CPE 486/586: Machine Learning for Engineers

03 Linear Algebra

Fall 2025

Rahul Bhadani

Outline

What is Linear Algebra?

- Linear algebra is the study of **linear equations** and their **transformations** using **vectors** and **matrices**.
- It is the mathematical language of the 21st century: essential for data science, engineering, computer graphics, and physics.
- **Key Objects of Study:**
  - **Vectors:** Elements in a vector space (e.g., arrows in $\mathbb{R}^n$).
  - **Matrices:** Tables of numbers used to represent linear maps and systems of equations.

Some Basic Notions

- A set is a collection of objects.
- The objects themselves are elements: $x \in A$

  We then have lots of set operations:

  - Union $\cup$: $A \cup B = \{x \mid x \in A$ or $x \in B\}$
  - Intersection $\cap$: $A \cap B = \{x \mid x \in A$ and $x \in B\}$
  - Product $\times$: $A \times B = \{(a, b) \mid a \in A, b \in B\}$
  - Difference $-$: $A - B = \{x \mid x \in A$ and $x \notin B\}$
  - Subset $\subseteq$: $A \subseteq X \Leftrightarrow x \in A \Rightarrow x \in X$
  - Complement ^{c}: $A \subseteq X$, $A^c = \{x \in X \mid x \notin A\}$
  - Cardinality $|\cdot|$: $|A| =$ number of elements in A

- Functions are rules assigning values in one set to elements of another: $f : A \to B$
  (So subspaces Γ of $A \times B$ s.t. for all $a \in A$, there is a unique $b \in B$ s.t. $(a, b) \in \Gamma$)

  We'll need a few more notions later.

Fields ($\mathbb{F}$)

A **Field** is a set $\mathbb{F}$ together with two operations: Addition (+) $\mathbb{F} \times \mathbb{F} \to \mathbb{F}$, and Multiplication $(\cdot) : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ satisfying

a1) $(a + b) + c = a + (b + c)$

a2) $\exists 0 \in \mathbb{F}$ s.t. $0 + a = a + 0 = a \; \forall a$

a3) $\forall a$, $\exists (-a)$ s.t $a + (-a) = (-a) + a = 0$

a4) $a + b = b + a$

a5) $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$, $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$

a6) $a \cdot (b \cdot c) = (a \cdot b) \cdot c$

a7) $\exists 1$ s.t $a \cdot 1 = 1 \cdot a = a$

a8) $\forall a \neq 0$, $\exists a^{-1}$ s.t. $a \cdot a^{-1} = a^{-1} \cdot a = 1$

a9) $a \cdot b = b \cdot a$

- § a1: a3 = Group
- § a1: a4 = Abelian group
- § a1: a7 = Unital ring
- § a1: a8 = Division ring

So $(\mathbb{F}, +)$ is an abelian group & $(\mathbb{F} \setminus \{0\}, \cdot)$ is an abelian group too.

Fields ($\mathbb{F}$)

- Think of a field as a set where you can add, subtract, multiply, and divide (except by zero).
- **Common Examples:**
  - $\mathbb{R}$: The set of **Real Numbers** (the most common field in applied linear algebra).
  - $\mathbb{C}$: The set of **Complex Numbers**.
  - $\mathbb{Q}$: The set of **Rational Numbers**.
  - $\mathbb{Z}_p$ or $\mathbb{F}_p$: Finite fields with p elements (used in coding theory and cryptography).

Vectors

Linearity

Linear Combination:

$$\sum_{i}^{n} w_i x_i \tag{1}$$

In machine learning, you can think of x_i has i-th feature in a sample, such as **Toggle Rate**, **Capacitive Load**, and **Operating Voltage** in case of a dataset concerning the power a semiconductor chip will consume.

Linearity

A System of Linear Equation:

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
&\vdots \\
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \\
a_{ij}, b_i \in \mathbb{R}
\end{aligned} \tag{2}
$$

You can understand this overall system of linear equation as m samples where I am predicitng b: **total power dissipation** of the semiconductor chip given several features.

Vectors

Simplistic Definition:

A vector is a tuple of one or more values called scalars. We will denote a vector by boldface lowercase letters, such as $\mathbf{v}$.

‘

‘Vectors are built from components, which are ordinary numbers. You can think of a vector as a list of numbers, and vector algebra as operations performed on the numbers in the list.”
– No Bullshit Guide To Linear Algebra, 2017.

Vectors

Column Vectors and Row Vectors

Column vectors can be written as $\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$ and row vectors can be written as $\mathbf{v} = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix}$

Vectors in Python

Column Vector with Python List

```python
# Column vector as list of lists
v = [[1], [2], [3]]  # 3 rows, 1 element each
print(f"Number of rows: {len(v)}")
print(f"Elements per row: {len(v[0])}")
print(v)
```

Output:

```
Number of rows: 3
Elements per row: 1
[[1], [2], [3]]
```

Vectors in Python

Column Vector with Python Array

```
from array import array
# Column vector as array (1D structure)
v = array('i', [1, 2, 3])  # 'i' for integers
print(f"Length: {len(v)}")
print(f"Type: {v.typecode}")
print(list(v))  # Convert to list for display
```

Output:

```
Length: 3
Type: i
[1, 2, 3]
```

Vectors in Python

Column Vector in Numpy

```python
import numpy as np
v = np.array([[1], [2],[3]]) # 3 rows 1 column
print(v.shape)
```

Output:

```
(3, 1)
```

Vectors in Python

Row Vector with Python List

```
v = [1, 2, 3]  # 1 row, 3 elements
print(f"Length: {len(v)}")
print(f"Elements: {v}")
# Alternative: Row vector as list containing one list
v_nested = [[1, 2, 3]]  # 1 row, 3 columns (like NumPy)
print(f"Rows: {len(v_nested)}")
print(f"Columns: {len(v_nested[0])}")
print(f"Elements: {v_nested}")
```

Vectors in Python

Row Vector with Python Array

Native array in Python doesn't distinguish between row and column representation.

Vectors in Python

Row Vector in Numpy

```
np.array([[1, 2, 3]]) # 1 row 3 columns
print(v.shape)
```

Output:

```
(1, 3)
```

Alternative Implementation of Vectors in Python

Vector as an array

We can have vector implementation in Python like a 1-D array where there is no distinction between column vectors and row vectors.
We will use this form for any operations on vectors when we don’t need to worry about whether they are column vectors or row vectors.

```
import numpy as np
v = np.array([1, 2,3]) # 3-tuple vector
print(v.shape)
```

Output:

```
(3, )
```

Operations on Vectors I

Elementwise Vector Multiplication

```
import numpy as np
v = np.array([[1], [2], [3]]) # 3 rows 1 column
w = np.array([[3], [4], [5]]) # 3 rows 1 column
u = v*w
print(u)
```

Output:

```
array([[ 3],
    [ 8],
    [15]])
```

Operations on Vectors II

Vector Dot Product (also known as inner product)

Dot product of two vectors $\mathbf{v} = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix}$ and $\mathbf{w} = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix}$ is defined as $u = v \cdot w = v_1 w_1 + v_2 w_2 + v_3 w_3$

```
import numpy as np
v = np.array([1, 2, 3]) # 3 rows 1 column
w = np.array([3, 4, 5]) # 3 rows 1 column
u = v.dot(w)
print(u)
```

Output:

```
26
```

Vector Norms I

We often need to calculate the length or magnitude of a vector. Different ways to calculate vector norms are called vector norms. The length or magnitude is a non-negative number. We will consider three kind of vector norms: L1 norm, L2 norm, and Max norm.

Vector Norms I

L1 Norm (Manhattan Norm)

The L1 norm of a vector $\mathbf{v} = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix}$ is defined as $\|v\|_1 = |v_1| + |v_2| + |v_3|$.

```python
import numpy as np
v = np.array([1, 2, 3])
norm_v = np.linalg.norm(v, ord=1)
print(norm_v)
```

Output:

```
6.0
```

Vector Norms II

L2 Norm (Euclidean Norm)

The L2 norm of a vector $\mathbf{v} = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix}$ is defined as $\|v\|_2 = \sqrt{v_1^2 + v_2^2 + v_3^2}$.

```python
import numpy as np
v = np.array([1, 2, 3])
norm_v = np.linalg.norm(v)
print(norm_v)
```

Output:

```
3.7416573867739413
```

Vector Norms III

Max Norm

The max norm (also known as infinity norm) of a vector $\mathbf{v} = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix}$ is defined as $\|v\|_\infty = \max(|v_1|, |v_2|, |v_3|)$.

```python
import numpy as np
v = np.array([1, 2, 3]) # 3 rows 1 column
norm_v = np.linalg.norm(v, np.inf)
print(norm_v)
```

Output:

```
3.0
```

Vector Spaces (V)

- A **Vector Space V** over a field $\mathbb{F}$ is a set with two operations:
  1. **Vector Addition** ($\mathbf{u} + \mathbf{v} \in V$).
  2. **Scalar Multiplication** ($c\mathbf{u} \in V$, where $c \in \mathbb{F}$).
- These operations must satisfy ten axioms (e.g., commutativity, existence of zero vector $\mathbf{0}$, inverse vector $-\mathbf{v}$).
- **Examples:**
  - $\mathbb{R}^n$: n-tuples of real numbers (Standard n-dimensional space).
  - $M_{m\times n}(\mathbb{F})$: The set of $m \times n$ matrices over $\mathbb{F}$.
  - $P_n(\mathbb{R})$: The set of all polynomials of degree $\leq n$ with real coefficients.

Linear Transformations

Def Let V and W be vector spaces. A linear transformation is a function $L : V \to W$ s.t.

$$\boxed{L(a\vec{v} + b\vec{w}) = aL(\vec{v}) + bL(\vec{w})}$$

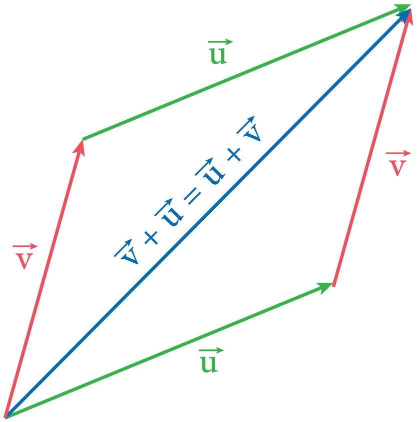
Linear transformations respect the structure.

Vector Spaces and Scalar Fields I

- A **scalar field** is a function that assigns a single number to each point in a region of space or spacetime.
- An example is the temperature distribution in space.
- $f(x, y, z) = x^2 + y^2 + z^2$ is an example of a scalar field.
- A **vector space** V over a scalar field $\mathbb{K}$ satisfies several properties, including:
  - It is **closed under summation** ($\vec{x} + \vec{y} \in V$ for any vectors $\vec{x}, \vec{y} \in V$).
  - It is **closed under scalar multiplication** ($a\vec{x} \in V$ for any scalar a and vector $\vec{x}$).

Vector Spaces and Scalar Fields II

Commutativity of Vector Addition



Subspaces and Linear Mappings I

- A subset W of a vector space V is a **subspace** if it is itself a vector space over the same scalar field, and $\vec{v}, \vec{w} \in W$ and $a, b \in \mathbb{F}$, $a\vec{v} + b\vec{w} \in W$
- A subspace is entirely contained within another vector space, written as $W \subset V$.
- A mapping $f : V \to W$ is a **linear mapping** if it preserves the structure of vector addition and scalar multiplication.
- Properties of linear mappings include:
  - $f(\vec{x} + \vec{y}) = f(\vec{x}) + f(\vec{y})$.
  - $f(a\vec{x}) = af(\vec{x})$.
  - $f(\vec{0}_V) = \vec{0}_W$.
  - $f(-\vec{x}) = -f(\vec{x})$.

Subspace Generation and Linear Dependence I

- The set of all linear combinations of vectors $\{\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n\}$ forms the smallest subspace containing these vectors. This is called the subspace **generated by** or **spanned by** the set.
- A set of vectors $A = \{\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n\}$ is **linearly dependent** if there exist scalars $x_1, x_2, \ldots, x_n$ (at least one of which is not zero) such that $x_1\vec{a}_1 + x_2\vec{a}_2 + \cdots + x_n\vec{a}_n = \vec{0}$.
- A set is **linearly independent** if the equation holds only when all scalars are zero ($x_1 = x_2 = \cdots = x_n = 0$).

Example

$\{2\vec{a}, 3\vec{a}\}$ for any vector $\vec{a}$ is linearly dependent because

$$3 \cdot 2\vec{a} + (-2) \cdot 3\vec{a} = \vec{0}$$

We choose 3 and -2 to show that a linear combination of them may give zero vectors.

Example 2: Linear Independence with SymPy

- To test for linear independence, you can solve the system of equations. For example, for $A = \{(1,2),(2,3)\} \subseteq \mathbb{R}^2$, the system is:

$$x \begin{bmatrix} 1 \\ 2 \end{bmatrix} + y \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{3}$$

$$\begin{aligned} x + 2y &= 0 \\ 2x + 3y &= 0 \end{aligned}$$

- This system has a unique solution of $x = y = 0$, so the set is linearly independent.
- We can use the `sympy` package in Python to solve this:

```python
from sympy import solve
from sympy.abc import x, y
ans = solve([x + 2*y, 2*x + 3*y], [x, y])
print(ans)
```

Example 3

For $A = \{(1,2),(2,4)\} \subseteq \mathbb{R}^2$, the system is:

$$\begin{aligned} x + 2y &= 0 \\ 2x + 4y &= 0 \end{aligned} \tag{4}$$

which has the solution $(x,y) = (2,-1)$ other than $(x,y) = (0,0)$. Hence A is not linearly dependent.

To Summarize:

Linear dependence of A is equivalent to **there exists a vector in A that is linear combination of the other vectors of A**. The linear independence of A is equivalent of **any vector in A never belongs to the subspace generated by the other vector but it**.

Basis and Dimension

- A set $X = \{\vec{a}_1, \dots, \vec{a}_n\}$ is a **basis** for a vector space V if it is both **linearly independent** and **generates V** (spans V).
- The **standard basis** for $\mathbb{R}^n$ is the set of vectors $\{\vec{e}_1, \vec{e}_2, \dots, \vec{e}_n\}$ where $\vec{e}_i$ has a 1 in the i-th position and 0s elsewhere.
- If a vector space V has a basis with m vectors, then any other basis of V will also have m vectors. This number m is the **dimension** of V, denoted as $m = \dim_{\mathbb{K}} V$.
- The dimension of the subspace generated by a set A is called the **rank** of A, denoted as rank(A).

Basis

Some points to note:

1. A set obtained by adding a new vector to X or removing any vector of X is no longer a basis of V.
2. A set obtained by replacing any vector of X with its nonzero scalar multiple remains a basis.
3. A set obtained by replacing any vector of X with a sum of it and a scalar multiple of another vector of X remains a basis.

Standard Basis

The set $\{\vec{e}_1, \vec{e}_2, \ldots, \vec{e}_n\}$ is given by:

$$\vec{e}_1 = (1, 0, \ldots, 0) \tag{5}$$
$$\vec{e}_2 = (0, 1, \ldots, 0) \tag{6}$$
$$\vdots \tag{7}$$
$$\vec{e}_n = (0, 0, \ldots, 1) \tag{8}$$

is a basis of $\mathbb{R}^n$. It is called as the **Standard basis.**

Let X be the basis of V.
Assume that we serialize the vectors in X as $\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n$ and fix this order.
For any vector $\vec{z} \in V$:

$$\vec{z} = x_1\vec{a}_1 + x_2\vec{a}_2 + \cdots + x_n\vec{a}_n$$

We call this the expansion of $\vec{z}$ on the basis of X.

Vector Representation

The vector $\vec{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{K}^n$ made of the expansion coefficient $x_1, x_2, \ldots, x_n$ is called the representation of $\vec{z}$ on the basis X.

With the standard basis, we could write:

$$\vec{x} = x_1\vec{e}_1 + x_2\vec{e}_2 + \cdots + x_n\vec{e}_n \tag{9}$$

$$\vec{x} = x_1(1, 0, \ldots, 0) + x_2(0, 1, \ldots, 0) + \cdots + x_n(0, 0, \ldots, 1) \tag{10}$$

Note that at this point we are currently not caring about whether it is a column vector or row vector

Bijective Mapping Between Spaces

$*$ $s\vec{z} + t\vec{y}$ implies $f_X(s\vec{z} + t\vec{y})$ where f_X is bijective mapping from V to $\mathbb{K}^n$ with

$$f_X(s\vec{z} + t\vec{y}) = sf_X(\vec{z}) + tf_X(\vec{y})$$

Bijective mapping refresher:

- Every element in $\mathbb{K}^n$ is mapped to by exactly one element in V.
- Each element in V maps to a unique element in $\mathbb{K}^n$.

Rank

Rank

The dimension of the subspace generated by $A = \{a_1, a_2, \ldots, a_n\}$ is called the **rank of** A. We denote it by $\text{rank}_{\mathbb{K}} A$ or $\text{rank}\, A$.

Some Points:

1. $\text{rank}\, A \leq n$
2. If A is linearly independent, then $\text{rank}\, A = n$
3. If A is linearly dependent, then $\text{rank}\, A < n$

If A is a subset of an m-dimensional vector space V, then:

1. $\text{rank}\, A \leq m$
2. If A generates V, then $\text{rank}\, A = m$.
3. If A doesn't generate V, then $\text{rank}\, A < m$.

Direct Sum

Direct Sum

Let $W_1, W_2, \ldots, W_k$ be subspaces of vector space V.
and $W = \{\vec{x}_1 + \vec{x}_2 + \cdots + \vec{x}_k \mid \vec{x}_1 \in W_1, \vec{x}_2 \in W_2, \ldots, \vec{x}_k \in W_k\}$
then W is a subspace of V.
W is called a **sum of subspaces**.

In this case, when every element $\vec{x}$ of W is uniquely expressed $\vec{x} = \vec{x}_1 + \vec{x}_2 + \cdots + \vec{x}_k$ with $\vec{x}_1 \in W_1, \vec{x}_2 \in W_2, \ldots$ then we call W the **direct sum** of $W_1, W_2, \ldots, W_k$.

We can also write $\mathbb{W} = \{W_1, W_2, \ldots, W_k\}$
and denote the direct sum as $W_1 \oplus W_2 \oplus \cdots \oplus W_k$.

Python Implementation: Direct Sum

In Python: direct sum can be written as

```
[1, 2] + [3, 4, 5]
```

Output: `[1, 2, 3, 4, 5]`

or using numpy

```
from numpy import array, concatenate
concatenate (array([1,2]), array([3,4,5]))
```

In sympy:

```
from sympy import Matrix
Matrix([1,2]).col_join(Matrix([3,4,5]))
```

NumPy Arrays and n-Dimensional Arrangements

We denote n-dimensional arrangement by arrays in NumPy Python.

1. 1-D arrangement is a sequence of elements arranged in a row.
2. 2D arrangement is a matrix in which elements are arranged vertically and horizontally in a 2-D plane.
3. 3D arrangement is a layout of elements arranged vertically, horizontally, and depth-wise in 3D space.

```
from numpy import array
A = array([1,2,3])                    # 1D
B = array([[1,2,3], [4,5,6]])         # 2D
C = array([[[1,2], [3,4]], [[5,6], [7,8]]])  #Two 2D Arrays
```

Vector Broadcasting in Python

Vector broadcasting is purely a computer operation.
Consider Python code:

```
v = np.array([[1,5,6]])     # row vector (2 columns)
w = np.array([[10,20,30]]).T     # column vector (3 rows)
v+w
```

Output:

```
array([[11, 15, 16],
       [21, 25, 26],
       [31, 35, 36]])
```

What is going on?
We are adding two vectors of dimension 1×3 and 3×1. Clearly there is a dimension mismatch but there doesn't seem to be an error!
Here, broadcasting operation is taking place even though there is a dimension mismatch.
Broadcasting essentially means to repeat an operation multiple times between one vector and each element of another vector.

Example

```
import numpy as np
v = np.array([[1,2,3]]).T  # col vector 3 rows 1 col 3x1
w = np.array([[10,20]])     #  row vector 1 row 2 col 1x2
v+w
```

It does the following operation on $v + w$:

$$\begin{aligned}&[1, 1] + [10, 20]\\&[2, 2] + [10, 20]\\&[3, 3] + [10, 20]\end{aligned} \tag{11}$$

Broadcasting allows for compact and efficient calculations in numerical coding.

Matrices

Matrices

Simplistic Definition:

For a positive integer $m, n \in \mathbb{R}$, a matrix $\mathbf{A}$ is $m \times n$ tuple of elements $a_{i,j}, i = 1, \cdot, m, j = 1, \cdot n$ which is ordered according to a rectangular scheme with m rows and n columns such that we can write:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix} \tag{12}$$

We will set a convention to use boldface uppercase letters to denote a matrix. You can see that a matrix can be constructed by placing multiple row vectors or column vectors as well.

Matrix in Python

A rectangular matrix

```
import numpy as np
A = np.array([[1, 2, 3], [4, 5, 6]])
print(A.shape)
```

Output:

```
(2, 3)
```

Elementwise Matrix Multiplication (Hadamard Product)

Two matrices with the same size can be multiplied together, and this is often called element-wise matrix multiplication or the Hadamard product. It is denoted using a small circle $\circ$:

$$\mathbf{C} = \mathbf{A} \circ \mathbf{B}$$

.

Hadamard Product

```
import numpy as np
A = np.array([[1, 2, 3],[4, 5, 6]])
B = np.array([[1, 2, 3],[4, 5, 6]])
C = A * B
```

Matrix-Matrix Multiplication (Or, Matrix Multiplication) I

Matrix multiplication is a binary operation that takes a pair of matrices and produces another matrix. If $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{B}$ is an $n \times k$ matrix, then their matrix product $\mathbf{AB}$ is an $m \times k$ matrix.
The element at the i-th row and j-th column of $\mathbf{C}$, denoted as c_{ij}, can be calculated as follows:

$$c_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj} \tag{13}$$

where a_{ir} is the element at the i-th row and r-th column of $\mathbf{A}$, and b_{rj} is the element at the r-th row and j-th column of $\mathbf{B}$. The summation runs over the r index from 1 to n.
In Python, you can use the ‘numpy‘ library to perform matrix multiplication using the ‘dot‘ function.

Matrix-Matrix Multiplication (Or, Matrix Multiplication) II

Matrix Multiplication

```
import numpy as np
A = np.array([[1, 2], [3, 4]]) # 2x2 matrix
B = np.array([[5, 6], [7, 8]]) # 2x2 matrix
C = A.dot(B)
print(C)
```

Output:

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

Types of Matrices I

Triangular Matrices

A triangular matrix is a special kind of square matrix where all the entries above the main diagonal are zero (lower triangular) or all the entries below the main diagonal are zero (upper triangular).
Example in Python:

```python
import numpy as np
A = np.array([[1, 2, 3], [0, 4, 5], [0, 0, 6]]) # Upper triangular
B = np.array([[1, 0, 0], [4, 5, 0], [7, 8, 9]]) # Lower triangular
print("A = ", A)
print("B = ", B)
```

Types of Matrices II

Identity Matrices

An identity matrix $\mathbf{I}$ is a square matrix in which all the elements of the principal diagonal are ones and all other elements are zeros.
Example in Python:

```python
import numpy as np
I = np.eye(3) # 3x3 identity matrix
print("I = ", I)
```

Types of Matrices III

Diagonal Matrices

A diagonal matrix is a matrix in which the entries outside the main diagonal are all zero.
Example in Python:

```python
import numpy as np
D = np.diag([1, 2, 3]) # Diagonal matrix with diagonal elements 1, 2, 3
print("D = ", D)
```

Types of Matrices IV

Orthogonal Matrices

An orthogonal matrix is a square matrix whose rows and columns are orthogonal unit vectors (i.e., orthonormal vectors), making it a column and row-stochastic matrix. Orthogonal matrices have the property that their transpose is equal to their inverse, i.e.

$$Q^\top = Q^{-1} \quad \text{or} \quad Q \cdot Q^\top = I$$

Types of Matrices V

Orthogonal Matrices: Example in Python

```python
import numpy as np
Q = np.array([[1, 0], [0, -1]]) # An example of orthogonal matrix
print("Q = ", Q)
print("Q Transpose = ", Q.T)
print("Q Inverse = ", np.linalg.inv(Q))
```

Matrices

Operations on Matrices

1 1 1 1 0 1 0 0 1 1 1 0 0 0 0
1 1 0 0 1 0 1 0 0 0 1 0 0 1 0

Operations on Matrices

Transpose

The transpose of a matrix is obtained by interchanging its rows into columns or columns into rows. It is denoted by $\mathbf{A}^T$. For example, if

Operations on Matrices

Transpose: Example

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \tag{14}$$

then,

$$A^\top = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix} \tag{15}$$

Operations on Matrices

Transpose: Example in Python

```
import numpy as np
A = np.array([[1, 2, 3], [4, 5, 6]])
print("A = ", A)
print("Transpose of A = ", A.T)
```

Operations on Matrices

Determinant

The determinant is a special number that can be calculated from a square matrix. The determinant helps us find the inverse of a matrix, tells us things about the matrix that are useful in systems of linear equations, calculus and more.

Operations on Matrices

Determinant

The determinant of a matrix gives us important information about the matrix. For example, a matrix is invertible (i.e., has an inverse) if and only if its determinant is non-zero. The determinant of a matrix also gives us the scale factor by which area (or volume, in higher dimensions) is transformed under the linear transformation represented by the matrix.
It's important to note that the determinant is only defined for square matrices. For non-square matrices, related concepts like the rank or the pseudoinverse are often used instead.

Operations on Matrices

A 2x2 Determinant

The determinant of a matrix $\mathbf{A}$ is often denoted as $|\mathbf{A}|$ or $\det(\mathbf{A})$.
For a 2x2 matrix:

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

The determinant is calculated as:

$$|\mathbf{A}| = ad - bc$$

Operations on Matrices

Determinant

For a 3x3 matrix:

$$\mathbf{A} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

The determinant is calculated as:

$$|\mathbf{A}| = a(ei - fh) - b(di - gf) + c(dh - eg)$$

For larger square matrices (4x4 and above), the determinant is usually computed using more advanced methods like the Laplace expansion, or LU decomposition. The determinant of a matrix can also be calculated using the product of its eigenvalues.

Operations on Matrices

Determinant: Example in Python

```
A = np.array([[1, 2], [3, 4]])
print("A = ", A)
print("Determinant of A = ", np.linalg.det(A))
```

Operations on Matrices

Inverse

The inverse of a square matrix $\mathbf{A}$ is denoted as $\mathbf{A}^{-1}$, and it is the matrix such that when it is multiplied by $\mathbf{A}$, the result is the identity matrix.

Example in Python:

```
A = np.array([[4, 7], [2, 6]])
print("A = ", A)
print("Inverse of A = ", np.linalg.inv(A))
```

Operations on Matrices

Trace

The trace of a square matrix $\mathbf{A}$, denoted as $\mathrm{tr}(\mathbf{A})$, is the sum of the elements on the main diagonal.

Example in Python:

```
A = np.array([[1, 2], [3, 4]])
print("A = ", A)
print("Trace of A = ", np.trace(A))
```

Operations on Matrices

Rank

The rank of a matrix $\mathbf{A}$ is the maximum number of linearly independent row vectors in the matrix.
Example in Python:

```python
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("A = ", A)
print("Rank of A = ", np.linalg.matrix_rank(A))
```

Applications of Matrices

Solving Systems of Linear Equations

Let's look at the systems of linear equations again.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \\ a_{ij}, b_i \in \mathbb{R} \end{aligned} \tag{16}$$

In Matrix form, we could write it as

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}_{m\times n} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}_{n\times 1} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}_{m\times 1} \tag{17}$$

$$A_{m\times n}X_{n\times 1} = B_{m\times 1}$$

How to Solve Systems of Linear Equations using Matrix

$$AX = B \tag{18}$$

1. Matrix Inverse
2. Gaussian Elimination

Matrix Inverse

$$X = A^{-1}B \tag{19}$$

Inverse can be calculated as

$$A^{-1} = \frac{1}{|A|} Adj A \tag{20}$$

where $|A|$ is the determininant of the matrix.

$Adj(A)$ is the adjoint of a matrix A (not the same as a adjoint Matrix. $adj(A)$ is also called adjugate matrix.

Example

Consider $A = \begin{bmatrix} -2 & 5 & 1 \\ 4 & 1 & 0 \\ -3 & 5 & 5 \end{bmatrix}$. $Adj(A)$ can be written as $Adj(A) = \begin{bmatrix} C_{11} & C_2 & C_{31} \\ C_{12} & C_{22} & C_{23} \\ C_{13} & C_{23} & C_{33} \end{bmatrix}$. C_{ij} are called cofactors.

$C_{11} = +\left| \begin{bmatrix} 1 & 0 \\ 5 & 5 \end{bmatrix} \right| = 5 - 0 = 5, \quad C_{12} = -\left| \begin{bmatrix} 4 & 0 \\ -3 & 5 \end{bmatrix} \right| = -(20 - 0) = -20,$
$C_{13} = +\left| \begin{bmatrix} 4 & 1 \\ -3 & 5 \end{bmatrix} \right| = 20 + 3 = 23$. Similarly, you can calculate for C_{ij}. This way,

$$Adj(A) = \begin{bmatrix} 5 & -20 & -1 \\ -20 & -7 & 4 \\ 23 & -5 & -22 \end{bmatrix}$$

General Formula for Solving Systems of Linear Equations by Inverse

1. First, calculate the cofactors $C_{ij} = (-1)^{i+j} M_{ij}$ where M_{ij} is the determininant of $(n-1) \times (n-1)$ matrix resulting from deleting row i and column j of A.
2. Divide cofactor matrix by $det(A) = |A|$.

$$\begin{aligned} AX &= B \\ A^{-1}AX &= A^{-1}B \\ IX &= A^{-1}B \\ X &= A^{-1}B \end{aligned} \tag{21}$$

⚠ Computing $Adj(A)$ for a high-dimensional matrix is computationally expensive. We need better methods.

We use Gaussian elimination method for solving large matrices. For that we need to understand Row Echelon Form, and elementary operations.

What is Row Echelon Form?

A matrix is in Row Echelon Form (REF) if it satisfies these rules:

1. **Leading Entry Rule:** Any non-zero row has a leading **1** (pivot)
2. **Staircase Rule:** Each pivot is right of the one above
3. **Zero Rule:** All-zero rows are at the bottom

$$\begin{bmatrix} \mathbf{1} & 2 & 3 & 4 \\ 0 & \mathbf{1} & 5 & 6 \\ 0 & 0 & 0 & \mathbf{1} \end{bmatrix} \qquad \begin{bmatrix} \mathbf{1} & 2 & 3 \\ 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 \end{bmatrix}$$

Elementary Row Operations

We use these operations to achieve REF:

Operation	**Notation**	**Effect**
Row Swap	$R_i \leftrightarrow R_j$	Swap two rows
Scaling	kR_i	Multiply row by non-zero k
Replacement	$R_i + kR_j$	Add multiple of one row to another

Key Property: These operations do **not** change the solution set of the system!

Step-by-Step Example: Starting Matrix

Let's transform this matrix into REF:

$$A = \begin{bmatrix} 0 & 3 & -6 & 6 & 4 \\ 3 & -7 & 8 & -5 & 8 \\ 3 & -9 & 12 & -9 & 6 \end{bmatrix}$$

Step 1: Get pivot in position (1,1)

Step 1: Create First Pivot

Swap Rows 1 and 2, then scale new Row 1:

$$R_1 \leftrightarrow R_2 \text{ then } \frac{1}{3}R_1$$

$$\to \begin{bmatrix} \mathbf{1} & -\frac{7}{3} & \frac{8}{3} & -\frac{5}{3} & \frac{8}{3} \\ 0 & 3 & -6 & 6 & 4 \\ 3 & -9 & 12 & -9 & 6 \end{bmatrix}$$

Step 2: Create Zeros Below Pivot

Eliminate the 3 in position (3,1):

$$-3R_1 + R_3$$

$$\rightarrow \begin{bmatrix} \mathbf{1} & -\frac{7}{3} & \frac{8}{3} & -\frac{5}{3} & \frac{8}{3} \\ 0 & 3 & -6 & 6 & 4 \\ \mathbf{0} & -2 & 4 & -4 & -2 \end{bmatrix}$$

Step 3: Create Second Pivot

Scale Row 2 to create pivot:

$$\frac{1}{3}R_2$$

$$\to \begin{bmatrix} \mathbf{1} & -\frac{7}{3} & \frac{8}{3} & -\frac{5}{3} & \frac{8}{3} \\ 0 & \mathbf{1} & -2 & 2 & \frac{4}{3} \\ 0 & -2 & 4 & -4 & -2 \end{bmatrix}$$

Step 4: Create More Zeros

Eliminate below second pivot:

$$2R_2 + R_3$$

$$\rightarrow \begin{bmatrix} \mathbf{1} & -\frac{7}{3} & \frac{8}{3} & -\frac{5}{3} & \frac{8}{3} \\ 0 & \mathbf{1} & -2 & 2 & \frac{4}{3} \\ 0 & \mathbf{0} & 0 & 0 & \frac{2}{3} \end{bmatrix}$$

Step 5: Final REF Form

Scale Row 3 for final pivot:

$$\frac{3}{2}R_3$$

$$A_{\text{REF}} = \begin{bmatrix} \mathbf{1} & -\frac{7}{3} & \frac{8}{3} & -\frac{5}{3} & \frac{8}{3} \\ 0 & \mathbf{1} & -2 & 2 & \frac{4}{3} \\ 0 & 0 & 0 & 0 & \mathbf{1} \end{bmatrix}$$

Success! Matrix is now in Row Echelon Form.

Calculating Matrix Inverse

To find the inverse A^{-1} of a matrix A, we need to solve:

$$AX = I_n$$

for the unknown matrix X, which will be A^{-1}.

We can set this up as an Augmented Matrix:

$$[A \mid I_n]$$

Goal: Use row operations to transform this into $[I_n \mid A^{-1}]$

The process is Gaussian Elimination (also called Gauss-Jordan Elimination), and we perform the operations on the entire augmented matrix.

Reduced Row Echelon Form

An equation system is in reduced REF if

1. It is in REF.
2. Every pivot is 1.
3. The pivot is the only non-zero entry in its column.

Example:

$$A = \begin{bmatrix} 1 & 3 & 0 & 0 & 3 \\ 0 & 0 & 1 & 0 & 9 \\ 0 & 0 & 0 & 1 & -4 \end{bmatrix}$$

Example: Matrix A and Its Augmented Matrix

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

We form the augmented matrix $[A|I_4]$:

$$\left[\begin{array}{cccc|cccc} 1 & 0 & 2 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 2 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right]$$

Our goal is to turn the left side into I_4. The right side will then become A^{-1}. We need to use eleme matrix operation to convert left side to reduce REF.

Step 1: Establish the First Pivot

Pivot position (1,1) is already 1. Now, we create zeros below it.

Operations:

$$R_2 = R_2 - R_1$$
$$R_3 = R_3 - R_1$$
$$R_4 = R_4 - R_1$$

$$\rightarrow \left[\begin{array}{cccc|cccc} 1 & 0 & 2 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 0 & -1 & 1 & 0 & 0 \\ 0 & 2 & -2 & 1 & -1 & 0 & 1 & 0 \\ 0 & 1 & -1 & 1 & -1 & 0 & 0 & 1 \end{array}\right]$$

Pivot: position (1,1) = 1

Zeros created below pivot in column 1

Step 2: Establish the Second Pivot

Pivot position (2,2) is already 1. Create zeros above and below it.

Operations:

$$R_3 = R_3 - 2R_2$$
$$R_4 = R_4 - R_2$$

$$\rightarrow \left[\begin{array}{cccc|cccc} 1 & 0 & 2 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 2 & 1 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & -1 & 0 & 1 \end{array}\right]$$

Pivots: positions (1,1) = 1 and (2,2) = 1

Zeros created below pivot in column 2

Step 3: Establish the Third Pivot

Pivot position (3,3) is 2. Scale Row 3 to make it 1.

Operation:

$$R_3 = \frac{1}{2}R_3$$

$$\rightarrow \left[\begin{array}{cccc|cccc} 1 & 0 & 2 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.5 & 0.5 & -1 & 0.5 & 0 \\ 0 & 0 & 1 & 1 & 0 & -1 & 0 & 1 \end{array}\right]$$

Now create zeros above and below the new pivot. Operations:

$$R_1 = R_1 - 2R_3$$
$$R_2 = R_2 + 2R_3$$
$$R_4 = R_4 - R_3$$

Step 3 (Cont'd): Results after Eliminations

$$\rightarrow \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & -1 & 0 & 2 & -1 & 0 \\ 0 & 1 & 0 & 1 & 0 & -1 & 1 & 0 \\ 0 & 0 & 1 & 0.5 & 0.5 & -1 & 0.5 & 0 \\ 0 & 0 & 0 & 0.5 & -0.5 & 0 & -0.5 & 1 \end{array}\right]$$

Pivots: positions (1,1), (2,2), (3,3) = 1

Zero created below pivot in column 3

Step 4: Establish the Fourth Pivot

Pivot position (4,4) is 0.5. Scale Row 4 to make it 1.

Operation:

$$R_4 = 2R_4$$

$$\rightarrow \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & -1 & 0 & 2 & -1 & 0 \\ 0 & 1 & 0 & 1 & 0 & -1 & 1 & 0 \\ 0 & 0 & 1 & 0.5 & 0.5 & -1 & 0.5 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & -1 & 2 \end{array}\right]$$

Now create zeros above the new pivot. Operations:

$$R_1 = R_1 + R_4$$
$$R_2 = R_2 - R_4$$
$$R_3 = R_3 - 0.5R_4$$

Step 4 (Cont'd): The Final Result - Reduced Row Echelon Form

After performing the final eliminations, we achieve our goal:

$$\left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & -1 & 2 & -2 & 2 \\ 0 & 1 & 0 & 0 & 1 & -1 & 2 & -2 \\ 0 & 0 & 1 & 0 & 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 1 & -1 & 0 & -1 & 2 \end{array}\right]$$

Success! The left side is I_4.
Therefore, the right side is $\mathbf{A}^{-1}$

We Have Found the Inverse!

We can now read off the inverse matrix from the augmented matrix:

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

Verification

We can verify this is correct by checking that $A \cdot A^{-1} = I_4$. Multiplying the original matrix A by this found matrix A^{-1} will indeed yield the identity matrix. Using this, we can find the solution of a system of linear equations.

Solving A System of Linear Equations Directly with Gaussian Elimination

Consider a system of linear equations as

$$\begin{aligned} 2x_1 + 3x_2 + 5x_3 &= 1 \\ 4x_1 - 2x_2 - 7x_3 &= 8 \\ 9x_1 + 5x_2 - 3x_3 &= 2 \end{aligned} \tag{22}$$

We can write augmented matrix as

$$\left[\begin{array}{ccc|c} 2 & 3 & 5 & 1 \\ 4 & -2 & -7 & 8 \\ 9 & 5 & -3 & 2 \end{array}\right]$$

Step 1: Goal for Column 1

Goal: Get a pivot of 1 in position (1,1) and zeros below it.

$$\left[\begin{array}{ccc|c} 2 & 3 & 5 & 1 \\ 4 & -2 & -7 & 8 \\ 9 & 5 & -3 & 2 \end{array}\right]$$

Strategy:

1. First, make the pivot **1** by scaling Row 1.
2. Then, eliminate the entries below it (4 and 9).

Step 1a: Create the First Pivot

Scale Row 1 by $\frac{1}{2}$ to get a 1 in position (1,1).

Operation: $R_1 = \frac{1}{2}R_1$

$$\rightarrow \left[\begin{array}{ccc|c} 1 & 1.5 & 2.5 & 0.5 \\ 4 & -2 & -7 & 8 \\ 9 & 5 & -3 & 2 \end{array}\right]$$

Pivot created at position (1,1) = 1

Step 1b: Eliminate Below the First Pivot

Use the new pivot to create zeros in the rest of column 1.

Operations:

$$R_2 = R_2 - 4R_1$$
$$R_3 = R_3 - 9R_1$$

$$\rightarrow \left[\begin{array}{ccc|c} 1 & 1.5 & 2.5 & 0.5 \\ 0 & -8 & -17 & 6 \\ 0 & -8.5 & -25.5 & -2.5 \end{array}\right]$$

Pivot: position (1,1) = 1

Zeros created below pivot in column 1

Step 2: Goal for Column 2

Goal: Get a pivot of 1 in position (2,2) and zeros below it.

$$\left[\begin{array}{ccc|c} 1 & 1.5 & 2.5 & 0.5 \\ 0 & -8 & -17 & 6 \\ 0 & -8.5 & -25.5 & -2.5 \end{array}\right]$$

Strategy:

1. First, make the pivot **1** by scaling Row 2.
2. Then, eliminate the entry below it (-8.5).

Step 2a: Create the Second Pivot

Scale Row 2 by $-\frac{1}{8}$ to get a 1 in position (2,2).

Operation: $R_2 = -\frac{1}{8}R_2$

$$\rightarrow \left[\begin{array}{ccc|c} 1 & 1.5 & 2.5 & 0.5 \\ 0 & 1 & 2.125 & -0.75 \\ 0 & -8.5 & -25.5 & -2.5 \end{array}\right]$$

Pivots: positions (1,1) = 1 and (2,2) = 1

Step 2b: Eliminate Below the Second Pivot

Use the new pivot to create a zero in the rest of column 2.

Operation: $R_3 = R_3 + 8.5R_2$

$$\rightarrow \left[\begin{array}{ccc|c} 1 & 1.5 & 2.5 & 0.5 \\ 0 & 1 & 2.125 & -0.75 \\ 0 & 0 & -7.4375 & -8.875 \end{array}\right]$$

Pivots: positions (1,1) and (2,2) = 1

Zero created below pivot in column 2

Step 3: Goal for Column 3 & Back-Substitution

Goal: Get a pivot of 1 in position (3,3). The matrix is now in Row Echelon Form (REF). We can then use back-substitution to solve.

$$\left[\begin{array}{ccc|c} 1 & 1.5 & 2.5 & 0.5 \\ 0 & 1 & 2.125 & -0.75 \\ 0 & 0 & -7.4375 & -8.875 \end{array}\right]$$

Strategy:

1. Scale Row 3 to get the final pivot.
2. Read the simplified system from the REF matrix.
3. Solve from the bottom up.

Step 3a: Create the Third Pivot

Scale Row 3 by $-\frac{1}{7.4375}$ to get a 1 in position (3,3).
(Note: $-\frac{1}{7.4375} \approx -0.1345$)

Operation: $R_3 = -\frac{1}{7.4375} R_3$

$$\rightarrow \left[\begin{array}{ccc|c} 1 & 1.5 & 2.5 & 0.5 \\ 0 & 1 & 2.125 & -0.75 \\ 0 & 0 & 1 & 1.193 \end{array}\right]$$

All pivots: positions (1,1), (2,2), (3,3) = 1

Step 4: Back-Substitution

The REF matrix corresponds to the simplified system:

$$\begin{aligned} x_1 + 1.5x_2 + 2.5x_3 &= 0.5 \\ x_2 + 2.125x_3 &= -0.75 \\ x_3 &= 1.193 \end{aligned}$$

Solve from the bottom up:

1. From Eq. 3: $x_3 = 1.193$
2. Substitute x_3 into Eq. 2: $x_2 = -0.75 - 2.125(1.193) \approx -3.285$
3. Substitute x_2, x_3 into Eq. 1: $x_1 = 0.5 - 1.5(-3.285) - 2.5(1.193) \approx 2.445$

Solution

$\mathbf{x_1 \approx 2.445}, \quad \mathbf{x_2 \approx -3.285}, \quad \mathbf{x_3 \approx 1.193}$

References

1. Chapter 2: Linear Algebra, *Mathematics for Machine Learning*, Deisenroth, Faisal & Ong.
2. Chapter 1 Vector Spaces, *Advanced Linear and Matrix Algebra*, Nathaniel Johnston
3. Chapter 2 Matrix Decompositions, *Advanced Linear and Matrix Algebra*, Nathaniel Johnston

PyTorch Implementation

Column Vectors

```
import torch
import numpy as np
v_col = torch.tensor([[1], [2], [3]], dtype=torch.float32)
print(f"Column vector v: \n{v_col}")
print(f"Shape: {v_col.shape}")
```

Row Vectors

```python
v_row = torch.tensor([[1, 2, 3]], dtype=torch.float32)
print(f"Row vector v: \n{v_row}")
print(f"Shape: {v_row.shape}")
```

1D Vector

```
v_1d = torch.tensor([1, 2, 3], dtype=torch.float32)
print(f"1D vector v: {v_1d}")
print(f"Shape: {v_1d.shape}")
```

Elementwise Vector Multiplication

```python
v = torch.tensor([[1], [2], [3]], dtype=torch.float32)
w = torch.tensor([[3], [4], [5]], dtype=torch.float32)
u_elementwise = v * w
print(f"v * w (elementwise) = \n{u_elementwise}")
```

Vector Dot Product

```
v_1d = torch.tensor([1, 2, 3], dtype=torch.float32)
w_1d = torch.tensor([3, 4, 5], dtype=torch.float32)
# Alternative dot product using matrix multiplication
dot_product = torch.dot(v_1d, w_1d)
dot_product_alt = v.T @ w
```

Vector Norms

```python
v = torch.tensor([1, 2, 3], dtype=torch.float32)
# L1 Norm (Manhattan Norm)
l1_norm = torch.norm(v, p=1)
# L2 Norm (Euclidean Norm)
l2_norm = torch.norm(v, p=2)  # or simply torch.norm(v)
# Max Norm (Infinity Norm)
max_norm = torch.norm(v, p=float('inf'))
```

Vector Broadcasting

```
# Example 1:
print("Broadcasting Example 1:")
v_row = torch.tensor([[1, 5, 6]], dtype=torch.float32)  # 1x3
w_col = torch.tensor([[10], [20], [30]], dtype=torch.float32)  # 3x1
result = v_row + w_col
# Example 2
print("\nBroadcasting Example 2:")
v_col = torch.tensor([[1], [2], [3]], dtype=torch.float32)  # 3x1
w_row = torch.tensor([[10, 20]], dtype=torch.float32)  # 1x2
result2 = v_col + w_row
```

Rectangular Matrix

```
A = torch.tensor([[1, 2, 3], [4, 5, 6]], dtype=torch.float32)
print(f"Matrix A: \n{A}")
print(f"Shape: {A.shape}")
```

Hadamard Product (Elementwise Multiplication)

```
A = torch.tensor([[1, 2, 3], [4, 5, 6]], dtype=torch.float32)
B = torch.tensor([[1, 2, 3], [4, 5, 6]], dtype=torch.float32)
C_hadamard = A * B
```

Matrix-Matrix Multiplication

```python
A = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
B = torch.tensor([[5, 6], [7, 8]], dtype=torch.float32)
C_matmul = torch.matmul(A, B)  # or A @ B
```

Upper Triangular Matrix

```
A_upper = torch.tensor([[1, 2, 3], [0, 4, 5], [0, 0, 6]], dtype=torch.float32)
```

Lower Triangular Matrix

```
B_lower = torch.tensor([[1, 0, 0], [4, 5, 0], [7, 8, 9]], dtype=torch.float32)
```

Identity Matrix

```
I = torch.eye(3)
```

Diagonal Matrix

```
D = torch.diag(torch.tensor([1, 2, 3], dtype=torch.float32))
```

Orthogonal Matrix

```
Q = torch.tensor([[1, 0], [0, -1]], dtype=torch.float32)
Q_transpose = Q.T
Q_inverse = torch.inverse(Q)
print(f"Q = \n{Q}")
print(f"Q^T = \n{Q_transpose}")
print(f"Q^(-1) = \n{Q_inverse}")
print(f"Q @ Q^T = \n{Q @ Q_transpose}")  # Should be identity
```

Matrix Transpose

```
A = torch.tensor([[1, 2, 3], [4, 5, 6]], dtype=torch.float32)
A_transpose = A.T
```

Matrix Determinant

```
A = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
det_A = torch.det(A)
```

Matrix Inverse

```python
A = torch.tensor([[4, 7], [2, 6]], dtype=torch.float32)
A_inv = torch.inverse(A)
print(f"A = \n{A}")
print(f"A^(-1) = \n{A_inv}")
print(f"A @ A^(-1) = \n{A @ A_inv}")  # Should be close to identity
```

Matrix Trace

```python
A = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
trace_A = torch.trace(A)
print(f"A = \n{A}")
print(f"tr(A) = {trace_A}")
```

Matrix Rank

```
A = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype=torch.float32)
rank_A = torch.linalg.matrix_rank(A)
print(f"A = \n{A}")
print(f"rank(A) = {rank_A}")
```

System of Linear Equations: AX = B

```python
A = torch.tensor([[2, 3, 5],
                  [4, -2, -7],
                  [9, 5, -3]], dtype=torch.float32)
B = torch.tensor([[1], [8], [2]], dtype=torch.float32)

# Method 1: Using matrix inverse
print("\nMethod 1: Using Matrix Inverse (X = A^(-1) * B):")
try:
    A_inv = torch.inverse(A)
    X_inverse = A_inv @ B
    print(f"A^(-1) = \n{A_inv}")
    print(f"Solution X = A^(-1) @ B = \n{X_inverse}")

    # Verification
    verification = A @ X_inverse
    print(f"Verification: A @ X = \n{verification}")
    print(f"Should equal B = \n{B}")
except:
    print("Matrix is singular (non-invertible)")
```

System of Linear Equations: AX = B

```python
# Method 2: Using torch.linalg.solve (more numerically stable)
print("\nMethod 2: Using torch.linalg.solve:")
X_solve = torch.linalg.solve(A, B)
print(f"Solution X = \n{X_solve}")

# Verification
verification_solve = A @ X_solve
print(f"Verification: A @ X = \n{verification_solve}")
```

Gaussian Elimination

```
# Create augmented matrix for the system
A_orig = torch.tensor([[2, 3, 5],
                        [4, -2, -7],
                        [9, 5, -3]], dtype=torch.float32)
B_orig = torch.tensor([[1], [8], [2]], dtype=torch.float32)
# Create augmented matrix
augmented = torch.cat([A_orig, B_orig], dim=1)
print(f"Augmented matrix = \n{augmented}")
# Manual Gaussian elimination steps
print("\nStep 1: Scale first row to get pivot = 1")
augmented[0] = augmented[0] / augmented[0, 0]
print(f"After scaling R1 by 1/2: \n{augmented}")
print("\nStep 2: Eliminate below first pivot")
augmented[1] = augmented[1] - 4 * augmented[0]
augmented[2] = augmented[2] - 9 * augmented[0]
print(f"After eliminating below pivot: \n{augmented}")
print("\nStep 3: Scale second row to get pivot = 1")
augmented[1] = augmented[1] / augmented[1, 1]
print(f"After scaling R2: \n{augmented}")
print("\nStep 4: Eliminate below second pivot")
augmented[2] = augmented[2] - augmented[2, 1] * augmented[1]
print(f"After eliminating below second pivot: \n{augmented}")
print("\nStep 5: Scale third row to get pivot = 1")
augmented[2] = augmented[2] / augmented[2, 2]
print(f"Final REF form: \n{augmented}")
```

Extract Solution Using Back Substitution

```python
# Extract solution using back-substitution
x3 = augmented[2, 3]
x2 = augmented[1, 3] - augmented[1, 2] * x3
x1 = augmented[0, 3] - augmented[0, 2] * x3 - augmented[0, 1] * x2

print(f"\nBack-substitution solution:")
print(f"x3 = {x3}")
print(f"x2 = {x2}")
print(f"x1 = {x1}")
print(f"Solution vector: [{x1}, {x2}, {x3}]")
```

Linear Independence Testing

```python
# Example 1: Linearly independent vectors
print("Example 1: Testing linear independence")
v1 = torch.tensor([1, 2], dtype=torch.float32)
v2 = torch.tensor([2, 3], dtype=torch.float32)

# Create matrix with vectors as columns
vectors_matrix = torch.stack([v1, v2], dim=1)
print(f"Matrix with vectors as columns: \n{vectors_matrix}")

# Check determinant (for square matrices)
det_vectors = torch.det(vectors_matrix)
print(f"Determinant: {det_vectors}")
print(f"Linearly independent? {abs(det_vectors) > 1e-6}")
```

Linear Independence Testing

```
# Example 2: Linearly dependent vectors
print("\nExample 2: Linearly dependent vectors")
v3 = torch.tensor([1, 2], dtype=torch.float32)
v4 = torch.tensor([2, 4], dtype=torch.float32)  # v4 = 2 * v3

dependent_matrix = torch.stack([v3, v4], dim=1)
print(f"Matrix with dependent vectors: \n{dependent_matrix}")

det_dependent = torch.det(dependent_matrix)
print(f"Determinant: {det_dependent}")
print(f"Linearly independent? {abs(det_dependent) > 1e-6}")
```

Basis and Dimensions

```python
# Standard basis for R^3
print("Standard basis for R^3:")
e1 = torch.tensor([1, 0, 0], dtype=torch.float32)
e2 = torch.tensor([0, 1, 0], dtype=torch.float32)
e3 = torch.tensor([0, 0, 1], dtype=torch.float32)

standard_basis = torch.stack([e1, e2, e3], dim=1)
print(f"Standard basis matrix: \n{standard_basis}")

# Express a vector in terms of standard basis
x = torch.tensor([5, 3, -2], dtype=torch.float32)
print(f"\nVector x = {x}")
print(f"x = {x[0]}*e1 + {x[1]}*e2 + {x[2]}*e3")
print(f"x = {x[0]}*{e1} + {x[1]}*{e2} + {x[2]}*{e3}")

# Verify
reconstructed = x[0]*e1 + x[1]*e2 + x[2]*e3
print(f"Reconstructed vector: {reconstructed}")
print(f"Original == Reconstructed? {torch.allclose(x, reconstructed)}")
```

The End