

TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI TP.HCM  
VIỆN CÔNG NGHỆ THÔNG TIN VÀ ĐIỆN, ĐIỆN TỬ

## PHƯƠNG PHÁP TOÁN CHO MÁY HỌC

### CHƯƠNG 1: MA TRẬN - ĐẠI SỐ TUYẾN TÍNH

TS. Trần Thế Vinh

# TỔNG QUAN VỀ PHƯƠNG PHÁP TOÁN CHO MÁY HỌC

**Máy học (Machine Learning, ML)** là một lĩnh vực dựa trên các nguyên tắc toán học để phân tích dữ liệu, tối ưu hóa mô hình và đưa ra dự đoán.

Các phương pháp toán học là nền tảng không thể thiếu trong Machine Learning. Chúng giúp biểu diễn dữ liệu, tối ưu hóa mô hình và cải thiện độ chính xác của các thuật toán. Các công cụ toán học quan trọng trong ML bao gồm đại số tuyến tính, xác suất - thống kê, giải tích ma trận và tối ưu hóa. Việc nắm vững các công cụ toán học này sẽ giúp xây dựng các mô hình mạnh mẽ và tối ưu hơn trong thực tế.

## 1. Đại số tuyến tính và không gian véc tơ

Đại số tuyến tính cung cấp nền tảng để biểu diễn dữ liệu và thực hiện các phép toán trên dữ liệu nhiều chiều. Ma trận và véc tơ giúp mô hình hóa dữ liệu, trong khi các phép toán như nhân ma trận, phân rã ma trận (SVD, Eigen) hỗ trợ trong giảm chiều dữ liệu và trích xuất đặc trưng. Không gian véc tơ và ánh xạ tuyến tính giúp biểu diễn dữ liệu trong các không gian khác nhau để tối ưu hóa mô hình.

## 2. Giải tích ma trận và đạo hàm

Đạo hàm véc tơ và ma trận là công cụ quan trọng để tối ưu hóa trong ML. Các thuật toán tối ưu hóa như Gradient Descent, Backpropagation trong mạng nơ-ron nhân tạo đều dựa trên đạo hàm để điều chỉnh trọng số mô hình. Ma trận Hessian và Jacobian được sử dụng để đánh giá độ hội tụ và tính ổn định của các thuật toán học máy.a

# TỔNG QUAN VỀ PHƯƠNG PHÁP TOÁN CHO MÁY HỌC

## 3. Xác suất và thống kê

Xác suất giúp mô hình hóa sự không chắc chắn trong dữ liệu, trong khi thống kê giúp phân tích và diễn giải dữ liệu. Các mô hình phân phối xác suất (Gaussian, Bernoulli, Poisson) hỗ trợ trong việc xây dựng mô hình dự đoán. Các khái niệm như Entropy, Cross-Entropy, KL Divergence rất quan trọng trong học có giám sát và mô hình Bayesian.

## 4. Tối ưu hóa trong Machine Learning

Tối ưu hóa là chìa khóa để tìm ra mô hình hiệu quả. Các phương pháp tối ưu hóa như Stochastic Gradient Descent (SGD), Adam, Adagrad giúp điều chỉnh trọng số mô hình nhanh chóng và hiệu quả. Các bài toán tối ưu phi tuyến tính xuất hiện trong Deep Learning và Reinforcement Learning giúp cải thiện hiệu suất mô hình.

## 5. Các phương pháp ước lượng và học Bayes

Ước lượng tham số giúp mô hình tổng quát hóa tốt hơn với dữ liệu thực tế. Các phương pháp như Maximum Likelihood Estimation (MLE), Maximum A Posteriori (MAP) giúp xác định tham số mô hình tối ưu. Bayesian Inference và Variational Inference giúp xử lý dữ liệu không chắc chắn và xây dựng mô hình dự đoán hiệu quả hơn.

# MA TRẬN - ĐẠI SỐ TUYẾN TÍNH TRONG ML

Đại số tuyến tính là nền tảng quan trọng trong ML. Các phép toán trên ma trận giúp biểu diễn dữ liệu, mô hình hóa các phép biến đổi và tối ưu hóa trong huấn luyện mô hình.

## Ứng dụng:

- **Phân rã ma trận** (SVD, Eigen Decomposition) được sử dụng trong giảm chiều dữ liệu (PCA), xử lý ảnh và nén dữ liệu.
  - **Hạng ma trận** giúp xác định số lượng thông tin độc lập trong dữ liệu, hỗ trợ giải hệ phương trình tuyến tính.
  - **Định thức và ma trận nghịch đảo** có vai trò quan trọng trong tính toán độ ổn định của hệ phương trình.



# ĐỊNH NGHĨA VỀ MA TRẬN

**Ma trận (Matrix)** là một bảng chữ nhật gồm các phần tử sắp xếp theo hàng và cột. Một ma trận A kích thước m x n có m hàng và n cột:

$$\forall A = (a_{ij})_{m \times n} \in R^{m \times n}$$

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

Mỗi phần tử  $a_{ij}$  đại diện cho giá trị nằm ở hàng thứ i và cột thứ j

Ví dụ:



$$A = \begin{bmatrix} -1 & 0 \\ 3 & -6 \\ 4 & 9 \end{bmatrix}$$

```
import numpy as np

# Nhập kích thước ma trận
m = int(input("Nhập số hàng (m): "))
n = int(input("Nhập số cột (n): "))

# Khởi tạo ma trận rỗng
A = np.zeros((m, n))

# Nhập từng phần tử
for i in range(m):
    for j in range(n):
        A[i, j] = float(input(f"Nhập phần tử A[{i+1}][{j+1}]: "))

print("Ma trận A:")
print(A)
```

bLTUf(A)  
bLTUf(...Ma trận A...)

# ĐỊNH NGHĨA VỀ MA TRẬN

## Ma trận chuyển vị

Ma trận chuyển vị của A, ký hiệu  $A^T$  là ma trận thu được bằng cách đổi hàng thành cột.

$$A^T = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}$$



Nói cách khác, phần tử tại vị trí  $(i,j)$  của A sẽ nằm ở vị trí  $(j,i)$  của  $A^T$

Ví dụ:

$$A^T = \begin{bmatrix} -1 & 3 & 4 \\ 0 & -6 & 9 \end{bmatrix}$$

```
# Ma trận chuyển vị A^T
```

```
A_T = A.T
```

```
print("Ma trận chuyển vị A^T:")
print(A_T)
```

# ĐỊNH NGHĨA VỀ MA TRẬN

```
import numpy as np

# Nhập kích thước ma trận
m = int(input("Nhập số hàng (m): "))
n = int(input("Nhập số cột (n): "))

# Nhập dữ liệu cho ma trận số phức
A = []
print(f"Nhập ma trận {m}x{n}, mỗi hàng nhập các số phức (vd: 1+2j 3-4j):")
for i in range(m):
    row = list(map(complex, input(f"Hàng {i+1}: ").split())))
    A.append(row)

# Chuyển thành mảng numpy
A = np.array(A, dtype=np.complex128)

# Tính ma trận chuyển vị liên hợp
A_H = A.conj().T

print("Ma trận A:")
print(A)
print("\nMa trận chuyển vị liên hợp A^H:")
print(A_H)
```

## Ma trận chuyển vị liên hợp

Ma trận chuyển vị liên hợp (Hermitian transpose), ký hiệu  $A^H$ , là ma trận thu được bằng cách lấy chuyển vị của A và liên hợp phức (conjugate) tất cả các phần tử. Nếu A là ma trận số thực thì  $A^H = A^T$ , nhưng nếu A chứa số phức thì:

- **Liên hợp phức** của một số phức  $a + bi$  là  $a - bi$ .
- **Ma trận chuyển vị liên hợp** của một ma trận số phức A là:

$$B = \begin{bmatrix} -1 + 2i & 0 \\ 3 & 6 - 3i \\ 4 + i & 9 - 4i \end{bmatrix}$$

Thì  $A^H$  là:

$$B^H = \begin{bmatrix} -1 - 2i & 3 & 4 - i \\ 0 & 6 + 3i & 9 + 4i \end{bmatrix}$$



# MỘT SỐ LOẠI MA TRẬN ĐẶC BIỆT

## Ma trận vuông (Square Matrix)

**Định nghĩa:** Ma trận có số hàng bằng số cột ( $m = n$ ).

Ví dụ:

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

**Ứng dụng:** Dùng trong giải hệ phương trình tuyến tính, biến đổi không gian.



```
import numpy as np

# Nhập kích thước ma trận vuông (n x n)
n = int(input("Nhập kích thước ma trận vuông (n):"))

# Nhập từng phần tử từ bàn phím
A = []
print(f"Nhập ma trận {n}x{n}, mỗi hàng nhập các số cách nhau bởi dấu cách:")
for i in range(n):
    row = list(map(float, input(f"Hàng {i+1}: ").split()))
    A.append(row)

# Chuyển thành mảng NumPy
A = np.array(A)

print("\nMa trận vuông A:")
print(A)
```

# MỘT SỐ LOẠI MA TRẬN ĐẶC BIỆT

```
import numpy as np

def row_echelon_form(A):
    A = A.astype(float) # Chuyển về dạng số thực để tránh lỗi chia
    rows, cols = A.shape
    pivot_row = 0

    for col in range(cols):
        # Tìm hàng có phần tử lớn nhất trong cột hiện tại (tránh chia cho 0)
        max_row = pivot_row + np.argmax(np.abs(A[pivot_row:, col]))

        # Nếu phần tử lớn nhất là 0 thì bỏ qua cột này
        if A[max_row, col] == 0:
            continue

        # Đổi hàng hiện tại với hàng có phần tử lớn nhất
        A[[pivot_row, max_row]] = A[[max_row, pivot_row]]

        # Biến đổi hàng sao cho phần tử tru bằng 1
        A[pivot_row] = A[pivot_row] / A[pivot_row, col]

        # Khử các phần tử bên dưới phần tử tru
        for row in range(pivot_row + 1, rows):
            A[row] -= A[row, col] * A[pivot_row]

        pivot_row += 1
        if pivot_row == rows:
            break

    return A

# Nhập ma trận từ bàn phím
m = int(input("Nhập số hàng (m): "))
n = int(input("Nhập số cột (n): "))

A = []
print(f"Nhập ma trận {m}x{n}, mỗi hàng cách nhau bởi dấu cách:")
for i in range(m):
    row = list(map(float, input(f"Hàng {i+1}: ").split()))
    A.append(row)

A = np.array(A)
A_ref = row_echelon_form(A.copy())

print("\nMa trận bậc thang (Row Echelon Form):")
print(A_ref)
```

## Ma trận bậc thang

Ma trận bậc thang là ma trận mà các phần tử bên dưới đường chéo chính đều là 0 theo quy tắc sau:

- Hàng toàn số 0 (nếu có) phải nằm dưới cùng
- Với 2 hàng khác 0 bất kỳ kè nhau, phần tử khác 0 đầu tiên của hàng trên phải nằm về bên trái cột chứa phần tử khác 0 đầu tiên của hàng dưới.

Ví dụ:

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

$$C = \begin{bmatrix} 2 & 0 & 1 \\ 0 & -1 & 9 \\ 0 & 0 & 4 \\ 0 & 0 & 0 \end{bmatrix}; D = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 5 \end{bmatrix}$$

# MỘT SỐ LOẠI MA TRẬN ĐẶC BIỆT

## Ma trận tam giác (Triangular Matrix)

**Định nghĩa:**

- Ma trận tam giác trên:** Các phần tử dưới đường chéo chính đều bằng 0.

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

- Ma trận tam giác dưới:** Các phần tử trên đường chéo chính đều bằng 0.

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

**Ứng dụng:**

- Giải hệ phương trình tuyến tính nhanh hơn với **phân rã LU**.
- Tính toán định thức dễ dàng hơn.



```
import numpy as np

# Nhập kích thước ma trận vuông
n = int(input("Nhập kích thước ma trận vuông (n): "))

# Tạo ma trận ngẫu nhiên
A = np.random.randint(1, 10, (n, n))

# Chuyển thành ma trận tam giác trên
upper_triangular = np.triu(A)
#Chuyển thành ma trận tam giác dưới
lower_triangular = np.tril(A)

print("\nMa trận ban đầu:")
print(A)

print("\nMa trận tam giác trên:")
print(upper_triangular)

print("\nMa trận tam giác dưới:")
print(lower_triangular)
```

# MỘT SỐ LOẠI MA TRẬN ĐẶC BIỆT

## Ma trận xác định dương (Positive Definite Matrix) & bán xác định dương

- Định nghĩa: Cho ma trận vuông A đối xứng, với mọi vector  $x \neq 0$ , thoả mãn điều kiện:
- Ma trận xác định dương:

$$x^T Ax > 0$$

### Ma trận bán xác định dương:

$$x^T Ax \geq 0$$

Ví dụ:

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

Với  $\forall$  vector  $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ , ta có:

$$x^T Ax = 2x_1^2 - 2x_1x_2 + 2x_2^2 > 0$$

### Ứng dụng:

- Xác minh tính lồi của hàm loss trong Machine Learning. Điều này giúp tối ưu gradient descent hội tụ nhanh hơn khi huấn luyện mô hình.
- Áp dụng trong **hồi quy Ridge và Lasso**. Giúp đảm bảo nghiệm duy nhất của phương trình tối ưu. Tránh vấn đề đa cộng tuyến, giúp mô hình ổn định hơn.

```
import numpy as np

def generate_positive_definite_matrix(n):
    B = np.random.rand(n, n) # Ma trận ngẫu nhiên
    A = np.dot(B.T, B) # Nhân với chuyển vị để tạo ma trận xác định dương
    return A

# Nhập kích thước ma trận
n = int(input("Nhập kích thước ma trận vuông (n): "))

# Tạo ma trận xác định dương
A = generate_positive_definite_matrix(n)

print("\nMa trận xác định dương:")
print(A)
```

# CÁC PHÉP TOÁN TRÊN MA TRẬN

## Cộng và nhân ma trận:

Công hai ma trận cùng kích thước bằng cách cộng từng phần tử tương ứng.

## Ví dụ:

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

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

**Nhân hai ma trận**  $A_{m \times p}$  và  $B_{p \times n}$  có công thức:

$$A = (a_{ij})_{m \times p}; B = (b_{ij})_{p \times n} \quad A \cdot B = (c_{ij})_{m \times n};$$

trong đó:  $c_{ij} = \sum_{k=1}^p a_{ik} \cdot b_{kj}; \forall i, j$

(A)

(B)

$$hang\ i \rightarrow \begin{bmatrix} \dots & \dots & \dots \end{bmatrix} \cdot \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix} \quad (cot\ j)$$

## Chú ý:

- Phép nhân 2 ma trận có kề đến thứ tự các nhân tử và được thực hiện khi và chỉ khi số cột của nhân tử trước bằng số hàng của nhân tử sau.
  - Có thể tồn tại  $A \cdot B$  mà không tồn tại  $B \cdot A$  và nếu tồn tại cả hai tích thì nói chung:  $A \cdot B \neq B \cdot A$ .
  - Với ma trận vuông  $A$  thì tích của  $k$  nhân tử  $A$  được viết:  $A \cdot A \cdots A = A^n$

```
import numpy as np
# Nhập kích thước ma trận
m = int(input("Nhập số hàng của ma trận A: "))
n = int(input("Nhập số cột của ma trận A (cũng là số hàng của B): "))
p = int(input("Nhập số cột của ma trận B: "))

# Nhập ma trận A (m x n)
print("\nNhập ma trận A:")
A = np.array([[int(input(f"A[{i}][{j}]: ")) for j in range(n)] for i in range(m)])
print(A)
# Nhập ma trận B (n x p)
print("\nNhập ma trận B:")
B = np.array([[int(input(f"B[{i}][{j}]: ")) for j in range(p)] for i in range(n)])
print(B)
# Nhân hai ma trận
C = np.dot(A, B) # Hoặc dùng A @ B

print("\nMa trận tích C = A x B:")
print(C)
```

# CÁC PHÉP TOÁN TRÊN MA TRẬN

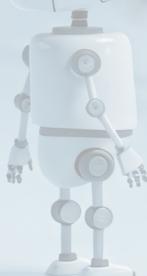
Ví dụ:

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

$$C = A \cdot B = \begin{bmatrix} 1 * 2 + 2 * 3 & 1 * 3 + 2 * 1 & 1 * 1 + 2 * 4 \\ 3 * 2 + 4 * 3 & 3 * 3 + 4 * 1 & 3 * 1 + 4 * 4 \\ 2 * 2 + 3 * 3 & 2 * 3 + 3 * 1 & 2 * 1 + 3 * 4 \end{bmatrix} = \begin{bmatrix} 8 & 5 & 9 \\ 18 & 13 & 19 \\ 13 & 9 & 14 \end{bmatrix}$$

Ứng dụng:

- Trong mạng Neural Network, các trọng số và đầu vào được nhân ma trận.
- Biểu diễn phép biến đổi ảnh (Image Transformation).
- Hồi quy tuyến tính: Tính dự đoán bằng  $y=Xb$



# MA TRẬN NGHỊCH ĐẢO

Định nghĩa: Ma trận nghịch đảo của ma trận vuông A bậc n (ký hiệu  $A^{-1}$ ) là một ma trận sao cho:

$$A^{-1} \cdot A = A \cdot A^{-1} = I_n$$

Với  $I_n$  là ma trận đơn vị bậc n

## Điều kiện tồn tại

- Ma trận A phải là ma trận vuông (nxn).
- Định thức của A khác 0 ( $\det(A) \neq 0$ )

## Ứng dụng:

- Giải hệ phương trình tuyến tính  $Ax = b \Rightarrow x = A^{-1}b$
- Tính toán tối ưu trong hồi quy tuyến tính:
- Công thức OLS:  $\theta = (X^T \cdot X)^{-1} \cdot X^T y$  trong đó X là ma trận dữ liệu, y là vector đầu ra.

```
import numpy as np

# Nhập kích thước ma trận vuông
n = int(input("Nhập kích thước ma trận vuông (n): "))

# Nhập ma trận A
print("\nNhập ma trận A:")
A = np.array([[float(input(f"A[{i}][{j}]: ")) for j in range(n)] for i in range(n)])

# Kiểm tra định thức
det_A = np.linalg.det(A)
if det_A == 0:
    print("\n[X] Ma trận không khả nghịch vì định thức = 0.")
else:
    # Tính nghịch đảo
    A_inv = np.linalg.inv(A)
    print("\n[✓] Ma trận nghịch đảo của A là:")
    print(A_inv)

    # Kiểm tra: A * A^-1 có ra ma trận đơn vị không?
    print("\n[Kiểm tra A * A^-1:]")
    print(np.round(np.dot(A, A_inv), 5)) # Làm tròn để dễ nhìn
```

```
b11=1.0; b21=0.0;b31=0.0;A=[[1,2,3],[4,5,6],[7,8,9]]# Tạo ma trận A
b12=0.0; b22=1.0;b32=0.0;A1=[[1,0,0],[0,1,0],[0,0,1]]# Tạo ma trận đơn vị I3
b13=0.0; b23=0.0;b33=1.0;A2=[[1,0,0],[0,1,0],[0,0,1]]# Tạo ma trận đơn vị I3
```

# MA TRẬN TRỰC GIAO (ORTHOGONAL MATRIX)

**Định nghĩa:** Một ma trận A có các cột trực giao với nhau và thỏa mãn:

$$A^T A = I$$

với  $I$  là ma trận đơn vị.

Ví dụ:

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

Ma trận trực giao là trường hợp đặc biệt của ma trận nghịch đảo khi  $A^T = A^{-1}$ . Nghĩa là ma trận nghịch đảo của nó cũng chính là ma trận chuyển vị.

## Ứng dụng:

- Dùng trong đồ họa máy tính để xoay hình ảnh.
- Giữ lại thông tin quan trọng của dữ liệu, loại bỏ nhiễu dữ liệu, giảm chiều dữ liệu bằng **Phân rã Giá trị Kì dị (SVD)** (gồm 2 ma trận trực giao và ma trận đường chéo chứa các giá trị kì dị)

```
def random_orthogonal_matrix(n):
    """
    Sinh một ma trận trực giao ngẫu nhiên kích thước (n x n).

    :param n: Kích thước của ma trận
    :return: Ma trận trực giao (NumPy array)
    """
    random_matrix = np.random.rand(n, n) # Tạo ma trận ngẫu nhiên
    Q, _ = np.linalg.qr(random_matrix) # Phân rã QR để lấy Q (ma trận trực giao)
    return Q

# Tạo ma trận trực giao 3x3
Q = random_orthogonal_matrix(3)
print("Ma trận trực giao:\n", Q)
```

# CÁC TÍNH CHẤT CỦA CÁC PHÉP TOÁN VỀ MA TRẬN

Phép cộng có tính chất giao hoán, kết hợp,  $A + (-A) = 0$ ;  $A + 0 = A$

Với hai ma trận  $A, B$ , hai số thực  $a, b$  thì:

- ❖  $a.(A + B) = a.A + a.B; (a + b).A = a.A + b.A$
- ❖  $(a.A)^T = a.A^T; (A + B)^T = A^T + B^T;$
- ❖  $(A.B)^T = B^T.A^T; (A^T)^T = A$
- ❖  $0.A = \mathcal{O}; A.\mathcal{O} = \mathcal{O}; \mathcal{O}.A = \mathcal{O}; 1.A = A; (-1).A = -A;$
- ❖  $A_{m \times n} \cdot I_n = A_{m \times n}; I_m \cdot A_{m \times n} = A_{m \times n}$
- ❖  $(A + B).C = A.C + B.C; C(A + B) = C.A + C.B$
- ❖ Nếu  $A, B$  là các ma trận vuông cùng cấp, khả nghịch thì  $A.B$  cũng khả nghịch và:  $(A.B)^{-1} = B^{-1} \cdot A^{-1}$



# ĐỊNH THỨC CỦA MA TRẬN VUÔNG

Định thức của một **ma trận vuông** là một giá trị **vô hướng** đặc trưng cho ma trận đó, ký hiệu là  $\det(A)$  hoặc  $|A|$ . Nó giúp xác định tính chất của ma trận như **khả nghịch** (có ma trận nghịch đảo hay không), **hạng của ma trận**, và **nghiệm của hệ phương trình tuyến tính**.

- **Với  $A = (a)$  (ma trận vuông cấp 1) thì  $\det(A) = a$**

- **Với  $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ :**

$$\det(A) = a_{11}a_{22} - a_{21}a_{12}$$

- **Với  $A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$ :**


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

# ĐỊNH THỨC MA TRẬN VUÔNG CẤP N

Triển khai Laplace:

Chọn triển khai theo hàng thứ i:

$$\det(A) = \sum_{j=1}^n (-1)^{i+j} a_{ij} M_{ij}$$

Chọn triển khai theo cột thứ j:

$$\det(A) = \sum_{i=1}^n (-1)^{i+j} a_{ij} M_{ij}$$

Trong đó:

- $M_{ij}$  là định thức của ma trận con thu được bằng cách loại bỏ hàng thứ i và cột thứ j của A.
- $(-1)^{i+j}$  là hệ số dấu giúp bảo đảm tính chính xác.

```
import numpy as np

def determinant_laplace(matrix):
    """
    Tính định thức của ma trận vuông bằng phương pháp khai triển Laplace (đệ quy).

    :param matrix: Ma trận vuông (numpy array hoặc list lồng nhau)
    :return: Giá trị định thức của ma trận
    """
    matrix = np.array(matrix)
    n = matrix.shape[0]

    # Điều kiện dừng: Ma trận 1x1
    if n == 1:
        return matrix[0, 0]

    # Điều kiện dừng: Ma trận 2x2
    if n == 2:
        return matrix[0, 0] * matrix[1, 1] - matrix[0, 1] * matrix[1, 0]

    det = 0
    for j in range(n): # Duyệt theo từng phần tử của hàng đầu tiên
        sub_matrix = np.delete(np.delete(matrix, 0, axis=0), j, axis=1) # Ma trận con
        cofactor = (-1) ** j * matrix[0, j] * determinant_laplace(sub_matrix)
        det += cofactor

    return det

# Nhập ma trận từ người dùng
def input_matrix():
    n = int(input("Nhập kích thước ma trận vuông n x n: "))
    matrix = []
    print("Nhập từng dòng của ma trận, cách nhau bởi dấu cách:")
    for i in range(n):
        row = list(map(float, input(f"Dòng {i+1}: ").split()))
        matrix.append(row)
    return np.array(matrix)

# Chạy chương trình
matrix = input_matrix()
det = determinant_laplace(matrix)
print("Định thức của ma trận là:", det)
```

# ĐỊNH THỨC MA TRẬN VUÔNG CẤP N

Chọn triển khai theo hàng thứ 1:

$$\det(A) = a_{11} \cdot A_{11} + a_{12} \cdot A_{12} + \dots + a_{1n} \cdot A_{1n}$$

Các tính chất của định thức: Giả sử  $A$  là ma trận vuông cấp  $n$

1.  $\det(A) = a_{i1} \cdot A_{i1} + a_{i2} \cdot A_{i2} + \dots + a_{in} \cdot A_{in}, \forall i = \overline{1, n}$  (1.2a)

$$= a_{1j} A_{1j} a_{2j} A_{2j} a_{nj} A_{nj}, \forall j = \overline{1, n} \quad (1.2b)$$

và:  $a_{i1} \cdot A_{k1} + \dots + a_{in} \cdot A_{kn} = a_{1j} A_{1k} a_{2j} A_{2k} a_{nj} A_{nk} = 0, \forall i \neq k \neq j$

2. Nếu  $A$  có 1 hàng 0 (cột 0) thì:  $\det(A) = 0$

3.  $\det(A^T) = \det(A)$

4. Nếu đổi chỗ 2 hàng (2 cột) của  $m$ .trận thì định thức chỉ đổi dấu. Đặc biệt nếu  $A$  có 2 hàng (2 cột) giống nhau thì  $\det(A) = 0$

5. Nếu nhân 1 hàng (cột) của  $A$  với số  $r$  thì định thức được nhân lên với  $r$ . Đặc biệt:  $\det(\alpha \cdot A) = \alpha^n \cdot \det(A)$

6. Nếu  $A$  có 2 hàng (2 cột) tương ứng tỉ lệ thì:  $\det(A) = 0$

7. Nếu  $A$  có 1 hàng (1 cột) mà mỗi phần tử là tổng của 2 số hạng thì:  $\det(A) = \det(A_1) + \det(A_2)$ , ( $A_1$  và  $A_2$  là các  $m$ .trận thu được từ  $A$  bằng cách giữ nguyên các hàng (cột) khác, còn hàng (cột) nói trên chỉ giữ lại số hạng thứ nhất và thứ 2 tương ứng). Minh họa:

$$\begin{vmatrix} a_{11} & b_{12} + c_{12} & a_{13} \\ a_{21} & b_{22} + c_{22} & a_{23} \\ a_{31} & b_{32} + c_{32} & a_{33} \end{vmatrix} = \begin{vmatrix} a_{11} & b_{12} & a_{13} \\ a_{21} & b_{22} & a_{23} \\ a_{31} & b_{32} & a_{33} \end{vmatrix} + \begin{vmatrix} a_{11} & c_{12} & a_{13} \\ a_{21} & c_{22} & a_{23} \\ a_{31} & c_{32} & a_{33} \end{vmatrix}$$

# ĐỊNH THỨC MA TRẬN VUÔNG CẤP N

8. Nếu  $A$  có 1 hàng (cột) là tổ hợp tuyến tính của các hàng (cột) khác thì:  $\det A = 0$ . Vậy khi nhân 1 hàng (cột) với số  $t$  bất kì rồi cộng vào 1 hàng (cột) khác thì  $\det$  không đổi.
9. Nếu  $A$  là  $m \times n$  tam giác thì:  $\det(A) = a_{11} \cdot a_{22} \cdots a_{nn}$

**Sơ đồ tính định thức của ma trận vuông  $A$ :**

- (1) Nếu  $A$  là ma trận tam giác thì  $\det(A) = a_{11} \cdot a_{22} \cdots a_{nn}$
- (2) Nếu  $A$  có dấu hiệu của t/c 2, 6, 8 thì  $\det A = 0$
- (3) Nếu không phải (1), (2) thì dùng các phép biến đổi không làm thay đổi định thức để đưa ma trận  $A$  về ma trận tam giác  $A^*$
- Suy ra  $\det A = \det A^*$

10.  $\det(A \cdot B) = \det(A) \cdot \det(B)$ .

Do đó:  $\det(A^m) = \{\det(A)\}^m$

11. Điều kiện cần và đủ để  $m$ .trận  $A$  khả nghịch là  $\det(A) \neq 0$ . Khi đó:

$$A^{-1} = \frac{1}{\det(A)} \cdot \begin{pmatrix} A_{11} & A_{21} & \dots & A_{n1} \\ A_{12} & A_{22} & \dots & A_{n2} \\ \vdots & \vdots & \dots & \vdots \\ A_{1n} & A_{2n} & \dots & A_{nn} \end{pmatrix}$$

với:  $A_{ij}$  là phần phụ đại số của phần tử  $a_{ij}$  trong  $m$ .trận  $A$ .



## ĐỊNH THỨC MA TRẬN VUÔNG CẤP N

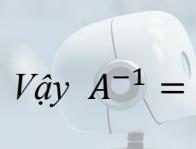
VD: Khảo sát tính khả nghịch và tìm ma trận nghịch đảo (nếu có) của

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

Giải: Có  $\det(A) = 2 \neq 0$ , nên  $A$  khả nghịch,  $A^{-1}$  tìm theo công thức:

$$A^{-1} = \frac{1}{\det A} \begin{bmatrix} A_{11} & A_{21} & A_{31} \\ A_{12} & A_{22} & A_{32} \\ A_{13} & A_{23} & A_{33} \end{bmatrix}$$

Có  $A_{11} = -3, A_{12} = 2, A_{13} = -1, A_{21} = -3, A_{22} = 2, A_{23} = 1, A_{31} = 4, A_{32} = -2, A_{33} = 0$ .


$$\text{Vậy } A^{-1} = \frac{1}{2} \begin{bmatrix} -3 & -3 & 4 \\ 2 & 2 & -2 \\ -1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} -3/2 & -3/2 & 2 \\ 1 & 1 & -1 \\ -1/2 & 1/2 & 0 \end{bmatrix}$$

### Ứng dụng:

- Kiểm tra tính khả nghịch của ma trận
- Giúp tối ưu trong DL: Sử dụng để xác định Jacobian trong lan truyền ngược (Backpropagation)
- Ứng dụng trong PCA: Định thức liên quan đến phép biến đổi tuyến tính giúp giảm chiều dữ liệu.
- Áp dụng trong tính toán giá trị riêng của ma trận trong ML.

# HẠNG CỦA MA TRẬN

- Hạng của ma trận A ( $\text{Rank}(A)$ ) là số lượng hàng hoặc cột độc lập tuyến tính.
- Nếu một hàng/cột có thể biểu diễn bằng tổng hợp tuyến tính của các hàng/cột khác thì nó không đóng góp vào hạng.
- Hạng luôn nhỏ hơn hoặc bằng số hàng và số cột của ma trận:

$$\text{rank}(A) \leq \min\{m, n\}.$$

## Phương pháp tìm hạng của ma trận:

1. Các phép biến đổi sơ cấp không làm thay đổi hạng của ma trận:
  - Đổi chỗ 2 hàng(hoặc 2 cột)
  - Nhân một hàng(hoặc 1 cột) với một số khác 0
  - Cộng một bội số của một hàng(hoặc một cột) vào một hàng(hoặc một cột) khác.
2. Đưa ma trận về dạng bậc thang, sau đó đếm số hàng khác 0 để tìm hạng ma trận ( $\text{rank}(A)$ ).



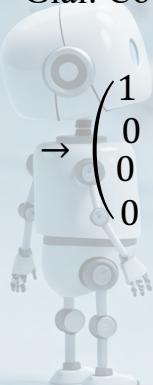
```
import numpy as np
def input_matrix():
    n, m = map(int, input("Nhập số hàng và số cột của ma trận (cách nhau bởi dấu cách): ").split())
    print("Nhập từng dòng của ma trận, cách nhau bởi dấu cách:")
    matrix = []
    for i in range(n):
        row = list(map(float, input(f"Dòng {i+1}: ").split()))
        matrix.append(row)
    return np.array(matrix)

# Nhập ma trận và tính hạng
A = input_matrix()
rank = np.linalg.matrix_rank(A)
print("Hạng của ma trận:", rank)
```

## HẠNG CỦA MA TRẬN

Vd 2: Tìm hạng của ma trận:  $\begin{pmatrix} 1 & 2 & 3 & 1 & 2 \\ 2 & 3 & 1 & 2 & 3 \\ 3 & 1 & 2 & 3 & 1 \\ 3 & 5 & 4 & 3 & 5 \end{pmatrix}$

Giải: Có sơ đồ biến đổi:  $A = \begin{pmatrix} 1 & 2 & 3 & 1 & 2 \\ 2 & 3 & 1 & 2 & 3 \\ 3 & 1 & 2 & 3 & 1 \\ 3 & 5 & 4 & 3 & 5 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 2 & 3 & 1 & 2 \\ 0 & -1 & -5 & 0 & -1 \\ 0 & -4 & -2 & 0 & -4 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$   
 $\rightarrow \begin{pmatrix} 1 & 2 & 3 & 1 & 2 \\ 0 & -1 & -5 & 0 & -1 \\ 0 & 0 & 18 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} = A^*$  (bậc thang). Vậy  $r(A) = r(A^*) = 3$



# HẠNG CỦA MA TRẬN

## Các tính chất:

- $\text{rank}(A \cdot B) \leq \min\{\text{rank}(A), \text{rank}(B)\}$
- $\text{rank}(A + B) \leq \text{rank}(A) + \text{rank}(B)$
- $\forall A \in \mathbb{R}^{m+n}, \forall B \in \mathbb{R}^{n+k}, \text{thì: } \text{rank}(A) + \text{rank}(B) - n \leq \text{rank}(A \cdot B)$
- Nếu A là m.trận vuông cấp n thì:  
A khả nghịch  $\Leftrightarrow \det(A) \neq 0 \Leftrightarrow \text{rank}(A) = n$

Ví dụ:

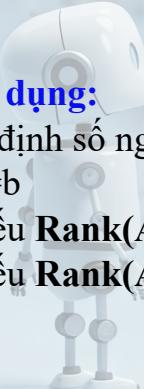
$$A = \begin{bmatrix} 1 & 2 & -1 \\ 1 & 2 & 1 \\ 2 & 3 & 0 \end{bmatrix}, \text{ có } \det(A) = 2 \neq 0, \text{ suy ra rank}(A) = 3$$

## Ứng dụng:

Xác định số nghiệm của hệ phương trình tuyến tính:

$$AX=b$$

- Nếu  $\text{Rank}(A) = \text{Rank}([A|b])$ , hệ có **nghiệm duy nhất hoặc vô số nghiệm**.
- Nếu  $\text{Rank}(A) < \text{Rank}([A|b])$ , hệ **vô nghiệm**.



# THUẬT TOÁN GAUSS-JORDAN TÌM MA TRẬN NGHỊCH ĐẢO

**Phương pháp Gauss-Jordan** dùng phép biến đổi hàng để tìm nghịch đảo. Các bước:

1. Gộp ma trận A với ma trận đơn vị  $I$ :

$$[A|I]$$

2. Biến đổi hàng để đưa A về dạng bậc thang rút gọn I:

$$[I|A^{-1}]$$

3. Khi  $A$  trở thành  $I$ , phần bên phải chính là  $A^{-1}$

## Ứng dụng:

- Giải hệ phương trình tuyến tính
- Tính ma trận hiệp phương sai trong thống kê
- Dự báo dữ liệu bằng mô hình ARIMA.

```
import numpy as np

def gauss_jordan_inverse(matrix):
    """
    Tìm ma trận nghịch đảo bằng phương pháp Gauss-Jordan.

    :param matrix: Ma trận vuông (numpy array)
    :return: Ma trận nghịch đảo nếu tồn tại, ngược lại báo lỗi
    """
    matrix = np.array(matrix, dtype=float)
    n = matrix.shape[0]

    # Tạo ma trận mở rộng [A | I]
    augmented_matrix = np.hstack((matrix, np.eye(n)))

    # Áp dụng phép biến đổi hàng
    for i in range(n):
        # Tim phần tử chính (pivot)
        pivot = augmented_matrix[i, i]
        if abs(pivot) < 1e-10: # Kiểm tra nếu pivot quá nhỏ hoặc bằng 0
            raise ValueError("Ma trận không khả nghịch!")
        
        # Chia hàng i cho pivot để pivot trở thành 1
        augmented_matrix[i] = augmented_matrix[i] / pivot

        # Dùng hàng i để khử các phần tử khác trong cột
        for j in range(n):
            if i != j:
                factor = augmented_matrix[j, i]
                augmented_matrix[j] -= factor * augmented_matrix[i]

    # Phần bên phải là ma trận nghịch đảo
    inverse_matrix = augmented_matrix[:, n:]
    return inverse_matrix

# Nhập ma trận từ người dùng
def input_matrix():
    n = int(input("Nhập kích thước ma trận vuông n x n: "))
    matrix = []
    print("Nhập từng dòng của ma trận, cách nhau bởi dấu cách:")
    for i in range(n):
        row = list(map(float, input(f"Dòng {i+1}: ").split()))
        matrix.append(row)
    return np.array(matrix)

# Chạy chương trình
try:
    A = input_matrix()
    A_inv = gauss_jordan_inverse(A)
    print("Ma trận nghịch đảo của A là:\n", A_inv)
except ValueError as e:
    print(e)
```

# THUẬT TOÁN GAUSS-JORDAN TÌM MA TRẬN NGHỊCH ĐẢO

Ví dụ 1: Khảo sát tính khả nghịch và tìm m.ma trận nghịch đảo (nếu có) của:

$$A = \begin{bmatrix} -1 & 0 & 1 & -1 \\ 2 & 1 & -3 & 4 \\ 1 & 2 & -2 & 8 \\ -2 & 0 & 2 & 1 \end{bmatrix}; B = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 3 & 5 \\ 3 & 3 & 4 & 5 \\ 1 & 1 & 1 & 2 \end{bmatrix};$$

Ví dụ 2: Với ma trận A, B trong ví dụ 1, hãy tìm ma trận X sao cho:  $(X - A).A^T = B$

Ví dụ 3: Cho A là ma trận đường chéo. Tìm  $A^k$

Ví dụ 4: Tìm điều kiện cho m để m.ma trận  $A = \begin{bmatrix} m+1 & 1 & 3 \\ 2 & m+2 & 0 \\ 2m & 1 & 3 \end{bmatrix}$  khả nghịch

