

Linear Algebra for Data Science

A Report Submitted in Partial Fulfilment of the Requirements for the
SN Bose Internship Program, 2025

Submitted by

Mansi Bansal

M.Sc. Mathematics

National Institute of Technology, Tiruchirapalli

Under the guidance of

Dr. Mausumi Sen

Professor

Department of Mathematics

National Institute of Technology Silchar



Department of Mathematics
NATIONAL INSTITUTE OF TECHNOLOGY SILCHAR
Assam

June-July, 2025

DECLARATION

“Linear Algebra for Data Science”

We declare that the art on display is mostly comprised of our own ideas and work, expressed in our own words. Where other people’s thoughts or words were used, we properly cited and noted them in the reference materials. We have followed all academic honesty and integrity principles.

Mansi Bansal
M.Sc Mathematics, NIT Trichy

Department of Mathematics
National Institute of Technology Silchar, Assam

ACKNOWLEDGEMENT

I feel incredibly fortunate to have worked under the guidance of Prof. Mausumi Sen, whose mentorship and perceptive feedback made this “Linear Algebra for Data Science” internship genuinely transformative. Her helpful suggestions for this entire effort and co-operation are gratefully thanked. She showed exceptional patience—even with her busy schedule, she always made time to guide me.

I’m also deeply grateful to NIT Silchar and the SN Bose Summer Internship Program 2025 for providing this invaluable opportunity. The supportive environment has been instrumental in shaping my growth as an aspiring data scientist. My thanks also go to my fellow interns and the faculty and staff at NIT Silchar for their collaboration and kindness.

Finally, a special shout-out to my family and friends—their unwavering encouragement and faith in me carried me through even the toughest moments.

Mansi Bansal
M.Sc Mathematics, NIT Trichy

Department of Computer Mathematics
National Institute of Technology Silchar, Assam

ABSTRACT

During the SN Bose Summer Internship Program 2025 at NIT Silchar, I undertook a focused internship on Linear Algebra for Data Science, guided by Dr. Mausumi Sen. This report presents my exploration into the foundational concepts of linear algebra—such as eigenvalues, matrix decompositions, and vector transformations—and their direct application in data science tasks. Over the course of eight weeks, I engaged in problem-solving sessions, implemented algorithms in Python, and developed a deeper understanding of how linear algebra enables machine learning models and dimensional reduction techniques.

This internship was valuable both in terms of learning and personal growth. With guidance from Dr. Mausumi Sen and support, I was able to connect some mathematical concepts with real-world data work. By working through problems and applying linear algebra in Python, I became more confident in using math to solve data-related challenges. This experience has strengthened my understanding in mathematics is key to doing meaningful work in data science.

Contents

1	Introduction	5
2	Basic Structures & Operations	6
2.1	Scalars, Vectors, and Matrices	6
2.2	Fundamental Operations	6
2.3	Types Of Matrices	9
2.4	Inverse and Determinant	10
3	Solving Linear Systems: Methods & Theory	12
3.1	Row echelon form (REF) & Reduced REF	12
3.2	Matrix Rank	12
3.3	Gaussian Elimination	13
3.4	Cramer's Rule	16
3.5	Homogenous & Non-homogenous System	17
4	Vector Spaces & Their Applications	19
4.1	Vector Space	19
4.2	Some Basic Concepts	20
4.3	Applications of Vector Spaces in Data Science	24
5	Eigenvalues & Diagonalization	25
5.1	Eigenvalues and Eigenvectors	25
5.2	Diagonalization	27
5.3	Applications of Eigen-Decomposition	28
6	Orthogonal & Projection	30
6.1	Orthogonality	30
6.2	Projection	30
6.3	Gram–Schmidt process	31
6.4	Applications of Orthogonality & Gram–Schmidt	31
7	Matrix Decomposition	33
7.1	LU Decomposition:	33
7.2	QR Decomposition	34
7.3	Cholesky Decomposition	36
7.4	Singular Value Decomposition (SVD)	37
8	Conclusion	39

Chapter 1

Introduction

In data science, we often work with datasets containing many columns or features, and linear algebra helps us manage and understand them more easily. In this report, we will see why linear algebra is important in data science, how its tools make data more meaningful and manageable, and explore its examples. Also learn about the challenges of working with high-dimensional data and strategies like dimensionality reduction to simplify analysis without losing valuable information.

Throughout this report, we'll discover how vectors and matrices serve as the primary structures for organizing data; how matrix operations—such as multiplication, decomposition, and transformation—enable efficient data manipulation; and why dimension reduction, using techniques like eigen-decomposition and PCA, is vital for simplifying complexity while capturing essential information .

We'll also see by solving a system of linear equations how we can estimate parameters and optimize objectives and how eigenvalues and eigenvectors guide us in understanding the variability within data—key for pattern discovery and feature extraction .

In this journey through Linear Algebra for Data Science, you'll learn both the theory and hands-on application—using Python to implement matrix manipulations, decompositions, and real-world data transformations. This report demonstrates how strong mathematical foundations translate directly into robust, interpretable, and scalable data science workflows.

Why linear algebra is essential in data science?

Linear algebra is essential because it treats datasets—whether tabular records, images, or text embeds—as vectors and matrices, allowing us to manipulate data efficiently through matrix operations. These operations form the backbone of many data tasks, such as transforming features, normalizing data, and combining variables into meaningful structures. Moreover, advanced techniques like PCA and SVD use eigenvalues and eigenvectors to reduce the number of dimensions while preserving most of the useful information, enabling simpler and faster data processing without significant loss of insight .

Chapter 2

Basic Structures & Operations

2.1 Scalars, Vectors, and Matrices

1. **Scalars:** A scalar is the most basic data unit, representing a single numerical value. It is a zero-dimensional quantity. Examples include a single temperature reading, a person's age (e.g., 30), etc.
2. **Vectors:** A vector is an ordered collection of scalars, representing a one-dimensional array of numbers. It is an ordered list of numbers and in computing data science (NumPy in python) it's commonly called 1 dimensional array.

```
import numpy as np
a=np.array([1,2]) # vector
print(a)
```

Figure 2.1: Vector in python using NumPy

3. **Matrices:** A Matrix is a rectangular array of numbers arranged in rows and columns. It is the most common and powerful structure for representing entire datasets in data science.

```
import numpy as np
matrix=np.array([[1,2,3],[4,5,6],[7,8,9]]) # matrix
print(matrix)
```

Figure 2.2: Matrix in python using NumPy

2.2 Fundamental Operations

Vector Operations

Vector operations are essential tools for handling data efficiently and extracting meaningful insights:

1. **Vector Addition and Subtraction:** We combine or compare data points element-wise to aggregate information or measure changes.

$$\vec{u} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}, \quad \vec{v} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \implies \vec{u} + \vec{v} = \begin{pmatrix} u_1 + v_1 \\ u_2 + v_2 \\ \vdots \\ u_n + v_n \end{pmatrix}$$

2. **Scalar Multiplication:** Scaling vectors by a constant helps normalize data or adjust feature magnitudes before analysis.

$$\alpha \in R, \quad \vec{v} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \implies \alpha \vec{v} = \begin{pmatrix} \alpha v_1 \\ \alpha v_2 \\ \alpha v_3 \end{pmatrix}$$

3. **Dot Product (Inner Product):** This operation yields a single number, measuring similarity or projection between vectors—fundamental for tasks like cosine similarity and feature comparisons.

$$\vec{v} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}, \quad \vec{w} = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} \implies \vec{v} \cdot \vec{w} = v_1 w_1 + v_2 w_2 + v_3 w_3$$

4. **Vector Norm:** Calculating a vector's length (its norm) is important for normalization, distance computations, and feature scaling .

$$\|\mathbf{v}\|_2 = \sqrt{v_1^2 + v_2^2 + v_3^2}$$

Now let's see these using python:

```
import numpy as np
a=np.array([1,2])           # vector
b=np.array([3,2])
print(a+b)                  # vector addition
print(a-b)                  # vector subtraction
print(2*a)                  # scalar multiplication
print(np.dot(a,b))          # dot product [a.b=(1.3)+(2.2)]
import math
print(math.sqrt(1**2+2**2)) # norm of a vector
```

Figure 2.3: Vector Operations in python using NumPy

Matrix Operations

Matrix operations are key tools that help data scientists efficiently transform, organize, and analyze complex datasets:

1. **Matrix Addition/Substraction:** To add/subtract two matrices, add/subtract the corresponding elements.

NOTE: Dimensions must be the same.

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix} \implies A + B = \begin{pmatrix} a+e & b+f \\ c+g & d+h \end{pmatrix}$$

```
import numpy as np
A=np.array([[1,2],[3,4]]) # matrix 1
B=np.array([[5,6],[7,8]]) # matrix 2
print(A+B)                # matrix addition
print(A-B)                # matrix subtraction
```

Figure 2.4: Matrix Addition/Substraction in python using NumPy

2. **Matrix Multiplication:** Matrix multiplication is a fundamental operation in linear algebra with profound implications across various fields, especially in data science and machine learning. Matrix multiplication involves a more complex process that combines rows from the first matrix with columns from the second.

Let us understand by an example:

for 2X2 matrix,

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix} \implies AB = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}$$

Example for 3X3 matrix

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

$$AB = \begin{pmatrix} 1 \cdot 9 + 2 \cdot 6 + 3 \cdot 3 & 1 \cdot 8 + 2 \cdot 5 + 3 \cdot 2 & 1 \cdot 7 + 2 \cdot 4 + 3 \cdot 1 \\ 4 \cdot 9 + 5 \cdot 6 + 6 \cdot 3 & 4 \cdot 8 + 5 \cdot 5 + 6 \cdot 2 & 4 \cdot 7 + 5 \cdot 4 + 6 \cdot 1 \\ 7 \cdot 9 + 8 \cdot 6 + 9 \cdot 3 & 7 \cdot 8 + 8 \cdot 5 + 9 \cdot 2 & 7 \cdot 7 + 8 \cdot 4 + 9 \cdot 1 \end{pmatrix} = \begin{pmatrix} 30 & 24 & 18 \\ 84 & 69 & 54 \\ 138 & 114 & 90 \end{pmatrix}$$

```
import numpy as np
A=np.array([[1,2],[3,4]]) # matrix 1
B=np.array([[5,6],[7,8]]) # matrix 2
C=A.dot(B)               # matrix multiplication
print(C)
```

Figure 2.5: Matrix Multiplication in python using NumPy

3. **Transpose of a Matrix:** The transpose of a matrix A, denoted A^T , essentially flips A over its diagonal by turning all rows into columns and vice versa. If A is an

m x n matrix then the transpose A^T is an n x m matrix.
Example:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \implies A^T = \begin{bmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \\ a_{13} & a_{23} \end{bmatrix}$$

```
import numpy as np
A=np.array([[0,2,3],[-2,0,6],[-3,-6,0],[8,6,5]]) # matrix
transpose_A=A.T                                # matrix transpose
print(transpose_A)
```

Figure 2.6: Matrix Transpose in python using NumPy

2.3 Types Of Matrices

1. **Identity Matrix:** Diagonal entries are 1 and rest are zeros.

Example: $I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

```
import numpy as np
A=np.eye(3)    # 3X3 identity matrix
print(A)
```

Figure 2.7: Identity Matrix in python using NumPy

2. **Zero Matrix:** A zero matrix, also known as a null matrix, is a matrix in which all the elements are zero.

Example: $A = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$

```
import numpy as np
A=np.zeros((3,3))    # 3X3 zeros matrix
print(A)
```

Figure 2.8: Null Matrix in python using NumPy

3. **Symmetric Matrix:** A symmetric matrix is a square matrix that is equal to its transpose matrix.

Mathematically, a matrix A is symmetric if $A = A^T$. The elements on the main diagonal (from top left to bottom right) are the same, and the elements are mirrored

about this main diagonal.

Example:

$$A = \begin{bmatrix} a & b & c \\ b & d & e \\ c & e & f \end{bmatrix}, \quad \text{where } A = A^\top$$

4. **Diagonal Matrix:** A diagonal matrix is a square matrix where all the elements outside the principal diagonal are zero. Diagonal matrix is also symmetric matrix. Example:

$$D = \begin{bmatrix} d_1 & & & \\ & d_2 & & \\ & & \ddots & \\ & & & d_n \end{bmatrix}$$

```
import numpy as np
A=np.diag((3,1,2))    # 3X3 diagonal matrix
print(A)
```

Figure 2.9: Diagonal Matrix in python using NumPy

5. **Upper and Lower Triangular Matrix:** An upper triangular matrix is a matrix where all the entries below the main diagonal are zero. A lower triangular matrix is a matrix where all the entries above the main diagonal are zero. Example: Upper and lower respectively.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} \ell_{11} & 0 & 0 \\ \ell_{21} & \ell_{22} & 0 \\ \ell_{31} & \ell_{32} & \ell_{33} \end{bmatrix}$$

```
import numpy as np
A=np.triu([[1,2,3],[4,5,6],[7,8,9]])    # 3X3 upper triangular matrix
print(A)
B=np.tril([[1,2,3],[4,5,6],[7,8,9]])    # 3X3 lower triangular matrix
print(B)
```

Figure 2.10: Triangular Matrix in python using NumPy

2.4 Inverse and Determinant

1. **Determinant:** A scalar value is computed from the elements of a square matrix and denoted as $\det(A)$ or $|A|$.

$$\det(A) = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

```
import numpy as np
A=np.array([[1,2,3],[-2,1,6],[-3,-6,0]]) # matrix
print(np.linalg.det(A))                 # determinant
```

Figure 2.11: Matrix Determinant in python using NumPy

1. **Inverse of Matrix:** The inverse of a square matrix A is indicated as A^{-1} , where $A \cdot A^{-1} = 1$. If determinant of matrix is 0 then it is invertible.

Example: For 2X2 matrix,

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \implies A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix}, \quad \det(A) = a_{11}a_{22} - a_{12}a_{21}$$

```
import numpy as np
A=np.array([[1,2,3],[-2,1,6],[-3,-6,0]]) # matrix
print(np.linalg.inv(A))                 # inverse
```

Figure 2.12: Matrix Determinant in python using NumPy

Chapter 3

Solving Linear Systems: Methods & Theory

3.1 Row echelon form (REF) & Reduced REF

A matrix is in **Row Echelon Form** if:

1. All nonzero rows are above any rows of all zeros.
2. The leading entry (first non-zero number from the left called pivot) of each nonzero row is to the right of the leading entry of the row above.
3. All entries below each leading entry are zeros.

A matrix is in **Reduced Row Echelon Form** if it meets all REF criteria plus:
If all above three conditions are satisfied with given below

1. Each leading entry is 1.
2. Each leading 1 is the only nonzero entry in its column (i.e., zeros both above and below).

3.2 Matrix Rank

The rank of a matrix is defined as the maximum number of linearly independent rows or columns in the matrix. It provides a measure of the “dimension” of the matrix.

How to compute rank?

1. Perform Gaussian elimination to row-echelon form;
the number of non-zero rows = rank.
2. For square matrices:
if determinant $\neq 0$, the matrix is full rank i.e., n ; else rank $< n$.
3. Also, rank = order of the largest non-zero minor (square submatrix).

Example:

$$\begin{pmatrix} 1 & 2 & 1 \\ -2 & -3 & 1 \\ 3 & 5 & 0 \end{pmatrix} \xrightarrow{2R_1+R_2 \rightarrow R_2} \begin{pmatrix} 1 & 2 & 1 \\ 0 & 1 & 3 \\ 3 & 5 & 0 \end{pmatrix} \xrightarrow{-3R_1+R_3 \rightarrow R_3} \begin{pmatrix} 1 & 2 & 1 \\ 0 & 1 & 3 \\ 0 & -1 & -3 \end{pmatrix}$$

$$\xrightarrow{R_2+R_3 \rightarrow R_3} \begin{pmatrix} 1 & 2 & 1 \\ 0 & 1 & 3 \\ 0 & 0 & 0 \end{pmatrix}$$

Rank of a given matrix is 2.

```
import numpy as np

# Define the matrix
A = np.array([[1, 2, 3], [2, 4, 6], [1, 0, 1]])

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

Figure 3.1: Matrix Rank in python using NumPy

3.3 Gaussian Elimination

The Gaussian Elimination method is a widely used technique for solving systems of linear equations. A system of linear equation involves multiple equation with unknown variables. The goal of solving such system is to find values of unknowns that satisfy all given equations simultaneously.

Categories of Linear Equation System are given below:

1. Consistent Independent System: Has exactly one solution (Unique solution).
2. Consistent Dependent System: Has infinite solutions.
3. Inconsistent System: Has no solution.

Gaussian elimination is a row reduction algorithm for solving linear systems. It involves a series of operations on the augmented matrix (which includes both coefficients& constants) to simplify it into a row echelon form or reduced row echelon form. This method can also help in determining rank, determinant and inverse of matrices.

Let's for example we have system of linear equations:

$$\begin{aligned}a_1x + b_1y + c_1z &= d_1, \\a_2x + b_2y + c_2z &= d_2, \\a_3x + b_3y + c_3z &= d_3.\end{aligned}$$

Matrix form of this system of linear equation:

$$\left[\begin{array}{ccc|c} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \end{array} \right]$$

Goal: Turn above matrix into row-echelon form like

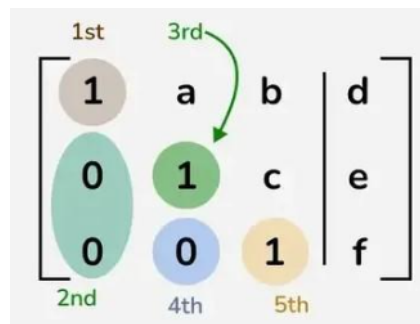
$$\left[\begin{array}{ccc|c} 1 & a & b & d \\ 0 & 1 & c & e \\ 0 & 0 & 1 & f \end{array} \right]$$

Once in this form we can say that $z=f$ and use back substitution to solve for y and x .

Elementary Row Operations:

1. Interchanging Rows: Swap two rows.
2. Multiplying a row by a scalar: Multiply all elements of a row by non-zero number.
3. Adding a scalar multiple of one row to another: Add or subtract a multiple of one row to/from another.

These operations simplify solving system without changing the solution set.



The diagram shows a matrix in row-echelon form with the following structure:

$$\left[\begin{array}{ccc|c} \text{1st} & & & \\ \text{1} & a & b & d \\ & \text{2nd} & & \\ 0 & \text{3rd} & & \\ & & & \\ 0 & 1 & c & e \\ & & & \\ 0 & 0 & 1 & f \\ & \text{4th} & \text{5th} & \end{array} \right]$$

Annotations include: '1st' above the first row, '2nd' below the second row, '3rd' above the third row, '4th' below the fourth row, and '5th' below the fifth row. A green arrow points from the '3rd' label to the '1' in the third row, second column.

Follow these steps:

1. Get a 1 in the first column, first row.
2. Use the 1 to get 0's in the remainder of the first column.
3. Get a 1 in the second column, second row.
4. Use the 1 to get 0's in the remainder of the second column.
5. Get a 1 in the third column, third row.

The resulting matrix is in row echelon form. A matrix is in reduced row-echelon form if all leading coefficients are 1, and the columns containing these leading coefficients are 1, and the columns containing these leading coefficients have zero elsewhere. This final form is unique, regardless of the sequence of row operations used. An example below illustrates this concept.

Example: To solve the system of equation.

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

Solution:- Step 1: Write augmented matrix:

$$\left[\begin{array}{cc|c} 1 & 1 & 3 \\ 3 & -2 & 4 \end{array} \right]$$

Step 2: Perform row operations: The goal is to simplify the augmented matrix to solve for x and y.

$$\left[\begin{array}{cc|c} 1 & 1 & 3 \\ 3 & -2 & 4 \end{array} \right] \xrightarrow{r_2 - 3r_1 \rightarrow r_2} \left[\begin{array}{cc|c} 1 & 1 & 3 \\ 0 & -5 & -5 \end{array} \right] \xrightarrow{r_2 \div (-5)} \left[\begin{array}{cc|c} 1 & 1 & 3 \\ 0 & 1 & 1 \end{array} \right]$$

Step 3: Using Back Substitution we get, solution: x=2 and y=1.

```
import numpy as np

# Example augmented matrix Ab (A|b)
Ab = np.array([[1,1,3],[3,-2,4]])
n=len(Ab)

# Gaussian elimination
for i in range(n):
    for j in range(i+1, n):
        factor = Ab[j][i] / Ab[i][i]
        Ab[j, i:] = Ab[j, i:] - factor * Ab[i, i:]

# Back substitution
x = np.zeros(n)
for i in range(n - 1, -1, -1):
    x[i] = (Ab[i, -1] - np.dot(Ab[i, i+1:n], x[i+1:n])) / Ab[i, i]

print("The solutions are:", x)
```

Figure 3.2: Implementation in python

3.4 Cramer's Rule

Cramer's rule is an explicit method for solving a system of linear equations with unique solutions using determinants.

Example:

$$\text{Solve the system: } \begin{cases} x + y = 3, \\ 2x + 5y = 7. \end{cases}$$

Solution:

$$A = \begin{pmatrix} 1 & 1 \\ 2 & 5 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 3 \\ 7 \end{pmatrix}.$$

$$A_1 = \begin{pmatrix} 3 & 1 \\ 7 & 5 \end{pmatrix}, \quad A_2 = \begin{pmatrix} 1 & 3 \\ 2 & 7 \end{pmatrix}.$$

$$\det(A) = 1 \cdot 5 - 2 \cdot 1 = 5 - 2 = 3,$$

$$\det(A_1) = 3 \cdot 5 - 7 \cdot 1 = 15 - 7 = 8,$$

$$\det(A_2) = 1 \cdot 7 - 2 \cdot 3 = 7 - 6 = 1.$$

$$x = \frac{\det(A_1)}{\det(A)} = \frac{8}{3}, \quad y = \frac{\det(A_2)}{\det(A)} = \frac{1}{3}.$$

Now let's see this method in python:

```
import numpy as np

A = np.array([[1,1], [2,5]])
b = np.array([3,7])
detA = np.linalg.det(A)

# Compute x using Cramer's Rule
A1 = np.array([[b[0], A[0,1]], [b[1], A[1,1]]])
detA1 = np.linalg.det(A1)
x = detA1/detA
print(x)

# Compute y using Cramer's Rule
A2 = np.array([[A[0,0], b[0]], [A[1,0], b[1]]])
detA2 = np.linalg.det(A2)
y = detA2/detA
print(y)
```

3.5 Homogenous & Non-homogenous System

1. **Homogenous System:** A system of linear equation is said to be homogenous if all the constant terms on the right hand side of the equations are zero. Mathematically has the form $Ax=0$,

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = 0 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = 0 \\ \vdots \quad \quad \quad \vdots \quad \quad \quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = 0 \end{cases}$$

Note: Homogenous system always consistent i.e., always have solution.

Types of solutions:

- (a) **Unique Solution:** The system has a unique (trivial) solution only if the coefficient matrix A has ****full column rank****, i.e.

$$\text{rank}(A) = n,$$

where n is the number of variables. Equivalently, if A is square and $\det(A) \neq 0$, the only solution to the homogeneous system $Ax = 0$ is

$$x = 0.$$

- (b) **Infinitely Many Solutions:** If

$$\text{rank}(A) < n,$$

then A has at least one free variable, and the homogeneous system $Ax = 0$ admits nontrivial solutions (i.e. solutions with $x \neq 0$). These generate infinitely many solutions. :contentReference[oaicite:0]index=0

2. **Non-homogenous System:** A system of linear equation is said to be non-homogenous if all the constant terms on the right hand side of the equations need not be zero. Mathematically has the form $Ax=b$.

Note: Non-homogeneous system can be consistent or inconsistent.

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2, \\ \vdots \quad \quad \quad \vdots \quad \quad \quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m. \end{cases}$$

Types of solutions:

- (a) **Unique Solution:** A system has a unique solution precisely when

$$\text{rank}(A) = \text{rank}([A \mid b]) = n,$$

i.e. the coefficient matrix A is full-rank (for an $n \times n$ system).

- (b) **Infinitely Many Solutions:** Solutions occur when

$$\text{rank}(A) = \text{rank}([A \mid b]) < n,$$

meaning there are fewer independent equations than variables—leading to free parameters.

- (c) **No Solution:** The system is inconsistent (no solution) exactly when

$$\text{rank}([A \mid b]) > \text{rank}(A),$$

indicating the augmented matrix has more independent rows than A itself.

Chapter 4

Vector Spaces & Their Applications

4.1 Vector Space

Definition

Let V be a nonempty set whose elements are called vectors, and let F be a field (e.g. R or C) whose elements are called scalars. A vector space (over F) consists of V together with two operations:

$$\begin{aligned}\text{addition: } V \times V &\rightarrow V, & (u, v) &\mapsto u + v, \\ \text{scalar multiplication: } F \times V &\rightarrow V, & (a, v) &\mapsto a v,\end{aligned}$$

such that, for all $u, v, w \in V$ and $a, b \in F$, the following **axioms** hold:

1. **Closure under addition:** $u + v \in V$.
2. **Closure under scalar multiplication:** $a v \in V$.
3. **Commutativity of addition:** $u + v = v + u$.
4. **Associativity of addition:** $(u + v) + w = u + (v + w)$.
5. **Existence of additive identity:** $\exists 0 \in V$ such that $v + 0 = v$.
6. **Existence of additive inverse:** $\forall v, \exists (-v) \in V$ such that $v + (-v) = 0$.
7. **Multiplicative identity:** $1 v = v$, where 1 is the unit in F .
8. **Associativity of scalar multiplication:** $(ab) v = a (b v)$.
9. **Distributivity over vector addition:** $a (u + v) = a u + a v$.
10. **Distributivity over scalar addition:** $(a + b) v = a v + b v$.

Examples of Vector Spaces

Below are some standard examples of vector spaces over a field F (typically R or C):

1. **Real numbers, R :** The set of real numbers is a one-dimensional vector space under standard addition and scalar multiplication.

2. **Euclidean space**, F^n : The set of all n -tuples (x_1, x_2, \dots, x_n) forms a vector space with component-wise operations. For instance, R^3 contains vectors (x, y, z) with real coordinates.
3. **Polynomials**, $P_n(F)$: The set of all polynomials of degree $\leq n$, such as $ax^2 + bx + c$, is a vector space. Addition and scalar multiplication work coefficient-wise.
4. **Matrices**, $M_{m \times n}(F)$: All $m \times n$ matrices with entries in F , with matrix addition and scalar multiplication defined entry-wise, form a vector space.

Subspaces

Let V be a vector space over a field F . A non-empty subset $W \subseteq V$ is called a subspace of V if it is itself a vector space using the operations inherited from V .

Subspace Test

To verify W is a subspace, it suffices to check:

1. $\mathbf{0} \in W$ (contains the zero vector),
2. $\forall u, v \in W : u + v \in W$ (closed under addition),
3. $\forall a \in F, v \in W : av \in W$ (closed under scalar multiplication).

If these conditions hold, then W automatically satisfies all vector-space axioms since they are inherited from V .

Examples of Subspaces:

1. $W = \{(x, y, z) \in R^3 \mid z = 0\}$ is a subspace of R^3 .
2. $W = \{(x, y) \in R^2 \mid x = y\}$ is a subspace of R^2 .
3. $W = C(R)$, the set of all continuous real-valued functions on R , is a subspace of the vector space of all real functions.
4. $W = \{(x, y, z) \in R^3 \mid x + y + z = 0\}$ is a subspace defined by a homogeneous linear condition.

4.2 Some Basic Concepts

Linear Combination

Given vectors $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k \in V$ over a field F , a linear combination is any vector of the form

$$c_1\mathbf{x}_1 + c_2\mathbf{x}_2 + \dots + c_k\mathbf{x}_k,$$

where $c_1, \dots, c_k \in F$.

Span and Spanning Sets

The span of the set $\{\mathbf{x}_1, \dots, \mathbf{x}_k\}$ is the set of all their linear combinations:

$$\text{span}\{\mathbf{x}_1, \dots, \mathbf{x}_k\} = \left\{ \sum_{i=1}^k c_i \mathbf{x}_i : c_i \in F \right\}.$$

A set is called a spanning set if its span equals the entire space V .

Linear Independence

The set $\{\mathbf{x}_1, \dots, \mathbf{x}_k\}$ is linearly independent if the only solution to

$$c_1 \mathbf{x}_1 + \dots + c_k \mathbf{x}_k = \mathbf{0}$$

is $c_1 = \dots = c_k = 0$. Equivalently, no nontrivial combination gives zero.

Basis & Dimension

A basis of V is a collection that is both

1. linearly independent, and
2. spanning.

The dimension of V , denoted $\dim V$, is the number of vectors in any basis of V .

Linear Transformation

A function $T : V \rightarrow W$ between vector spaces is a linear transformation if, for all $\mathbf{u}, \mathbf{v} \in V$ and all scalars $c \in F$,

$$\begin{aligned} T(\mathbf{u} + \mathbf{v}) &= T(\mathbf{u}) + T(\mathbf{v}), \\ T(c\mathbf{u}) &= cT(\mathbf{u}). \end{aligned}$$

This ensures T preserves the vector-space structure and finite linear combinations:

$$T\left(\sum_i c_i \mathbf{x}_i\right) = \sum_i c_i T(\mathbf{x}_i).$$

Kernel & Image

$$\ker(T) = \{\mathbf{v} \in V \mid T(\mathbf{v}) = \mathbf{0}\}, \quad \text{im}(T) = \{T(\mathbf{v}) \mid \mathbf{v} \in V\}.$$

These are subspaces of V and W respectively.

Column Space & Null Space

For a matrix A , its column space is $\text{col}(A) = \text{im}(A)$, and its null space is $\text{null}(A) = \ker(A)$.

Rank–Nullity Theorem

For a linear map $T : V \rightarrow W$ with $\dim(V)$ finite,

$$\text{rank}(T) + \text{nullity}(T) = \dim(V),$$

where $\text{rank} = \dim(\text{im}(T))$ and $\text{nullity} = \dim(\ker(T))$.

```
import numpy as np
A = np.array([[1, 2, 3], [4, 5, 6]])
rank = np.linalg.matrix_rank(A)
nullity = A.shape[1] - rank
print("Rank:", rank)
print("Nullity:", nullity)
```

Figure 4.1: finding rank & nullity using RNT

Tips:

- *Span* = “smallest subspace containing the set.”
- *Basis* = “a minimal spanning set.”
- *Rank-Nullity* = partitioning V into kernel + image (dimensions sum up).

Example:

Define a Linear Transformation $L : R^2 \rightarrow R^3$,

$$L(x, y) = (x + y, x + 2y, y).$$

Solution:**Column Space / Image**

L as a matrix,

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 0 & 1 \end{pmatrix}, \quad L(x, y) = A \begin{pmatrix} x \\ y \end{pmatrix} = x \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} + y \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}.$$

Thus,

$$\text{im}(L) = \text{span} \left\{ \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \right\},$$

a 2-dimensional subspace of R^3 .

Rank and Basis of Image

The two column vectors are linearly independent, so they form a basis of $\text{im}(L)$, and hence $\text{rank}(L) = 2$.

Kernel / Null Space

Solve $L(x, y) = (0, 0, 0)$:

$$x + y = 0, \quad x + 2y = 0, \quad y = 0 \Rightarrow x = y = 0.$$

So $\ker(L) = \{(0, 0)\}$, meaning L is injective and $\text{nullity}(L) = 0$.

Rank–Nullity Theorem

Since $\dim(R^2) = 2$,

$$\text{rank}(L) + \text{nullity}(L) = 2 + 0 = 2,$$

confirming that $\text{rank}(L) + \text{nullity}(L) = \dim(\text{domain})$.

Summary:

- Linear combination: $(x, y) \mapsto x\mathbf{v}_1 + y\mathbf{v}_2$.
- Span: Image is span of $\{\mathbf{v}_1, \mathbf{v}_2\}$.
- Linear independence: \mathbf{v}_1 and \mathbf{v}_2 independent.
- Basis: $\{\mathbf{v}_1, \mathbf{v}_2\}$ is a basis of $\text{im}(L)$.
- Dimension: $\dim(\text{im}(L)) = 2$.
- Kernel: Only the zero vector maps to zero.
- Rank–Nullity: $2 + 0 = 2$.

Now let's see this example in python using sympy and numpy.

```
import numpy as np
from sympy import Matrix # sympy for exact nullspace

# Define the matrix for L: R^2 → R^3
A = np.array([[1, 1], [1, 2], [0, 1]], dtype=float)

# Compute the rank = dimension of column space
rank = np.linalg.matrix_rank(A)
print(f"Rank (columns independent): {rank}") # should be 2

# Column space basis (via pivot columns)
# Use SVD to extract independent columns
U, S, Vt = np.linalg.svd(A, full_matrices=False)
basis_image = A[:, :rank]
print("Basis for im(L):")
print(basis_image)

# Null space using sympy for exact solution
ns = Matrix(A).nullspace()
print("Null space basis:")
for v in ns:
    print(v)
nullity = len(ns)
print(f"Nullity = {nullity}")

# Verify Rank-Nullity theorem
dim_domain = A.shape[1] # number of columns = 2
print(f"Rank + Nullity = {rank} + {nullity} = {rank + nullity}")
print(f"Dimension of domain: {dim_domain}")
assert rank + nullity == dim_domain
```

Figure 4.2: above example

4.3 Applications of Vector Spaces in Data Science

Vector spaces form the mathematical backbone of many data science techniques. Key applications include:

1. **Dimensionality Reduction (e.g. PCA)** Principal Component Analysis finds an orthonormal basis capturing maximum variance in data, enabling projection into a lower-dimensional subspace for visualization, noise reduction, and preprocessing.
2. **Feature Representation and Similarity** Data—such as images, text, or user transactions—is represented as high-dimensional vectors. Vector operations like dot-product, Euclidean distance, or cosine similarity underpin clustering, nearest-neighbor searches, recommender systems, and similarity-based ranking.
3. **Natural Language Processing (Word Embeddings & LSA)** Words and documents are embedded in high-dimensional vector spaces (e.g., Word2Vec, GloVe). Semantic relationships (e.g., $king - man + woman \approx queen$) emerge naturally. Latent Semantic Analysis employs matrix decomposition (SVD) on document-term matrices to uncover latent topics.
4. **Support Vector Machines (SVMs)** Classification models rely on vector spaces to represent data points and decision boundaries. Kernels implicitly map data to higher-dimensional spaces to achieve linear separability.
5. **Vector Search Recommender Systems** Systems using vector databases (e.g., FAISS, Pinecone) store high-dimensional embeddings to efficiently perform similarity searches—powering semantic search, chatbot retrieval, and personalized recommendations.

Chapter 5

Eigenvalues & Diagonalization

5.1 Eigenvalues and Eigenvectors

Let A be an $n \times n$ square matrix over a field F .

Definition

A nonzero vector $\mathbf{v} \in F^n$ is called an eigenvector of A if there exists a scalar $\lambda \in F$ such that

$$A\mathbf{v} = \lambda \mathbf{v}.$$

In this case, λ is called the corresponding eigenvalue. Eigenvectors point along directions that are scaled (possibly reversed) by A , without being rotated into a different direction.

Characteristic Equation

Since $\mathbf{v} \neq \mathbf{0}$, the matrix $(A - \lambda I)$ must be singular:

$$\det(A - \lambda I) = 0.$$

This characteristic polynomial in λ yields the eigenvalues as its roots.

Finding Eigenvectors

For each eigenvalue λ , solve

$$(A - \lambda I) \mathbf{v} = \mathbf{0}$$

for nonzero \mathbf{v} ; this finds the corresponding eigenvectors (forming the λ -eigenspace).

Key Properties

- Eigenvectors for distinct eigenvalues are linearly independent. Hence an $n \times n$ matrix can have up to n independent eigenvectors.
- The set of all eigenvalues $\{\lambda_1, \dots, \lambda_n\}$ is called the spectrum of A ; its elements are the roots of the characteristic polynomial.

- If A is diagonalizable, it can be written as

$$A = VDV^{-1},$$

where D is a diagonal matrix containing the eigenvalues, and columns of V are the corresponding eigenvectors.

Example:

Let

$$A = \begin{pmatrix} 1 & 2 \\ 5 & 4 \end{pmatrix}.$$

We seek eigenvalues λ and eigenvectors $\mathbf{v} = \begin{pmatrix} a \\ b \end{pmatrix}$, satisfying

$$A\mathbf{v} = \lambda\mathbf{v}.$$

Characteristic Equation

$$\det(A - \lambda I) = \begin{vmatrix} 1 - \lambda & 2 \\ 5 & 4 - \lambda \end{vmatrix} = (1 - \lambda)(4 - \lambda) - 10 = \lambda^2 - 5\lambda - 6 = 0.$$

Solving gives

$$(\lambda - 6)(\lambda + 1) = 0 \implies \lambda_1 = 6, \lambda_2 = -1.$$

Eigenvalues are: 6 & -1. Now finding eigenvectors corresponding to them.

Eigenvector for $\lambda = 6$

Solve $(A - 6I)\mathbf{v} = \mathbf{0}$:

$$\begin{pmatrix} -5 & 2 \\ 5 & -2 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \implies -5a + 2b = 0 \implies b = \frac{5}{2}a.$$

Choose $a = 2$ to avoid fractions, giving $b = 5$. Thus, an eigenvector is

$$\mathbf{v}_1 = \begin{pmatrix} 2 \\ 5 \end{pmatrix}.$$

Eigenvector for $\lambda = -1$

Solve $(A + I)\mathbf{v} = \mathbf{0}$:

$$\begin{pmatrix} 2 & 2 \\ 5 & 5 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \implies 2a + 2b = 0 \implies a = -b.$$

Choose $a = 1$, hence $b = -1$. An eigenvector is

$$\mathbf{v}_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}.$$

```
import numpy as np

m = np.array([[1, 2],[5, 4]])
# finding eigenvalues and eigenvectors
w, v = np.linalg.eig(m)
print("Eigen values:\n",w)
print("Eigenvectors:\n",v)
```

Figure 5.1: Eigenvalues & eigenvectors in python

Final Results

Eigenvalues: $\lambda_1 = 6, \quad \lambda_2 = -1,$ Eigenvectors: $\mathbf{v}_1 = \begin{pmatrix} 2 \\ 5 \end{pmatrix}, \quad \mathbf{v}_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}.$

5.2 Diagonalization

What Does “Diagonalizable” Mean?

A square matrix A is diagonalizable if it is similar to a diagonal matrix, that is, if there exists an invertible matrix P and a diagonal matrix D such that

$$A = P D P^{-1},$$

or equivalently,

$$P^{-1}AP = D.$$

Here, the columns of P are eigenvectors of A , and the diagonal entries of D are the corresponding eigenvalues.

When Is a Matrix Diagonalizable?

- A necessary and sufficient condition: A is diagonalizable if and only if it has n linearly independent eigenvectors.
- A sufficient (but not necessary) condition: if all eigenvalues of A are distinct, then A is guaranteed to be diagonalizable.
- Let λ be an eigenvalue of an $n \times n$ matrix A . Its algebraic multiplicity $\mu_A(\lambda)$ is its multiplicity as a root of $\det(A - \lambda I) = 0$, while its geometric multiplicity $\gamma_A(\lambda)$ is $\dim \ker(A - \lambda I)$, the dimension of its eigenspace. We always have

$$1 \leq \gamma_A(\lambda) \leq \mu_A(\lambda).$$

Moreover, A is diagonalizable if and only if for every eigenvalue λ ,

$$\gamma_A(\lambda) = \mu_A(\lambda),$$

which guarantees a full set of n independent eigenvectors.

How to Diagonalize – Step by Step Given an $n \times n$ matrix A :

1. Find eigenvalues $\lambda_1, \dots, \lambda_n$ by solving $\det(A - \lambda I) = 0$.
2. For each eigenvalue λ_i , solve $(A - \lambda_i I)\mathbf{v} = 0$ to obtain eigenvectors \mathbf{v}_i .
3. If you obtain n linearly independent eigenvectors $\{\mathbf{v}_i\}$, form

$$P = (\mathbf{v}_1 \ \cdots \ \mathbf{v}_n), \quad D = \text{diag}(\lambda_1, \dots, \lambda_n).$$

4. Then verify $A = P D P^{-1}$ (equivalently $P^{-1}AP = D$).

Why Diagonalization Matters

Diagonalizing A greatly simplifies computations, for example:

$$A^k = (P D P^{-1})^k = P D^k P^{-1}$$

where D^k is computed by raising each diagonal entry to the power k .

5.3 Applications of Eigen-Decomposition

Eigenvalues and eigenvectors play a central role in multiple data-science tasks:

1. **Principal Component Analysis (PCA)** PCA computes the eigendecomposition of the covariance matrix to find orthogonal directions (principal components) capturing maximal variance. Data is projected onto the top k eigenvectors—reducing dimensionality while preserving most information.
2. **Dimensionality Reduction in Machine Learning** Beyond PCA, eigenvector-based decompositions (e.g., SVD, spectral methods) are widely used for feature extraction, denoising, and pattern discovery in high-dimensional datasets.

How?

Dimensionality Reduction is a technique used to transform data from higher dimension to lower dimension i.e., it helps to reduce the number of features while retaining key information. Technique like principal component analysis (PCA), singular value decomposition (SVD), linear discriminant analysis (LDA) convert data into lower dimensional space while preserving important details.

This technique divided into two categories:

- (a) **Feature Selection:** This involves selecting a subset of the original features that are most relevant. It removes irrelevant features and improving model efficiency.
 - (b) **Feature Extraction:** This involves creating new features by combining or transforming the original features.
3. **Spectral Clustering & Graph Analysis** Techniques such as spectral clustering use the eigenvectors of the graph Laplacian (or similarity matrix) to embed data into a lower-dimensional space. Clustering in this space helps identify communities or clusters effectively.

4. **Image Compression & Reconstruction** Via Singular Value Decomposition (SVD)—a close relative of eigendecomposition—images are approximated using only the largest singular values/vectors, achieving lossy compression with minimal quality loss.
5. **Stability and Dynamic Mode Analysis** In fields like systems biology and neural network analysis, eigenvalues of Hessians or Jacobians detect dynamic behaviors—such as convergence rates, resonant modes, and stability properties.

Chapter 6

Orthogonal & Projection

6.1 Orthogonality

Orthogonal Vectors & Matrices

- Two vectors $\mathbf{u}, \mathbf{v} \in R^n$ are *orthogonal* if

$$\langle \mathbf{u}, \mathbf{v} \rangle = \mathbf{u}^T \mathbf{v} = 0.$$

- A matrix Q is *orthogonal* if

$$Q^T Q = I \iff Q^{-1} = Q^T,$$

meaning its columns form an orthonormal set. Such matrices preserve both lengths and angles.

Orthogonal Sets

A set of nonzero vectors $\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_p\} \subset R^n$ is called *orthogonal* if

$$\mathbf{u}_i \cdot \mathbf{u}_j = 0, \quad \forall i \neq j.$$

If, additionally, each \mathbf{u}_i is of unit length, the set is called *orthonormal*. Orthogonal sets are automatically linearly independent.

6.2 Projection

Orthogonal Expansion & Projection

For an orthogonal basis $S = \{\mathbf{u}_1, \dots, \mathbf{u}_p\}$ of a subspace $W \subseteq R^n$, any vector $\mathbf{y} \in W$ can be uniquely expressed as:

$$\mathbf{y} = \sum_{j=1}^p c_j \mathbf{u}_j, \quad c_j = \frac{\mathbf{y} \cdot \mathbf{u}_j}{\mathbf{u}_j \cdot \mathbf{u}_j}.$$

The *orthogonal projection* of any $\mathbf{x} \in R^n$ onto W is

$$\hat{\mathbf{x}} = \sum_{j=1}^p \frac{\mathbf{x} \cdot \mathbf{u}_j}{\mathbf{u}_j \cdot \mathbf{u}_j} \mathbf{u}_j,$$

giving the decomposition $\mathbf{x} = \hat{\mathbf{x}} + \mathbf{z}$, where $\mathbf{z} \perp W$.

Projection onto a Line

When W is the line spanned by \mathbf{u} , the projection simplifies to:

$$\text{proj}_W(\mathbf{x}) = \frac{\mathbf{x} \cdot \mathbf{u}}{\mathbf{u} \cdot \mathbf{u}} \mathbf{u},$$

and the residual $\mathbf{x} - \text{proj}_W(\mathbf{x})$ is orthogonal to \mathbf{u} .

Projection Matrix

If $\{\mathbf{u}_1, \dots, \mathbf{u}_p\}$ is an orthogonal basis of W , then the projection operator $P : R^n \rightarrow W$ has the matrix form:

$$P = \sum_{j=1}^p \frac{1}{\|\mathbf{u}_j\|^2} \mathbf{u}_j \mathbf{u}_j^T.$$

This matrix satisfies:

$$P^T = P, \quad P^2 = P, \quad \text{rank}(P) = p,$$

and it orthogonally projects any \mathbf{x} onto W .

6.3 Gram–Schmidt process

Given a set of linearly independent vectors $\{\mathbf{v}_1, \dots, \mathbf{v}_k\} \subset R^n$, the Gram–Schmidt process constructs an *orthonormal* basis $\{\mathbf{e}_1, \dots, \mathbf{e}_k\}$ for the same subspace.

Example

Given $\mathbf{v}_1 = (3, 1)^T$, $\mathbf{v}_2 = (2, 2)^T$:

$$\mathbf{u}_1 = (3, 1)^T, \quad \mathbf{u}_2 = \mathbf{v}_2 - \frac{\langle \mathbf{v}_2, \mathbf{u}_1 \rangle}{\|\mathbf{u}_1\|^2} \mathbf{u}_1 = \begin{pmatrix} 2 \\ 2 \end{pmatrix} - \frac{8}{10} \begin{pmatrix} 3 \\ 1 \end{pmatrix} = \begin{pmatrix} -\frac{2}{5} \\ \frac{6}{5} \end{pmatrix}.$$

After normalization, $\mathbf{e}_1 = \frac{1}{\sqrt{10}}(3, 1)^T$, $\mathbf{e}_2 = \frac{1}{\sqrt{10}}(-1, 3)^T$, which form an orthonormal basis of $\text{span}\{\mathbf{v}_1, \mathbf{v}_2\}$.

6.4 Applications of Orthogonality & Gram–Schmidt

Orthogonality and related techniques such as Gram–Schmidt play crucial roles across various data science tasks:

1. **Feature Decorrelation & Stability** Orthogonal feature sets are uncorrelated, which improves interpretability and numeric conditioning in regression models. By orthogonalizing features (e.g., via Gram–Schmidt), one can reduce variance inflation and stabilize learning :contentReference[oaicite:1]index=1.
2. **Dimensionality Reduction** Gram–Schmidt underlies QR-based PCA: we can obtain an orthonormal basis approximating principal components. This simplifies high-dimensional data to a lower-dimensional subspace while retaining maximal variance :contentReference[oaicite:2]index=2.


```

import numpy as np

def gram_schmidt(A):
    """
    Orthogonalize the columns of A and normalize them.
    Returns a matrix Q whose columns are orthonormal.
    """
    A = A.astype(np.float64)
    (n, m) = A.shape
    Q = np.zeros((n, m))

    for j in range(m):
        v = A[:, j].copy()
        for i in range(j):
            # Subtract projection of v onto previously computed Q[:, i]
            v -= np.dot(Q[:, i], A[:, j]) * Q[:, i]
        norm = np.linalg.norm(v)
        if norm < 1e-14:
            # Dependent vector → skip or leave zero
            Q[:, j] = np.zeros(n)
        else:
            Q[:, j] = v / norm
    return Q

# Example usage:
A = np.column_stack(([3, 2], [1, 2]))
Q = gram_schmidt(A)
print("Orthonormal Q:")
print(Q)
print("Check orthogonality (Q^T Q):")
print(np.round(Q.T @ Q, 8))

```

Figure 6.1: Gram-Schmidt process in python

3. **QR Decomposition for Least Squares Regression** In linear regression or overdetermined systems, orthonormalizing columns of the design matrix A via Gram-Schmidt yields $A = QR$. Solving for x becomes stable and efficient via $x = R^{-1}Q^T b$:contentReference[oaicite:3]index=3.
4. **Uncorrelated Signal Extraction** In time-series and signal-processing, orthogonal bases derived from Gram-Schmidt (or SVD/PCA) enable noise reduction, source separation, and efficient compact representations. :contentReference[oaicite:4]index=4
5. **Feature Selection and Representation Learning** Orthogonalization can be used within feature-selection algorithms and unsupervised extraction methods to ensure that learned features (e.g., via Gram-Schmidt-inspired methods) are unique and informative :contentReference[oaicite:5]index=5.

Chapter 7

Matrix Decomposition

7.1 LU Decomposition:

A square matrix A can be factored into a lower-triangular L (with unit diagonal) and an upper-triangular U :

$$A = LU.$$

This is the matrix form of Gaussian elimination and is commonly used to solve $A\mathbf{x} = \mathbf{b}$ efficiently via: 1. $L\mathbf{z} = \mathbf{b}$ (forward substitution), 2. $U\mathbf{x} = \mathbf{z}$ (backward substitution). These steps avoid recomputing elimination for multiple \mathbf{b} 's, offering significant performance gains.

Example:

Given

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 4 & 3 & -1 \\ 3 & 5 & 3 \end{pmatrix}, \quad X = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \quad C = \begin{pmatrix} 1 \\ 6 \\ 4 \end{pmatrix}.$$

1. Compute L and U via Gaussian elimination: Perform elimination on A without row swaps to obtain:

$$U = \begin{pmatrix} 1 & 1 & 1 \\ 0 & -1 & -5 \\ 0 & 0 & -10 \end{pmatrix}, \quad L = \begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 3 & -2 & 1 \end{pmatrix},$$

where multipliers are $l_{21} = 4$, $l_{31} = 3$, and $l_{32} = -2$. Such an LU decomposition exists since no row swaps were necessary.

2. Forward substitution: $LZ = C$ Let $Z = (z_1, z_2, z_3)^T$. Solve

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 3 & -2 & 1 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 6 \\ 4 \end{pmatrix}$$

gives:

$$z_1 = 1, \quad 4z_1 + z_2 = 6 \implies z_2 = 2, \quad 3z_1 - 2z_2 + z_3 = 4 \implies z_3 = 5.$$

Hence, $Z = (1, 2, 5)^T$.

3. Backward substitution: $U X = Z$ Solve

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & -1 & -5 \\ 0 & 0 & -10 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 5 \end{pmatrix}$$

to get:

$$\begin{aligned} -10x_3 &= 5 &\Rightarrow x_3 &= -0.5, \\ -1x_2 - 5x_3 &= 2 &\Rightarrow x_2 &= 0.5, \\ x_1 + x_2 + x_3 &= 1 &\Rightarrow x_1 &= 1. \end{aligned}$$

4. Final solution

$$X = \begin{pmatrix} 1 \\ 0.5 \\ -0.5 \end{pmatrix}.$$

Summary

$$A = LU, \quad LX = C \xrightarrow{\text{forward sub}} Z \xrightarrow{\text{back sub}} X.$$

This is a standard approach using LU decomposition to efficiently solve a

7.2 QR Decomposition

QR decomposition decomposes a matrix A into an orthogonal matrix Q and an upper triangular matrix R . Mathematically, $A = QR$.

QR Decomposition via Gram–Schmidt Method

Given a full-column-rank matrix $A = [a_1 \ a_2 \ a_3]$, the goal is to find an orthonormal matrix Q and an upper-triangular matrix R such that:

$$A = QR \quad \text{with} \quad Q^T Q = I,$$

using the Gram–Schmidt process.

1. Compute q_1 :

$$q_1 = \frac{a_1}{\|a_1\|}, \quad \text{set } r_{11} = \|a_1\|.$$

2. Orthogonalize a_2 :

$$r_{12} = q_1^T a_2, \quad v_2 = a_2 - r_{12} q_1, \quad q_2 = \frac{v_2}{\|v_2\|}, \quad r_{22} = \|v_2\|.$$

3. Orthogonalize a_3 :

$$\begin{aligned} r_{13} &= q_1^T a_3, & r_{23} &= q_2^T a_3, \\ v_3 &= a_3 - r_{13} q_1 - r_{23} q_2, & q_3 &= \frac{v_3}{\|v_3\|}, & r_{33} &= \|v_3\|. \end{aligned}$$

4. Form Q and R :

$$Q = [q_1 \ q_2 \ q_3], \quad R = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ 0 & r_{22} & r_{23} \\ 0 & 0 & r_{33} \end{pmatrix}.$$

Example:

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

Solution: **Step 1: Normalize a_1**

$$a_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad q_1 = \frac{a_1}{\|a_1\|} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

Step 2: Orthogonalize a_2

$$\begin{aligned} r_{12} &= q_1^T a_2 = 0, \\ v_2 &= a_2 - r_{12}q_1 = \begin{bmatrix} 0 \\ 2 \\ -1 \end{bmatrix}, \\ q_2 &= \frac{v_2}{\|v_2\|} = \frac{1}{\sqrt{5}} \begin{bmatrix} 0 \\ 2 \\ -1 \end{bmatrix}. \end{aligned}$$

Step 3: Orthogonalize a_3

$$\begin{aligned} r_{13} &= q_1^T a_3 = 2, \\ r_{23} &= q_2^T a_3 = \frac{1}{\sqrt{5}} \begin{bmatrix} 0 & 2 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix} = \frac{-1}{\sqrt{5}}, \\ v_3 &= a_3 - r_{13}q_1 - r_{23}q_2 = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix} - 2 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} - \left(-\frac{1}{\sqrt{5}}\right) \frac{1}{\sqrt{5}} \begin{bmatrix} 0 \\ 2 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \frac{-2}{\sqrt{5}} \end{bmatrix}, \\ q_3 &= \frac{v_3}{\|v_3\|} = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}. \end{aligned}$$

Step 4: Construct Q and R

$$Q = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{2}{\sqrt{5}} & \frac{1}{\sqrt{2}} \\ 0 & \frac{-1}{\sqrt{5}} & \frac{-1}{\sqrt{2}} \end{bmatrix}, \quad R = \begin{bmatrix} 1 & 0 & 2 \\ 0 & \sqrt{5} & -\frac{1}{\sqrt{5}} \\ 0 & 0 & \frac{2}{\sqrt{5}} \end{bmatrix}.$$

We can verify that $A = QR$.

```
import numpy as np
arr = np.array([[1,0,2], [0,2,0],[0,-1,1]])

# Find the QR factor of array
q, r = np.linalg.qr(arr)
print('\nQ:\n', q)
print('\nR:\n', r)
print(np.allclose(arr, np.dot(q, r))) # to check result is correct or not
```

Figure 7.1: QR - Decomposition in python

7.3 Cholesky Decomposition

It is a decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose. Mathematically, $A = LL^T$

Example:

Given

$$A = \begin{pmatrix} 10 & 4 & 5 \\ 4 & 6 & 7 \\ 5 & 7 & 21 \end{pmatrix},$$

we seek a lower-triangular matrix L with positive diagonal entries such that $A = LL^T$.

Step 1: Compute L_{11}

$$L_{11} = \sqrt{A_{11}} = \sqrt{10} \approx 3.1623.$$

Step 2: Compute off-diagonal entries in column 1

$$L_{21} = \frac{A_{21}}{L_{11}} = \frac{4}{3.1623} \approx 1.2649, \quad L_{31} = \frac{A_{31}}{L_{11}} = \frac{5}{3.1623} \approx 1.5811.$$

Step 3: Compute L_{22}

$$L_{22} = \sqrt{A_{22} - L_{21}^2} = \sqrt{6 - (1.2649)^2} \approx \sqrt{4.0000} \approx 2.0000.$$

Step 4: Compute L_{32}

$$L_{32} = \frac{A_{32} - L_{31}L_{21}}{L_{22}} = \frac{7 - (1.5811)(1.2649)}{2.0000} \approx 2.0000.$$

Step 5: Compute L_{33}

$$L_{33} = \sqrt{A_{33} - L_{31}^2 - L_{32}^2} = \sqrt{21 - (1.5811)^2 - (2.0000)^2} \approx \sqrt{21 - 2.5000 - 4.0000} \approx 3.0000.$$

Resulting matrix L :

$$L \approx \begin{pmatrix} 3.1623 & 0 & 0 \\ 1.2649 & 2.0000 & 0 \\ 1.5811 & 2.0000 & 3.0000 \end{pmatrix}.$$

Verification

$$LL^T \approx \begin{pmatrix} 10 & 4 & 5 \\ 4 & 6 & 7 \\ 5 & 7 & 21 \end{pmatrix} = A,$$

confirming a correct Cholesky decomposition. (All values rounded to 4 decimal places for clarity.)

```
import numpy as np
A = np.array([[10,4,5], [4,6,7], [5,7,21]])
L = np.linalg.cholesky(A)
print(L)
```

Figure 7.2: Cholesky - Decomposition in python

7.4 Singular Value Decomposition (SVD)

For any real $m \times n$ matrix A , the singular value decomposition is given by

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

where

- U is an $m \times m$ orthogonal matrix (columns are left singular vectors),
- V is an $n \times n$ orthogonal matrix (columns are right singular vectors),
- Σ is an $m \times n$ diagonal matrix with non-negative entries $\sigma_1 \geq \sigma_2 \geq \dots$, called the singular values of A .

This factorization always exists (even for non-square A), and the singular values are specifically the square roots of the eigenvalues of $A^T A$ (or AA^T).

Example: SVD of $A = \begin{pmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{pmatrix}$

We aim to find $A = U \Sigma V^T$ in five structured steps.

Step 1: Compute AA^T

$$AA^T = \begin{pmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{pmatrix} \begin{pmatrix} 3 & 2 \\ 2 & 3 \\ 2 & -2 \end{pmatrix} = \begin{pmatrix} 17 & 8 \\ 8 & 17 \end{pmatrix}.$$

Step 2: Eigenvalues \rightarrow Singular values The characteristic equation is $\lambda^2 - 34\lambda + 225 = 0$, giving $\lambda_1 = 25$, $\lambda_2 = 9$. Thus the singular values are

$$\sigma_1 = 5, \quad \sigma_2 = 3.$$

Step 3: Right singular vectors v_i From eigenvectors of $A^T A$ (or AA^T):

$$v_1 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \quad v_2 = \frac{1}{\sqrt{18}} \begin{pmatrix} 1 \\ -1 \\ 4 \end{pmatrix}, \quad v_3 = \begin{pmatrix} \frac{2}{3} \\ -\frac{2}{3} \\ -\frac{1}{3} \end{pmatrix},$$

resulting in $V = [v_1 \ v_2 \ v_3]$.

Step 4: Left singular vectors u_i Compute each as $u_i = \frac{1}{\sigma_i} A v_i$, yielding:

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

Step 5: Assemble U , Σ & V

$$\Sigma = \begin{pmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix}, \quad A = U \Sigma V^T.$$

Verification completes the SVD decomposition.

```
import numpy as np
A = np.array([[3,2,2], [2,3,-2]])
U, Sigma, Vt = np.linalg.svd(A)
print("U:\n", U)
print("Sigma:\n", np.diag(Sigma))
print("V^T:\n", Vt)
```

Figure 7.3: SVD in python

Chapter 8

Conclusion

This document has methodically guided the reader through the core concepts and computational techniques of linear algebra, building from foundational ideas to powerful matrix decompositions. In Chapter 2, we established the basic algebraic structures—scalars, vectors, and matrices—and introduced essential operations such as addition, scalar multiplication, dot products, and matrix multiplication. Chapter 3 focused on solving systems of linear equations: exploring Gaussian elimination, row echelon and reduced row echelon forms, matrix rank, Cramer’s Rule, and distinguishing between homogeneous and non-homogeneous systems—thereby laying the groundwork for understanding when and how solutions exist and whether they are unique. Chapter 4 formalized the concept of vector spaces, offering axiomatic definitions and illuminating subspaces through concrete examples; it underscored how span, basis, and dimension underpin many data science and machine learning applications. Chapter 5 harnessed the power of orthogonality, detailing the Gram–Schmidt process and QR decomposition to derive stable, efficient orthonormal bases—crucial for numerical computations and data transformations. Chapter 6 delved into eigenvalues, eigenvectors, and diagonalization, articulating both the mathematical conditions for their existence and their practical applications in areas like principal component analysis, dynamical systems, and graph-based learning. Interwoven throughout were advanced matrix factorizations—including LU, Cholesky, and singular value decomposition (SVD)—demonstrating their theoretical elegance and utility in solving systems, optimizing performance, and compressing high-dimensional data.

Collectively, these chapters equip the reader with both the mathematical foundations and algorithmic techniques that drive modern computational and data-driven methodologies. By blending rigorous theory with clear examples and computational algorithms, this document serves as a sturdy foundation for further exploration in scientific computation, data analysis, and engineering disciplines.

Bibliography

- [1] Hamm, K. (2023). *Advanced Linear Algebra For Data Science* [Lecture Notes, University of Arizona].
- [2] Garcia Vidal, J., Barcelo Ordinas, J.M., & Ferrer Cid, P. (2023). *Review of Linear Algebra and Applications to Data Science*. SANS-MIRI Technical Report.