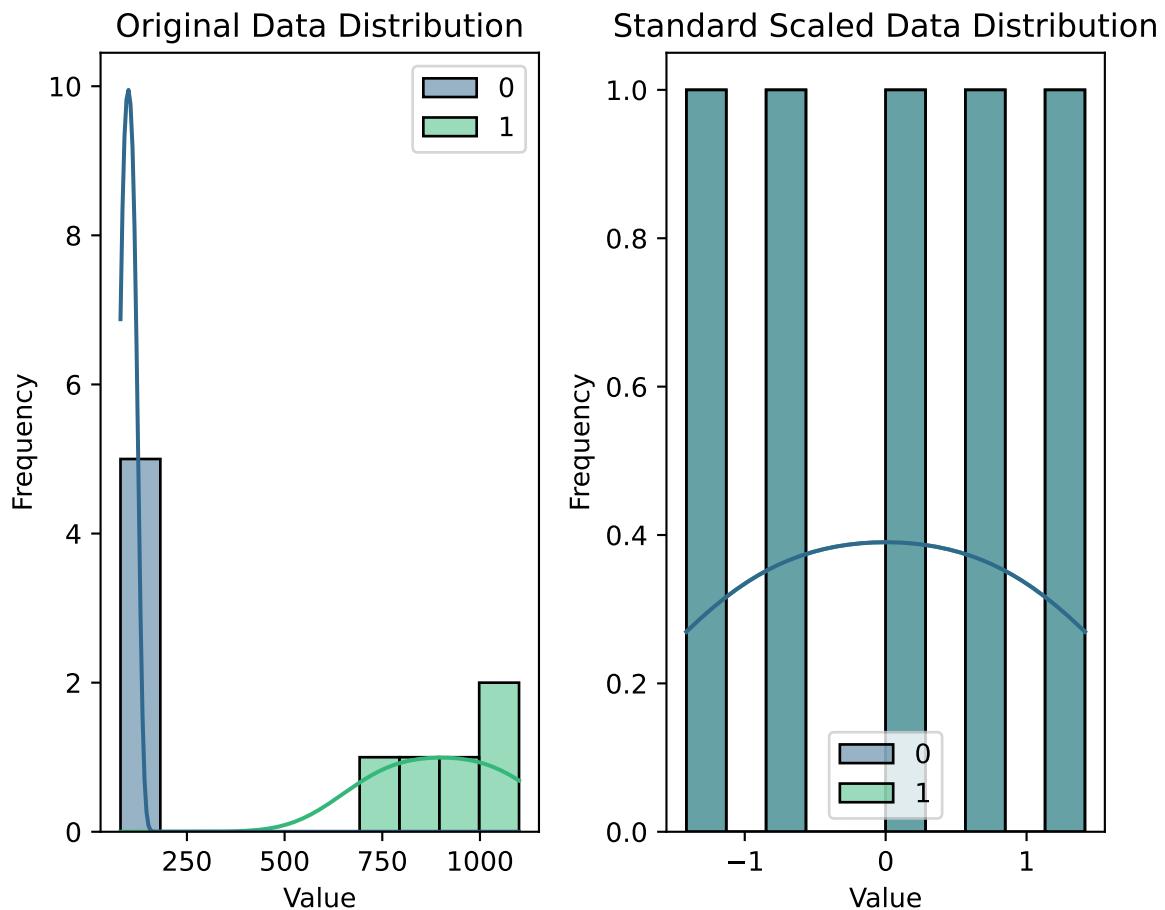


Figure 2.6: Impact of standard scaling on the distribution

A scatter plot showing the compare the impact of scaling on the given distribution is shown in Fig 2.7.

```
# 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()
```

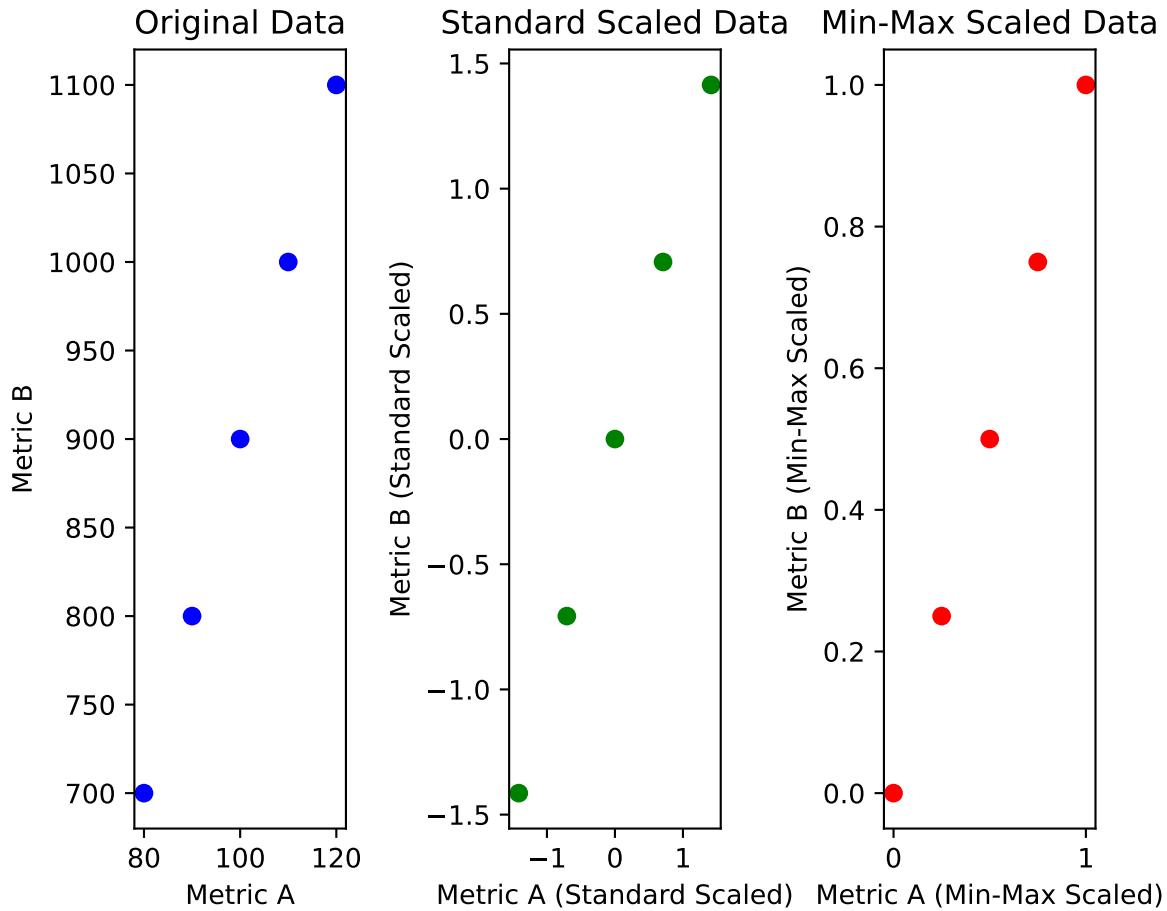


Figure 2.7: Comparison of impact of scaling on the distribution

From the Fig 2.7, 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 2.8. 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()

```

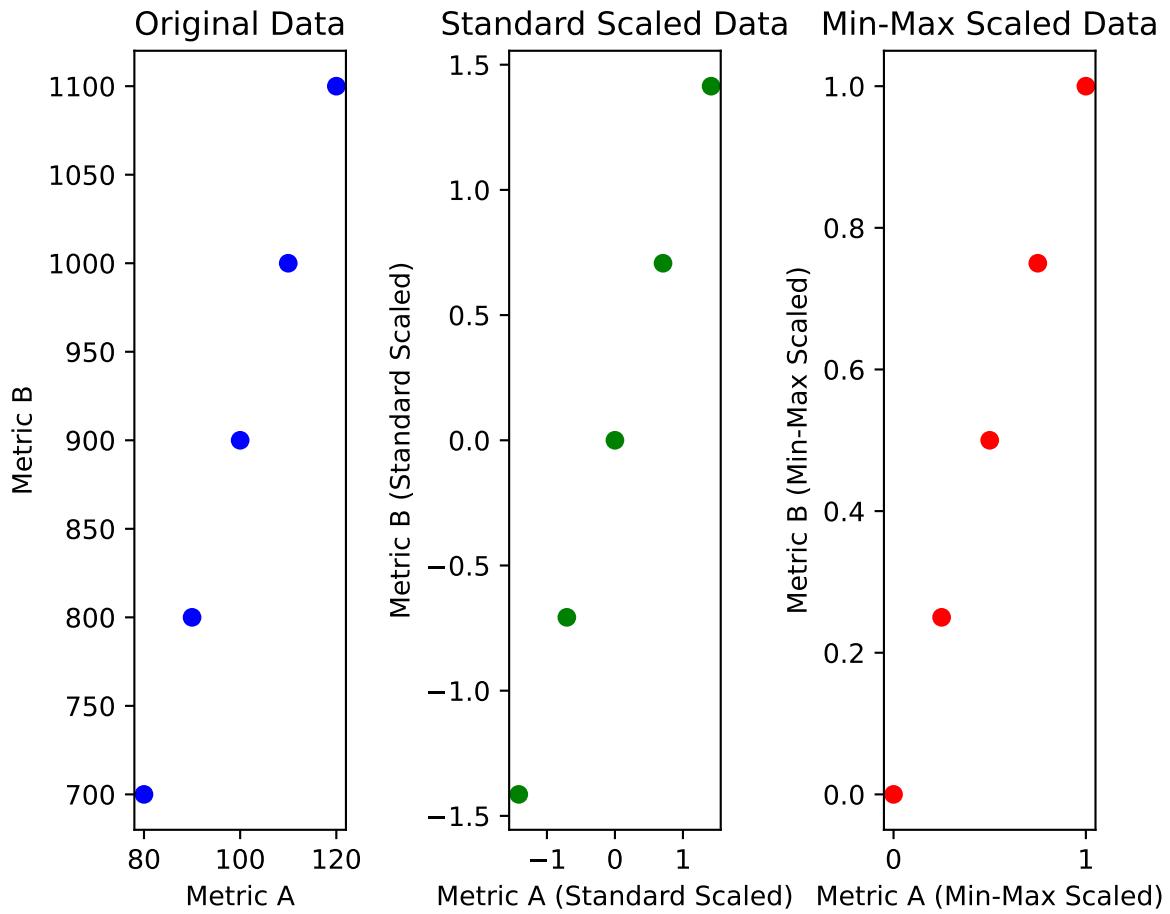


Figure 2.8: Comparison of Min-max and standard scalings with original data

2.2.2 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.

2.2.2.1 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 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 2.9.

```

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()

```

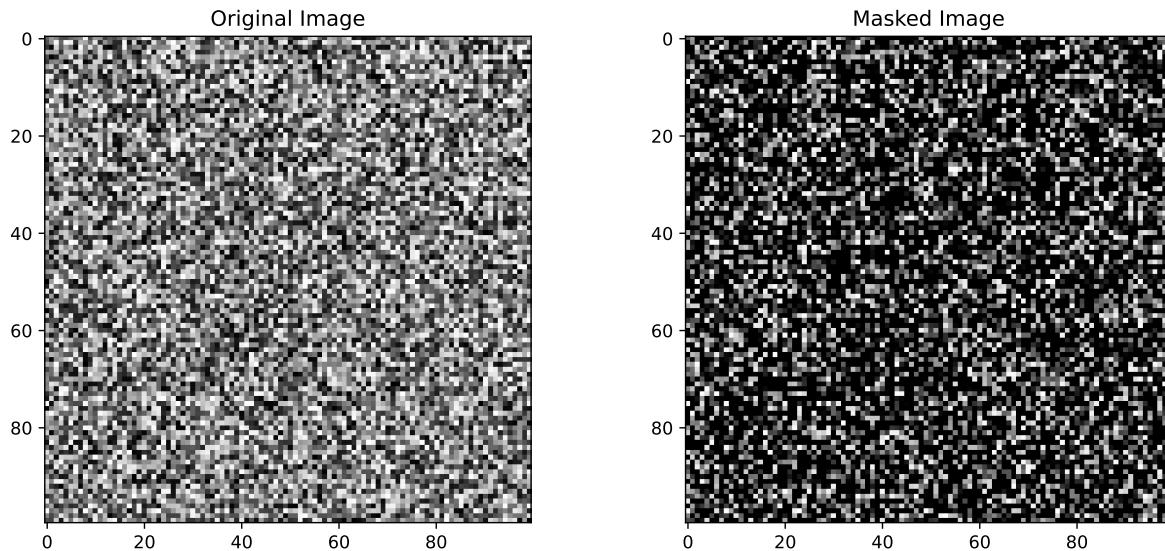


Figure 2.9: Demonstration of Masking in DIP using Hadamard Product

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:

```
[[0.21483693 0.63643729 0.1640106 ... 0.02561905 0.09535167 0.31433024]
 [0.82910948 0.16571217 0.66594546 ... 0.7116377 0.9727802 0.51264758]
 [0.32414978 0.28995349 0.12031392 ... 0.15894448 0.75138141 0.32713744]
 ...
 [0.87266776 0.09486363 0.4808236 ... 0.03146275 0.2415561 0.20886348]
 [0.74710401 0.53470408 0.35241462 ... 0.33015554 0.9989092 0.40753009]
 [0.19149587 0.34822228 0.4458629 ... 0.95356444 0.80168576 0.59783154]]
```

Dropout Mask:

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

Dropped Activations:

```
[[0.21483693 0.63643729 0. ... 0. 0. 0.31433024]]
```

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

```
[0.82910948 0.          0.          ... 0.          0.          0.51264758]
[0.32414978 0.28995349 0.          ... 0.15894448 0.75138141 0.32713744]
...
[0.          0.09486363 0.4808236 ... 0.03146275 0.2415561  0.20886348]
[0.74710401 0.53470408 0.          ... 0.          0.9989092  0.          ]
[0.          0.34822228 0.          ... 0.          0.80168576 0.59783154]]
```

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([[0.0790221014936448, 0.00718531166293288, 0.0166996377344117, 0.00520561939844207, 0.000520561939844207], [0.00718531166293288, 0.0166996377344117, 0.00520561939844207, 0.000520561939844207, 0.000520561939844207], [0.0166996377344117, 0.00520561939844207, 0.000520561939844207, 0.000520561939844207, 0.000520561939844207], [0.00520561939844207, 0.000520561939844207, 0.000520561939844207, 0.000520561939844207, 0.000520561939844207], [0.000520561939844207, 0.000520561939844207, 0.000520561939844207, 0.000520561939844207, 0.000520561939844207]])

Scaling Factors:

Matrix([[0.438367433833123, 0.0626773883983843, 0.910115630373666, 0.0176575403529717, 0.4650000000000001], [0.0626773883983843, 0.910115630373666, 0.0176575403529717, 0.4650000000000001, 0.000520561939844207], [0.910115630373666, 0.0176575403529717, 0.4650000000000001, 0.000520561939844207, 0.000520561939844207], [0.0176575403529717, 0.4650000000000001, 0.000520561939844207, 0.000520561939844207, 0.000520561939844207], [0.4650000000000001, 0.000520561939844207, 0.000520561939844207, 0.000520561939844207, 0.000520561939844207]])

Scaled Covariance Matrix:

Matrix([[0.0544300985573108, 0.0269188103439923, 0.0823955469804108, 0.0158508897355430, 0.000520561939844207], [0.0269188103439923, 0.0823955469804108, 0.0158508897355430, 0.000520561939844207, 0.000520561939844207], [0.0823955469804108, 0.0158508897355430, 0.000520561939844207, 0.000520561939844207, 0.000520561939844207], [0.0158508897355430, 0.000520561939844207, 0.000520561939844207, 0.000520561939844207, 0.000520561939844207], [0.000520561939844207, 0.000520561939844207, 0.000520561939844207, 0.000520561939844207, 0.000520561939844207]])

2.2.2.2 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:

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

2.2.2.3 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 below.

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:

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

2.2.2.4 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 $A = (1 \ 2 \ 3)$ and $B = (4 \ 5 \ 6)$, 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 fot 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

```

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):

386.3044855527312

Similarity of signals:

0.8702338627622932

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([2.89165172, 2.08918362, 1.55601815, 1.44236323, 3.40699879,
       2.84807247, 2.26731485, 2.46982449, 2.2899128 , 2.56753326,
       1.22926403, 2.70436052, 1.97289096, 2.56851253, 2.55385748,
       1.92265134, 3.01357752, 2.38393559, 3.41387825, 1.88689726,
       2.04392636, 2.96433464, 2.37252747, 3.97184718, 1.96680413,
       3.33656637, 2.11220016, 2.61513997, 2.76786909, 1.74075651,
       1.91839649, 2.54846057, 2.85069027, 1.84310853, 2.16813079,
       3.07432487, 1.80016765, 1.84702094, 3.20090016, 1.44002096,
       3.31871771, 1.95864186, 1.90897797, 2.94610451, 1.86605326,
       1.26562938, 2.37764562, 2.12074577, 3.13957921, 3.35120285,
       3.37781332, 2.31719735, 2.18708433, 1.83888247, 1.47957327,
       1.86854415, 1.55335521, 2.69159557, 1.56916182, 2.23125577,
       2.56902465, 3.09698601, 1.82605486, 1.72654174, 3.30271371,
       1.70723132, 2.53945302, 1.73910653, 3.05362647, 2.11144697,
       2.65170652, 0.87911842, 2.1677809 , 3.28300058, 3.36245039,
       3.06305754, 3.33353194, 2.10306988, 1.93529557, 1.28040658,
       2.25155266, 2.75280138, 2.91985166, 3.12861766, 2.93057724,
       2.67334454, 4.20012845, 2.22660831, 2.54153042, 2.29615488,
       0.86481013, 3.11484692, 3.2745528 , 1.98949285, 3.24098065,
       1.6876918 , 2.70078153, 2.48652566, 2.39127066, 2.16415242])
```

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.

```
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
```

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

```

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:

0.0854440533421158	0.000151030261045823	-0.000570342606188518	-0.00313020728710101	-0.00
0.000151030261045823	0.0711324611510939	0.006667183584056	-0.00385363810045373	-0.01
-0.000570342606188518	0.006667183584056	0.0783129281906814	-0.00588563141830825	0.005
-0.00313020728710101	-0.00385363810045373	-0.00588563141830825	0.0798838166499781	-0.00
-0.00437895019166552	-0.0114369796582531	0.00522547200908714	-0.00333917863973172	0.083
0.00638091620359299	0.00073134845699356	0.00618532251276825	0.00257622852488847	-0.000
0.0131390070055006	0.0121287250603349	0.014614893149089	-0.0120051835912404	-0.00
0.00775498491266611	0.0134933619121732	0.0125110531175336	-0.00972370455486007	-0.00
0.00626305768643138	-0.0121630567928363	-0.000679598306186701	0.0101460663094745	0.013
-0.0160534771087747	-0.0125474923555661	-0.015675098439215	-0.00498152136814472	0.001

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

2.2.2.5 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 ,

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

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:

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

2.2.2.6 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:

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

$$B = [3 \ 4]$$

Solution:

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

Problem 2:

Find the outer product of:

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

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

Solution:

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

Problem 3:

Find the outer product of:

$$A = [1 \ 2]$$

$$B = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

Solution:

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

Problem 4:

Find the outer product of:

$$A = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$B = [1 \ -1]$$

Solution:

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

Problem 5:

Find the outer product of:

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

$$B = [5 \ -2]$$

Solution:

$$A \otimes B = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \otimes [5 \ -2]$$

$$= \begin{bmatrix} 2 \cdot 5 & 2 \cdot -2 \\ 3 \cdot 5 & 3 \cdot -2 \end{bmatrix}$$

$$= \begin{bmatrix} 10 & -4 \\ 15 & -6 \end{bmatrix}$$

Problem 6:

Find the outer product of:

$$A = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

$$B = [2 \ -1 \ 0]$$

Solution:

$$A \otimes B = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \otimes [2 \ -1 \ 0]$$

$$= \begin{bmatrix} 1 \cdot 2 & 1 \cdot -1 & 1 \cdot 0 \\ 0 \cdot 2 & 0 \cdot -1 & 0 \cdot 0 \\ 1 \cdot 2 & 1 \cdot -1 & 1 \cdot 0 \end{bmatrix}$$

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

Problem 7:

Find the outer product of:

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

$$B = \begin{bmatrix} 2 & 0 \\ 3 & -1 \end{bmatrix}$$

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:

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

$$B = [1 \quad -2 \quad 3]$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 3 \\ 4 \end{bmatrix} \otimes [1 \quad -2 \quad 3] \\ &= \begin{bmatrix} 3 \cdot 1 & 3 \cdot -2 & 3 \cdot 3 \\ 4 \cdot 1 & 4 \cdot -2 & 4 \cdot 3 \end{bmatrix} \\ &= \begin{bmatrix} 3 & -6 & 9 \\ 4 & -8 & 12 \end{bmatrix} \end{aligned}$$

Problem 9:

Find the outer product of:

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

$$B = [4 \quad -2]$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 2 \\ 3 \\ -1 \end{bmatrix} \otimes [4 \quad -2] \\ &= \begin{bmatrix} 2 \cdot 4 & 2 \cdot -2 \\ 3 \cdot 4 & 3 \cdot -2 \\ -1 \cdot 4 & -1 \cdot -2 \end{bmatrix} \\ &= \begin{bmatrix} 8 & -4 \\ 12 & -6 \\ -4 & 2 \end{bmatrix} \end{aligned}$$

Problem 10:

Find the outer product of:

$$A = \begin{bmatrix} 0 \\ 5 \end{bmatrix}$$

$$B = [3 \quad 1]$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 0 \\ 5 \end{bmatrix} \otimes [3 \quad 1] \\ &= \begin{bmatrix} 0 \cdot 3 & 0 \cdot 1 \\ 5 \cdot 3 & 5 \cdot 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 \\ 15 & 5 \end{bmatrix} \end{aligned}$$

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)

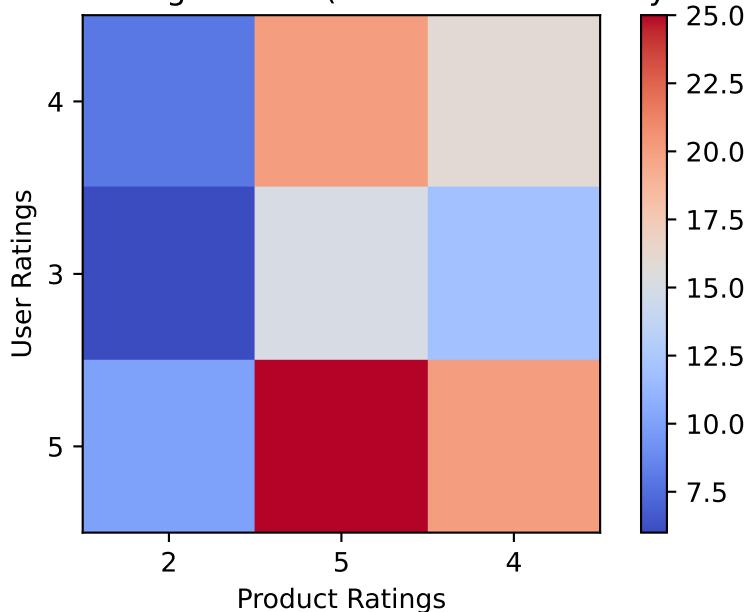
# Plot the result
plt.imshow(predicted_ratings, cmap='coolwarm', interpolation='nearest')
plt.colorbar()
plt.title('Predicted Ratings Matrix (Recommendation System)')
plt.xlabel('Product Ratings')
plt.ylabel('User Ratings')
plt.xticks(ticks=np.arange(len(product_ratings)), labels=product_ratings)
```

```
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

```

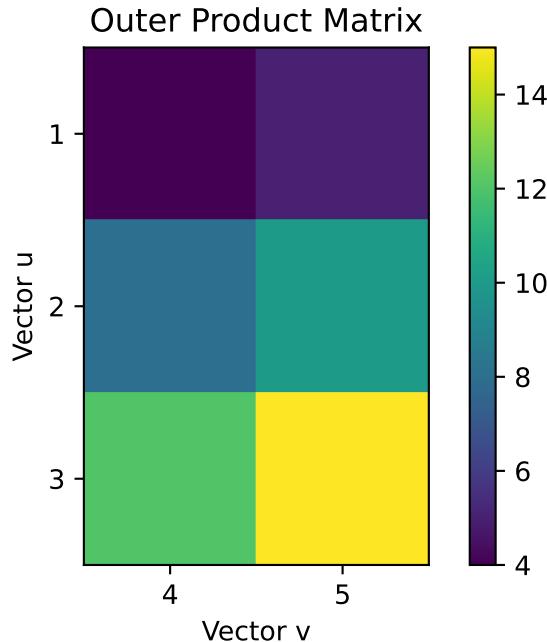


Figure 2.10: Demonstration of Outer Product and its Properties

2.2.2.7 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:

$$A \otimes B = A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ a_{21}B & a_{22}B & \cdots & a_{2n}B \\ \vdots & \vdots & \ddots & vdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{bmatrix}$$

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 $C \in \mathbb{R}^{r \times s}$:

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

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 $D \in \mathbb{R}^{t \times u}$:

$$(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$$

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 :

$$\|C\|_F = \sqrt{\sum_{i=1}^4 \sum_{j=1}^4 |c_{ij}|^2}$$

$$\|C\|_F = \sqrt{0^2 + 5^2 + 0^2 + 10^2 + 6^2 + 7^2 + 12^2 + 14^2 + 0^2 + 15^2 + 0^2 + 20^2 + 18^2 + 21^2 + 24^2 + 28^2}$$

$$\|C\|_F = \sqrt{0 + 25 + 0 + 100 + 36 + 49 + 144 + 196 + 0 + 225 + 0 + 400 + 324 + 441 + 576 + 784}$$

$$\begin{aligned} \|C\|_F &= \sqrt{2896} \\ \|C\|_F &\approx 53.87 \end{aligned}$$

2.2.2.8 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:

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

Solution:

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

Problem 4:

Find the Kronecker product of:

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

Solution:

$$\begin{aligned}
A \otimes B &= [0 \ 1] \otimes \begin{bmatrix} 1 & -1 \\ 2 & 0 \end{bmatrix} \\
&= \left[0 \cdot \begin{bmatrix} 1 & -1 \\ 2 & 0 \end{bmatrix} \quad 1 \cdot \begin{bmatrix} 1 & -1 \\ 2 & 0 \end{bmatrix} \right] \\
&= \begin{bmatrix} 0 & 0 & 1 & -1 \\ 0 & 0 & 2 & 0 \end{bmatrix}
\end{aligned}$$

Problem 5:

Find the Kronecker product of:

$$\begin{aligned}
A &= \begin{bmatrix} 2 \\ 3 \end{bmatrix} \\
B &= [4 \ -2]
\end{aligned}$$

Solution:

$$\begin{aligned}
A \otimes B &= \begin{bmatrix} 2 \\ 3 \end{bmatrix} \otimes [4 \ -2] \\
&= \left[2 \cdot \begin{bmatrix} 4 & -2 \end{bmatrix} \quad 3 \cdot \begin{bmatrix} 4 & -2 \end{bmatrix} \right] \\
&= \begin{bmatrix} 8 & -4 \\ 12 & -6 \end{bmatrix}
\end{aligned}$$

Problem 6:

Find the Kronecker product of:

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

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:

$$A = [0 \ 1]$$

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

Solution:

$$\begin{aligned}
A \otimes B &= [0 \ 1] \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\
&= \left[0 \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad 1 \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right] \\
&= \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}
\end{aligned}$$

Problem 9:

Find the Kronecker product of:

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

Solution:

$$\begin{aligned}
A \otimes B &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \\
&= \left[1 \cdot \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad 0 \cdot \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \right] \\
&= \left[0 \cdot \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad 1 \cdot \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \right] \\
&= \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}
\end{aligned}$$

Problem 10:

Find the Kronecker product of:

$$\begin{aligned}
A &= \begin{bmatrix} 2 & -1 \\ 3 & 4 \end{bmatrix} \\
B &= \begin{bmatrix} 0 & 5 \\ -2 & 3 \end{bmatrix}
\end{aligned}$$

Solution:

$$\begin{aligned} A \otimes B &= \begin{bmatrix} 2 & -1 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 0 & 5 \\ -2 & 3 \end{bmatrix} \\ &= \begin{bmatrix} 2 \cdot \begin{bmatrix} 0 & 5 \\ -2 & 3 \end{bmatrix} & -1 \cdot \begin{bmatrix} 0 & 5 \\ -2 & 3 \end{bmatrix} \\ 3 \cdot \begin{bmatrix} 0 & 5 \\ -2 & 3 \end{bmatrix} & 4 \cdot \begin{bmatrix} 0 & 5 \\ -2 & 3 \end{bmatrix} \end{bmatrix} \\ &= \begin{bmatrix} 0 & 10 & 0 & -5 \\ -4 & 6 & 2 & -3 \\ 0 & 15 & 0 & 20 \\ -6 & 9 & -8 & 12 \end{bmatrix} \end{aligned}$$

2.2.2.9 Connection Between Outer Product and Kronecker Product

1. Conceptual Connection:

- The **outer product** is a special case of the **Kronecker product**. Specifically, if **A** is a column vector and **B** is a row vector, then **A** is a $m \times 1$ matrix and **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 **A** and **B**, the Kronecker product involves taking the outer product of each element of **A** with the entire matrix **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 $\mathbf{u} \otimes \mathbf{v}$.

i 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.

2.2.2.10 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 $\mathbf{B}_{1,:} = [0 \ 1]$:

$$\mathbf{A}_{:,1} \otimes \mathbf{B}_{1,:} = \begin{bmatrix} 0 & 1 \\ 0 & 3 \end{bmatrix}$$

- 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 2.4 and Table 2.5 .

Table 2.4: User Preference

User/Item	Electronics	Clothing	Books
User 1	5	3	4
User 2	2	4	5
User 3	3	4	4

Table 2.5: 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
User 1 Electronics	3	0	0	4
User 1 Clothing	0	3	3	0
User 1 Books	0	3	3	0
User 2 Electronics	4	0	0	5
User 2 Clothing	0	4	4	0
User 2 Books	0	4	4	0
User 3 Electronics	4	0	0	4
User 3 Clothing	0	4	4	0
User 3 Books	0	4	4	0
	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_F3  
Electronics_F2  
Books_F3
```

A simple visualization of this recommendation system is shown in Fig 2.11.

```
# 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)
```

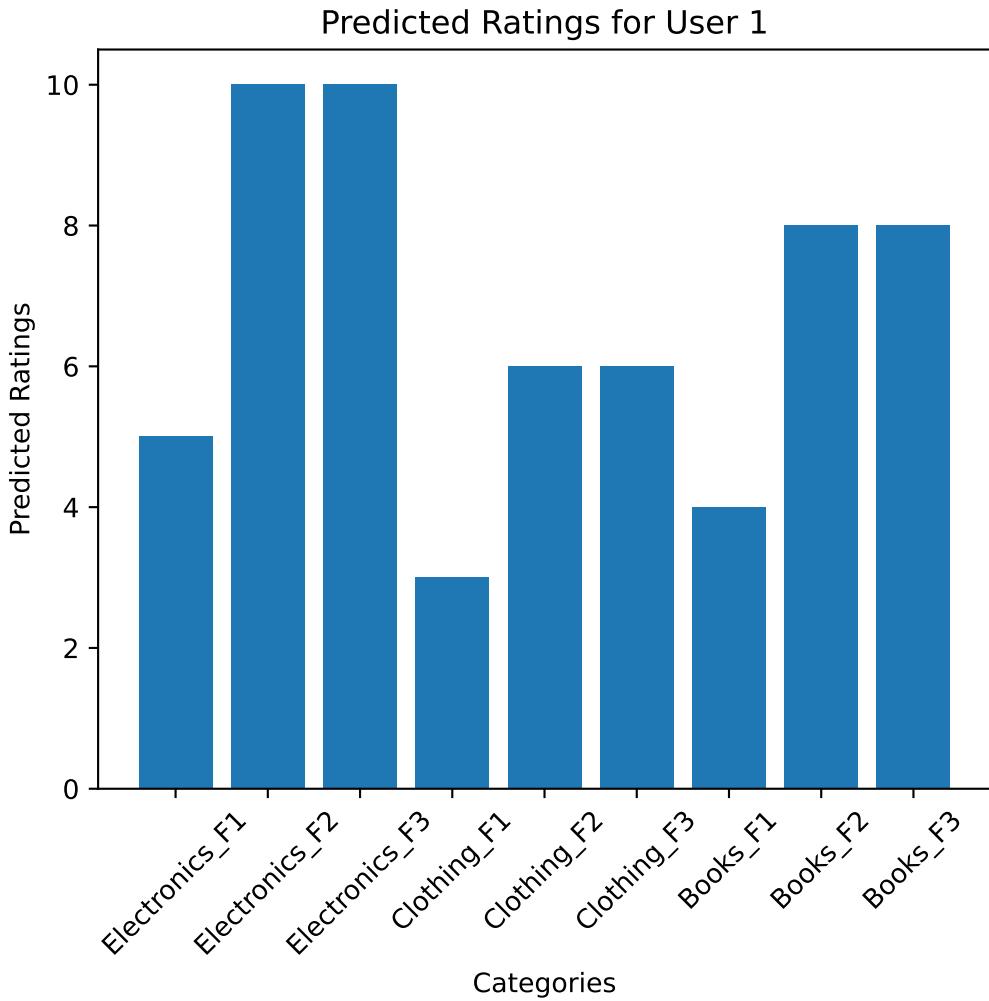


Figure 2.11: EDA for the Recommendation System

This micro project illustrate one of the popular use of Kronecker product on ML application.

2.2.3 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.

2.2.3.1 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 2.6.

Table 2.6: 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:

$$\begin{aligned} A &= CR \\ &= \begin{bmatrix} 1 & 4 \\ 3 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \end{aligned}$$

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 decompostion denote 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 thereby 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 correspondence 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 versa 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 represented 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 $x = [1 \ 1 \ -1]^T$.

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.

2.2.4 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 2.1 (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.

2.2.5 Fundamental Subspaces

In section ([note-ortho?](#)), we have seen that for any matrix A , there are 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 2.1 (Four Fundamental Subspaces). Let $T : \mathbb{R}^n \rightarrow \mathbb{R}^m$ 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- 2.12.

A video session from Strang’s session is here:

<https://youtu.be/rwLOfdfc4dw?si=DsJb8KJTF05hHc76>

2.2.5.1 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 .

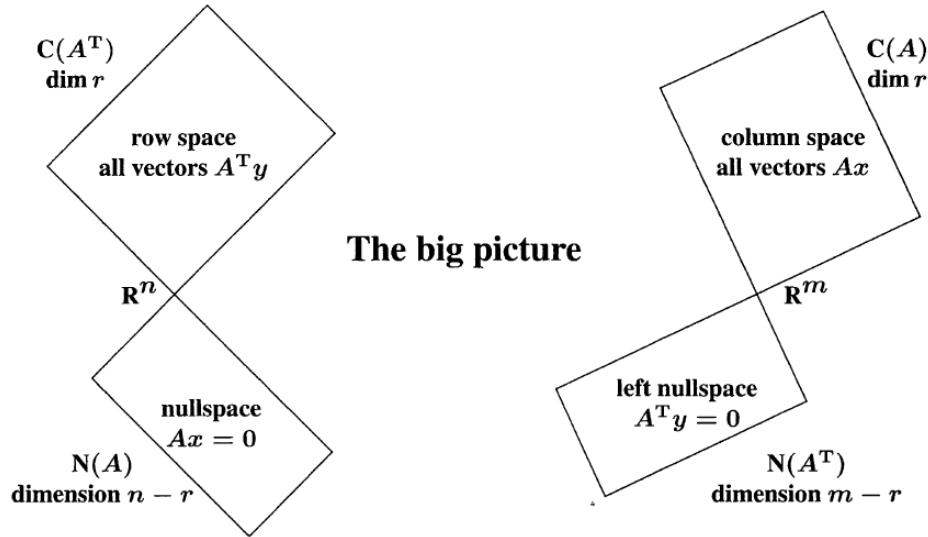


Figure 2.12: The Big Picture of Fundamental Subspaces

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	$\ker(A)$	$n - r$	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
----------------------------	----------------	-------------	---------	-----------------------------

2.2.5.2 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

```

2.2.5.3 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:
[[7.0000000e+00 8.0000000e+00 9.0000000e+00]
 [0.0000000e+00 8.57142857e-01 1.71428571e+00]
 [0.0000000e+00 5.55111512e-17 1.11022302e-16]]
```

Column Space of A:

```
[[1 2]
 [4 5]
 [7 8]]
```

2.2.5.4 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 2.2 (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 $Ax = b$, where A is an $m \times n$ matrix and 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 $[A|b]$. Mathematically, this can be expressed as:

$$\text{rank}(A) = \text{rank}([A|b])$$

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:

$$\text{rank}(A) \neq \text{rank}([A|b])$$

In this case, the equations represent parallel or conflicting constraints that cannot be satisfied simultaneously.

i 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 $X_N \in N(A)$.

2.2.5.5 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
            return "Inconsistent (rank mismatch)", None, None
    else:
        # Singular matrix
        return "Singular matrix (rank < n)", None, None
```

```

        # 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]]

2.3 Module review

1. Define the Hadamard Product in Linear Algebra and Provide a Suitable Example.

- **Hint:** Element-wise product of matrices of the same dimensions.

- **Example:** For $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$, $A \circ B = \begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix}$.

2. Give Two Applications of the Hadamard Product in Machine Learning.

- **Hint:** Gradient updates and feature scaling in deep learning.

3. Find the Outer Product of Two Vectors $S_1 = [1, 2, 7]$ and $S_2 = \begin{bmatrix} 7 \\ 2 \\ 1 \end{bmatrix}$.
- **Hint:** Compute $S_1 \cdot S_2^T$.
4. Define and Differentiate Between the Dot Product and the Outer Product of Two Vectors.
- **Hint:** Dot product results in a scalar; outer product results in a matrix.
5. Write a Pseudocode to Compute the Hadamard Product of Two Matrices.
- **Hint:** Use nested loops for element-wise multiplication.
6. Compute the Row Norms of Matrix $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$.
- **Hint:** Use $\|A_{i*}\| = \sqrt{\sum_j A_{ij}^2}$.
7. Find the Column Norms of Matrix $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$.
- **Hint:** Use $\|A_{*j}\| = \sqrt{\sum_i A_{ij}^2}$.
8. Compute the Frobenius Norm of Matrix $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$.
- **Hint:** $\|A\|_F = \sqrt{\sum_{i,j} A_{ij}^2}$.
9. Prove the Hadamard Product Is Commutative and Associative.
- **Hint:** Based on properties of element-wise operations.
10. Explain the Use of the Hadamard Product in Convolutional Neural Networks.
- **Hint:** Used in depth-wise convolutions and attention mechanisms.
11. For Vectors $u = [1, 2, 3]$ and $v = [4, 5, 6]$, Compute $u \circ v$ and $u \cdot v$.
- **Hint:** Hadamard product is element-wise, dot product is summation.
12. Write Pseudocode for Outer Product of Two Vectors.
- **Hint:** Compute $u_i \cdot v_j$ for all i, j .
13. Discuss the Role of the Outer Product in Tensor Decomposition.
- **Hint:** Used to represent rank-1 tensors.
14. Find Hadamard Product of $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $B = \begin{bmatrix} 2 & 0 \\ 1 & 5 \end{bmatrix}$.

- **Hint:** Multiply corresponding elements.
15. Use Python to Compute Outer Product of $x = [1, 2]$ and $y = [3, 4]$.
- **Hint:** Use NumPy's `np.outer(x, y)`.
16. Write all the fundamental subspaces of $A = \begin{bmatrix} 1 & 2 & 5 \\ 3 & 4 & 10 \\ 5 & 6 & 16 \end{bmatrix}$.
- **Hint:** Determine the column space, null space, row space, and left null space using Gaussian elimination and rank of A .
17. In a recommendation system, the user preference is $u = [4, 3, 5]$ and the item score is $v = [2, 5, 4]$. Find the user-item interaction score and its Frobenius norm.
- **Hint:** Compute the outer product $u \cdot v^T$ and the Frobenius norm $\|u \cdot v^T\|_F = \sqrt{\sum u_i v_j^2}$.
18. Verify the rank-nullity theorem for $A = \begin{bmatrix} 7 & -3 & 5 \\ 9 & 11 & 2 \\ 16 & 8 & 7 \end{bmatrix}$.
- **Hint:** Compute the rank of A , the dimension of its null space, and verify $\text{rank}(A) + \text{nullity}(A) = \text{number of columns of } A$.
19. Define the Kronecker product of two matrices. Find the Kronecker product of $A = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix}$ and $B = \begin{bmatrix} 1 & 0 & 1 & 8 \\ 2 & -2 & 2 & -2 \\ -4 & 0 & 3 & -1 \end{bmatrix}$ in block matrix form.
- **Hint:** Use the definition $A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B \\ a_{21}B & a_{22}B \end{bmatrix}$.
20. Explain and compute the rank of the Kronecker product of two matrices \$A\$ and \$B\$, where \$A\$ is 2×2 and \$B\$ is 3×3 .
- **Hint:** Use the property $\text{rank}(A \otimes B) = \text{rank}(A) \cdot \text{rank}(B)$.
21. Find the row space and column space of $A = \begin{bmatrix} 2 & 4 & 6 \\ 1 & 2 & 3 \\ 0 & 0 & 0 \end{bmatrix}$.
- **Hint:** Row space is spanned by independent rows; column space is spanned by independent columns.
22. Determine if $A = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$ is invertible. If not, explain why using its fundamental subspaces.

- **Hint:** Check if the null space is trivial or if the determinant of A is zero.
23. Write a pseudocode to calculate the Frobenius norm of any $m \times n$ matrix.
- **Hint:** Use $\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n A_{ij}^2}$.
24. Find the projection of $b = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix}$ onto the column space of $A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$.
- **Hint:** Use $P = A(A^T A)^{-1} A^T b$.
25. Write a pseudocode to compute the outer product of two vectors x and y .
- **Hint:** Use a nested loop or NumPy's `np.outer()` function to compute the product.

3 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.

3.1 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.

3.1.1 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.

3.2 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`

3.2.0.1 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)
```

3.2.1 Define different types of numpy arrays

As the first step to understand different types of arrays in NumPy let us consider the following examples.

3.2.1.1 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.

3.2.1.2 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.

3.2.1.3 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.

3.2.1.4 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.

3.2.1.5 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.

3.2.1.6 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. :::

3.2.1.7 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.

3.2.1.8 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)
```

```
[[0.57387774 0.10857888 0.05443869]
 [0.14123473 0.34655442 0.28668155]
 [0.37835477 0.91975636 0.40674866]]
```

`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.

3.2.2 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.

3.2.3 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.

3.2.3.1 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.

3.2.3.2 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)
```

```
[[[2 8 3 0 7]
 [7 6 6 7 8]
 [9 5 4 3 0]
 [8 4 8 5 4]]]
```

```
[[5 3 4 4 1]
 [6 0 2 5 0]
 [2 5 5 5 7]
 [0 8 1 4 1]]]
```

```
[[8 9 8 8 7]
 [7 6 8 1 6]
 [5 9 2 1 8]
 [9 0 5 4 3]]]
```

```
[[[1 4 4 4 3]
 [0 6 3 9 4]
 [3 1 4 8 4]
 [3 7 1 9 2]]]
```

```
[[6 9 3 9 4]
 [6 6 6 6 5]
 [2 7 3 2 4]
 [0 0 3 0 9]]]
```

```
[[3 4 1 2 9]
 [8 9 1 1 1]
 [8 0 2 1 5]
 [1 0 9 2 2]]]
```

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.

3.2.3.3 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
```

3.2.4 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
```

3.2.4.1 Acessing 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 third column
```

9

```
x2[2, -1] ## access the element in the 3rd row and last column
```

1

3.2.4.2 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!

3.2.4.3 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[::2,::2]) # slice alternate elements in first three rows and first
                     ← three columns
```

```
[[1 3]
 [5 7]]
```

i 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]
```

3.2.4.4 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]]
```

3.2.4.5 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]]
```

i 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]])
```

3.2.5 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.

3.2.5.1 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 as `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]])
```

3.2.5.2 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," Right array:\n",right)
```

```
Left array:
 [[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
Right array:
 [[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

3.2.6 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 Answer Questions (LAQ)

Q1: Explain how array slicing works in NumPy.

Ans: Array slicing in NumPy is a method to access or modify a subset of elements from a larger array. The syntax for slicing is `array[start:stop:step]`, where:

`start` is the index from which slicing begins (inclusive), `stop` is the index where slicing ends (exclusive), `step` is the interval between indices to include in the slice. For example, in a 1D array, `arr[1:5:2]` will return every second element between the indices 1 and 4.

Q2: Discuss the difference between the `.reshape()` function and the `.ravel()` function in NumPy.

Ans: The `.reshape()` function changes the shape of an array without modifying its data, allowing a multi-dimensional array to be flattened or reshaped into any compatible shape. On the other hand, `.ravel()` returns a flattened 1D version of an array, but it tries to avoid copying the data by returning a flattened view where possible. If modifying the flattened array is necessary, `ravel()` returns a copy instead.

Q3: Explain how NumPy handles broadcasting during array operations.

Ans: Broadcasting in NumPy is a method to perform element-wise operations on arrays of different shapes. Smaller arrays are “broadcast” across the larger array by repeating their elements to match the shape of the larger array. For example, when adding a scalar to a 2D array, the scalar is added to each element of the array by broadcasting the scalar to match the array’s shape. Similarly, operations between arrays of different shapes follow the broadcasting rules to make them compatible.

Q4: Describe how you would split an array in NumPy using the `np.split()` function. Provide an example.

Ans: The `np.split()` function in NumPy divides an array into multiple sub-arrays based on the indices provided. The syntax is: `np.split(array, indices)` Here, `array` is the array to be split, and `indices` is a list of indices where the split will occur. For example:

```
arr = np.array([1, 2, 3, 4, 5, 6])
np.split(arr, [2, 4])
```

This splits the array at indices 2 and 4, resulting in three sub-arrays: [1, 2], [3, 4], and [5, 6].

Q5: What are the key differences between `np.hsplit()` and `np.vsplit()`? Provide examples.

Ans: `np.hsplit()` horizontally splits an array along its columns, while `np.vsplit()` vertically splits an array along its rows. For example, if we have a 2D array:

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

np.hsplit(arr, 3) splits the array into three columns, each with two rows: [[1], [4]], [[2], [5]], [[3], [6]]. np.vsplit(arr, 2) splits the array into two sub-arrays along rows: [[1, 2, 3]] and [[4, 5, 6]].

Q6: How can you create a 2D array with random integers between 1 and 10 using NumPy? Provide an example.

Ans: A 2D array with random integers between 1 and 10 can be created using np.random.randint(low, high, size). Example:

```
np.random.randint(1, 10, size=(3, 3))
```

This generates a 3x3 array with random integers between 1 and 9.

Q7: Describe how you would reshape an array from 1D to 2D in NumPy.

Ans: Reshaping an array from 1D to 2D in NumPy can be done using the .reshape() function. For example, given a 1D array:

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

To reshape it into a 2D array with 2 rows and 3 columns:

```
arr.reshape(2, 3)
```

This results in [[1, 2, 3], [4, 5, 6]].

Q8: Explain the concept of stacking arrays in NumPy using np.stack(). Provide an example.

Ans: np.stack() joins arrays along a new axis, unlike hstack() and vstack(), which concatenate along existing axes. For example:

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
np.stack((arr1, arr2), axis=0)
```

This stacks the arrays along a new axis, resulting in $[[1, 2, 3], [4, 5, 6]]$.

Q9: How does NumPy's array_split() differ from split()? Provide an example.

Ans: The array_split() function allows unequal splitting of an array, whereas split() requires the splits to result in equal-sized sub-arrays. For example:

```
arr = np.array([1, 2, 3, 4, 5])
np.array_split(arr, 3)
```

This will split the array into three parts: [1, 2], [3, 4], and [5].

Q10: How would you flatten a multi-dimensional array into a 1D array in NumPy?

Ans: You can flatten a multi-dimensional array using the .ravel() or .flatten() methods. Example using ravel():

```
arr = np.array([[1, 2], [3, 4]])
arr.ravel()
```

This flattens the array into [1, 2, 3, 4].

Q11: Discuss the importance of NumPy in scientific computing and how it handles large datasets efficiently.

Ans: NumPy is crucial in scientific computing because it provides efficient storage and operations for large datasets through its n-dimensional array objects. It uses continuous memory blocks, making array operations faster than traditional Python lists, and supports a variety of mathematical functions and broadcasting, which simplifies computation.

NumPy operates efficiently by:

- Avoiding type checking at each operation due to its homogeneous data type constraint.
- Leveraging vectorization to reduce the need for explicit loops in operations.
- Providing optimized C and Fortran libraries for core computations.

Example of large dataset handling:

```
large_array = np.random.rand(1000000)
sum_large_array = np.sum(large_array) # Efficient summation
```

This efficiency makes NumPy a foundation for data-driven scientific applications like machine learning, signal processing, and simulations.

Q12: What is the difference between a view and a copy in NumPy? Why does this matter in array operations?

Ans: A view is a reference to the original array, meaning changes in the view will affect the original array. A copy creates a new, independent array.

Example:

```
arr = np.array([1, 2, 3])
view = arr[:2] # Creates a view
copy = arr[:2].copy() # Creates a copy
view[0] = 99 # This will change arr
```

Views are more memory-efficient, but changes to them affect the original data, whereas copies do not.

Q13: How are higher-dimensional arrays handled in NumPy, and how can they be reshaped and indexed? Provide a practical example.

Ans: Higher-dimensional arrays (tensors) in NumPy can be created and manipulated like 1D and 2D arrays. You can reshape tensors using `reshape()` and index them similarly, using one index for each dimension.

```
tensor = np.arange(24).reshape(2, 3, 4) # 3D tensor with shape (2, 3, 4)
element = tensor[1, 2, 3] # Access element at specified indices
```

we can reshape tensors:

```
reshaped_tensor = tensor.reshape(4, 6)
```

3.3 Some important NumPy function for Linear Algebra

Let's start with some basic matrix operations. Suppose we have two matrices, A and B, and we want to add them together. In NumPy, we can do this with the simple command, `A+B`. Let's look into the detailed computational steps.

```
# Addition and Subtraction
matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])
print(matrix1 + matrix2) # prints [[6, 8], [10, 12]]
```

```
[[ 6  8]
 [10 12]]
```

Similary, matrix difference and other matrix operations can be illustrated as follows.

```
print(matrix1 - matrix2) # prints [[-4, -4], [-4, -4]]
```

```
[[ -4 -4]
 [ -4 -4]]
```

```
# Scalar Multiplication
matrix = np.array([[1, 2], [3, 4]])
print(2 * matrix) # prints [[2, 4], [6, 8]]
```

```
[[2 4]
 [6 8]]
```

```
# Matrix Multiplication
matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])
print(np.dot(matrix1, matrix2)) # prints [[19, 22], [43, 50]]
```

```
[[19 22]
 [43 50]]
```

```
# Matrix Hadamards product
matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])
print(matrix1*matrix2)
```

```
[[ 5 12]
 [21 32]]
```

```
# Transpose
matrix = np.array([[1, 2, 3], [4, 5, 6]])
print(np.transpose(matrix)) # prints [[1, 4], [2, 5], [3, 6]]
```

```
[[1 4]
 [2 5]
 [3 6]]
```

```
# inverse of a matrix
a = np.array([[1, 2], [3, 4]])
a_inv = np.linalg.inv(a)
```

Next, let's talk about vectors. A vector is simply a matrix with one column. They're often used to represent things like forces or velocities in physics. In NumPy, we can represent vectors as arrays with one dimension.

```
vector = np.array([1, 2, 3])
print(vector) # prints [1, 2, 3]
```

```
[1 2 3]
```

Let's say we have two vectors, \vec{u} and \vec{v} , and we want to compute their dot product (i.e., the sum of the products of their corresponding entries). We can do this with the command:

```
vector1 = np.array([1, 2, 3])
vector2 = np.array([4, 5, 6])
print(np.dot(vector1, vector2)) # prints 32
```

32

Like that there are some other operations too.

```
# Cross Product
vector1 = np.array([1, 2, 3])
vector2 = np.array([4, 5, 6])
print(np.cross(vector1, vector2)) # prints [-3, 6, -3]
```

```
[-3 6 -3]
```

Note

Norm: The norm of a vector is a scalar that represents the “length” of the vector. In NumPy, we can compute the norm using the `numpy.linalg.norm` function. The inner product of two vectors is a matrix that is computed by multiplying the first vector by the transpose of the second vector. In NumPy, we can compute the inner product using the `numpy.inner` function.

```
# finding norm
vector = np.array([1, 2, 3])
print(np.linalg.norm(vector)) # prints 3.74165738677
```

3.7416573867739413

```
#finding inner product
vector1 = np.array([1, 2, 3])
vector2 = np.array([4, 5, 6])
print(np.inner(vector1, vector2)) # prints 32
```

32

To handle higher dimensional mutrix multiplication, one can use `matmul()` function. The `np.matmul()` function is another way to perform matrix multiplication. Unlike `np.dot()`, it handles higher-dimensional arrays correctly by broadcasting. The syntax for this operation is `np.matmul(a, b)`.

```
A = np.array([[1, 0], [0, 1]])
B = np.array([[4, 1], [2, 2]])

# Matrix multiplication using matmul
result = np.matmul(A, B)
print(result)
```

$\begin{bmatrix} 4 & 1 \\ 2 & 2 \end{bmatrix}$

The function `np.linalg.inv()` computes the inverse of a square matrix. Syntax for this function is `np.linalg.inv(a)`. An example is shown below.

```
A = np.array([[1, 2], [3, 4]])  
  
# Compute inverse  
inv_A = np.linalg.inv(A)  
print(inv_A)
```

```
[[ -2.    1. ]  
 [ 1.5   -0.5]]
```

The `np.linalg.det()` function computes the determinant of a square matrix. The determinant is useful for solving linear systems and understanding matrix properties. The syntax is `np.linalg.det(a)`.

```
A = np.array([[1, 2], [3, 4]])  
  
# Compute determinant  
det_A = np.linalg.det(A)  
print(det_A)
```

```
-2.0000000000000004
```

The `np.linalg.solve()` function solves a linear matrix equation or system of linear scalar equations. It finds the vector x that satisfies $Ax = b$. Syntax for this function is `np.linalg.solve(A, b)`.

```
A = np.array([[3, 1], [1, 2]])  
b = np.array([9, 8])  
  
# Solve system of equations  
x = np.linalg.solve(A, b)  
print(x)
```

```
[2. 3.]
```

This function computes the QR decomposition of a matrix. QR decomposition is used to solve linear systems, least squares problems, and compute eigenvalues. Syntax for this function is `np.linalg.qr()`.

```

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

# QR decomposition
Q, R = np.linalg.qr(A)
print("Q:", Q)
print("R:", R)

```

```

Q: [[-0.12309149  0.90453403  0.40824829]
 [-0.49236596  0.30151134 -0.81649658]
 [-0.86164044 -0.30151134  0.40824829]]
R: [[-8.12403840e+00 -9.60113630e+00 -1.10782342e+01]
 [ 0.00000000e+00  9.04534034e-01  1.80906807e+00]
 [ 0.00000000e+00  0.00000000e+00 -8.88178420e-16]]

```

The `np.linalg.lstsq()` function solves a linear least-squares problem, which is useful in regression tasks. The syntax for the function is `np.linalg.lstsq(a, b, rcond=None)`.

```

A = np.array([[1, 1], [1, 2], [2, 2], [2, 3]])
b = np.array([6, 8, 9, 11])

# Least-squares solution
x, residuals, rank, s = np.linalg.lstsq(A, b, rcond=None)
print("Solution:", x)

```

Solution: [2.09090909 2.54545455]

The Kronecker product is a matrix operation used in various applications like tensor products and matrix calculus. Syntax for this function is `np.kron(a, b)`.

```

A = np.array([[1, 2], [3, 4]])
B = np.array([[0, 5], [6, 7]])

kronecker = np.kron(A, B)
print(kronecker)

```

```

[[ 0  5  0 10]
 [ 6  7 12 14]
 [ 0 15  0 20]
 [18 21 24 28]]

```

Cosine similarity is used to find the cosine of the angle between two vectors. The mathematical formula for this operation is $\cos \theta = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| \cdot |\vec{b}|}$. The python function to calculate the cosine similarity is shown below.

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

cosine_sim = np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))
print(cosine_sim)
```

0.9746318461970762

3.3.1 Linear Regression using NumPy

The `np.polyfit()` function fits a polynomial of a specified degree to the data, making it useful for regression. Syntax for this function is `np.polyfit(x, y, deg)`. Where x Independent variable (input), y is the dependent variable (output) and deg degree of the fitting polynomial. A simple example is given below.

```
x = np.array([0, 1, 2, 3, 4])
y = np.array([1, 3, 5, 7, 9])

# Linear fit (degree 1)
coefficients = np.polyfit(x, y, 1)
print("Coefficients:", coefficients)
```

Coefficients: [2. 1.]

3.3.2 Some interesting handy matrix operations using numpy arrays

In matrix decomposition, we need the matrix representation, $A - \lambda I$. For any matrix, we can do this by just `A-lambda np.eye(3)`. This can be deomonstarted here.

```
A=np.array([[1,2,3],[3,4,5],[7,6,7]])
lamda=3
A-lamda*np.eye(3)
```

```
array([[-2.,  2.,  3.],
       [ 3.,  1.,  5.],
       [ 7.,  6.,  4.]])
```

Task: Create a matrix, A using numpy and find the covariance , $cov(A)$ using matrix operation.

```
# creating a random matrix
import numpy as np
A=np.arange(16).reshape(4,4)
A
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Now find $A - \bar{A}$.

```
A_bar=np.mean(A, axis=0) # calculating column-wise sum
A_bar
```

```
array([6., 7., 8., 9.])
```

```
# calculating A-A bar with outer product operation
print(A-np.outer(A_bar,np.ones(4)).T)
```

```
[[ -6. -6. -6. -6.]
 [-2. -2. -2. -2.]
 [ 2.  2.  2.  2.]
 [ 6.  6.  6.  6.]]
```

i Note

The same can be done using reshaping method

```
#calculating A-A_bar using broadcasting
X=A-np.mean(A, axis=0).reshape(1,4)
print(X)
```

```
[[[-6. -6. -6. -6.]
 [-2. -2. -2. -2.]
 [ 2.  2.  2.  2.]
 [ 6.  6.  6.  6.]]]
```

Calculating the covariance.

```
#manually calculating covariance
CoV=(1/3)*np.dot(X.T,X)
CoV
```

```
array([[26.66666667, 26.66666667, 26.66666667, 26.66666667],
       [26.66666667, 26.66666667, 26.66666667, 26.66666667],
       [26.66666667, 26.66666667, 26.66666667, 26.66666667],
       [26.66666667, 26.66666667, 26.66666667, 26.66666667]])
```

We can verify the same using default function as follows.

```
#calculating covariance using numpy function
np.cov(A, rowvar=False)
```

```
array([[26.66666667, 26.66666667, 26.66666667, 26.66666667],
       [26.66666667, 26.66666667, 26.66666667, 26.66666667],
       [26.66666667, 26.66666667, 26.66666667, 26.66666667],
       [26.66666667, 26.66666667, 26.66666667, 26.66666667]])
```

Note

It is interesting to compare the two ways of flattening an array using `reshape()`.

```
#comparing two ways of flattening a matrix using numpy
A.reshape(-1)==A.reshape(16,)

array([ True,  True,  True,  True,  True,  True,  True,  True,
       True,  True,  True,  True,  True,  True])
```

3.4 Basics of SciPy Library for Computational Linear Algebra

Following the comprehensive exploration of the NumPy library, which forms the foundation of array operations and basic linear algebra computations, it is essential to expand into more advanced tools for scientific computing. The **SciPy** library builds upon NumPy, offering a vast collection of functions and utilities specifically designed for higher-level operations in scientific and technical computing. While NumPy provides efficient array handling and basic matrix operations, SciPy extends these capabilities by incorporating advanced functions for optimization, integration, interpolation, and linear algebra, among other tasks (Virtanen et al. 2020).

For computational linear algebra, SciPy provides specialized modules like `scipy.linalg`, which can handle everything from solving linear systems to eigenvalue decompositions and matrix factorizations. This transition from NumPy to SciPy enables the handling of more complex problems efficiently and allows users to leverage optimized algorithms for large-scale numerical computations. By integrating SciPy into the workflow, computations can be carried out more robustly, expanding on the basic linear algebra concepts introduced through NumPy with advanced techniques necessary for practical applications.

3.4.1 Basic Matrix operations

SciPy builds on the functionality of NumPy, offering more sophisticated and optimized algorithms, particularly suited for numerical computing tasks. While NumPy provides essential operations for linear algebra, SciPy's `scipy.linalg` module extends these capabilities with more advanced functions. SciPy functions are often better optimized for large-scale systems, making them highly efficient for computational linear algebra applications.

As the first step to use SciPy, we need to import the (only) necessary submodules for our specific tasks. In our discussion, we consider only linear algebra. So we import the `linalg` submodule as follows

```
#import scipy
import numpy as np # for matrix definition
from scipy import linalg
#print(scipy.__version__) # check version
```

Now let's discuss various SciPy functions for linear algebra with examples.

3.4.1.1 Computing the Determinant

The determinant is a scalar value that can be computed from the elements of a square matrix and is often used to determine whether a system of linear equations has a unique solution.

Syntax: `scipy.linalg.det(A)`

```
# example
A = np.array([[1, 2], [3, 4]])
det_A = linalg.det(A)
print(det_A)
```

-2.0

 Note

Similar functionality is provided by `np.linalg.det()`. However, `scipy.linalg.det()` is often preferred when working with very large matrices due to the efficiency of SciPy's backend implementations.

3.4.1.2 Solving Linear Systems of Equations

One of the fundamental tasks in linear algebra is solving a system of linear equations of the form $AX = b$, where A is a matrix and b is a vector or matrix of known values. >*Syntax:* `scipy.linalg.solve(A, b)`

```
#example
A = np.array([[3, 1], [1, 2]])
b = np.array([9, 8])
x = linalg.solve(A, b)
print(x)
```

[2. 3.]

 Note

NumPy's `np.linalg.solve()` also provides this functionality, but SciPy's version is better suited for larger and more complex matrices because it uses more efficient algorithms for decomposing the matrix.

3.4.1.3 Matrix Inversion

Matrix inversion is a critical operation in many linear algebra problems, particularly in solving systems of linear equations. >*Syntax:* `scipy.linalg.inv(A)`

```
A = np.array([[1, 2], [3, 4]])
inv_A = linalg.inv(A)
print(inv_A)
```

```
[[ -2.    1. ]
 [ 1.5 -0.5]]
```

3.4.1.4 Kronecker Product

The Kronecker product is used in various applications, including constructing block matrices and expanding the dimensionality of matrices.

Syntax: `scipy.linalg.kron(A, B)`

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[0, 5], [6, 7]])
kron_product = linalg.kron(A, B)
print(kron_product)
```

```
[[ 0  5  0 10]
 [ 6  7 12 14]
 [ 0 15  0 20]
 [18 21 24 28]]
```

3.4.1.5 Eigenvalues and Eigenvectors

Eigenvalues and eigenvectors are fundamental in many areas of linear algebra, including solving systems of differential equations and performing dimensionality reduction in machine learning.

Syntax: `scipy.linalg.eig(A)`

```
A = np.array([[3, 2], [4, 1]])
eigenvalues, eigenvectors = linalg.eig(A)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:", eigenvectors)
```

```
Eigenvalues: [ 5.+0.j -1.+0.j]
Eigenvectors: [[ 0.70710678 -0.4472136 ]
 [ 0.70710678  0.89442719]]
```

3.5 Sparse Matrices

Sparse matrices are often useful in numerical simulations dealing with large systems, if the problem can be described in matrix form where the matrices or vectors mostly contains zeros. Scipy has a good support for sparse matrices, with basic linear algebra operations (such as equation solving, eigenvalue calculations, etc.).

There are many possible strategies for storing sparse matrices in an efficient way. Some of the most common are the so-called coordinate form (COO), list of list (LIL) form, and compressed-sparse column CSC (and row, CSR). Each format has some advantages and disadvantages. Most computational algorithms (equation solving, matrix-matrix multiplication, etc.) can be efficiently implemented using CSR or CSC formats, but they are not so intuitive and not so easy to initialize. So often a sparse matrix is initially created in COO or LIL format (where we can efficiently add elements to the sparse matrix data), and then converted to CSC or CSR before used in real calculations.

For more information about these sparse formats, see e.g. http://en.wikipedia.org/wiki/Sparse_matrix

3.5.1 Sparse Matrix operations in SciPy

Sparse matrices are a key feature of SciPy, providing an efficient way to store and manipulate large matrices with a significant number of zero elements. SciPy offers a variety of sparse matrix formats and supports operations like matrix multiplication, addition, transposition, and solving systems of equations.

Here is a guide to working with sparse matrices in SciPy.

Types of Sparse Matrices in SciPy

SciPy provides different types of sparse matrices depending on the use case:

1. **CSR (Compressed Sparse Row) Matrix:** Efficient for row slicing and matrix-vector products.
2. **CSC (Compressed Sparse Column) Matrix:** Efficient for column slicing and fast arithmetic operations.
3. **COO (Coordinate) Matrix:** Suitable for constructing sparse matrices by specifying individual entries.
4. **DIA (Diagonal) Matrix:** For matrices where non-zero elements are primarily on the diagonals.
5. **LIL (List of Lists) Matrix:** Good for constructing matrices incrementally.

3.5.1.1 Importing Sparse Matrices

```
from scipy.sparse import csr_matrix, csc_matrix, coo_matrix
import numpy as np
```

Creating Sparse Matrices

```
# Create a dense matrix
dense_matrix = np.array([[0, 0, 3], [4, 0, 0], [0, 5, 6]])

# Convert dense matrix to CSR format
csr = csr_matrix(dense_matrix)

# Display CSR matrix
print(csr)
```

```
<Compressed Sparse Row sparse matrix of dtype 'int32'
 with 4 stored elements and shape (3, 3)>
Coords      Values
(0, 2)      3
(1, 0)      4
(2, 1)      5
(2, 2)      6
```

```
# creating sparse matrix in COO format
# Define row indices, column indices, and values
row = np.array([0, 1, 2, 2])
col = np.array([2, 0, 1, 2])
data = np.array([3, 4, 5, 6])
```

```
# Create COO sparse matrix
coo = coo_matrix((data, (row, col)), shape=(3, 3))

print(coo)
```

```
<COOOrdinate sparse matrix of dtype 'int32'
  with 4 stored elements and shape (3, 3)>
Coords      Values
(0, 2)      3
(1, 0)      4
(2, 1)      5
(2, 2)      6
```

Basic Operations with Sparse Matrices

The basic matrix operations can be performed on the sparse matrix too. The difference is that in the case of sparse matrices, the respective operations will be done only on non-zero entries. Now look into the basic matrix operations thorough following examples.

Matrix Multiplication:

```
A = csr_matrix([[1, 0, 0], [0, 0, 1], [0, 2, 0]])
B = csr_matrix([[4, 5], [0, 0], [7, 8]])

# Matrix multiplication (dot product)
result = A.dot(B)
print(result.toarray()) # Convert to dense array for display
```

```
[[4 5]
 [7 8]
 [0 0]]
```

Transposition:

```
# Transpose the matrix
transposed = A.transpose()

print(transposed.toarray())
```

```
[[1 0 0]
 [0 0 2]
 [0 1 0]]
```

Addition:

```
# Adding two sparse matrices
C = csr_matrix([[0, 1, 2], [3, 0, 0], [0, 0, 5]])
D = csr_matrix([[0, 1, 0], [0, 0, 0], [2, 0, 5]])

sum_matrix = C + D

print(sum_matrix.toarray())
```

```
[[ 0  2  2]
 [ 3  0  0]
 [ 2  0 10]]
```

Solving Sparse Linear Systems

We can solve systems of linear equations using sparse matrices with the `spsolve()` function:

```
from scipy.sparse.linalg import spsolve

# Create a sparse matrix (A) and a dense vector (b)
A = csr_matrix([[3, 1, 0], [1, 2, 0], [0, 0, 1]])
b = np.array([5, 5, 1])

# Solve the system Ax = b
x = spsolve(A, b)

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

Solution x: [1. 2. 1.]

3.5.1.2 Conversion from one sparse matrix system to another

We can convert between different sparse matrix formats using the `.tocsc()`, `.tocoo()`, `.todia()`, and similar methods:

```
# Convert CSR to COO format
coo = A.tocoo()
print(coo)
print("The matrix is :\n",coo.toarray())

<COOrdinate sparse matrix of dtype 'int32'
 with 5 stored elements and shape (3, 3)>
Coords      Values
(0, 0)      3
(0, 1)      1
(1, 0)      1
(1, 1)      2
(2, 2)      1
The matrix is :
[[3 1 0]
 [1 2 0]
 [0 0 1]]
```

Summary of sparse matrix operations

Function	Description
<code>csr_matrix()</code>	Compressed Sparse Row matrix.
<code>csc_matrix()</code>	Compressed Sparse Column matrix.
<code>coo_matrix()</code>	Coordinate format matrix.
<code>spsolve()</code>	Solves sparse linear systems.
<code>spdiags()</code>	Extracts or constructs diagonal sparse matrices.
<code>lil_matrix()</code>	List of lists sparse matrix.

3.6 Visualization Libraries

Data visualization libraries in Python empower developers and data scientists to create compelling visual representations of data. Popular libraries include Matplotlib, which offers versatile 2-D plotting capabilities, Seaborn for statistical graphics, Bokeh for interactive web applications, Altair for declarative visualizations, and Plotly for web-based interactive charts and dashboards.

3.6.1 Matplotlib: A Comprehensive Data Visualization Library in Python

Matplotlib is a powerful Python library that enables developers and data scientists to create a wide range of static, animated, and interactive visualizations. Whether we're exploring data, presenting insights, or building scientific plots, Matplotlib has us covered. Let's delve into its features:

1. **Publication-Quality Plots:** Matplotlib allows us to create professional-quality plots suitable for research papers, presentations, and publications. Customize colors, fonts, and styles to match our requirements.
2. **Versatility:** With Matplotlib, one can create line plots, scatter plots, bar charts, histograms, pie charts, and more. It supports 2-D plotting and can handle complex visualizations.
3. **Interactive Figures:** While it's known for static plots, Matplotlib also offers interactivity. We can zoom, pan, and explore data points within the plot.
4. **Customization:** Fine-tune every aspect of our plot, from axis labels and titles to grid lines and legends. Annotations and text can be added seamlessly.
5. **Integration with Jupyter:** Matplotlib integrates well with Jupyter notebooks, making it a favorite among data scientists and analysts.

Background

Matplotlib is a multi-platform data visualization library built on NumPy arrays, and designed to work with the broader SciPy stack. It was conceived by John Hunter in 2002, originally as a patch to IPython for enabling interactive MATLAB-style plotting via gnuplot from the IPython command line. IPython's creator, Fernando Perez, was at the time scrambling to finish his PhD, and let John know he wouldn't have time to review the patch for several months. John took this as a cue to set out on his own, and the Matplotlib package was born, with version 0.1 released in 2003. It received an early boost when it was adopted as the plotting package of choice of the Space Telescope Science Institute (the folks behind the Hubble Telescope), which financially supported Matplotlib's development and greatly expanded its capabilities.

One of Matplotlib's most important features is its ability to play well with many operating systems and graphics backends. Matplotlib supports dozens of backends and output types, which means you can count on it to work regardless of which operating system you are using or which output format you wish. This cross-platform, everything-to-everyone approach has been one of the great strengths of Matplotlib. It has led to a large user base, which in turn has led to an active developer base and Matplotlib's powerful tools and ubiquity within the scientific Python world.

In recent years, however, the interface and style of Matplotlib have begun to show their age. Newer tools like ggplot and ggviz in the R language, along with web visualization toolkits based on D3js and HTML5 canvas, often make Matplotlib feel clunky and old-

fashioned. Still, I'm of the opinion that we cannot ignore Matplotlib's strength as a well-tested, cross-platform graphics engine. Recent Matplotlib versions make it relatively easy to set new global plotting styles, and people have been developing new packages that build on its powerful internals to drive Matplotlib via cleaner, more modern APIs—for example, Seaborn, ggpy, HoloViews, Altair, and even Pandas itself can be used as wrappers around Matplotlib's API. Even with wrappers like these, it is still often useful to dive into Matplotlib's syntax to adjust the final plot output. For this reason, I believe that Matplotlib itself will remain a vital piece of the data visualization stack, even if new tools mean the community gradually moves away from using the Matplotlib API directly.

3.6.1.1 General Matplotlib Tips

Before we dive into the details of creating visualizations with Matplotlib, there are a few useful things you should know about using the package.

1. Importing the `matplotlib` module

Just as we use the `np` shorthand for NumPy and the `pd` shorthand for Pandas, we will use some standard shorthands for Matplotlib imports:

```
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
%matplotlib inline
```

i Note

The `plt` interface is what we will use most often, as we shall see throughout this chapter.

3.6.1.2 Setting Styles

We will use the `plt.style` directive to choose appropriate aesthetic styles for our figures. Here we will set the `classic` style, which ensures that the plots we create use the classic Matplotlib style:

```
plt.style.use('classic')
plt.style.use('default')
plt.style.use('seaborn')
```

Throughout this section, we will adjust this style as needed. Note that the stylesheets used here are supported as of Matplotlib version 1.5; if you are using an earlier version of Matplotlib, only the default style is available.

A simple example of loading the matplotlib module and setting theme is shown below.

```
import matplotlib.pyplot as plt
plt.style.use('classic')
```

3.6.2 How to Display Your Plots?

A visualization you can't see won't be of much use, but just how you view your Matplotlib plots depends on the context. The best use of Matplotlib differs depending on how you are using it; roughly, the three applicable contexts are using Matplotlib in a script, in an IPython terminal, or in a Jupyter notebook.

3.6.3 Plotting from a Jupyter Notebook

The Jupyter notebook is a browser-based interactive data analysis tool that can combine narrative, code, graphics, HTML elements, and much more into a single executable document.

If you are using Matplotlib from within a script, the function `plt.show` is your friend. `plt.show` starts an event loop, looks for all currently active `Figure` objects, and opens one or more interactive windows that display your figure or figures.

The `plt.show` command does a lot under the hood, as it must interact with your system's interactive graphical backend. The details of this operation can vary greatly from system to system and even installation to installation, but Matplotlib does its best to hide all these details from you.

One thing to be aware of: the `plt.show` command should be used *only once* per Python session, and is most often seen at the very end of the script. Multiple `show` commands can lead to unpredictable backend-dependent behavior, and should mostly be avoided.

i Note

Using `plt.show` in IPython's Matplotlib mode is not required.

Plotting interactively within a Jupyter notebook can be done with the `%matplotlib` command, and works in a similar way to the IPython shell. You also have the option of embedding graphics directly in the notebook, with two possible options:

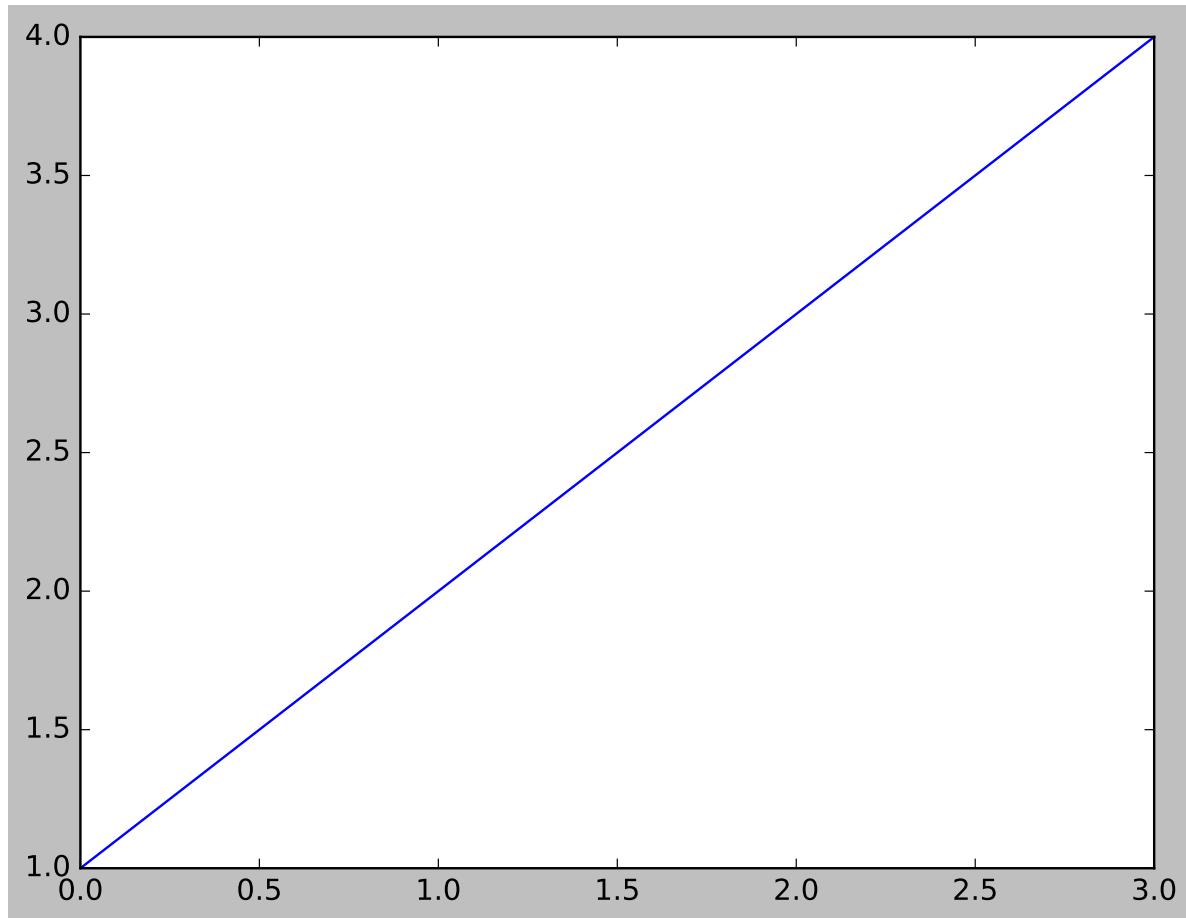
- `%matplotlib inline` will lead to *static* images of your plot embedded in the notebook.

- `%matplotlib notebook` will lead to *interactive* plots embedded within the notebook.

For this discussion, we will generally stick with the default, with figures rendered as static images (see the following figure for the result of this basic plotting example):

```
%matplotlib inline
```

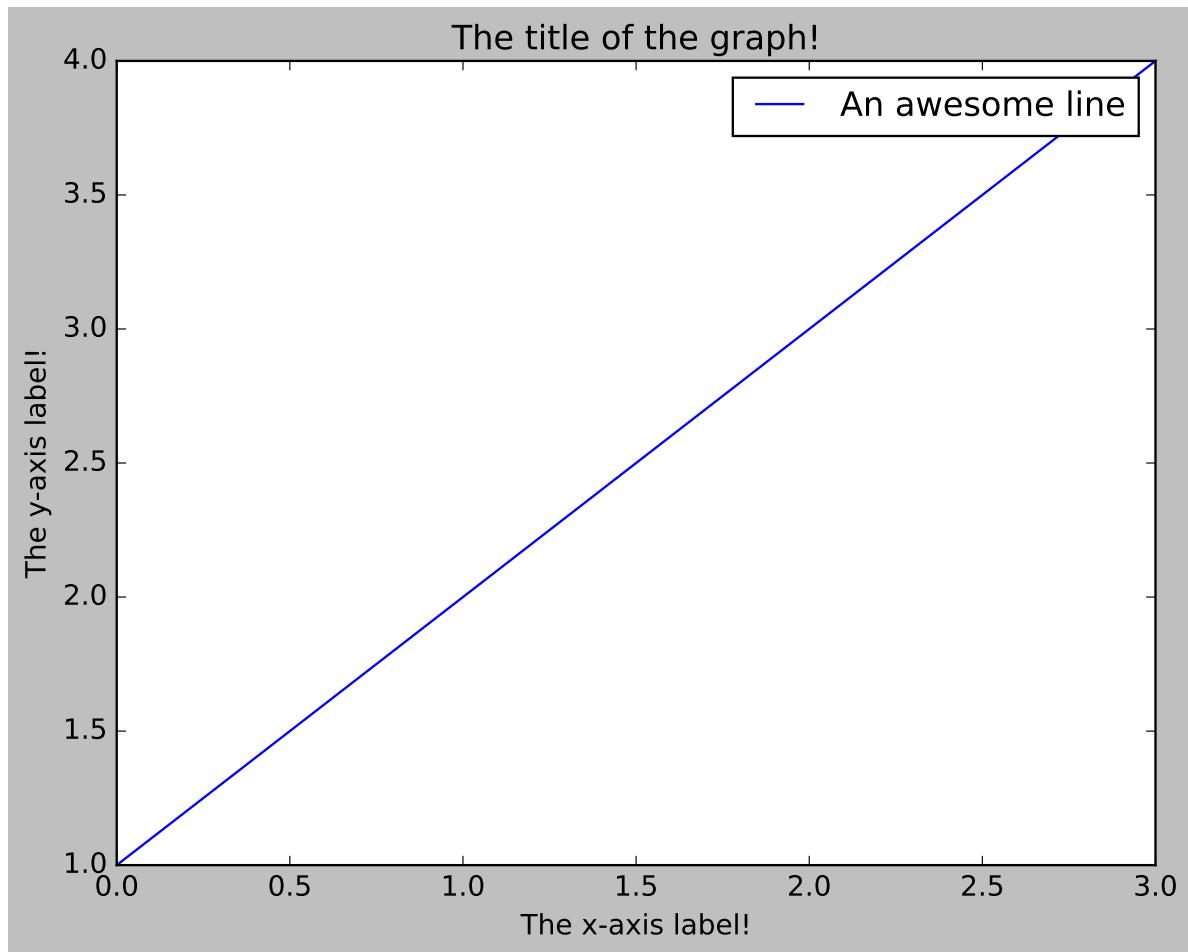
```
plt.plot([1,2,3,4])  
plt.show()
```



3.6.3.1 Adding titles, axis labels, and a legend

Let's redraw this plot but now with a title, axis labels, and a legend:

```
x_vals = [1, 2, 3, 4]
plt.plot(x_vals, label="An awesome line")
plt.ylabel('The y-axis label!')
plt.xlabel('The x-axis label!')
plt.title("The title of the graph!")
plt.legend()
plt.show()
```

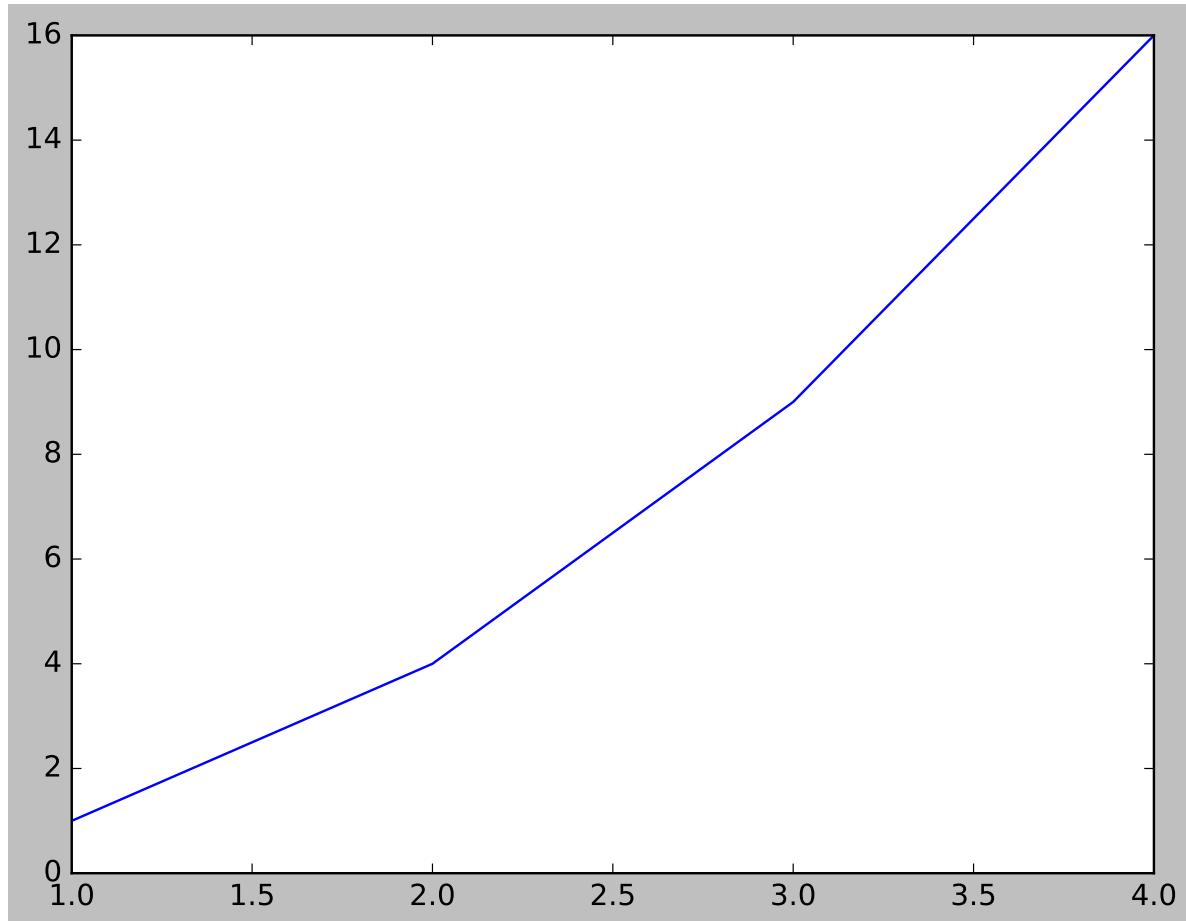


3.6.3.2 Adding both x and y data

You may be wondering why the x-axis ranges from 0-3 and the y-axis from 1-4. If you provide a single list or array to the `plot()` command, matplotlib assumes it is a sequence of y values, and automatically generates the x values for you.

`plot()` is a versatile command, and will take an arbitrary number of arguments. For example, to plot x versus y, you can issue the command:

```
x_vals = [1,2,3,4]
y_vals = [1, 4, 9, 16]
plt.plot(x_vals, y_vals)
plt.show()
```

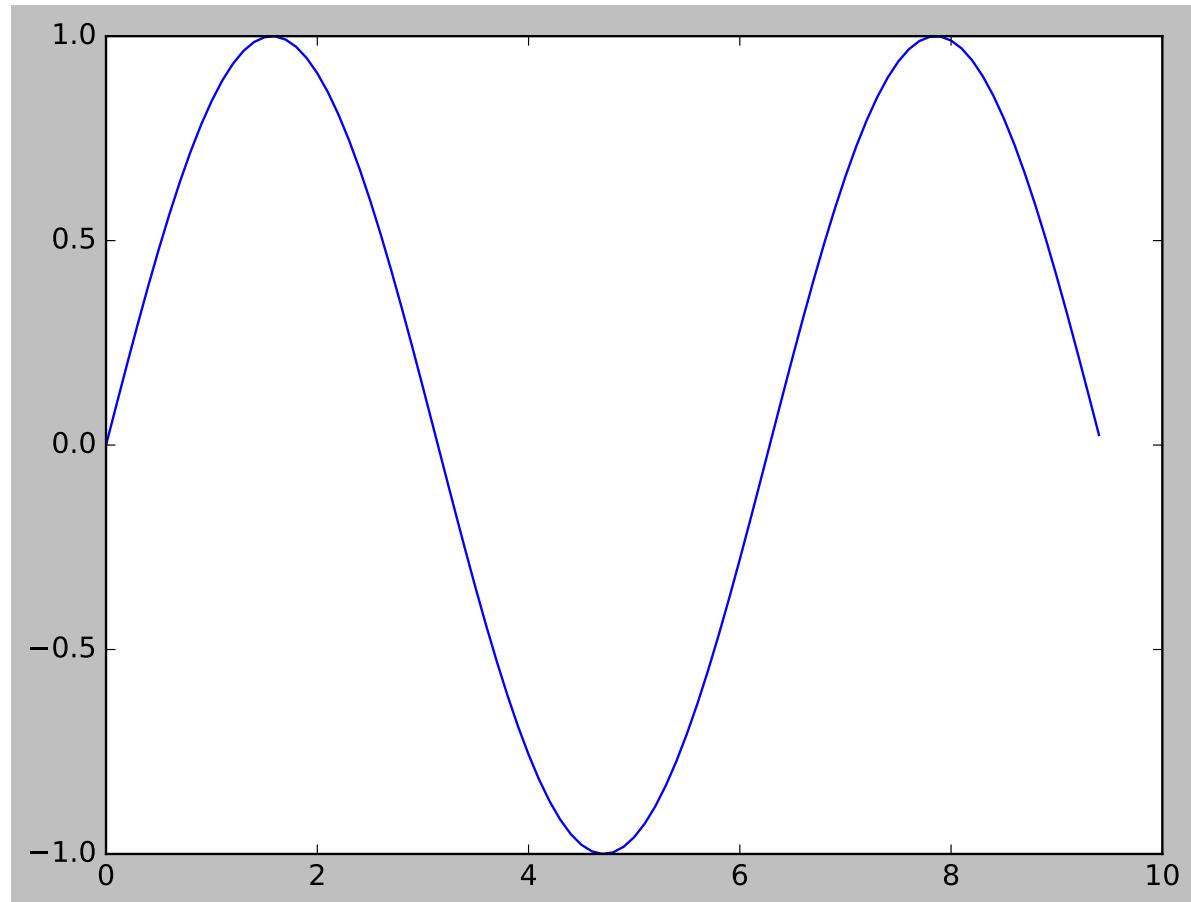


More explicit examples are shown below.

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

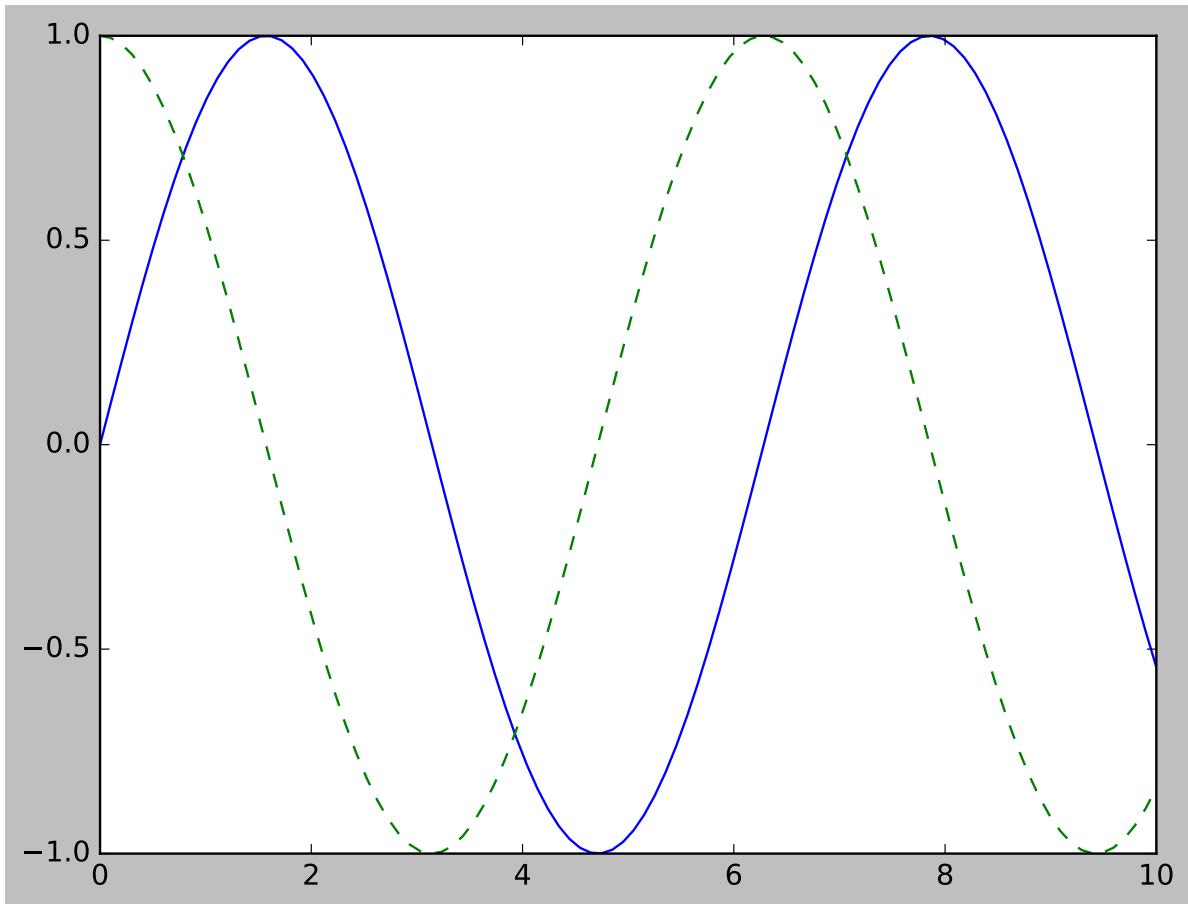
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)
```

```
# Plot the points using matplotlib
plt.plot(x, y)
plt.show() # You must call plt.show() to make graphics appear.
```



```
import numpy as np
x = np.linspace(0, 10, 100)

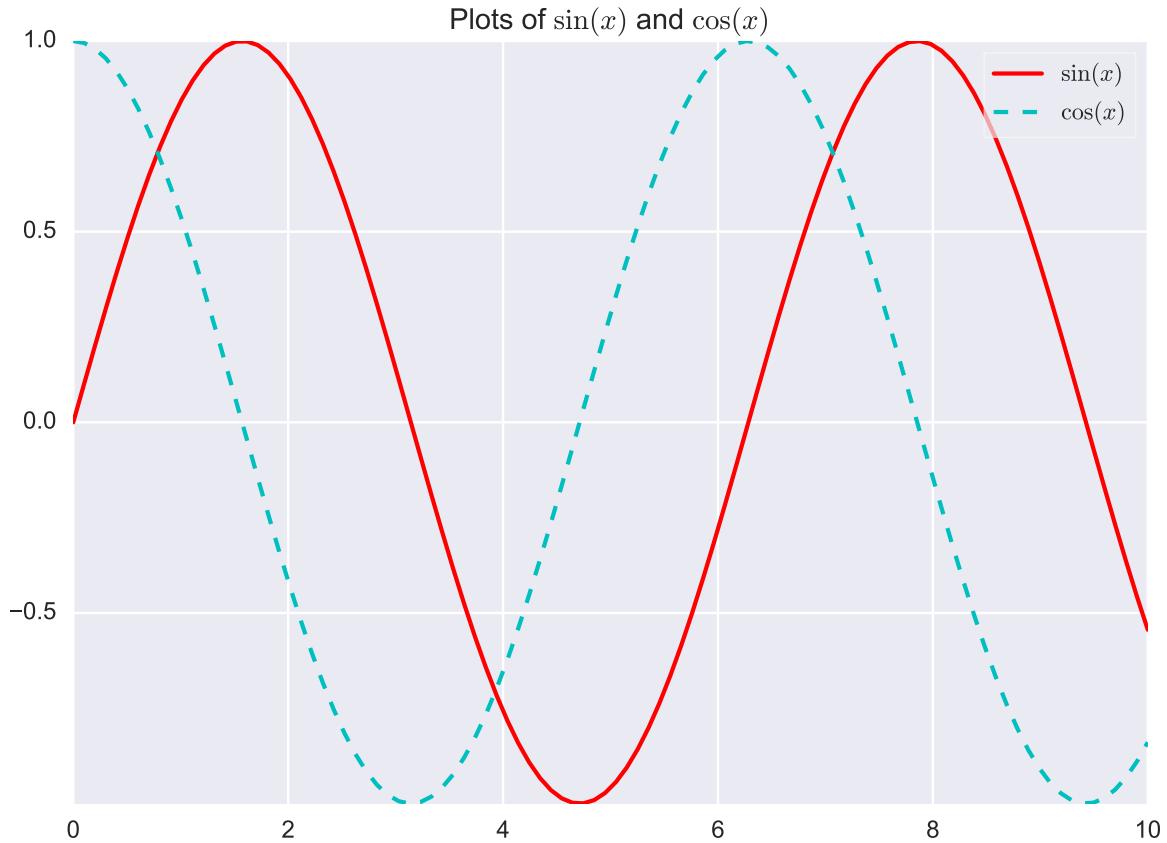
fig = plt.figure()
plt.plot(x, np.sin(x), '-')
plt.plot(x, np.cos(x), '--');
plt.show()
```



Now let's change the theme to `seaborn` and create more plots with additional features.

```
plt.style.use('seaborn-v0_8')

import numpy as np
fig=plt.figure()
x = np.linspace(0, 10, 100)
plt.plot(x, np.sin(x), 'r-', label=r'$\sin(x)$') # r stands for colour and r
# in label stands for row text
plt.plot(x, np.cos(x), 'c--', label=r'$\cos(x)$')
plt.title(r'Plots of $\sin(x)$ and $\cos(x)$')
plt.axis('tight')
plt.legend(frameon=True, loc='upper right', ncol=1, framealpha=.7)
plt.show()
```



Saving Figures to File

One nice feature of Matplotlib is the ability to save figures in a wide variety of formats. Saving a figure can be done using the `savefig()` command. In `savefig()`, the file format is inferred from the extension of the given filename. Depending on what backends you have installed, many different file formats are available. The list of supported file types can be found for your system by using the following method of the figure canvas object. Following function return all supported formats.

```
fig.canvas.get_supported_filetypes()
```

```
{'eps': 'Encapsulated Postscript',
'jpg': 'Joint Photographic Experts Group',
'jpeg': 'Joint Photographic Experts Group',
'pdf': 'Portable Document Format',
'pgf': 'PGF code for LaTeX',
'png': 'Portable Network Graphics',
```

```
'ps': 'Postscript',
'raw': 'Raw RGBA bitmap',
'rgba': 'Raw RGBA bitmap',
'svg': 'Scalable Vector Graphics',
'svgz': 'Scalable Vector Graphics',
'tif': 'Tagged Image File Format',
'tiff': 'Tagged Image File Format',
'webp': 'WebP Image Format'}
```

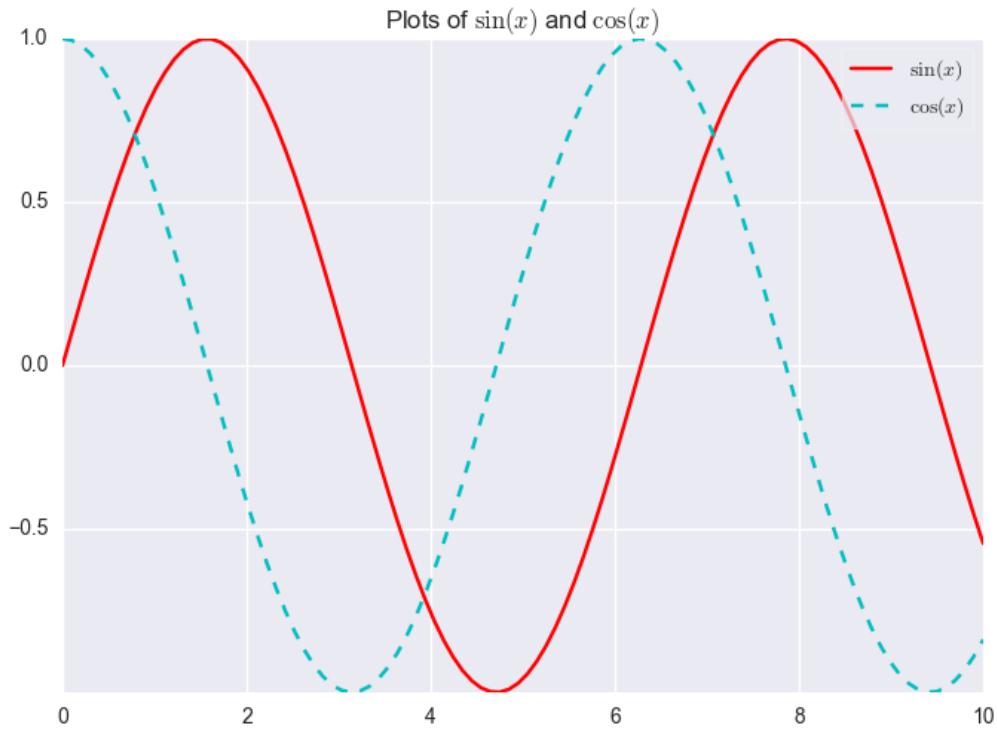
Note that when saving your figure, it's not necessary to use `plt.show()` or related commands discussed earlier.

For example, to save the previous figure as a PNG file, you can run this:

```
fig.savefig('my_figure.png')
```

To confirm that it contains what we think it contains, let's use the IPython `Image` object to display the contents of this file:

```
from IPython.display import Image
Image('my_figure.png')
```



3.6.3.3 MATLAB-style Interface

Matplotlib was originally written as a Python alternative for MATLAB users, and much of its syntax reflects that fact. The MATLAB-style tools are contained in the `pyplot` (`plt`) interface. For example, the following code will probably look quite familiar to MATLAB users:

3.6.3.4 Plotting multiple charts

You can create multiple plots within the same figure by using `subplot`

Let's consider an example of two plots on same canvas.

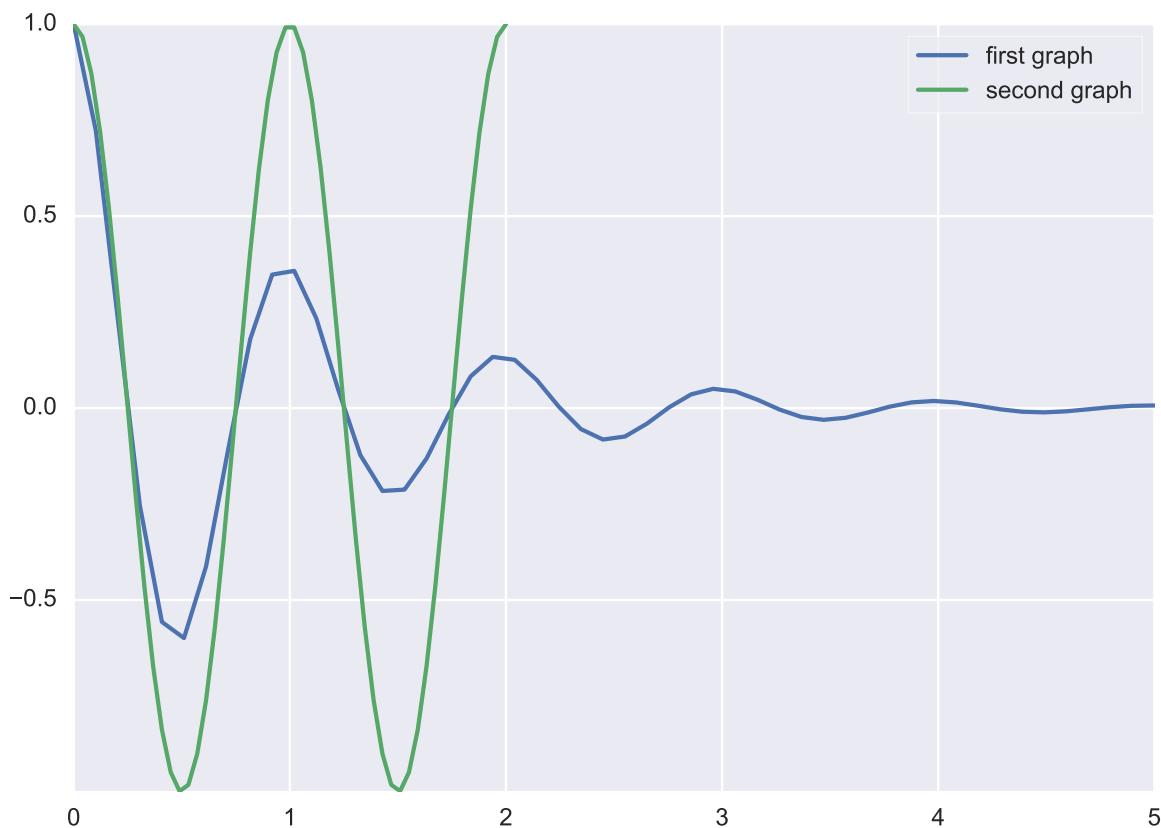
```
import matplotlib.pyplot as plt
import numpy as np

# Create some fake data.
x1 = np.linspace(0.0, 5.0)
y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
```

```

x2 = np.linspace(0.0, 2.0)
y2 = np.cos(2 * np.pi * x2)
plt.plot(x1,y1,label="first graph")
plt.plot(x2,y2,label="second graph")
plt.axis('tight')
plt.legend(frameon=True, loc='upper right', ncol=1, framealpha=.7)
plt.show()

```



Now let's represent these two plots in separate subplots as shown below.

```

fig, (ax1, ax2) = plt.subplots(2, 1)
fig.suptitle('A tale of 2 subplots')

ax1.plot(x1, y1, 'o-')
ax1.set_ylabel('Damped oscillation')

ax2.plot(x2, y2, '.-')

```

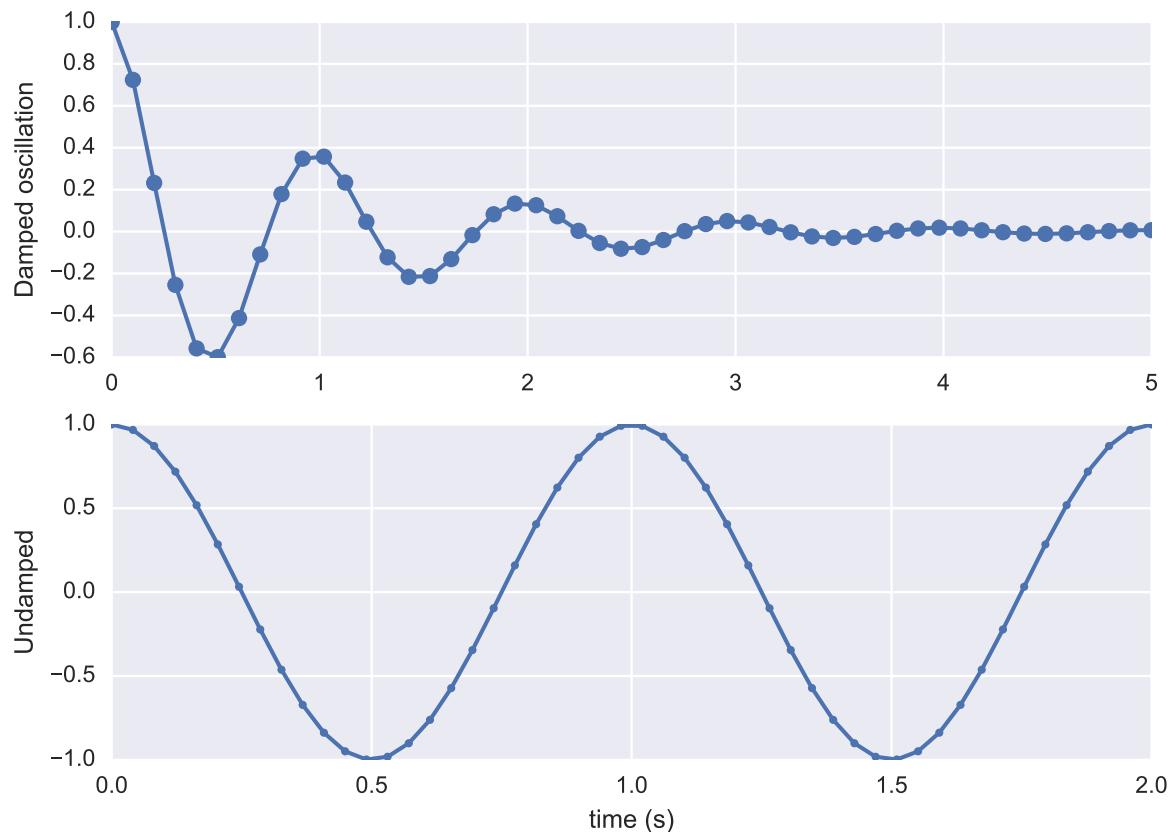
```

ax2.set_xlabel('time (s)')
ax2.set_ylabel('Undamped')

plt.show()

```

A tale of 2 subplots



Another approach is shown below.

```

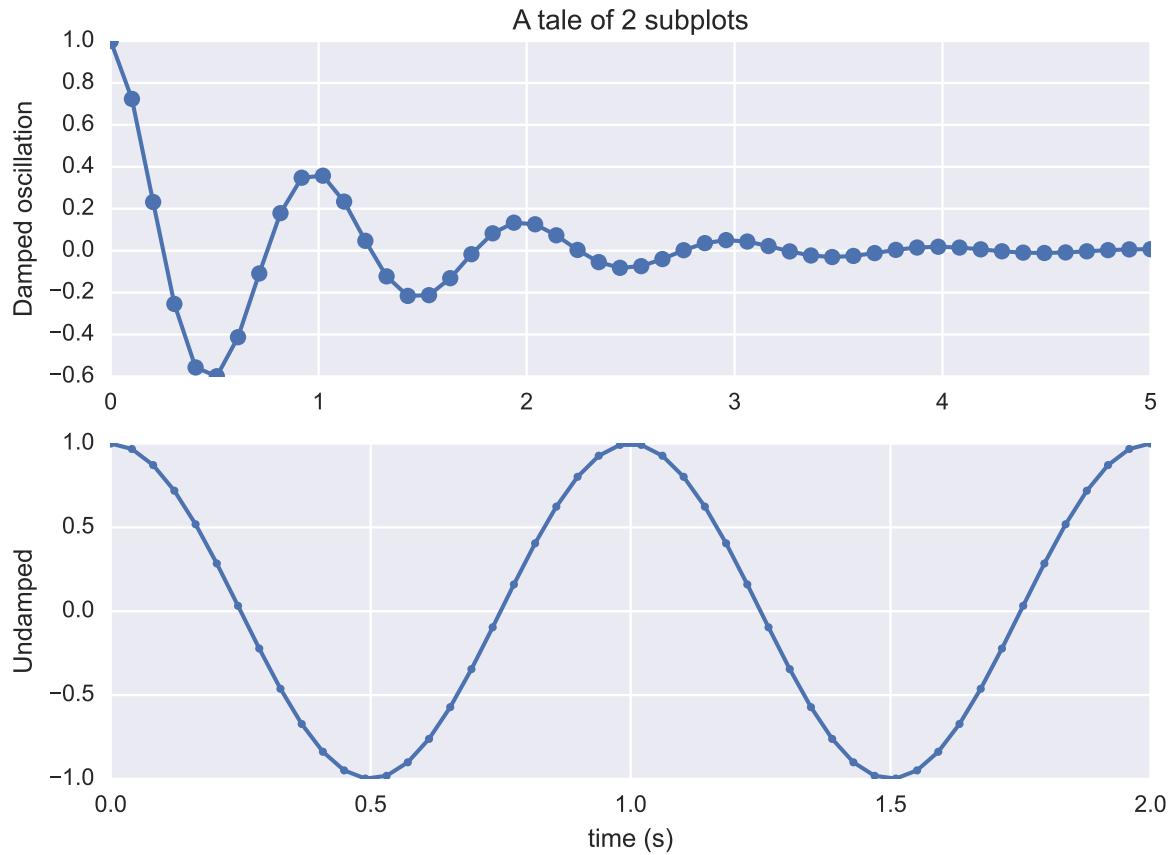
plt.subplot(2, 1, 1)
plt.plot(x1, y1, 'o-')
plt.title('A tale of 2 subplots')
plt.ylabel('Damped oscillation')

plt.subplot(2, 1, 2)
plt.plot(x2, y2, '.-')
plt.xlabel('time (s)')

```

```
plt.ylabel('Undamped')
```

```
plt.show()
```



Another example is shown below.

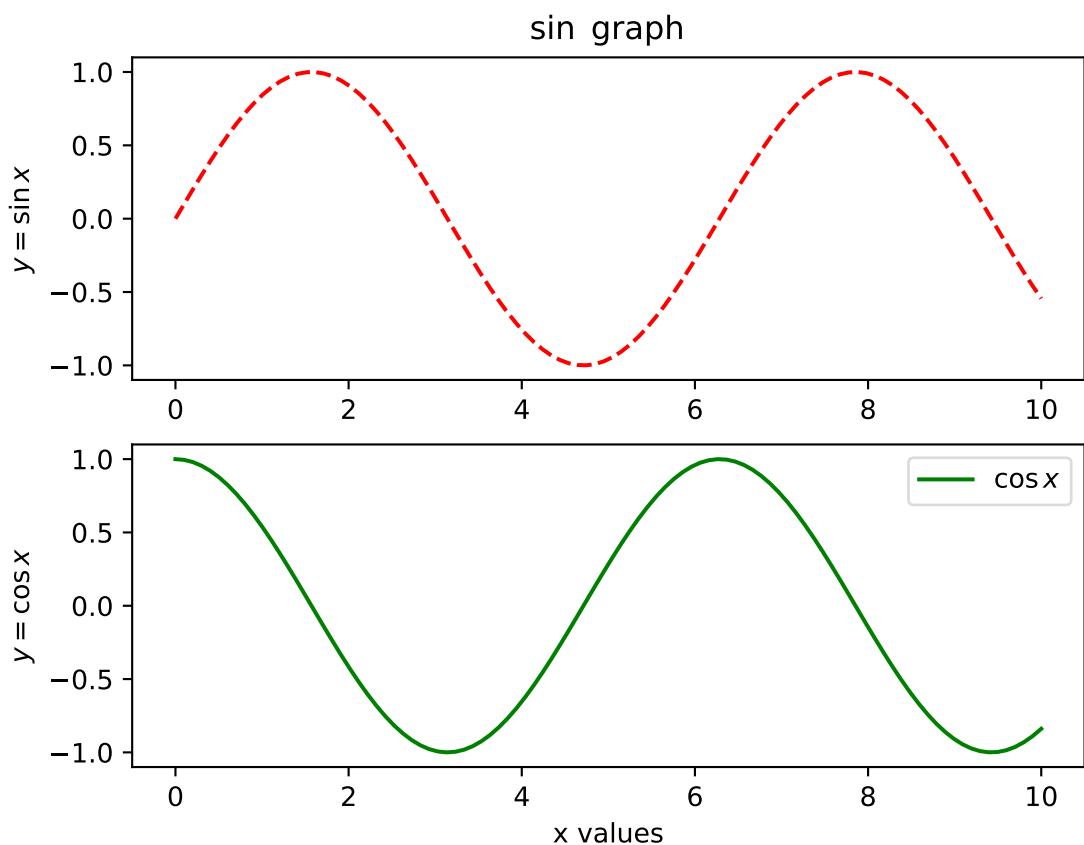
```
# First create a grid of plots
# ax will be an array of two Axes objects
plt.style.use('default')
fig, ax = plt.subplots(2)

# Call plot() method on the appropriate object
ax[0].plot(x, np.sin(x), 'r--', label=r'$\sin x$')
ax[0].set_title(r'$\sin $ graph')
ax[1].set_xlabel("x values")
ax[0].set_ylabel(r'$y=\sin x$')
```

```

ax[1].plot(x, np.cos(x), 'g-', label=r'$\cos x$')
ax[1].set_ylabel(r'$y=\cos x$')
plt.axis('tight')
plt.legend(frameon=True, loc='upper right', ncol=1, framealpha=.7)
plt.show()

```



3.6.3.5 Simple Scatter Plots

Another commonly used plot type is the simple scatter plot, a close cousin of the line plot. Instead of points being joined by line segments, here the points are represented individually with a dot, circle, or other shape. We'll start by setting up the notebook for plotting and importing the functions we will use:

```

import numpy as np
x = np.linspace(0, 10, 100)

```

```

fig = plt.figure()
plt.plot(x, np.sin(x), '-o', label=r'$\sin(x)$',)
plt.plot(x, np.cos(x), 'p', label=r'$\cos(x)$')
plt.title('Plots of $\sin(x)$ and $\cos(x)$' )
plt.axis('tight')
plt.legend(frameon=True, loc='upper right', ncol=1, framealpha=.7)
plt.show()

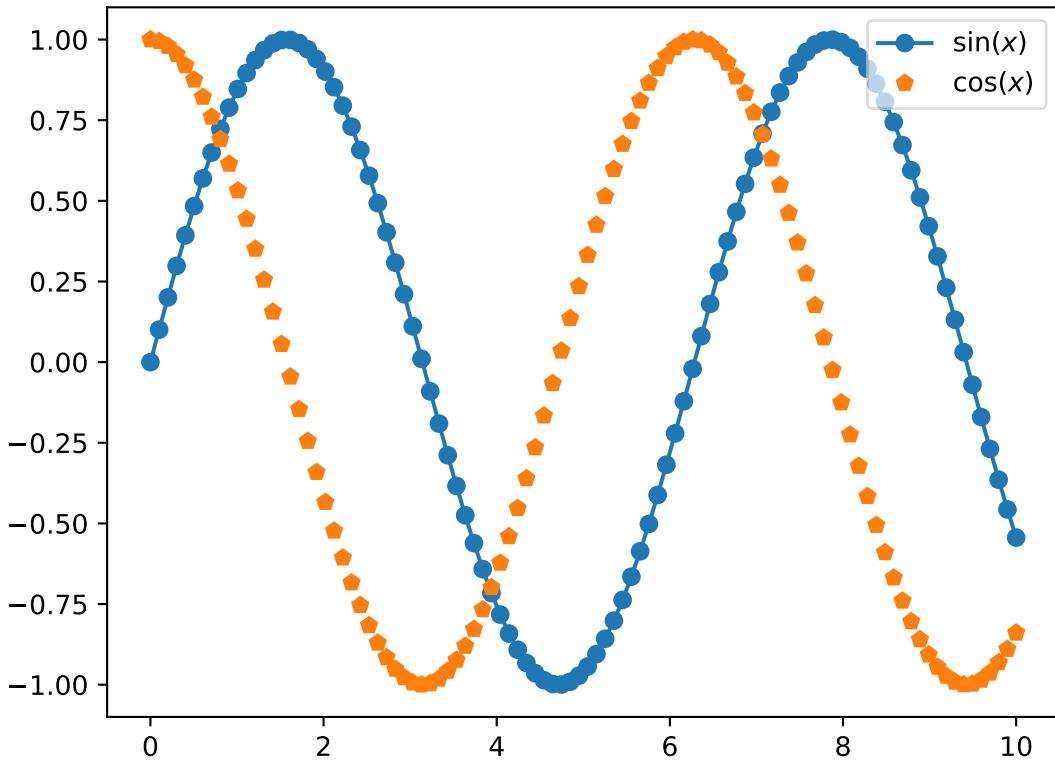
```

<>:6: SyntaxWarning: invalid escape sequence '\s'

<>:6: SyntaxWarning: invalid escape sequence '\s'

C:\Users\SIJUKSWAMY\AppData\Local\Temp\ipykernel_12808\3327894231.py:6: SyntaxWarning: invalid escape sequence '\s'

Plots of $\sin(x)$ and $\cos(x)$



3.6.3.6 Scatter Plots with plt.scatter

A second, more powerful method of creating scatter plots is the `plt.scatter` function, which can be used very similarly to the `plt.plot` function:

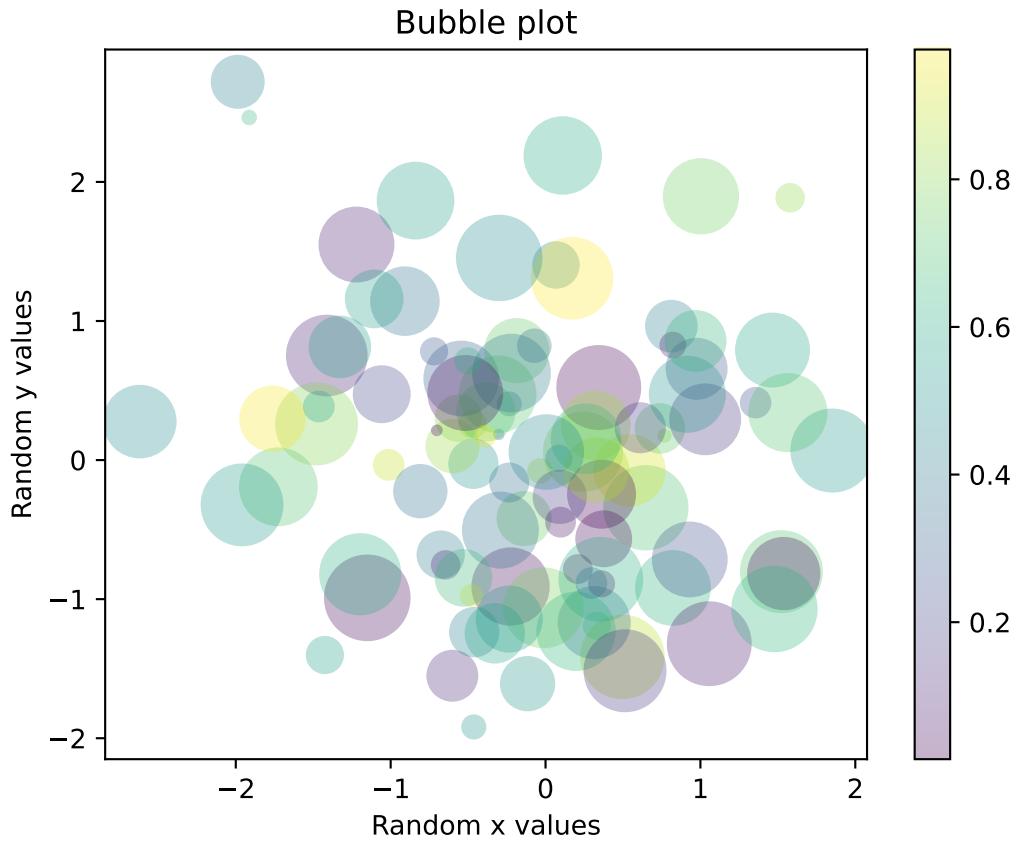
i Note

The primary difference of `plt.scatter` from `plt.plot` is that it can be used to create scatter plots where the properties of each individual point (size, face color, edge color, etc.) can be individually controlled or mapped to data.

Let's show this by creating a random scatter plot with points of many colors and sizes. In order to better see the overlapping results, we'll also use the `alpha` keyword to adjust the transparency level:

```
rng = np.random.RandomState(42)
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)

plt.scatter(x, y, c=colors, s=sizes, alpha=0.3,
            cmap='viridis',label=" Random Y values")
plt.xlabel('Random x values')
plt.ylabel('Random y values')
plt.title('Bubble plot' )
plt.colorbar(); # show color scale
```

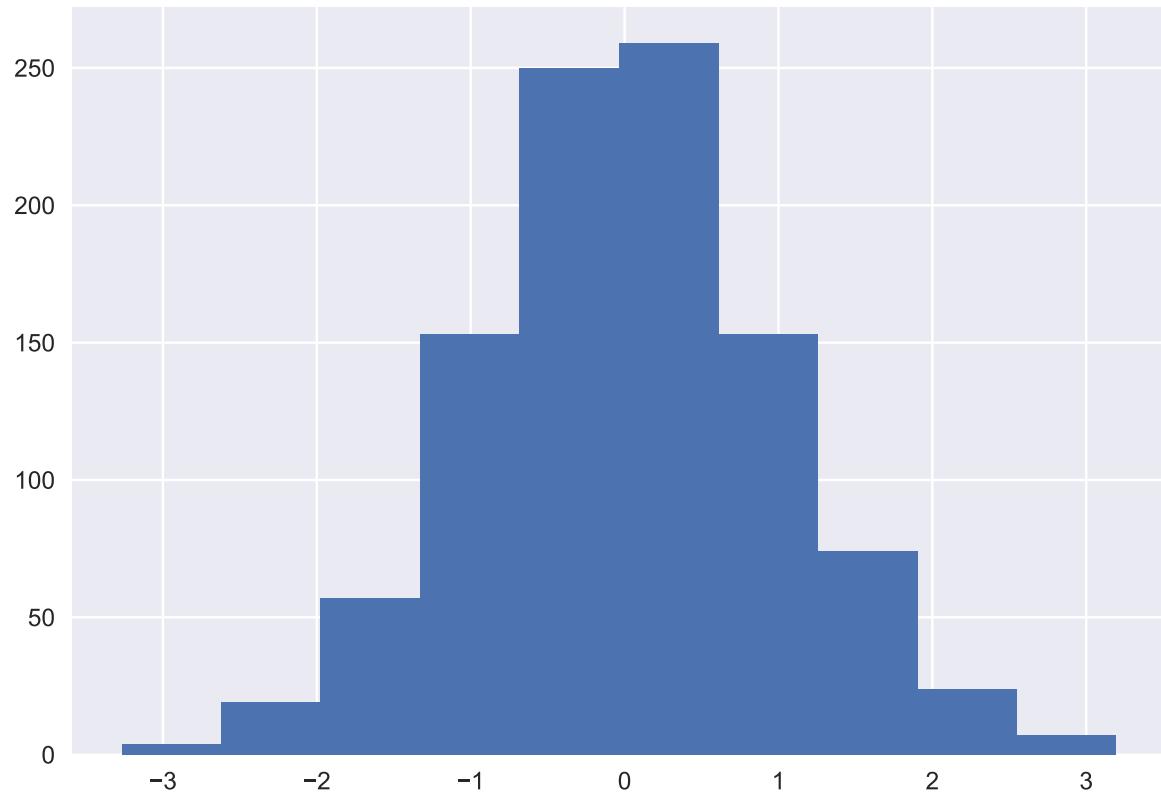


3.6.4 Histograms, Binnings, and Density

A simple histogram can be a great first step in understanding a dataset.

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-v0_8')
data = np.random.randn(1000)
plt.hist(data)
```

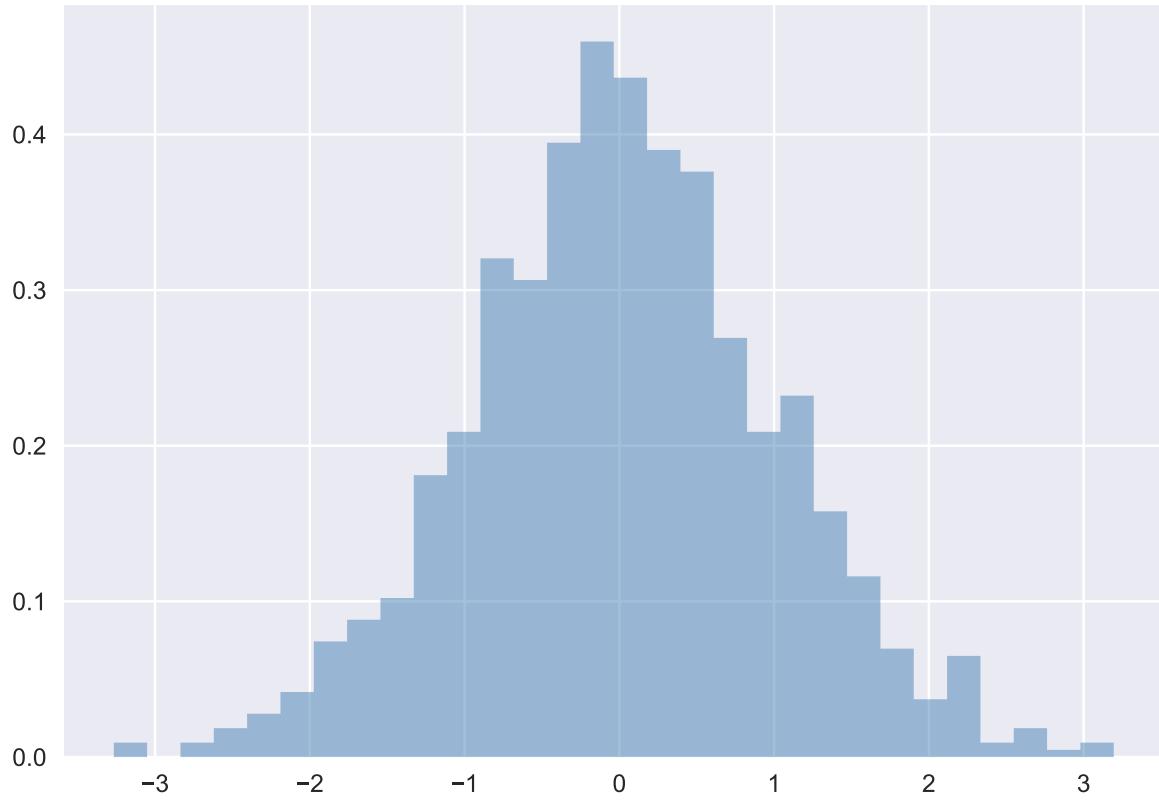
(array([4., 19., 57., 153., 250., 259., 153., 74., 24., 7.]),
 array([-3.26644452, -2.62036015, -1.97427577, -1.3281914 , -0.68210703,
 -0.03602265, 0.61006172, 1.2561461 , 1.90223047, 2.54831484,
 3.19439922]),
 <BarContainer object of 10 artists>)



3.6.4.1 Customizing Histograms

The `hist()` function has many options to tune both the calculation and the display; here's an example of a more customized histogram:

```
plt.hist(data, bins=30, density=True, alpha=0.5,  
         histtype='stepfilled', color='steelblue',  
         edgecolor='none');
```



The plt.hist docstring has more information on other customization options available. This combination of histtype='stepfilled' along with some transparency alpha to be very useful when comparing histograms of several distributions:

```
w1 = np.random.normal(0, 0.8, 1000)
w2 = np.random.normal(-2, 1, 1000)
w3 = np.random.normal(3, 1.2, 1000)

kwargs = dict(histtype='stepfilled', alpha=0.3, density=True, bins=40)

plt.hist(w1, **kwargs, label='w1')
plt.hist(w2, **kwargs, label='w2')
plt.hist(w3, **kwargs, label='w3')
plt.legend()
```



3.6.5 Working with datafiles

Consider the pokemon dataset for this job. The main features of this dataset are:

Defense: This column represents the base damage resistance against normal attacks. Higher values indicate that the Pokémon can withstand more physical damage.

Sp. Atk (Special Attack): This column shows the base modifier for special attacks. Pokémon with higher Special Attack values can deal more damage with special moves.

Sp. Def (Special Defense): This column indicates the base damage resistance against special attacks. Higher values mean the Pokémon can better resist damage from special moves.

Speed: This column determines which Pokémon attacks first in each round. Pokémon with higher Speed values will generally attack before those with lower values.

Stage: This column represents the evolutionary stage of the Pokémon. It typically ranges from 1 to 3, with 1 being the base form and 3 being the final evolved form. Some Pokémon may have additional stages, such as Mega Evolutions or Gigantamax forms.

Legendary: This is a boolean column that identifies whether the Pokémon is legendary. It is marked as `True` for legendary Pokémon and `False` for non-legends ones.

These columns provide valuable insights into the strengths and characteristics of each Pokémon, helping players strategize and build their teams effectively.

Now, let's read our data into a `Pandas` data frame. We will relax the limit on display columns and rows using the `set_option()` method in Pandas:

```
import pandas as pd
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)
```

```
df=pd.read_csv("https://raw.githubusercontent.com/sijuswamy/PyWorks/main/Pok
← emon.csv",encoding = 'utf_8')
df.head()
```

	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp.Atk	Sp.Def	Speed	Stage	Legen
0	Bulbasaur	Grass	Poison	318	45	49	49	65	65	45	1	False
1	Ivysaur	Grass	Poison	405	60	62	63	80	80	60	2	False
2	Venusaur	Grass	Poison	525	80	82	83	100	100	80	3	False
3	Charmander	Fire	Poison	309	39	52	43	60	50	65	1	False
4	Charmeleon	Fire	NaN	405	58	64	58	80	65	80	2	False

Since the `Legendary` feature contains the string `True` and `False`. But they are part of the logical data type in python. So let's replace these values with `TRUE` and `FALSE` strings as follows.

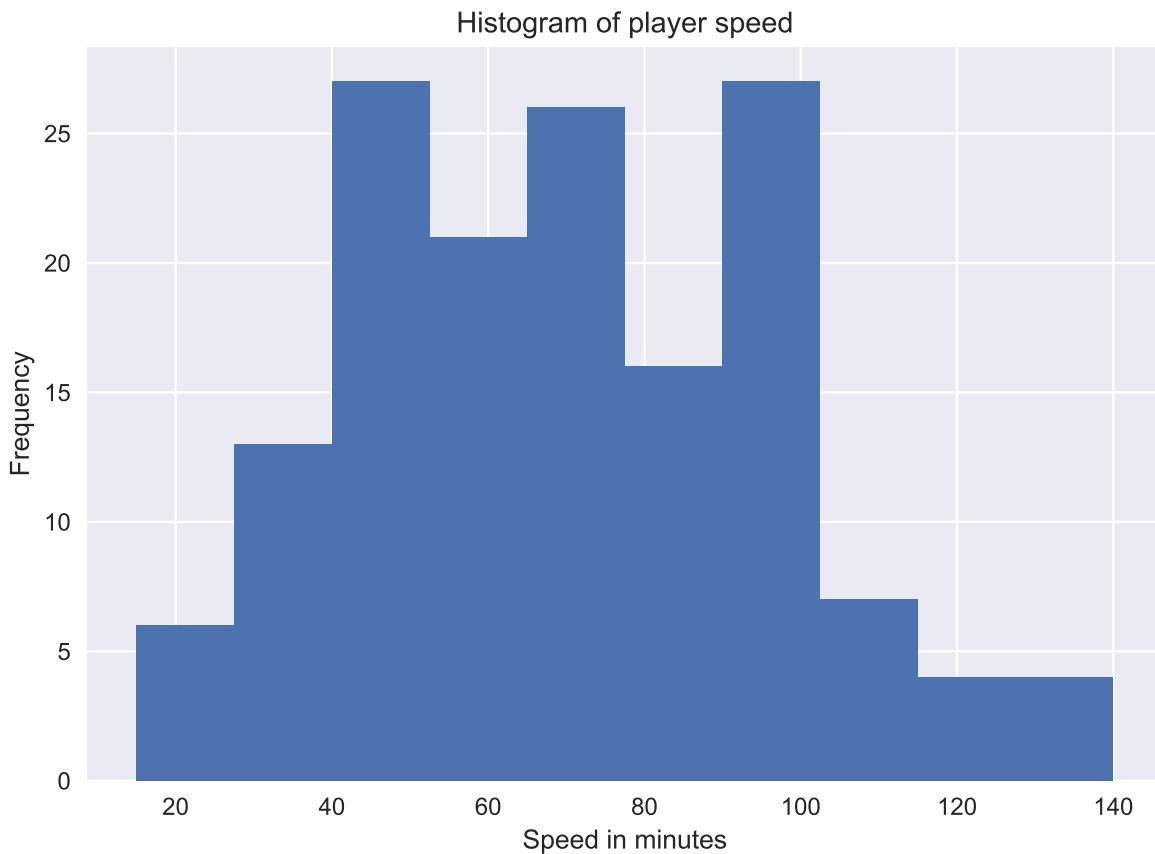
```
booleanDictionary = {True: 'TRUE', False: 'FALSE'}
df = df.replace(booleanDictionary)
df.head()
```

	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp.Atk	Sp.Def	Speed	Stage	Legen
0	Bulbasaur	Grass	Poison	318	45	49	49	65	65	45	1	FALS
1	Ivysaur	Grass	Poison	405	60	62	63	80	80	60	2	FALS
2	Venusaur	Grass	Poison	525	80	82	83	100	100	80	3	FALS
3	Charmander	Fire	Poison	309	39	52	43	60	50	65	1	FALS
4	Charmeleon	Fire	NaN	405	58	64	58	80	65	80	2	FALS

Creating a histogram

We can generate a histogram for any of the numerical columns by calling the `hist()` method on the `plt` object and passing in the selected column in the data frame. Let's do this for the speed column, which corresponds to speed of the player.

```
plt.hist(df['Speed'])
plt.xlabel('Speed in minutes')
plt.ylabel('Frequency')
plt.title('Histogram of player speed')
plt.show()
```



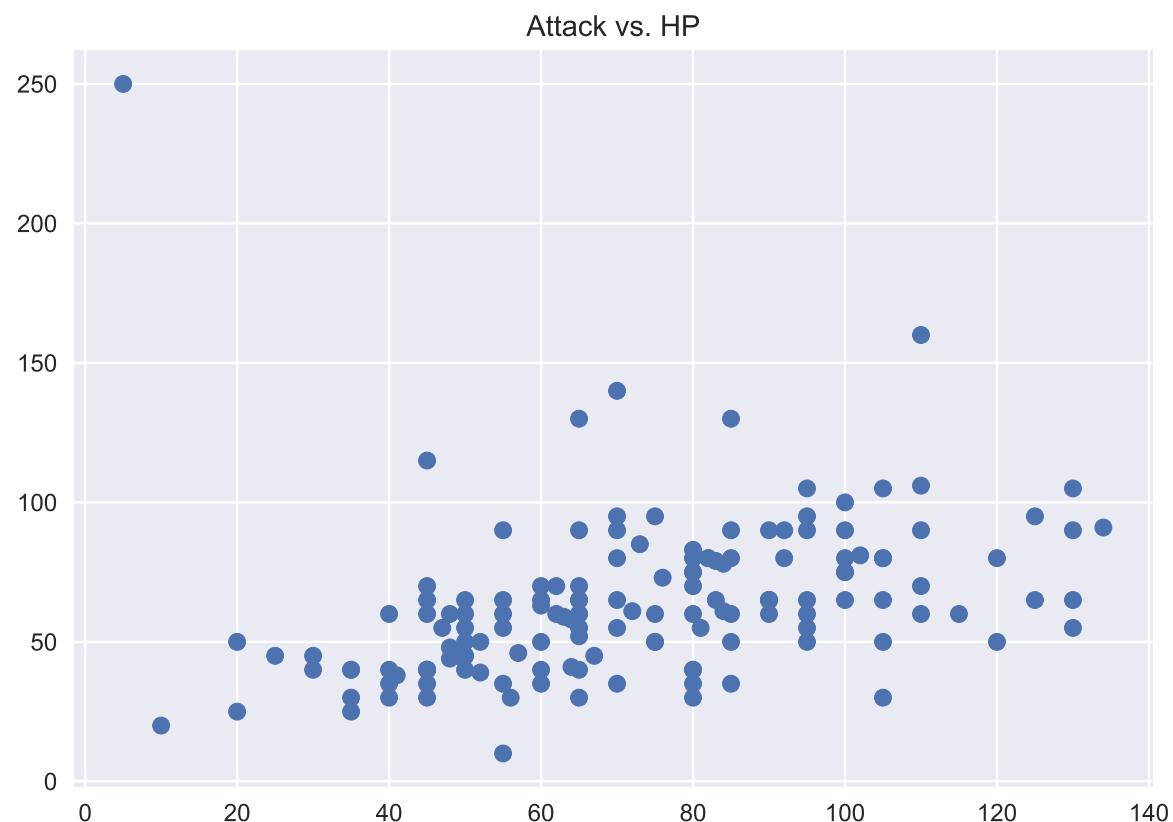
Scatterplot of Attack vs HP

To generate a scatter plot in Matplotlib, we simply use the `scatter()` method on the `plt` object. Let's also label the axes and give our plot a title:

```

plt.scatter(df['Attack'], df['HP'])
plt.title('Attack vs. HP')
plt.show()

```



Barchart

Bar charts are another useful visualization tool for analyzing categories in data. To visualize categorical columns, we first should count the values. We can use the counter method from the collections modules to generate a dictionary of count values for each category in a categorical column. Let's do this for the Legendary column.

```

from collections import Counter

print(Counter(df[('Legendary')]))

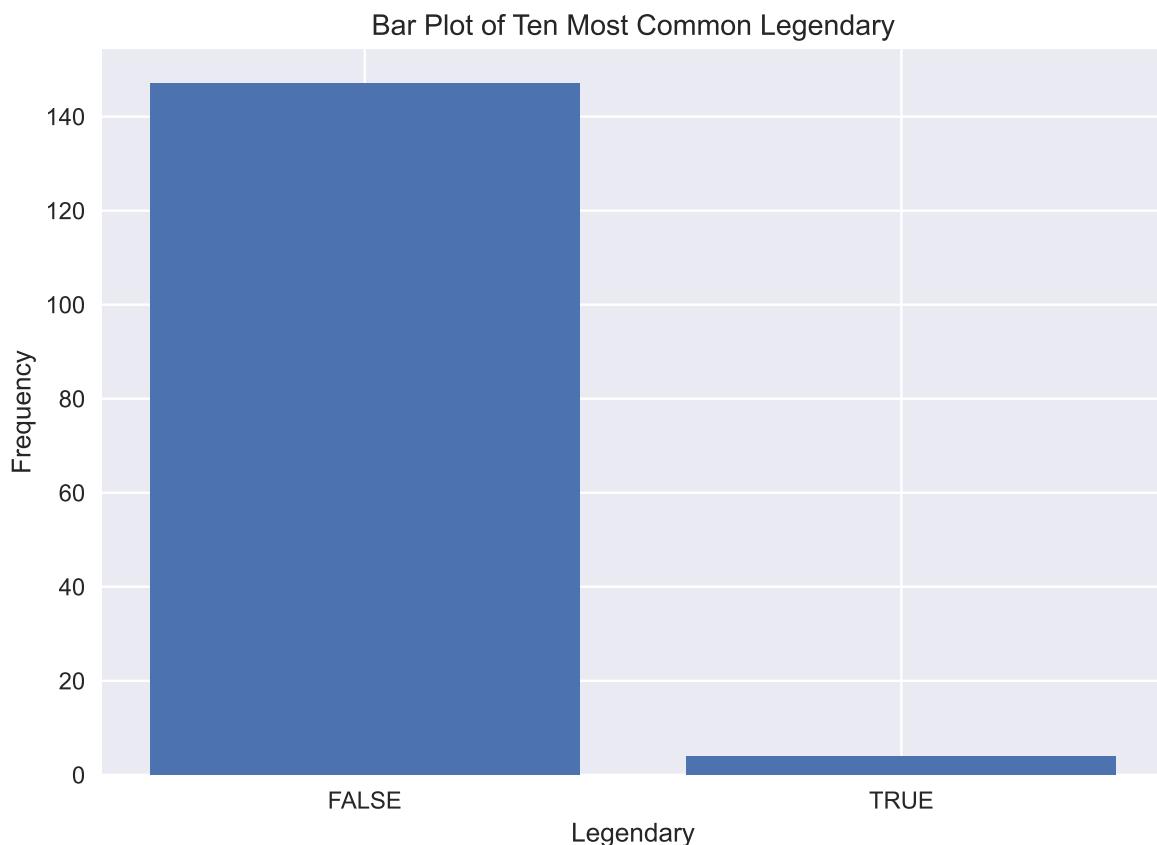
```

```
Counter({'FALSE': 147, 'TRUE': 4})
```

Note

We can filter this dictionary using the `most_common` method. Let's look at the 10 most common nationality values (you can also use the `least_common` method to analyze infrequent nationality values)

```
Legendary_dict = dict(Counter(df[('Legendary')]).most_common(2))
plt.bar(Legendary_dict.keys(), Legendary_dict.values())
plt.xlabel('Legendary')
plt.ylabel('Frequency')
plt.title('Bar Plot of Ten Most Common Legendary')
# plt.xticks(rotation=90)
plt.show()
```



Generating Pie Charts With Matplotlib

Pie charts are a useful way to visualize proportions in your data. So first we need to create the dictionary of proportion then feed it to the pie chart.

```
prop = dict(Counter(df['Legendary']))

for key, values in prop.items():

    prop[key] = (values)/len(df)*100

print(prop)
```

```
{'FALSE': 97.35099337748345, 'TRUE': 2.6490066225165565}
```

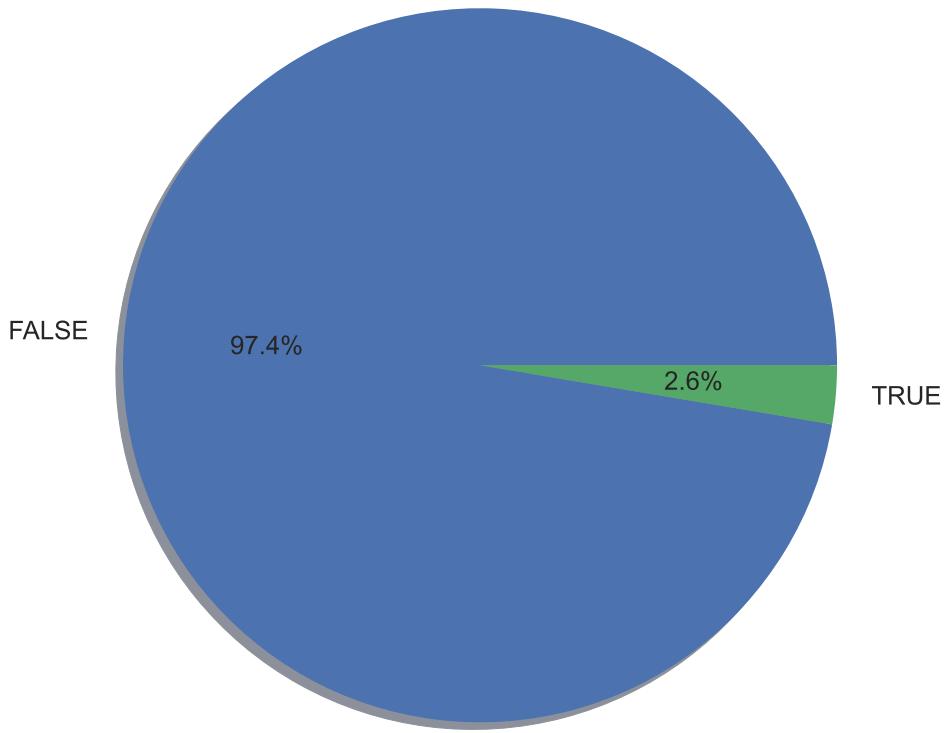
```
fig1, ax1 = plt.subplots()

ax1.pie(prop.values(), labels=prop.keys(), autopct='%.1f%%',

         shadow=True, startangle=0)

ax1.axis('equal') # Equal aspect ratio ensures that pie is drawn as a
                  # circle.

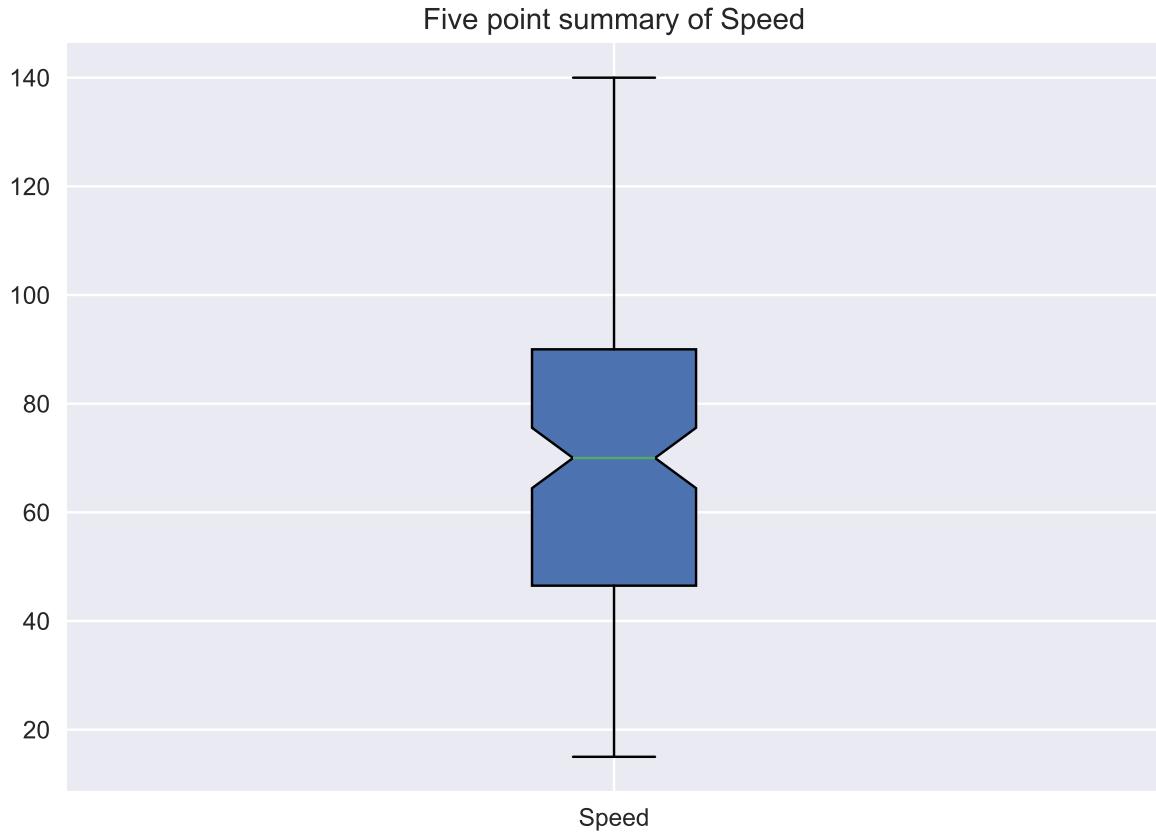
plt.show()
```



Box plots Box plots are helpful in visualizing the statistical summaries. The following code demonstrate the way of creating the box plot.

```
plt.title("Five point summary of Speed")
plt.boxplot(df['Speed'],patch_artist=True, notch=True,labels=['Speed'])
plt.show()
```

```
C:\Users\SIJUKSWAMY\AppData\Local\Temp\ipykernel_12808\1029903557.py:2: MatplotlibDeprecationWarning:
```



3.6.6 Data Visualization With Seaborn

Seaborn is a library built on top of Matplotlib that enables more sophisticated visualization and aesthetic plot formatting. Once you've mastered Matplotlib, you may want to move up to Seaborn for more complex visualizations.

For example, simply using the `Seaborn set()` method can dramatically improve the appearance of your Matplotlib plots. Let's take a look.

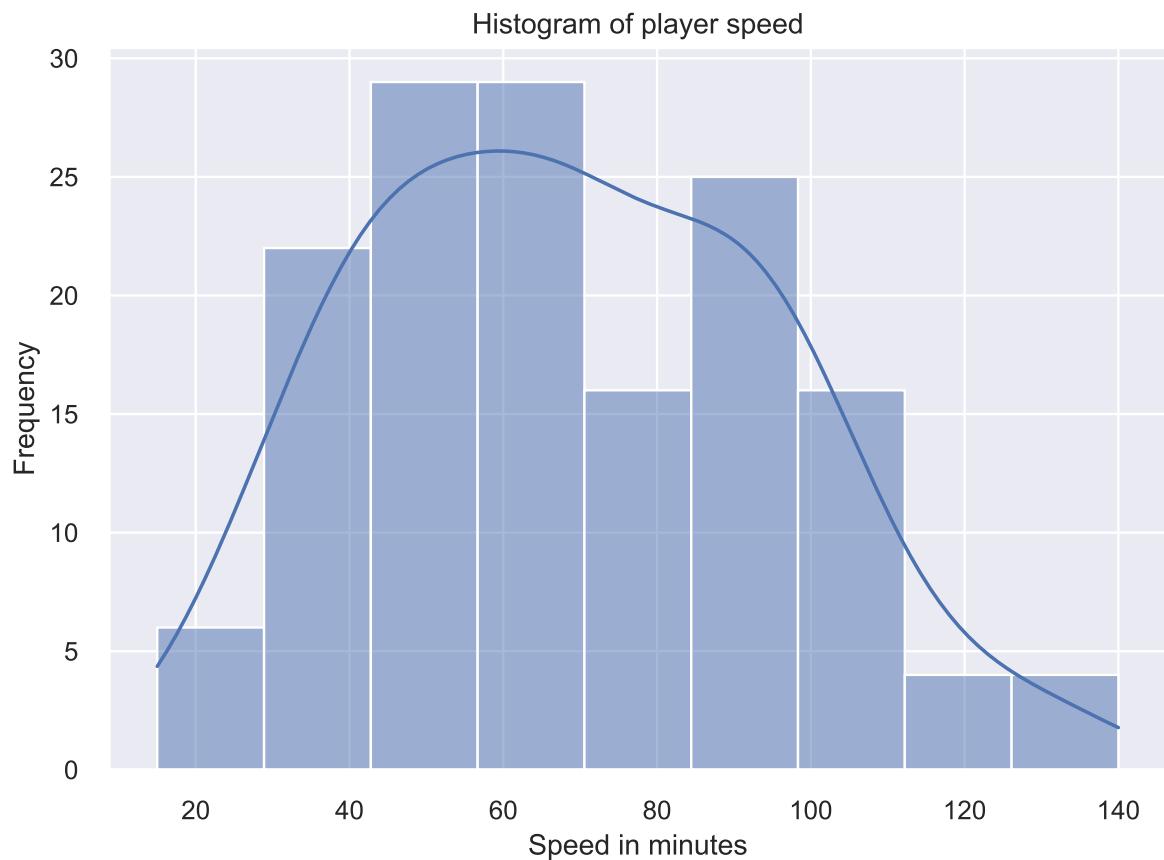
First, `import Seaborn as sns` and reformat all of the figures we generated. At the top of your script, write the following code and rerun:

```
import seaborn as sns
sns.set()
plt.show()
```

3.6.7 Histograms With Seaborn

To regenerate our histogram of the overall column, we use the `histplot` method on the Seaborn object:

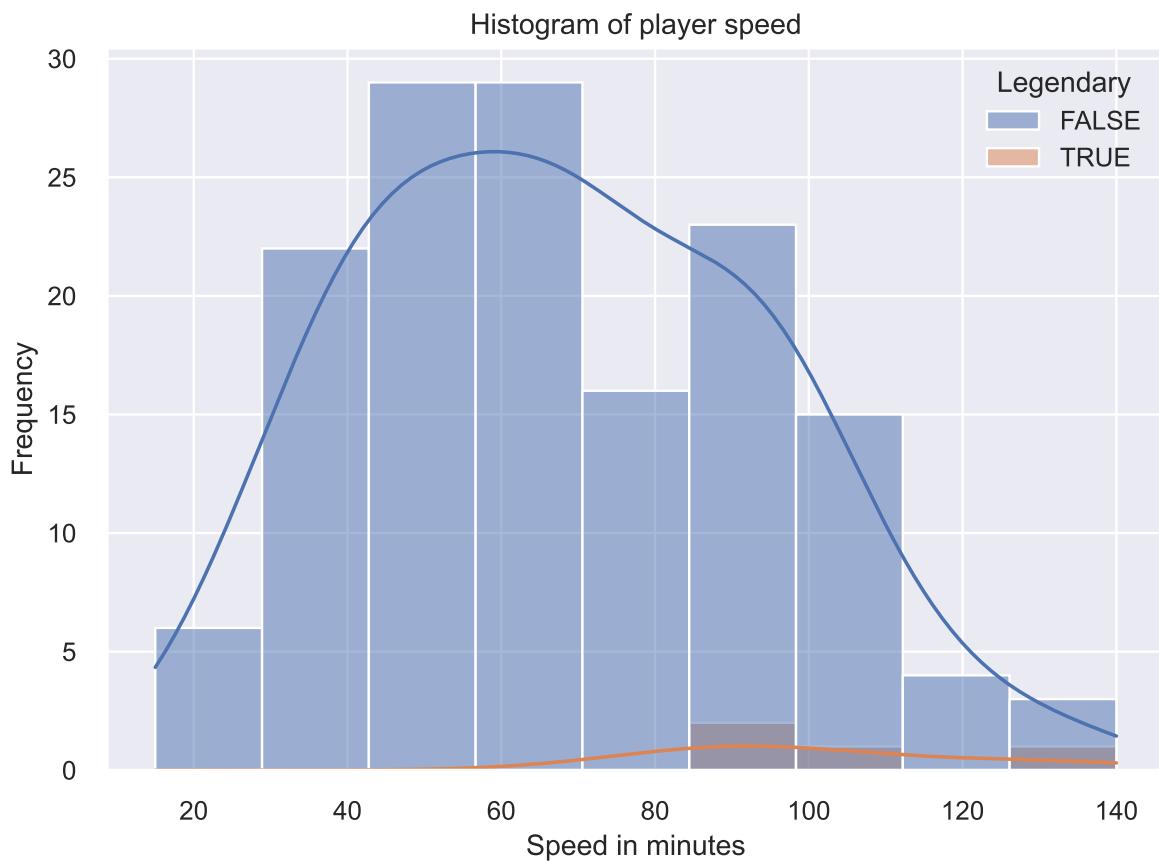
```
sns.histplot(df['Speed'], kde=True)
plt.xlabel('Speed in minutes')
plt.ylabel('Frequency')
plt.title('Histogram of player speed')
plt.show()
```



Now let's modify the histogram by including the feature Legendary.

```
sns.histplot(x='Speed', hue='Legendary', kde=True, data=df)
plt.xlabel('Speed in minutes')
plt.ylabel('Frequency')
```

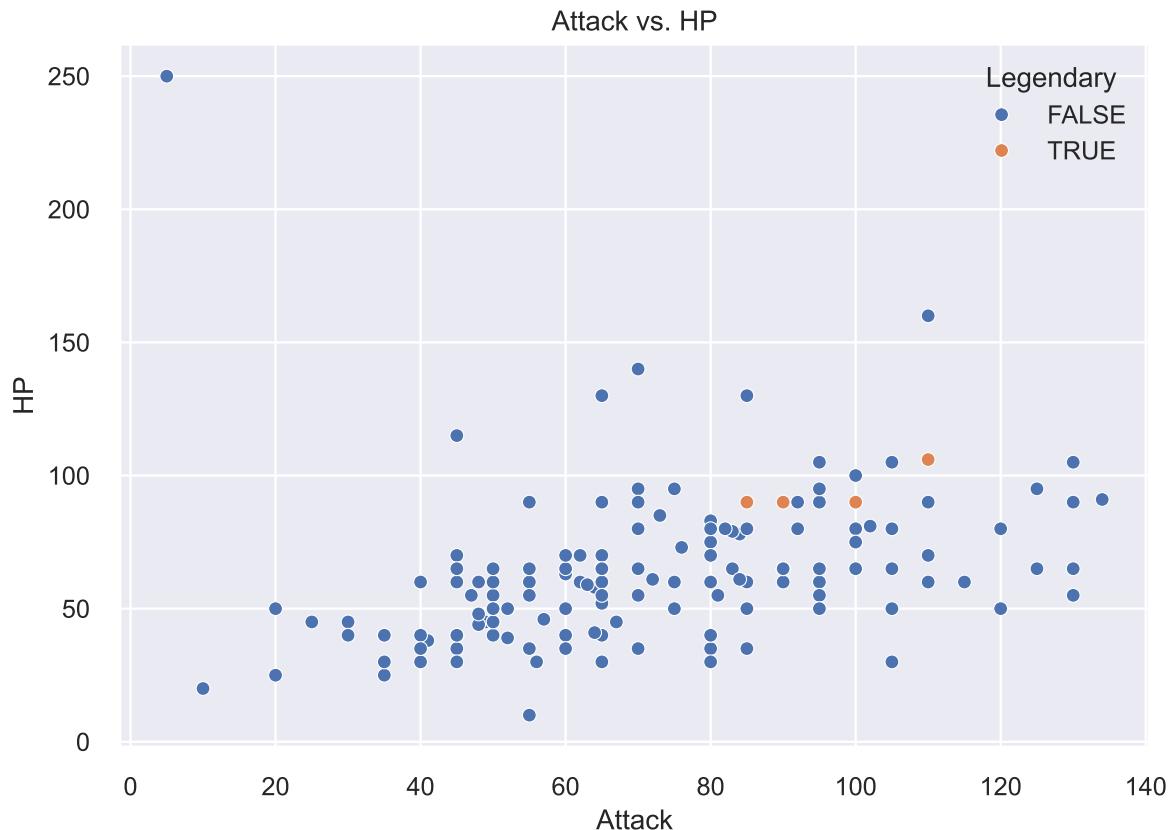
```
plt.title('Histogram of player speed')
plt.show()
```



3.6.8 Scatter Plots With Seaborn

Seaborn also makes generating scatter plots straightforward. Let's recreate the scatter plot from earlier:

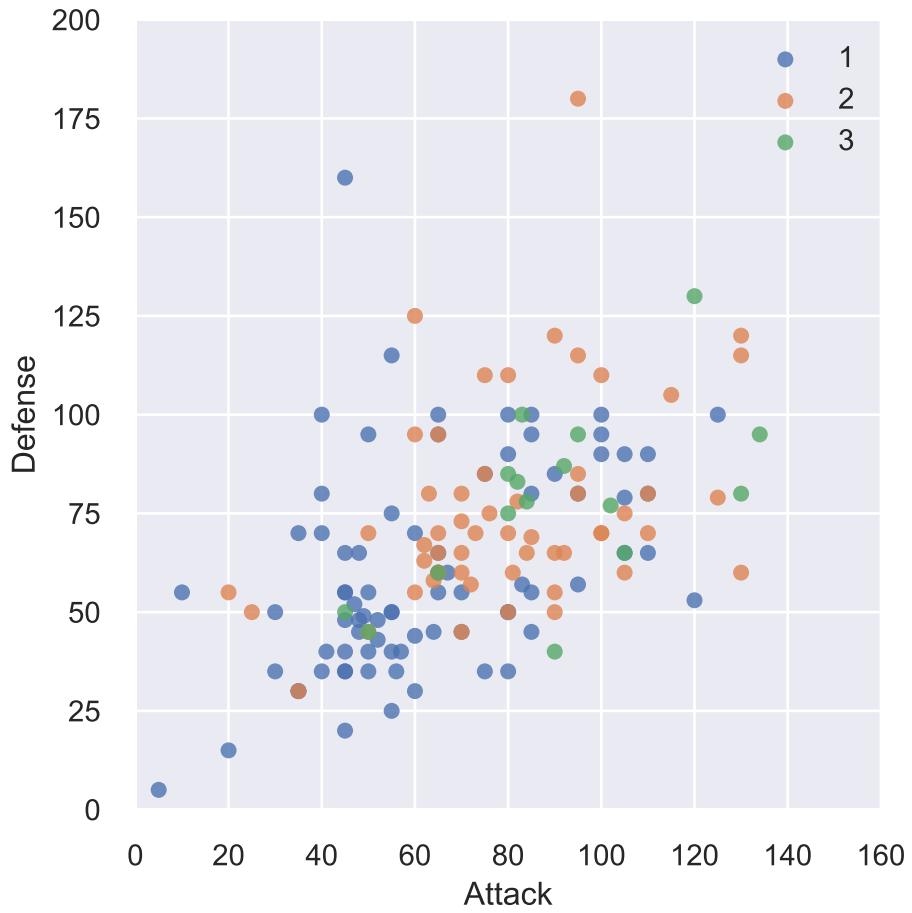
```
sns.scatterplot(x='Attack', y='HP', hue='Legendary', data=df)
plt.title('Attack vs. HP')
plt.show()
```



In the similar way, let's compare the Attack and Defense stats for our Pokémon over Stages

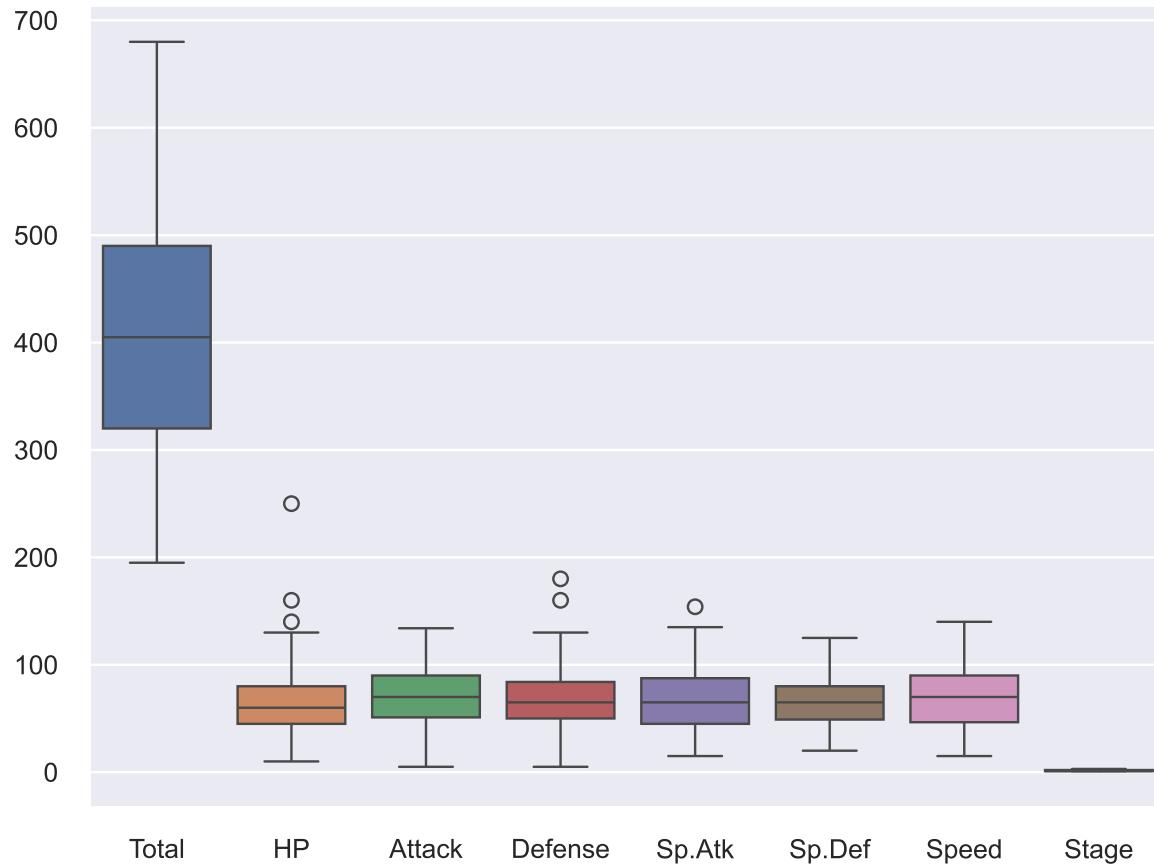
```
# Plot using Seaborn
sns.lmplot(x='Attack', y='Defense', data=df,
            fit_reg=False, legend=False,
            hue='Stage')

# Tweak using Matplotlib
plt.ylim(0, 200)
plt.xlim(0, 160)
plt.legend(loc='upper right')
plt.show()
```



Now let's witness the power of seaborn in creating boxplots of all numerical features in single line of code!

```
# Boxplot
plt.figure(figsize=(8,6)) # Set plot dimensions
sns.boxplot(data=df)
plt.show()
```



3.7 Module review

1. Write the NumPy methods for (a) matrix multiplication and (b) element-wise multiplication with suitable examples.
 - **Hint:** Use `numpy.matmul(A, B)` or `A @ B` for matrix multiplication and `numpy.multiply(A, B)` for element-wise multiplication.
2. What is the determinant of a matrix? Write the Python code to compute the determinant of $A = \begin{bmatrix} 3 & 2 \\ 5 & 7 \end{bmatrix}$ using NumPy.
 - **Hint:** Use `numpy.linalg.det(A)` to find the determinant.
3. Explain the difference between `numpy.dot()` and `numpy.matmul()` with suitable examples.
 - **Hint:** Discuss their usage for both vectors and matrices.

4. What is the condition number of a matrix? Write Python code to compute the condition number of $A = \begin{bmatrix} 4 & 7 \\ 2 & 6 \end{bmatrix}$.
- **Hint:** Use `numpy.linalg.cond(A)` to calculate the condition number.
5. Describe the process of normalizing a matrix. Write Python code to normalize the rows of $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$.
- **Hint:** Use `numpy.linalg.norm` and divide each row by its norm.
6. What is a transpose of a matrix? Write Python code to compute the transpose of $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$.
- **Hint:** Use `numpy.transpose(A)` or `A.T`.
7. Write Python code to compute the inverse of $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ using NumPy. Also, verify the result by multiplying A with its inverse.
- **Hint:** Use `numpy.linalg.inv(A)` for the inverse and `numpy.matmul()` to check the identity property.
8. Explain the role of the trace of a matrix in linear algebra. Write Python code to compute the trace of $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$.
- **Hint:** Use `numpy.trace(A)` to compute the trace.
9. What is matrix slicing? Demonstrate how to extract the first two rows and last two columns from $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$.
- **Hint:** Use NumPy slicing syntax: `A[:2, -2:]`.
10. Explain the difference between solving $Ax = b$ using `numpy.linalg.solve()` and directly computing $A^{-1}b$. Which method is computationally more efficient?
- **Hint:** Discuss how `numpy.linalg.solve(A, b)` avoids explicitly calculating the inverse of A , making it more efficient.
11. Write the NumPy methods for (a) outer product and (b) inner product with suitable examples.
- **Hint:** Use `numpy.outer(a, b)` for the outer product and `numpy.inner(a, b)` for the inner product. Example: $a = [1, 2]$ and $b = [3, 4]$.

12. What is the pseudo-inverse of a matrix? Write the Python code chunk to solve a system

of linear equations $Ax = b$, where $A = \begin{bmatrix} 2 & 3 \\ -1 & 3 \\ 2 & 4 \end{bmatrix}$ and $b = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$, using the pseudo-inverse.

Justify the reason for using the pseudo-inverse instead of A^{-1} .

- **Hint:** The pseudo-inverse is computed using the Moore-Penrose method (`numpy.linalg.pinv`). It is useful for non-square or rank-deficient matrices.

13. Explain various array concatenation routines from the NumPy library with suitable examples.

- **Hint:** Use `numpy.concatenate`, `numpy.vstack`, `numpy.hstack`, and `numpy.dstack` to merge arrays along different axes.

14. Describe the major differences between NumPy and SymPy in advanced linear algebra operations.

- **Hint:** NumPy focuses on numerical computations with high performance, while SymPy provides symbolic computation and exact results.

15. Demonstrate the use of `numpy.linalg.norm` to compute different norms of a matrix $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$.

- **Hint:** Use `ord='fro'` for the Frobenius norm, `ord=1` for the 1-norm, and `ord=np.inf` for the infinity norm.

16. What is the Kronecker product of two matrices? Find the Kronecker product of $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $B = \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix}$ using NumPy.

- **Hint:** Use `numpy.kron(A, B)` to compute the Kronecker product.

17. Explain how sparse matrices are represented in SciPy. Provide examples of common formats such as CSR, COO, and CSC.

- **Hint:** Use `scipy.sparse.csr_matrix`, `scipy.sparse.coo_matrix`, and `scipy.sparse.csc_matrix` for different formats.

18. Write Python code to compute the determinant and rank of a matrix $A = \begin{bmatrix} 5 & 2 \\ 1 & 4 \end{bmatrix}$ using NumPy.

- **Hint:** Use `numpy.linalg.det(A)` for determinant and `numpy.linalg.matrix_rank(A)` for rank.

19. What are broadcasting rules in NumPy? Explain with examples how broadcasting simplifies matrix operations.

- **Hint:** Broadcasting allows element-wise operations on arrays of different shapes. Examples include element-wise addition of a scalar and an array, and element-wise multiplication of arrays of different shapes.

20. Illustrate the computation of eigenvalues and eigenvectors of $A = \begin{bmatrix} 6 & 2 \\ 2 & 3 \end{bmatrix}$ using NumPy.

- **Hint:** Use `numpy.linalg.eig(A)` to compute eigenvalues and eigenvectors.

4 Linear Algebra for Advanced Applications

4.1 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.

4.2 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 $u_{11}, u_{12}, u_{13}, u_{22}, u_{23}, u_{33}, \dots$

4.2.1 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:

- $L = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix}$
- $U = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}$

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:

$$l_{21}u_{11} = a_{21} \Rightarrow l_{21} = \frac{a_{21}}{u_{11}}$$

$$l_{21}u_{12} + u_{22} = a_{22} \Rightarrow u_{22} = a_{22} - l_{21}u_{12}$$

$$l_{21}u_{13} + u_{23} = a_{23} \Rightarrow u_{23} = a_{23} - l_{21}u_{13}$$

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: $- u_{11} = a_{11}, u_{12} = a_{12}, u_{13} = a_{13} - l_{21} = \frac{a_{21}}{u_{11}}, u_{22} = a_{22} - l_{21}u_{12}, u_{23} = a_{23} - l_{21}u_{13} - l_{31} = \frac{a_{31}}{u_{11}}, l_{32} = \frac{a_{32} - l_{31}u_{12}}{u_{22}}, u_{33} = a_{33} - l_{31}u_{13} - l_{32}u_{23}$

4.2.2 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: - $L = \begin{pmatrix} 1 & 0 & 0 \\ 1.5 & 1 & 0 \\ 0.5 & 0.67 & 1 \end{pmatrix}$ - $U = \begin{pmatrix} 4 & 3 & 2 \\ 0 & -1.5 & -2 \\ 0 & 0 & 2.67 \end{pmatrix}$

4.2.3 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]]

```

i Note

Since there are many row transformations that reduce a given matrix into row echelon form. So the LU decomposition is not unique.

4.2.4 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

$$x = \begin{pmatrix} \frac{19}{9} \\ \frac{1}{3} \\ \frac{7}{18} \end{pmatrix}.$$

4.3 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 $A = LU$.

4.4 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]])
```

5 Spectral Decomposition

5.1 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.

5.2 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.

5.3 Spectral Decomposition: Detailed Concepts

5.3.1 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.

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 :

$$A\mathbf{v} = k\lambda\mathbf{v} \text{ for eigenvalue } \lambda.$$

5.3.2 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:

$$A = Q\Lambda Q^\top$$

Here, Q is an orthogonal matrix of eigenvectors (i.e., $Q^\top Q = I$), and Λ is a diagonal matrix of eigenvalues.

5.3.3 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).

5.3.4 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 .

5.3.5 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.

5.4 Mathematical Requirements for Spectral Decomposition

5.4.1 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.

5.4.2 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:

$$EV(\lambda = \lambda_1) = \begin{bmatrix} \lambda_1 - d \\ c \end{bmatrix}$$

5.4.3 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:

$$\lambda^3 - \text{tr}(A)\lambda^2 + (\text{sum of principal minors})\lambda - \det(A) = 0,$$

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

$\lambda = 3$, the coefficient matrix of $(A - \lambda I)X = 0$ 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

cross product of these two rows. $\therefore v_1 = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$.

When $\lambda = -1$, the coefficient matrix of $(A - \lambda I)X = 0$ 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

$(A - \lambda I)X = 0$ 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. $\therefore v_2 = \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}$.

5.4.4 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]]
```

5.4.5 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.]]
```

5.4.6 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:

$$f(A) = Qf(\Lambda)Q^\top$$

where $f(\Lambda)$ is the function applied element-wise to the eigenvalues in the diagonal matrix Λ .

6 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.

i 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.

6.0.1 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.

6.0.2 Python method for DR 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]]
```

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):

```
[[ 1.0000000e+00 -5.04131884e-17 -3.39864191e-17]
 [-5.04131884e-17  1.0000000e+00  2.30881074e-17]
 [-3.39864191e-17  2.30881074e-17  1.0000000e+00]]
```

6.1 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:

$$Ax = b$$

where: - A is the coefficient matrix, - x is the vector of unknowns (with size n), - b is the vector of constants (with size m).

6.1.1 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.

6.1.2 Challenges in Solving Overdetermined Systems

- 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.
- 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.

6.1.3 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.

6.1.4 Solving an Overdetermined System using QR Decomposition

Given an overdetermined system represented as $Ax = b$, 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:

$$Rx = Q^T b$$

3. Solve the Triangular System:

- Solve for 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.]

6.1.5 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:

$$\begin{aligned}x + 2y &= 3 \\2x + 4y &= 6 \\3x + 1y &= 5\end{aligned}$$

```

# 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: [1.0000000e+00 -2.73316113e-15 3.80986098e-15]
 Problem 5 - Ordinary Least Squares Solution: [1.0000000e+00 -6.77505297e-15 1.06049542e-15]

6.2 Module review

1. Diagonalize the matrix $A = \begin{bmatrix} 1 & 1 & 3 \\ 1 & 5 & 1 \\ 3 & 1 & 1 \end{bmatrix}$ and write the spectral decomposition of A .

Hint:

- To diagonalize a matrix, find the eigenvalues and eigenvectors of the matrix.
 - Compute the eigenvectors of A to form the matrix P , and diagonalize A using the formula $A = PDP^{-1}$ where D is the diagonal matrix of eigenvalues.
 - Spectral decomposition expresses the matrix as a sum of eigenvalue-weighted outer products of eigenvectors.
-

2. Find the eigenvalues and eigenvectors of the matrix $A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ and write the spectral decomposition.

Hint:

- To find the eigenvalues, solve the characteristic equation $\det(A - \lambda I) = 0$.
 - Once you have the eigenvalues λ , find the eigenvectors by solving $(A - \lambda I)v = 0$ for each eigenvalue.
 - Express A as a sum of eigenvalue-weighted outer products of its eigenvectors.
-

3. Compute the QR decomposition of the matrix $A = \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix}$.

Hint:

- Use Gram-Schmidt process or the `numpy.linalg.qr()` function to compute the QR decomposition.
 - The matrix A is factored into an orthogonal matrix Q and an upper triangular matrix R such that $A = QR$.
-

4. Derive the LU decomposition of the matrix $A = \begin{bmatrix} 4 & 3 & 1 \\ 6 & 3 & 2 \\ 3 & 1 & 2 \end{bmatrix}$.

Hint:

- Use Gaussian elimination to decompose the matrix into lower and upper triangular matrices.
 - The matrix A is written as $A = LU$, where L is a lower triangular matrix with 1s on the diagonal, and U is an upper triangular matrix.
-

5. Verify the rank-nullity theorem for the matrix $A = \begin{bmatrix} 7 & -3 & 5 \\ 9 & 11 & 2 \\ 16 & 8 & 7 \end{bmatrix}$.

Hint:

- The rank-nullity theorem states that the sum of the rank and nullity (dimension of the null space) of a matrix equals the number of its columns.
 - Compute the rank of the matrix A by finding the number of linearly independent rows or columns, and use this to determine the nullity.
-

6. What are the five important matrix decompositions in linear algebra? Compare them in terms of application.

Hint:

- The five important matrix decompositions include:
 - **LU Decomposition:** Used for solving systems of linear equations, especially in computational methods.
 - **QR Decomposition:** Primarily used in solving least squares problems and orthogonalization.
 - **Eigenvalue Decomposition:** Useful in spectral analysis and principal component analysis (PCA).
 - **Singular Value Decomposition (SVD):** Crucial for data compression, noise reduction, and dimensionality reduction.
 - **Cholesky Decomposition:** Applied in numerical optimization, especially for solving linear systems with symmetric, positive-definite matrices.
 - Compare their efficiency, use cases, and computational costs in different applications such as image processing, machine learning, etc.
-

7. If $A = \begin{bmatrix} 1 & 2 & 4 \\ 0 & 3 & 4 \\ 1 & -1 & -1 \end{bmatrix}$, find the eigenvalues and eigenvectors of A .

Hint:

- To find the eigenvalues of A , solve the characteristic equation $\det(A - \lambda I) = 0$. - After finding the eigenvalues λ , substitute them back into the equation $(A - \lambda I)v = 0$ to find the corresponding eigenvectors. - Eigenvectors should be normalized if required.
-

8. Perform the QR decomposition of the matrix $A = \begin{bmatrix} 4 & 1 \\ 3 & 2 \end{bmatrix}$.

Hint:

- Use Gram-Schmidt process or the `numpy.linalg.qr()` function to compute the QR decomposition. - The matrix A is decomposed into $A = QR$, where Q is an orthogonal matrix and R is an upper triangular matrix.
-

9. Compute the LU decomposition of the matrix $A = \begin{bmatrix} 2 & 3 & 1 \\ 3 & 4 & 2 \\ 1 & 2 & 3 \end{bmatrix}$.

Hint:

- Use Gaussian elimination or the `scipy.linalg.lu()` function to find the LU decomposition.
 - The matrix A is decomposed as $A = LU$, where L is a lower triangular matrix with 1s on the diagonal and U is an upper triangular matrix.
-

10. Find the Singular Value Decomposition (SVD) of the matrix $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$.

Hint:

- Use the `numpy.linalg.svd()` function to compute the SVD of matrix A . - The SVD decomposes A into $A = U\Sigma V^T$, where U and V are orthogonal matrices and Σ is a diagonal matrix containing the singular values.
-

7 Practical Uses Cases

7.1 Singular Value Decomposition (SVD) – An Intuitive and Mathematical Approach

Singular Value Decomposition (SVD) is one of the most powerful matrix factorization tools in linear algebra, extensively used in areas like data compression, signal processing, machine learning, and more. SVD generalizes the concept of diagonalization to non-square matrices, decomposing any $m \times n$ matrix A into three matrices with well-defined geometric interpretations.

7.2 The SVD Theorem

For any real or complex $m \times n$ matrix A , SVD states that:

$$A = U\Sigma V^T$$

Where: - U is an $m \times m$ orthogonal matrix (or unitary in the complex case), - Σ is an $m \times n$ diagonal matrix, with non-negative real numbers (the singular values of A) on the diagonal, - V^T is the transpose (or conjugate transpose in the complex case) of an $n \times n$ orthogonal matrix V .

These are range and null spaces for both the column and the row spaces.

$$\mathbf{C}^n = \mathcal{R}(\mathbf{A}^*) \oplus \mathcal{N}(\mathbf{A}) \quad (7.1)$$

$$\mathbf{C}^m = \mathcal{R}(\mathbf{A}) \oplus \mathcal{N}(\mathbf{A}^*) \quad (7.2)$$

The singular value decomposition provides an orthonormal basis for the four fundamental subspaces.

7.3 Intuition Behind SVD

The SVD can be understood geometrically:

- The columns of V form an orthonormal basis of the input space.
- The matrix Σ scales and transforms this space along the principal axes.
- The columns of U form an orthonormal basis of the output space, representing how the transformed vectors in the input space map to the output space.

SVD essentially performs three steps on any vector x :

1. **Rotation:** V^T aligns x with the principal axes.

2. **Scaling:** Σ scales along these axes.
3. **Rotation:** U maps the result back to the output space.

7.4 Spectral Decomposition vs. SVD

- **Spectral Decomposition** (also known as **Eigendecomposition**) applies to **square** matrices and decomposes a matrix A into $A = Q\Lambda Q^{-1}$, where Q is an orthogonal matrix of eigenvectors, and Λ is a diagonal matrix of eigenvalues.
- **SVD**, on the other hand, applies to **any** matrix (square or rectangular) and generalizes this idea by using singular values (which are always non-negative) instead of eigenvalues.

7.4.1 Comparison:

- **Eigenvectors and Eigenvalues:** Spectral decomposition only works if A is square and diagonalizable. It gives insight into the properties of a matrix (e.g., whether it is invertible).
- **Singular Vectors and Singular Values:** SVD works for any matrix and provides a more general and stable decomposition, useful even for non-square matrices.

7.5 Steps to Find U , Σ , and V^T

Given a matrix A , the SVD factors U , Σ , and V^T can be computed as follows:

1. **Compute $A^T A$ and find the eigenvalues and eigenvectors:**
 - The matrix V is formed from the eigenvectors of $A^T A$.
 - The singular values σ_i are the square roots of the eigenvalues of $A^T A$.
2. **Construct Σ :**

- Σ is a diagonal matrix where the non-zero entries are the singular values $\sigma_1, \sigma_2, \dots$, arranged in decreasing order.

3. Compute AA^T and find the eigenvectors:

- The matrix U is formed from the eigenvectors of AA^T .

4. Transpose V :

- The matrix V^T is simply the transpose of V .

7.6 Example

Let's consider a simple example where A is a 2×2 matrix:

$$A = \begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}$$

7.6.1 Step 1: Compute $A^T A$

$$A^T A = \begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix} = \begin{pmatrix} 10 & 6 \\ 6 & 10 \end{pmatrix}$$

Find the eigenvalues of $A^T A$:

$$\det(A^T A - \lambda I) = \det \begin{pmatrix} 10 - \lambda & 6 \\ 6 & 10 - \lambda \end{pmatrix} = 0$$

$$(10 - \lambda)^2 - 36 = 0 \quad \Rightarrow \quad \lambda = 16, \lambda = 4$$

The eigenvalues of $A^T A$ are 16 and 4, so the singular values of A are $\sigma_1 = 4$ and $\sigma_2 = 2$.

7.6.2 Step 2: Find V from the eigenvectors of $A^T A$

Solve $(A^T A - \lambda I)v = 0$ for each eigenvalue:

- For $\lambda = 16$, the eigenvector is $v_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$,
- For $\lambda = 4$, the eigenvector is $v_2 = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$.

Thus,

$$V = \begin{pmatrix} 1/\sqrt{2} & -1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{pmatrix}$$

7.6.3 Step 3: Construct Σ

The singular values $\sigma_1 = 4$ and $\sigma_2 = 2$, so:

$$\Sigma = \begin{pmatrix} 4 & 0 \\ 0 & 2 \end{pmatrix}$$

7.6.4 Step 4: Find U from the eigenvectors of AA^T

Similarly, compute AA^T :

$$AA^T = \begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix} = \begin{pmatrix} 10 & 6 \\ 6 & 10 \end{pmatrix}$$

Solve for the eigenvectors of AA^T (same as $A^T A$):

The eigenvectors are $u_1 = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix}$ and $u_2 = \begin{pmatrix} -1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix}$.

Thus,

$$U = \begin{pmatrix} 1/\sqrt{2} & -1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{pmatrix}$$

7.6.5 Step 5: Final SVD

We can now write the SVD of A as:

$$A = U\Sigma V^T$$

Where:

$$U = \begin{pmatrix} 1/\sqrt{2} & -1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{pmatrix}, \quad \Sigma = \begin{pmatrix} 4 & 0 \\ 0 & 2 \end{pmatrix}, \quad V^T = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ -1/\sqrt{2} & 1/\sqrt{2} \end{pmatrix}$$

Python code to find SVD of this example is given below.

```
import numpy as np

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

# Perform SVD decomposition
U, Sigma, VT = np.linalg.svd(A)

# Create Sigma matrix from singular values
Sigma_matrix = np.zeros((A.shape[0], A.shape[1]))
np.fill_diagonal(Sigma_matrix, Sigma)

# Print results
print("Matrix A:")
print(A)
print("\nU matrix:")
print(U)
print("\nSigma matrix:")
print(Sigma_matrix)
print("\nV^T matrix:")
print(VT)

# Verify the decomposition A = U * Sigma * V^T
A_reconstructed = U @ Sigma_matrix @ VT
print("\nReconstructed A (U * Sigma * V^T):")
print(A_reconstructed)
```

```

Matrix A:
[[3 1]
 [1 3]]

U matrix:
[[-0.70710678 -0.70710678]
 [-0.70710678  0.70710678]]

Sigma matrix:
[[4. 0.]
 [0. 2.]]

V^T matrix:
[[-0.70710678 -0.70710678]
 [-0.70710678  0.70710678]]

Reconstructed A (U * Sigma * V^T):
[[3. 1.]
 [1. 3.]]

```

7.7 Reconstructing Matrix A Using SVD

Given the Singular Value Decomposition (SVD) of a matrix A , the matrix can be reconstructed as a linear combination of low-rank matrices using the left singular vectors u_i , singular values σ_i , and the right singular vectors v_i^T .

The formula to reconstruct the matrix A is:

$$A = \sum_{i=1}^r \sigma_i u_i v_i^T$$

where: - r is the rank of the matrix A (i.e., the number of non-zero singular values), - σ_i is the i -th singular value from the diagonal matrix Σ , - u_i is the i -th column of the matrix U (left singular vectors), - v_i^T is the transpose of the i -th row of the matrix V^T (right singular vectors).

7.7.1 Breakdown of Terms:

- $u_i \in \mathbb{R}^m$ is a column vector from the matrix U (size $m \times 1$),
- $v_i^T \in \mathbb{R}^n$ is a row vector from the matrix V^T (size $1 \times n$),
- $\sigma_i \in \mathbb{R}$ is a scalar representing the i -th singular value.

Each term $\sigma_i u_i v_i^T$ represents a **rank-1 matrix** (the outer product of two vectors). The sum of these rank-1 matrices reconstructs the original matrix A .

7.7.2 Example:

For a matrix A , its SVD is represented as:

$$A = U\Sigma V^T = \sum_{i=1}^r \sigma_i u_i v_i^T$$

If the rank of A is 2, then the reconstructed form of A would be:

$$A = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T$$

Each term $\sigma_i u_i v_i^T$ corresponds to a **low-rank approximation** that contributes to the final matrix. By summing these terms, the full matrix A is obtained.

Python code demonstrating reconstruction is given below:

```
import numpy as np

# Define the matrix A and convert it to float64
A = np.array([[3, 1],
              [1, 3]], dtype=np.float64)

# Perform SVD
U, Sigma, VT = np.linalg.svd(A)

# Reconstruct A using the singular values and singular vectors
A_reconstructed = np.zeros_like(A) # This will be float64 now
for i in range(len(Sigma)-1):
    A_reconstructed += Sigma[i] * np.outer(U[:, i], VT[i, :])

print("Original matrix A:")
print(A)

print("\nReconstructed A from rank-1 matrices:")
print(A_reconstructed)
```

```
Original matrix A:
```

```
[[3. 1.]  
 [1. 3.]]
```

```
Reconstructed A from rank-1 matrices:
```

```
[[2. 2.]  
 [2. 2.]]
```

7.8 Singular Value Decomposition in Image Processing

7.8.1 Image Compression

SVD is widely used for compressing images. By approximating an image with a lower rank matrix, significant amounts of data can be reduced without a substantial loss in quality. The largest singular values and their corresponding singular vectors are retained, allowing for effective storage and transmission.

7.8.2 Noise Reduction

SVD helps in denoising images by separating noise from the original image data. By reconstructing the image using only the most significant singular values and vectors, the impact of noise (often associated with smaller singular values) can be minimized, resulting in a clearer image.

7.8.3 Image Reconstruction

In applications where parts of an image are missing or corrupted, SVD can facilitate reconstruction. By analyzing the singular values and vectors, missing data can be inferred and filled in, preserving the structural integrity of the image.

7.8.4 Facial Recognition

SVD is employed in facial recognition systems as a means to extract features. By decomposing facial images into their constituent parts, SVD helps identify key features that distinguish different faces, enhancing recognition accuracy.

7.8.5 Image Segmentation

In image segmentation, SVD can aid in clustering pixels based on their attributes. By reducing dimensionality, it helps identify distinct regions in an image, facilitating the separation of objects and backgrounds.

7.8.6 Color Image Processing

SVD can be applied to color images by treating each color channel separately. This allows for efficient manipulation, compression, and analysis of color images, improving overall processing performance.

7.8.7 Pattern Recognition

SVD is utilized in pattern recognition tasks where it helps to identify and classify patterns within images. By simplifying the data representation, SVD enhances the efficiency and accuracy of recognition algorithms.

7.8.8 Example

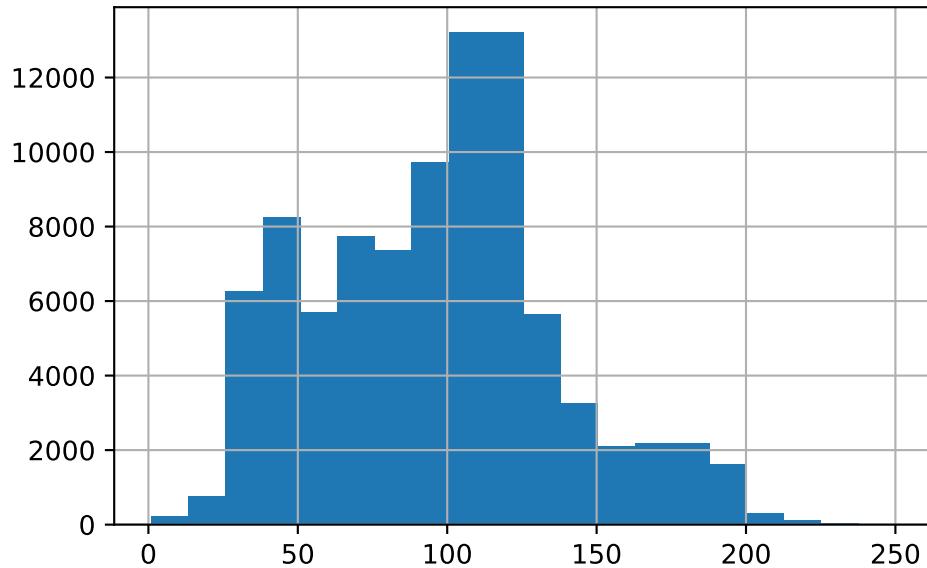
```
from PIL import Image
import urllib.request
import matplotlib.pyplot as plt
urllib.request.urlretrieve(
    'http://lenna.org/len_top.jpg',
    "input.jpg")

img = Image.open("input.jpg")
```

```
# convert to grayscale
imggray = img.convert('LA')
plt.figure(figsize=(8,6))
plt.imshow(imggray);
```



```
# creating image histogram
import pandas as pd
import numpy as np
imgmat = np.array(list(imggray.getdata(band=0)), float)
A=pd.Series(imgmat)
A.hist(bins=20)
```



```
# printing the pixel values
print(imgmat)
```

[80. 80. 79. ... 100. 94. 99.]

```
# dimension of the gray scale image matrix
imgmat.shape
```

(90000,)

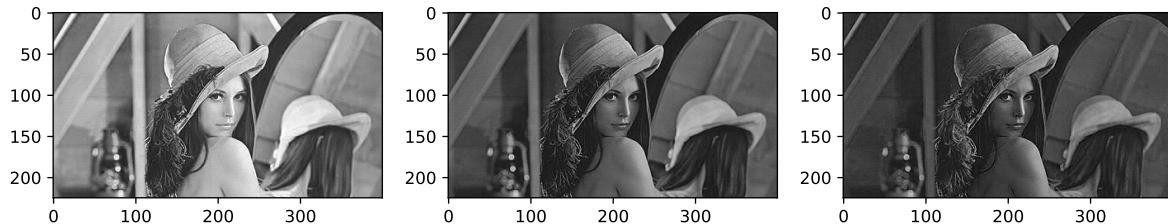
```
##loading an image and show it using matrices of pixel values
from skimage import io
f = "http://lenna.org/len_top.jpg" #url of the image
a = io.imread(f) # read the image to a tensor
c1=a[:, :, 0] # channel 1
c2=a[:, :, 1] # channel 2
c3=a[:, :, 2] # channel 3
print(c1)
# dimension of channel-1
c1.shape
```

[[109 109 108 ... 54 60 67]

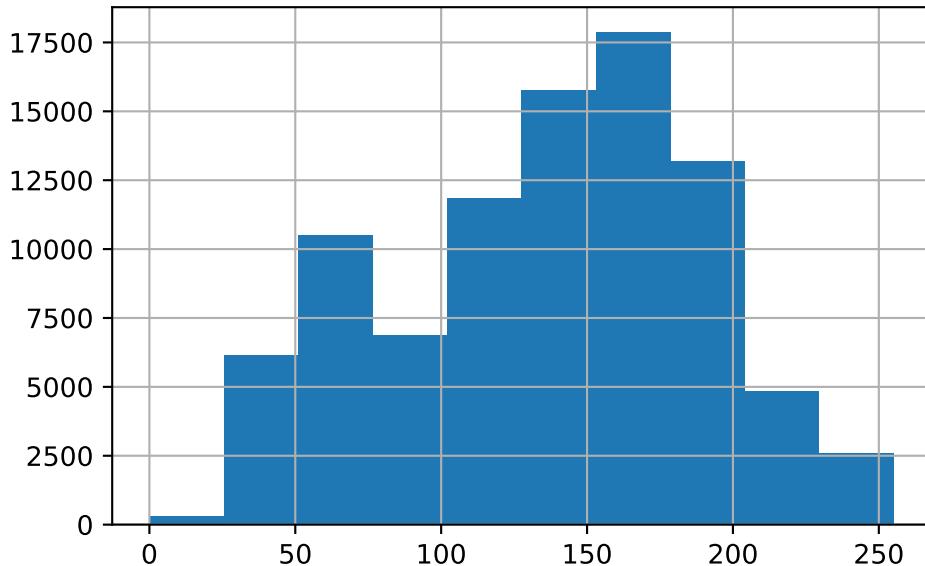
```
[112 111 107 ... 52 55 61]  
[111 110 107 ... 51 54 60]  
...  
[130 129 133 ... 122 119 125]  
[128 127 132 ... 125 119 123]  
[133 130 127 ... 139 133 140]]
```

(225, 400)

```
fig = plt.figure(figsize=(12, 3))  
ax1 = fig.add_subplot(131)  
ax2 = fig.add_subplot(132)  
ax3 = fig.add_subplot(133)  
ax1.imshow(c1, cmap='gray', vmin = 0, vmax = 255, interpolation='none')  
ax2.imshow(c2, cmap='gray', vmin = 0, vmax = 255, interpolation='none')  
ax3.imshow(c3, cmap='gray', vmin = 0, vmax = 255, interpolation='none')  
plt.show()
```



```
c1_array=np.array(list(c1)).reshape(-1)  
pd.Series(c1_array).hist()
```



```
## an application of matrix addition
plt.imshow(0.34*c1-0.2*c2-0.01*c3, cmap='gray', vmin = 0, vmax =
    255, interpolation='none')
plt.show()
```



```
#converting a grayscale image to numpy array
imgmat = np.array(list(imggray.getdata(band=0)), float)
imgmat.shape = (imggray.size[1], imggray.size[0])
```

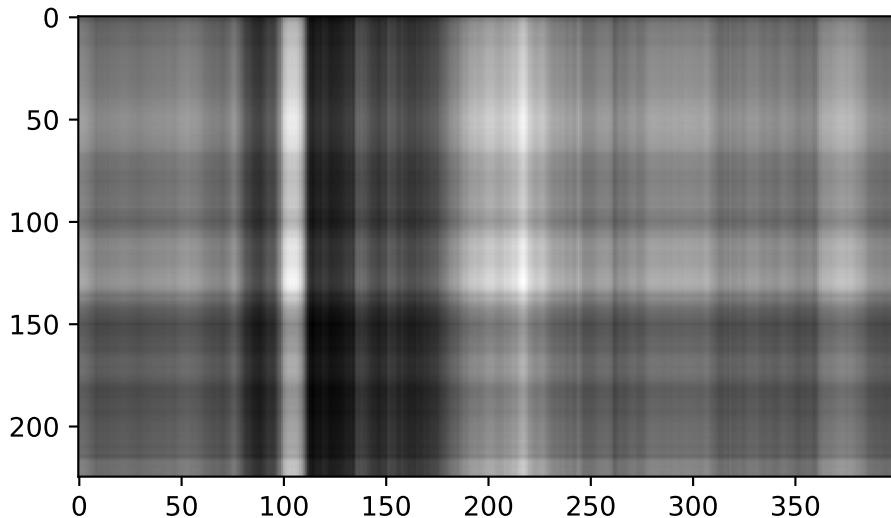
```
imgmat = np.matrix(imgmat)
plt.figure(figsize=(8,6))
plt.imshow(imgmat, cmap='gray');
```



As promised, one line of command is enough to get the singular value decomposition. U and V are the left-hand side and the right-hand side matrices, respectively. ‘sigma’ is a vector containing the diagonal entries of the matrix Σ . The other two lines reconstruct the matrix using the first singular value only. You can already guess the rough shape of the original image.

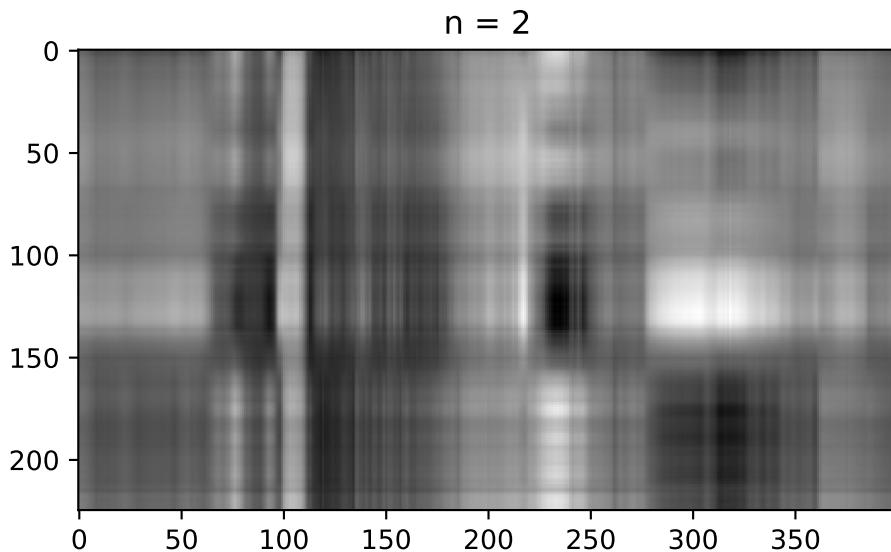
```
U, sigma, V = np.linalg.svd(imgmat)

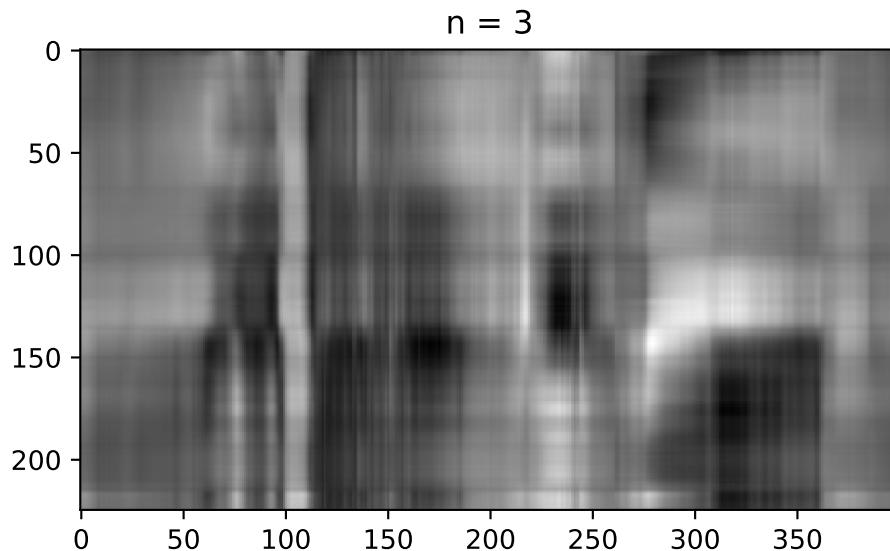
reconstimg = np.matrix(U[:, :1]) * np.diag(sigma[:1]) * np.matrix(V[:1, :])
plt.imshow(reconstimg, cmap='gray');
```



Let's see what we get when we use the second, third and fourth singular value as well.

```
for i in range(2, 4):
    reconstimg = np.matrix(U[:, :i]) * np.diag(sigma[:i]) * np.matrix(V[:i,
    :])
    plt.imshow(reconstimg, cmap='gray')
    title = "n = %s" % i
    plt.title(title)
    plt.show()
```

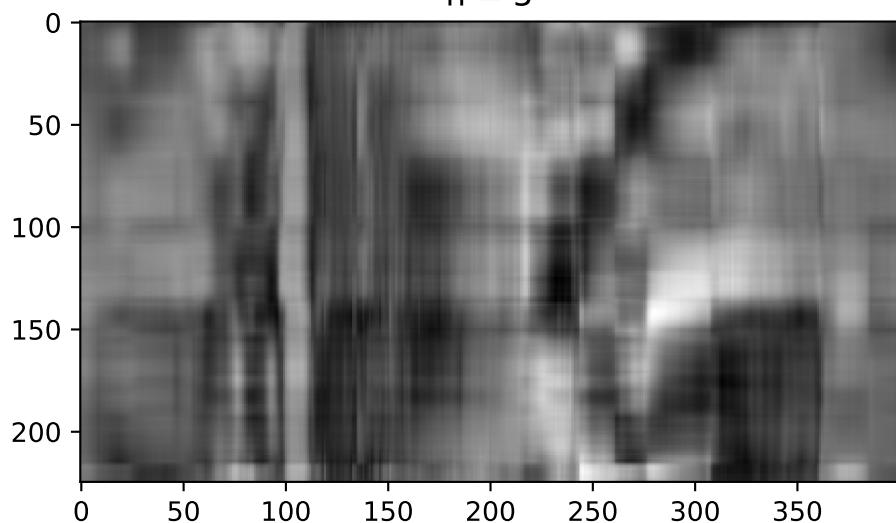




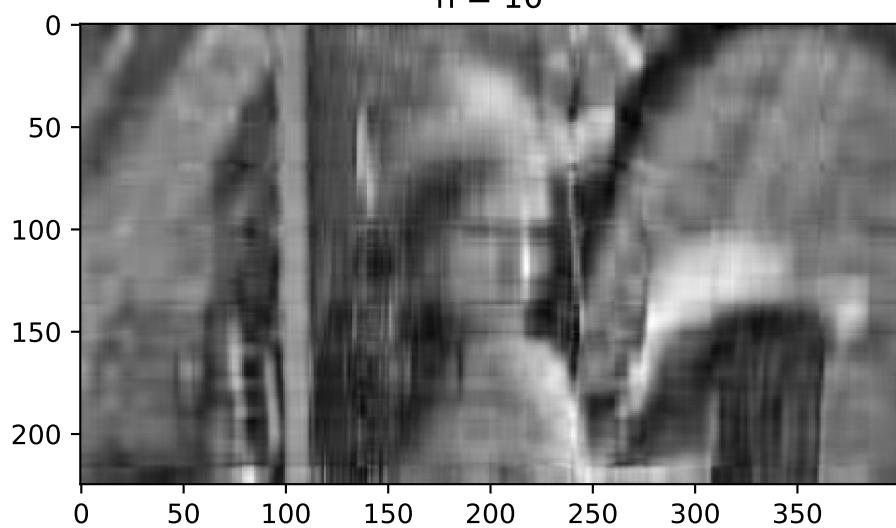
Now we let i run from 5 to 51, using a step width of 5. For $i = 50$, we already get a pretty good image!

```
for i in range(5, 51, 5):
    reconstimg = np.matrix(U[:, :i]) * np.diag(sigma[:i]) * np.matrix(V[:i,
    :])
    plt.imshow(reconstimg, cmap='gray')
    title = "n = %s" % i
    plt.title(title)
    plt.show()
```

$n = 5$



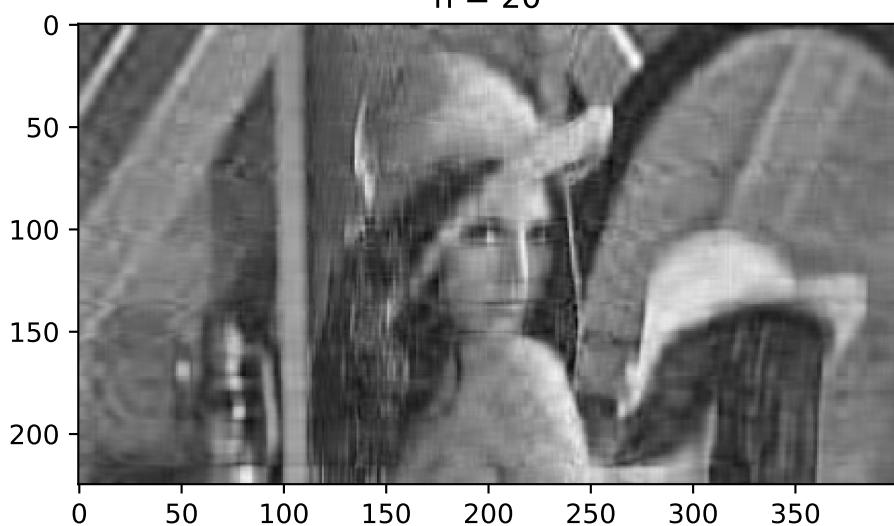
$n = 10$



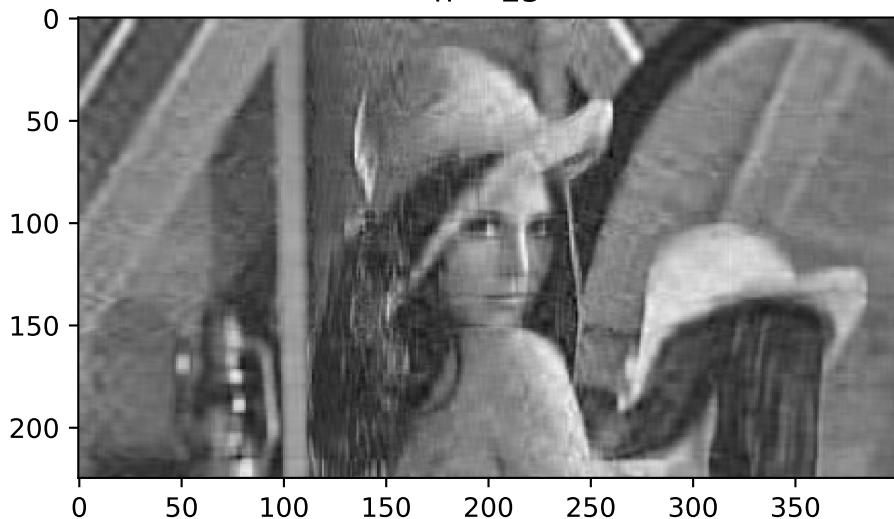
$n = 15$



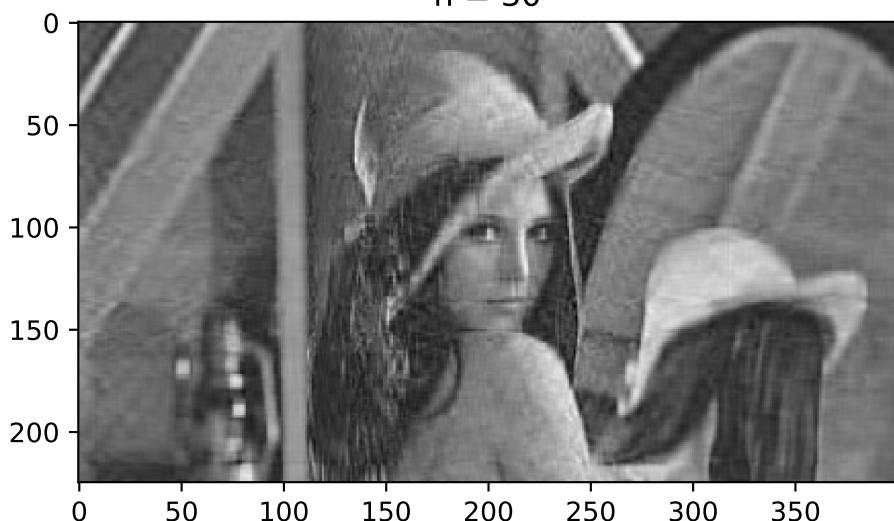
$n = 20$



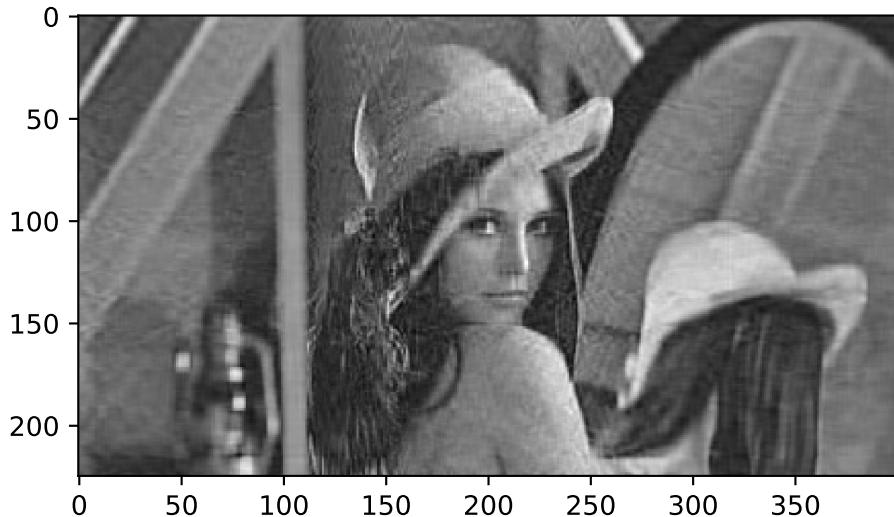
$n = 25$



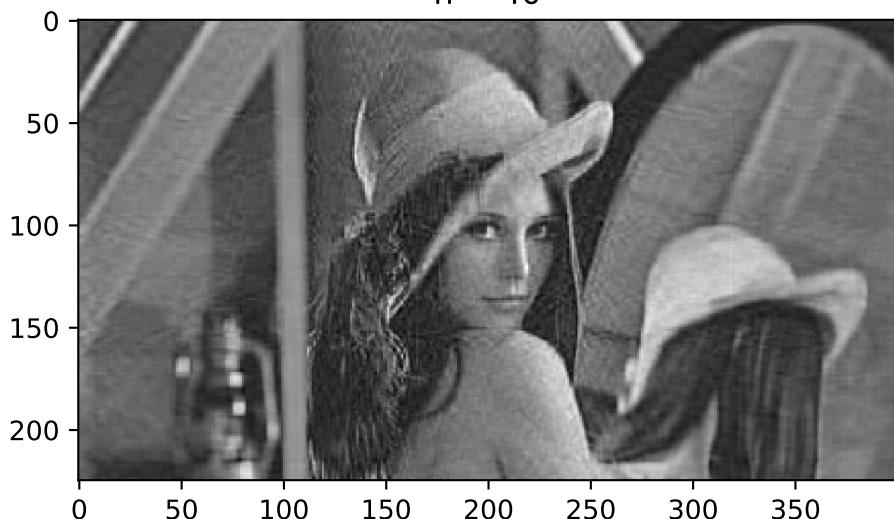
$n = 30$

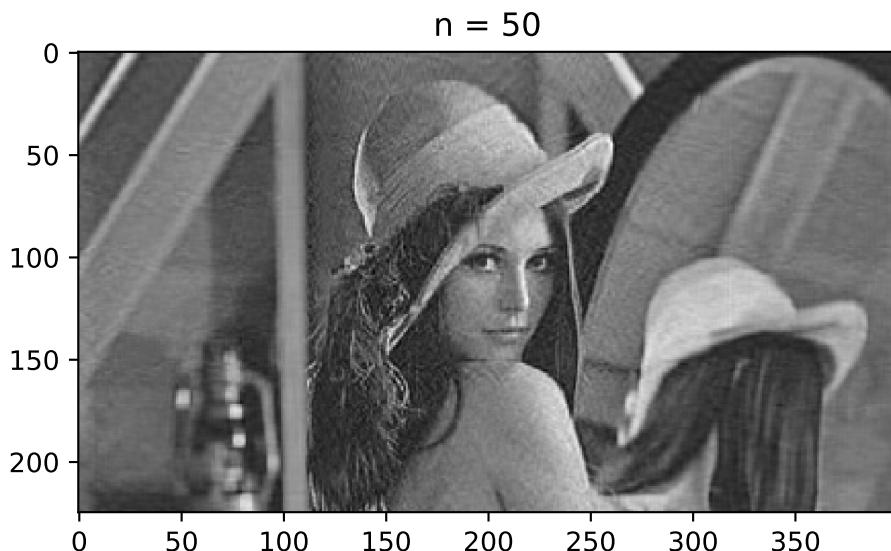
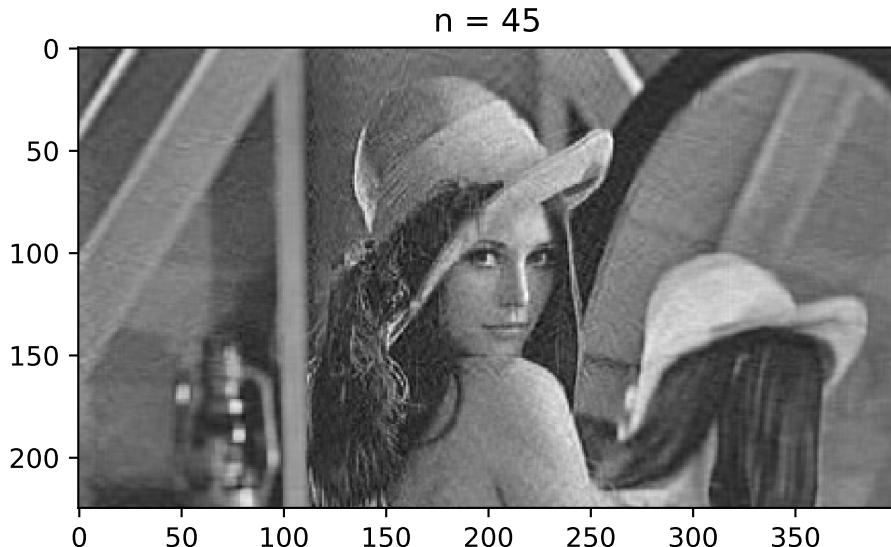


$n = 35$



$n = 40$





But how many singular values do we have after all? The following command gives us the number of entries in `sigma`. As it is the diagonal matrix, it is stored as a vector and we do not save the zero entries. We now output the number of singular values (the length of the vector `sigma`, containing the diagonal entries), as well as the size of the matrices U and V .

```
print("We have %d singular values." % sigma.shape)
print("U is of size", U.shape, ".")
print("V is of size", V.shape, ".")
print("The last, or smallest entry in sigma is", sigma[224])
```

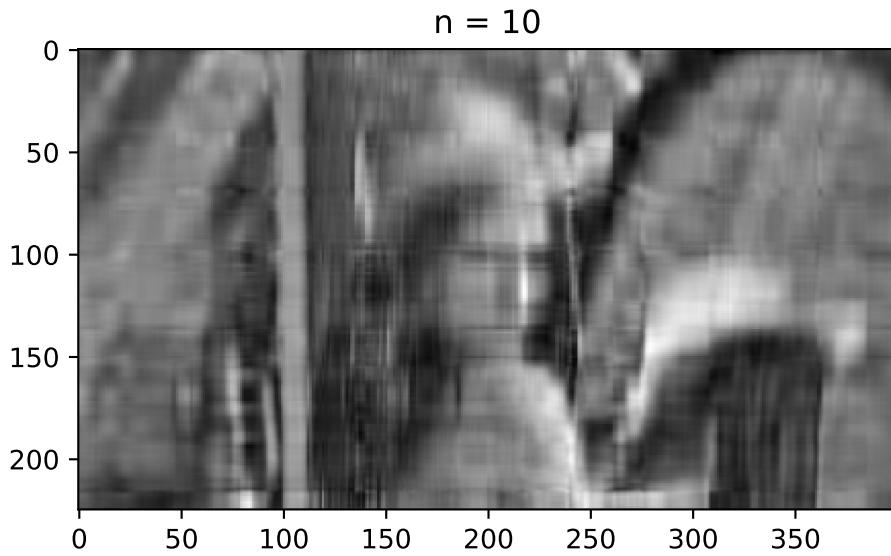
```
We have 225 singular values.
U is of size (225, 225) .
V is of size (400, 400) .
The last, or smallest entry in sigma is 9.637679189276597
```

As Python stores the whole singular value decomposition, we do not really save space. But as you saw in the first theoretical exercise of the 10th series, we do not have to compute the whole matrices U and V if we know that we only want to reconstruct the rank k approximation. How many numbers do you have to store for the initial matrix of the picture? How many numbers do you have to store if you want to reconstruct the rank k approximation only?

Use the following Cell to find an i large enough that you are satisfied with the quality of the image. Check, how much percent of the initial size you have to store. If your picture has a different resolution, you will have to correct the terms.

```
i = 10
reconstimg = np.matrix(U[:, :i]) * np.diag(sigma[:i]) * np.matrix(V[:i, :])
plt.imshow(reconstimg, cmap='gray')
title = "n = %s" % i
plt.title(title)
plt.show()

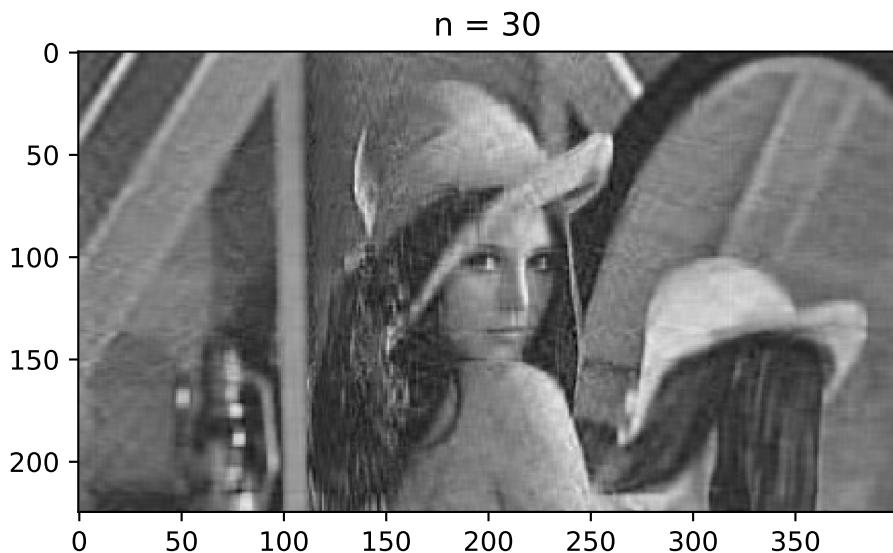
numbers = 400*i + i + 225* i
print("For this quality, we have to store %d numbers." % numbers)
```



For this quality, we have to store 6260 numbers.

If you really want to have a good quality, say you want to reconstruct using $r - 1$ singular values, where r is the total number of singular values, is it still a good idea to use the singular value decomposition?

```
i = 30
reconstimg = np.matrix(U[:, :i]) * np.diag(sigma[:i]) * np.matrix(V[:i, :])
plt.imshow(reconstimg, cmap='gray')
title = "n = %s" % i
plt.title(title)
plt.show()
```



7.9 Takeaway

Singular Value Decomposition provides a general framework for decomposing any matrix into orthogonal components, revealing the underlying structure of the matrix. SVD has numerous applications in machine learning, signal processing, and more. The method to find the matrices U , Σ , and V^T involves using the eigenvalues and eigenvectors of $A^T A$ and AA^T .

7.10 Principal Component Analysis

Principal Component Analysis (PCA) is fundamentally a technique used for dimensionality reduction, feature extraction, and data visualization. While it's widely used in various fields, its mathematical foundation lies in Singular Value Decomposition (SVD). This connection between PCA and SVD helps in understanding the underlying mechanics and simplifies PCA computations in high-dimensional spaces. In this section, we will explore PCA as a special case of SVD, present the mathematical derivations, and conclude with an advanced illustration involving image compression.

7.11 Principal Component Analysis as a Special Case of SVD

Principal Component Analysis (PCA) is a widely used technique for dimensionality reduction, feature extraction, and data compression. It is based on the mathematical foundations of linear algebra, particularly eigenvalue decomposition and singular value decomposition (SVD).

7.12 Problem Setting for PCA

Let $X \in \mathbb{R}^{m \times n}$ represent the data matrix, where: - m is the number of data samples, - n is the number of features for each sample.

To begin, we center the data by subtracting the mean of each feature from the dataset:

$$X_{\text{centered}} = X - \bar{X}$$

where \bar{X} is the mean vector for the columns of X .

7.13 Covariance Matrix

The covariance matrix C of the centered data X_{centered} is given by:

$$C = \frac{1}{m-1} X_{\text{centered}}^T X_{\text{centered}} \in \mathbb{R}^{n \times n}$$

The covariance matrix captures the relationships between the features of the data. PCA identifies the directions (principal components) that correspond to the maximum variance in the data.

7.14 Eigenvalue Decomposition

PCA reduces to finding the eigenvalue decomposition of the covariance matrix C :

$$Cv_i = \lambda_i v_i$$

where: - v_i is the i -th eigenvector, - λ_i is the i -th eigenvalue associated with v_i .

The eigenvectors v_i represent the principal components, and the eigenvalues λ_i represent the variance captured by each principal component.

7.15 Singular Value Decomposition (SVD)

PCA can be interpreted as a special case of Singular Value Decomposition (SVD). The SVD of the centered data matrix X_{centered} is:

$$X_{\text{centered}} = U\Sigma V^T$$

where: - $U \in \mathbb{R}^{m \times m}$ contains the left singular vectors, - $\Sigma \in \mathbb{R}^{m \times n}$ is the diagonal matrix of singular values, - $V \in \mathbb{R}^{n \times n}$ contains the right singular vectors.

The right singular vectors V are equivalent to the eigenvectors of the covariance matrix C , and the singular values σ_i are related to the eigenvalues λ_i by:

$$\lambda_i = \sigma_i^2$$

PCA Derivation Summary

To summarize:

1. Center the data by subtracting the mean of each feature.
2. Compute the covariance matrix $C = \frac{1}{m-1} X_{\text{centered}}^T X_{\text{centered}}$.
3. Perform eigenvalue decomposition of C to find the eigenvectors and eigenvalues.
4. Alternatively, perform SVD of X_{centered} . The right singular vectors are the principal components, and the singular values are related to the variance.

7.16 Applications of PCA

PCA has several practical applications:

- **Dimensionality Reduction:** PCA reduces the number of dimensions in a dataset while retaining the most important information.
- **Image Compression:** PCA can compress images by keeping only the principal components that capture the most variance.
- **Feature Extraction:** PCA is used in machine learning to extract the most significant features from high-dimensional datasets, speeding up computations and preventing overfitting.

7.17 Image Compression using PCA

Consider an image represented as a matrix $A \in \mathbb{R}^{n \times n}$, where each entry is a pixel intensity. To apply PCA for image compression:

1. Flatten the image into a matrix where each row represents a pixel.
2. Compute the covariance matrix of the image.
3. Perform eigenvalue decomposition or SVD on the covariance matrix.
4. Select the top k eigenvectors and project the image onto these eigenvectors.
5. Reconstruct the image using the reduced data.

7.17.1 Sample problems

Consider the following dataset:

$$X = \begin{pmatrix} 2 & 3 \\ 3 & 5 \\ 5 & 7 \end{pmatrix}$$

7.17.2 Step 1: Center the Data

Calculate the mean of each feature (column):

$$\text{Mean} = \left(\frac{\underline{2+3+5}}{3} \right) = \begin{pmatrix} 3.33 \\ 5 \end{pmatrix}$$

Subtract the mean from each data point to center the data:

$$X_{\text{centered}} = X - \text{Mean} = \begin{pmatrix} 2 - 3.33 & 3 - 5 \\ 3 - 3.33 & 5 - 5 \\ 5 - 3.33 & 7 - 5 \end{pmatrix} = \begin{pmatrix} -1.33 & -2 \\ -0.33 & 0 \\ 1.67 & 2 \end{pmatrix}$$

7.17.3 Step 2: Calculate the Covariance Matrix

Using the formula:

$$C = \frac{1}{n-1} X_{\text{centered}}^T X_{\text{centered}}$$

where $n = 3$ (the number of data points).

1. Transpose of Centered Data:

$$X_{\text{centered}}^T = \begin{pmatrix} -1.33 & -0.33 & 1.67 \\ -2 & 0 & 2 \end{pmatrix}$$

2. Multiply X_{centered}^T by X_{centered} :

$$X_{\text{centered}}^T X_{\text{centered}} = \begin{pmatrix} (-1.33)(-1.33) + (-0.33)(-0.33) + (1.67)(1.67) & (-1.33)(-2) + (-0.33)(0) + (1.67)(2) \\ (-2)(-1.33) + (0)(-0.33) + (2)(1.67) & (-2)(-2) + (0)(0) + (2)(2) \end{pmatrix}$$

Calculating each entry:

- First entry:

$$(-1.33)^2 + (-0.33)^2 + (1.67)^2 = 1.7689 + 0.1089 + 2.7889 = 4.6667$$

- Second entry:

$$(-1.33)(-2) + (1.67)(2) = 2.66 + 3.34 = 6$$

Thus,

$$X_{\text{centered}}^T X_{\text{centered}} = \begin{pmatrix} 4.67 & 6 \\ 6 & 8 \end{pmatrix}$$

3. Calculate the Covariance Matrix:

$$C = \frac{1}{3-1} \begin{pmatrix} 4.67 & 6 \\ 6 & 8 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 4.67 & 6 \\ 6 & 8 \end{pmatrix} = \begin{pmatrix} 2.335 & 3 \\ 3 & 4 \end{pmatrix}$$

7.17.4 Step 3: Calculate Eigenvalues and Eigenvectors

1. Eigenvalue Equation:

The characteristic polynomial is given by:

$$\det(C - \lambda I) = 0$$

where I is the identity matrix. So:

$$C - \lambda I = \begin{pmatrix} 2.335 - \lambda & 3 \\ 3 & 4 - \lambda \end{pmatrix}$$

The determinant is:

$$\det(C - \lambda I) = (2.335 - \lambda)(4 - \lambda) - (3)(3)$$

Expanding this, we get:

$$\det(C - \lambda I) = (2.335 \cdot 4 - 2.335\lambda - 4\lambda + \lambda^2 - 9) = \lambda^2 - (6.335)\lambda + (9.34 - 9) = \lambda^2 - 6.335\lambda + 0.34$$

2. Finding Eigenvalues:

Solving the characteristic equation:

$$\lambda^2 - 6.335\lambda + 0.34 = 0$$

Using the quadratic formula:

$$\lambda = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{6.335 \pm \sqrt{(-6.335)^2 - 4 \cdot 1 \cdot 0.34}}{2}$$

Calculate $b^2 - 4ac$:

$$(-6.335)^2 - 4 \cdot 1 \cdot 0.34 \approx 40.096225 - 1.36 = 38.736225$$

Thus, the eigenvalues are:

$$\lambda_1 = \frac{6.335 + \sqrt{38.736225}}{2}, \quad \lambda_2 = \frac{6.335 - \sqrt{38.736225}}{2}$$

3. Finding Eigenvectors:

For each eigenvalue λ , solve:

$$(C - \lambda I)\mathbf{v} = 0$$

Let's denote the eigenvalues we find as λ_1 and λ_2 . For each λ :

- Substitute λ into $(C - \lambda I)$ and set up the equation to find the eigenvectors.

7.17.5 Python Implementation

Here's the Python code to perform PCA step by step, corresponding to the tasks outlined above:

```
import numpy as np

# Given dataset
X = np.array([[2, 3],
              [3, 5],
              [5, 7]])

# Step 1: Center the data
mean = np.mean(X, axis=0)
X_centered = X - mean

# Step 2: Calculate the covariance matrix
cov_matrix = np.cov(X_centered, rowvar=False)

# Step 3: Calculate eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

# Display the results
print("Mean:\n", mean)
print("Centered Data:\n", X_centered)
print("Covariance Matrix:\n", cov_matrix)
print("Eigenvalues:\n", eigenvalues)
print("Eigenvectors:\n", eigenvectors)

# Step 4: Project data onto principal components
X_pca = X_centered @ eigenvectors
```

```
print("PCA Result (Projected Data):\n", X_pca)
```

```
Mean:  
[3.33333333 5.  
 ]  
Centered Data:  
[[-1.33333333 -2.  
 ]  
[-0.33333333 0.  
 ]  
[ 1.66666667 2.  
 ]]  
Covariance Matrix:  
[[2.33333333 3.  
 ]  
[3. 4.  
 ]]  
Eigenvalues:  
[0.05307638 6.28025695]  
Eigenvectors:  
[[-0.79612934 -0.60512649]  
 [ 0.60512649 -0.79612934]]  
PCA Result (Projected Data):  
[[-0.14874719 2.39909401]  
 [ 0.26537645 0.20170883]  
[-0.11662926 -2.60080284]]
```

7.18 Principal Component Analysis (PCA) for Image Reconstruction

In this example, we will use PCA to reduce the dimensionality of an image and then reconstruct it using only a fraction of the principal components. We will demonstrate this with the famous Lena image.

Step 1: Load and Display the Image

First, let's load the image using the PIL library and display it.

```
from PIL import Image  
import urllib.request  
import matplotlib.pyplot as plt  
  
# Load the image from URL  
urllib.request.urlretrieve('http://lenna.org/len_top.jpg', "input.jpg")
```

```
# Open and display the image
img = Image.open("input.jpg")
plt.imshow(img)
plt.title("Original Image")
plt.axis('off')
plt.show()
```

Original Image



Step 2: Convert the Image to Grayscale

To simplify the PCA process, we will work with the grayscale version of the image.

```
# Convert image to grayscale
img_gray = img.convert('L')
plt.imshow(img_gray, cmap='gray')
plt.title("Grayscale Image")
plt.axis('off')
plt.show()

# Convert to NumPy array
img_array = np.array(img_gray)
print("Image Shape:", img_array.shape)
```

Grayscale Image



Image Shape: (225, 400)

Step 3: Apply PCA to the Grayscale Image

We will apply PCA to compress the image by reducing the number of principal components used for reconstruction.

```
from sklearn.decomposition import PCA
import numpy as np

# Flatten the image into 2D (pixels x features)
img_flattened = img_array / 255.0 # Normalize pixel values
pca = PCA(n_components=50) # Choose 50 components

# Fit PCA on the image and transform
img_transformed = pca.fit_transform(img_flattened)

# Print the explained variance ratio
print("Explained Variance Ratio:", np.sum(pca.explained_variance_ratio_))
```

Explained Variance Ratio: 0.9480357612223579

Step 4: Reconstruct the Image using the Reduced Components

Now we use the transformed PCA components to reconstruct the image and compare it with the original.

```

# Reconstruct the image from PCA components
img_reconstructed = pca.inverse_transform(img_transformed)

# Rescale the image back to original pixel values (0-255)
img_reconstructed = (img_reconstructed * 255).astype(np.uint8)

# Plot the reconstructed image
plt.imshow(img_reconstructed, cmap='gray')
plt.title("Reconstructed Image with 50 Components")
plt.axis('off')
plt.show()

```

Reconstructed Image with 50 Components



Step 5: Compare the Original and Reconstructed Images

Finally, let's compare the original grayscale image with the PCA-reconstructed image to see how well it retains essential details.

```

# Plot original and reconstructed side by side
fig, ax = plt.subplots(1, 2, figsize=(10, 5))

# Original Grayscale Image
ax[0].imshow(img_array, cmap='gray')
ax[0].set_title("Original Grayscale Image")
ax[0].axis('off')

# Reconstructed Image

```

```
ax[1].imshow(img_reconstructed, cmap='gray')
ax[1].set_title("Reconstructed Image (50 Components)")
ax[1].axis('off')

plt.show()
```

Original Grayscale Image



Reconstructed Image (50 Components)



7.19 Micro Projects

1. Linear Regression

Link to GitHub

https://github.com/sijuswamy/AIML_Files

7.20 Sample questions for Lab Work

This lab exam covers Spectral Decomposition, Singular Value Decomposition (SVD), and Principal Component Analysis (PCA). For each question, perform the necessary computations and provide your answers.

7.20.1 Question 1: Spectral Decomposition of a Symmetric Matrix

For the matrix

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

find its spectral decomposition. Calculate the eigenvalues, eigenvectors, and express A as PDP^{-1} , where D is a diagonal matrix of eigenvalues and P contains the eigenvectors.

7.20.2 Question 2: Diagonalization of a Matrix

Given

$$B = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

find the eigenvalues and eigenvectors, and use them to express B in its spectral decomposition form.

7.20.3 Question 3: Eigenvalues and Eigenvectors of a Square Matrix

For the matrix

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

find the eigenvalues and eigenvectors. Verify that the matrix can be reconstructed from its eigenvalues and eigenvectors.

7.20.4 Question 4: Orthogonal Matrix Decomposition

For the matrix

$$D = \begin{bmatrix} 5 & 4 \\ 4 & 5 \end{bmatrix}$$

find its spectral decomposition and confirm whether the eigenvectors form an orthogonal matrix.

7.20.5 Question 5: Singular Value Decomposition (SVD)

For the matrix

$$E = \begin{bmatrix} 3 & 1 \\ -1 & 3 \end{bmatrix}$$

compute its Singular Value Decomposition. Write the matrix as $U\Sigma V^T$, where U and V are orthogonal matrices and Σ is a diagonal matrix with singular values.

7.20.6 Question 6: Rank-1 Approximation Using SVD

Using the matrix

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

perform a rank-1 approximation. Calculate the Frobenius norm of the difference between F and its rank-1 approximation.

7.20.7 Question 7: Matrix Compression Using SVD

For the matrix

$$G = \begin{bmatrix} 10 & 20 & 30 \\ 20 & 30 & 40 \\ 30 & 40 & 50 \end{bmatrix}$$

find a compressed form of G using only the most significant singular value. Provide the resulting approximation.

7.20.8 Question 8: SVD Data Reconstruction

For the matrix

$$H = \begin{bmatrix} 4 & 11 \\ 14 & 8 \\ 1 & 5 \end{bmatrix}$$

find its Singular Value Decomposition. Reconstruct the matrix from the SVD components.

7.20.9 Question 9: Principal Component Analysis (PCA)

Consider the dataset

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

Perform Principal Component Analysis (PCA) by first centering the data and then computing the principal components.

7.20.10 Question 10: Dimensionality Reduction with PCA

Using the dataset

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

apply PCA to reduce the data to one dimension. Project the data onto the principal component found.

7.21 Hint to Solutions.

```
import numpy as np
import scipy.linalg

# 1. Spectral Decomposition for Matrix A
A = np.array([[1, 2, 6], [3, 5, 6], [4, 6, 9]])
eigenvalues, eigenvectors = np.linalg.eig(A)
D = np.diag(eigenvalues)
P = eigenvectors
P_inv = np.linalg.inv(P)
reconstructed_A = P @ D @ P_inv
print("Eigenvalues:\n", eigenvalues)
```

```

print("Eigenvectors:\n", eigenvectors)
print("Reconstructed A:\n", reconstructed_A)

# 2. Diagonalization of Matrix B
B = np.array([[2, 1], [1, 2]])
eigenvalues_B, eigenvectors_B = np.linalg.eig(B)
D_B = np.diag(eigenvalues_B)
P_B = eigenvectors_B
P_inv_B = np.linalg.inv(P_B)
reconstructed_B = P_B @ D_B @ P_inv_B
print("\nEigenvalues of B:\n", eigenvalues_B)
print("Eigenvectors of B:\n", eigenvectors_B)
print("Reconstructed B:\n", reconstructed_B)

# 3. Eigenvalues and Eigenvectors for Matrix C
C = np.array([[4, 0, 0], [0, 3, 1], [0, 1, 2]])
eigenvalues_C, eigenvectors_C = np.linalg.eig(C)
D_C = np.diag(eigenvalues_C)
P_C = eigenvectors_C
reconstructed_C = P_C @ D_C @ np.linalg.inv(P_C)
print("\nEigenvalues of C:\n", eigenvalues_C)
print("Eigenvectors of C:\n", eigenvectors_C)
print("Reconstructed C:\n", reconstructed_C)

# 4. Orthogonal Matrix Decomposition of D
D = np.array([[5, 4], [4, 5]])
eigenvalues_D, eigenvectors_D = np.linalg.eig(D)
P_D = eigenvectors_D
reconstructed_D = P_D @ np.diag(eigenvalues_D) @ P_D.T
print("\nEigenvalues of D:\n", eigenvalues_D)
print("Eigenvectors of D (orthogonal):\n", P_D)
print("Reconstructed D:\n", reconstructed_D)

# 5. SVD of Matrix E
E = np.array([[3, 1], [-1, 3]])
U, S, Vt = np.linalg.svd(E)
Sigma = np.zeros((U.shape[0], Vt.shape[0]))
Sigma[:len(S), :len(S)] = np.diag(S)
reconstructed_E = U @ Sigma @ Vt
print("\nU:\n", U)
print("Sigma:\n", Sigma)
print("V^T:\n", Vt)

```

```

print("Reconstructed E:\n", reconstructed_E)

# 6. Rank-1 Approximation Using SVD for Matrix F
F = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
U_F, S_F, Vt_F = np.linalg.svd(F)
rank_1_F = np.outer(U_F[:, 0] * S_F[0], Vt_F[0, :])
frobenius_norm = np.linalg.norm(F - rank_1_F, 'fro')
print("\nRank-1 Approximation of F:\n", rank_1_F)
print("Frobenius Norm of Difference:\n", frobenius_norm)

# 7. SVD for Image Compression (Matrix G)
G = np.array([[10, 20, 30], [20, 30, 40], [30, 40, 50]])
U_G, S_G, Vt_G = np.linalg.svd(G)
compressed_G = np.outer(U_G[:, 0] * S_G[0], Vt_G[0, :])
print("\nCompressed Matrix G:\n", compressed_G)

# 8. SVD Data Reconstruction for Matrix H
H = np.array([[4, 11], [14, 8], [1, 5]])
U_H, S_H, Vt_H = np.linalg.svd(H)
Sigma_H = np.zeros((U_H.shape[0], Vt_H.shape[0]))
Sigma_H[:len(S_H), :len(S_H)] = np.diag(S_H)
reconstructed_H = U_H @ Sigma_H @ Vt_H
print("\nReconstructed H:\n", reconstructed_H)

# 9. PCA on Small Dataset X
X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
X_centered = X - np.mean(X, axis=0)
U_X, S_X, Vt_X = np.linalg.svd(X_centered)
print("\nCentered Data X:\n", X_centered)
print("Principal Components of X:\n", Vt_X.T)

# 10. Dimensionality Reduction with PCA for Dataset Y
Y = np.array([[2, 4], [3, 8], [5, 7], [6, 2], [7, 5]])
Y_centered = Y - np.mean(Y, axis=0)
U_Y, S_Y, Vt_Y = np.linalg.svd(Y_centered)
Y_projected = Y_centered @ Vt_Y.T[:, :1]
print("\n1D Projection of Y:\n", Y_projected)

```

Eigenvalues:
[15.41619849 -1. 0.58380151]
Eigenvectors:

```
[[ -0.38597937 -0.94280904 0.67025624]
 [-0.54141317 0.23570226 -0.71674952]
 [-0.74692149 0.23570226 0.19242323]]
```

Reconstructed A:

```
[[1. 2. 6.]
 [3. 5. 6.]
 [4. 6. 9.]]
```

Eigenvalues of B:

```
[3. 1.]
```

Eigenvectors of B:

```
[[ 0.70710678 -0.70710678]
 [ 0.70710678 0.70710678]]
```

Reconstructed B:

```
[[2. 1.]
 [1. 2.]]
```

Eigenvalues of C:

```
[1.38196601 3.61803399 4. ]
```

Eigenvectors of C:

```
[[ 0. 0. 1. ]
 [ 0.52573111 -0.85065081 0. ]
 [-0.85065081 -0.52573111 0. ]]
```

Reconstructed C:

```
[[4. 0. 0.]
 [0. 3. 1.]
 [0. 1. 2.]]
```

Eigenvalues of D:

```
[9. 1.]
```

Eigenvectors of D (orthogonal):

```
[[ 0.70710678 -0.70710678]
 [ 0.70710678 0.70710678]]
```

Reconstructed D:

```
[[5. 4.]
 [4. 5.]]
```

U:

```
[[ -0.9486833 0.31622777]
 [ 0.31622777 0.9486833 ]]
```

Sigma:

```
[[3.16227766 0. ]
 [0. 3.16227766]]
```

```

V^T:
[[ -1. -0.]
 [ 0.  1.]]
Reconstructed E:
[[ 3.  1.]
 [-1.  3.]]
Rank-1 Approximation of F:
[[1.73621779 2.07174246 2.40726714]
 [4.2071528 5.02018649 5.83322018]
 [6.6780878 7.96863051 9.25917322]]
Frobenius Norm of Difference:
1.068369514554709

Compressed Matrix G:
[[14.27105069 20.7349008 27.19875091]
 [20.7349008 30.12645113 39.51800146]
 [27.19875091 39.51800146 51.83725201]]

Reconstructed H:
[[ 4. 11.]
 [14. 8.]
 [ 1. 5.]]
Centered Data X:
[[ -3. -3.]
 [-1. -1.]
 [ 1.  1.]
 [ 3.  3.]]
Principal Components of X:
[[ 0.70710678 -0.70710678]
 [ 0.70710678 0.70710678]]

1D Projection of Y:
[[-0.34611992]
 [-3.22299204]
 [-1.32087901]
 [ 3.45810614]
 [ 1.43188483]]

```

7.22 Module review

1. Write the Singular Value Decomposition of $A = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$ and reconstruct the highest scaled component of A .

Hint:

- To perform SVD, use `numpy.linalg.svd()` to decompose the matrix A into $A = U\Sigma V^T$, where U and V are orthogonal matrices and Σ is a diagonal matrix containing the singular values.
 - After performing the SVD, reconstruct the highest scaled component of A using the first singular value and its corresponding singular vectors.
-

2. Explain the Principal Component Analysis (PCA) as a special case of the Singular Value Decomposition (SVD).

Hint:

- PCA is a dimensionality reduction technique that projects data onto a set of orthogonal axes called principal components.
 - SVD is used in PCA to decompose the data matrix X into $X = U\Sigma V^T$. The columns of U are the principal components, and Σ contains the scaling factors.
 - PCA involves selecting the top components based on the largest singular values in Σ .
-

3. Explain the applications of the Singular Value Decomposition (SVD) in image processing.

Hint:

- SVD is used in image compression by approximating the original image with a reduced set of singular values and vectors, effectively reducing storage requirements.
 - It is also used in noise reduction, where small singular values are discarded, preserving only the significant components of an image.
 - SVD is widely applied in face recognition and feature extraction as it reduces the dimensionality while retaining important features of the image.
-

4. Write the steps in Principal Component Analysis (PCA) of the matrix A so as to get the first two principal components using NumPy functions.

Hint:

- Center the matrix by subtracting the mean of each column from the respective column elements.
 - Compute the covariance matrix of the centered data.
 - Perform Singular Value Decomposition (SVD) on the covariance matrix using `numpy.linalg.svd()`.
 - The first two principal components are the first two columns of the matrix U obtained from the SVD of the covariance matrix.
 - Optionally, plot the projected data to visualize the results.
-

5. Given a matrix $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, perform PCA and extract the top two principal components.

Hint:

- Center the matrix by subtracting the mean of each column.
 - Compute the covariance matrix.
 - Perform SVD on the covariance matrix using `numpy.linalg.svd()`.
 - The first two principal components will be the first two columns of the matrix U .
 - Project the original data onto these components to reduce the dimensionality.
-

6. Explain the difference between the Singular Value Decomposition (SVD) and Eigenvalue Decomposition (EVD).

Hint:

- SVD is applicable to any matrix, while EVD is only defined for square matrices.
 - SVD decomposes a matrix into three components: $A = U\Sigma V^T$, where U and V are orthogonal matrices, and Σ is a diagonal matrix containing singular values.
 - EVD decomposes a square matrix into $A = V\Lambda V^{-1}$, where V contains the eigenvectors and Λ is a diagonal matrix with eigenvalues.
 - SVD is more general and can be applied to non-square and non-symmetric matrices.
-

7. Given the matrix $A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$, perform Singular Value Decomposition (SVD) and extract the first singular value.

Hint:

- Use `numpy.linalg.svd()` to compute the SVD of A .
- The singular values are the entries of the diagonal matrix Σ . The first singular value corresponds to the largest singular value in Σ .
- Reconstruct A using the computed U , Σ , and V^T .

-
8. How is the Singular Value Decomposition (SVD) used in Latent Semantic Analysis (LSA) for text mining?

Hint:

- LSA is used for extracting the underlying structure in text data by reducing the dimensionality of the term-document matrix.
- In LSA, the term-document matrix is decomposed using SVD to obtain a low-rank approximation of the matrix.
- This allows for capturing the most important latent semantic structures by selecting the largest singular values and corresponding vectors, leading to better clustering and classification of text data.

-
9. For a given matrix $A = \begin{bmatrix} 1 & 2 & 4 \\ 0 & 1 & 3 \\ 1 & 1 & 5 \end{bmatrix}$, compute the rank of the matrix using its Singular Value Decomposition (SVD).

Hint:

- The rank of the matrix A is equal to the number of non-zero singular values in Σ .
- Use `numpy.linalg.svd()` to perform the SVD of A and count how many of the singular values are greater than a small threshold (e.g., 10^{-10}).
- The rank is the number of non-zero singular values in the diagonal matrix Σ .

-
10. Using the matrix $A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \\ 3 & 5 \end{bmatrix}$, calculate the singular values and vectors using SVD and visualize the result.

Hint:

- Apply `numpy.linalg.svd()` to compute the singular values and vectors of A .
- Use the `matplotlib` library to visualize the data points and the transformation through the singular vectors.
- The singular values indicate the importance of each principal component, and the vectors describe the directions of the largest variance in the data.

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>.
- Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. <https://doi.org/10.1093/comjnl/27.2.97>.
- Strang, Gilbert. 2020. *Linear Algebra for Everyone*. SIAM.
- . 2022. *Introduction to Linear Algebra*. SIAM.
- Virtanen, Pauli, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, et al. 2020. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python.” *Nature Methods* 17: 261–72. <https://doi.org/10.1038/s41592-019-0686-2>.