

# 게임 수학 – 강의 7

## 행렬식과 역행렬

동명대학교 게임공학과  
강영민

# 행렬식

- 정사각 행렬에서 정의됨

- 행렬식의 표현

- 행렬  $A$ 의 행렬식 =  $\det A$  혹은  $|A|$

- 행렬식을 계산하기 위해 필요한 개념

- 소행렬식<sup>minor</sup>

- 여인자<sup>cofactor</sup>

- 소행렬식

- $A \in \mathbb{R}^{m \times n}$ : 이 행렬은  $m \times n$ 개의 소행렬식(minor)  $M_{ij}$ 를 가짐
  - 각  $M_{ij}$ 는  $A$  행렬의  $i$  행 벡터 전체와  $j$  열 벡터 전체를 제거하고 얻어지는 행렬( $\in \mathbb{R}^{m-1 \times n-1}$ )의 행렬식

- 여인자

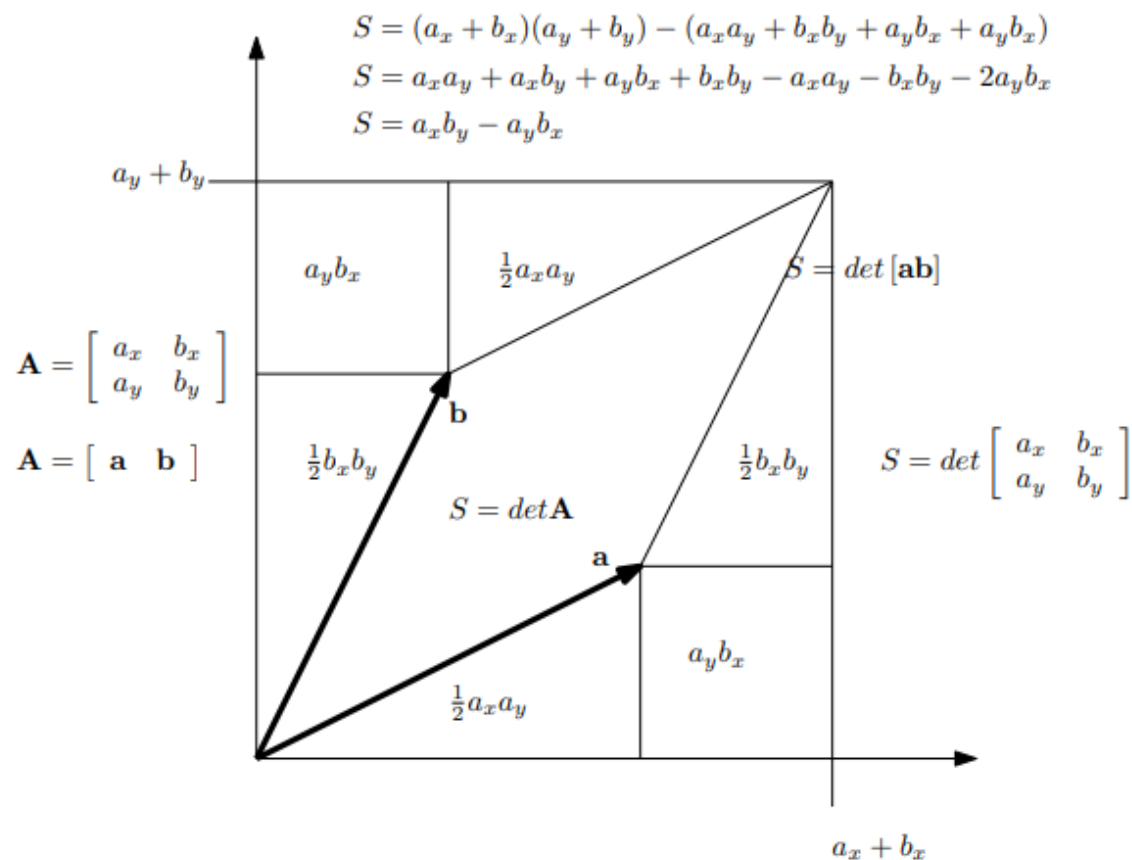
- 행렬  $A$ 의 여인자는 소행렬식이 구해지는 위치마다 결정
  - 다음과 같이 정의되는  $m \times n$ 개의 여인자  $C_{ij}$ 가 존재
  - $C_{ij} = (-1)^{i+j} M_{ij}$

# 행렬식의 기하적 의미

두 열 벡터  $\mathbf{a} = (a_x a_y)^T$ 와  $\mathbf{b} = (b_x, b_y)^T$ 를 열로 하는 행렬  $\mathbf{A}$

$$\mathbf{A} = \begin{bmatrix} a_x & b_x \\ a_y & b_y \end{bmatrix}$$

이 두 벡터를 두 개의 변으로 하는 평행사변형의 넓이가 행렬  $\mathbf{A}$ 의 행렬식

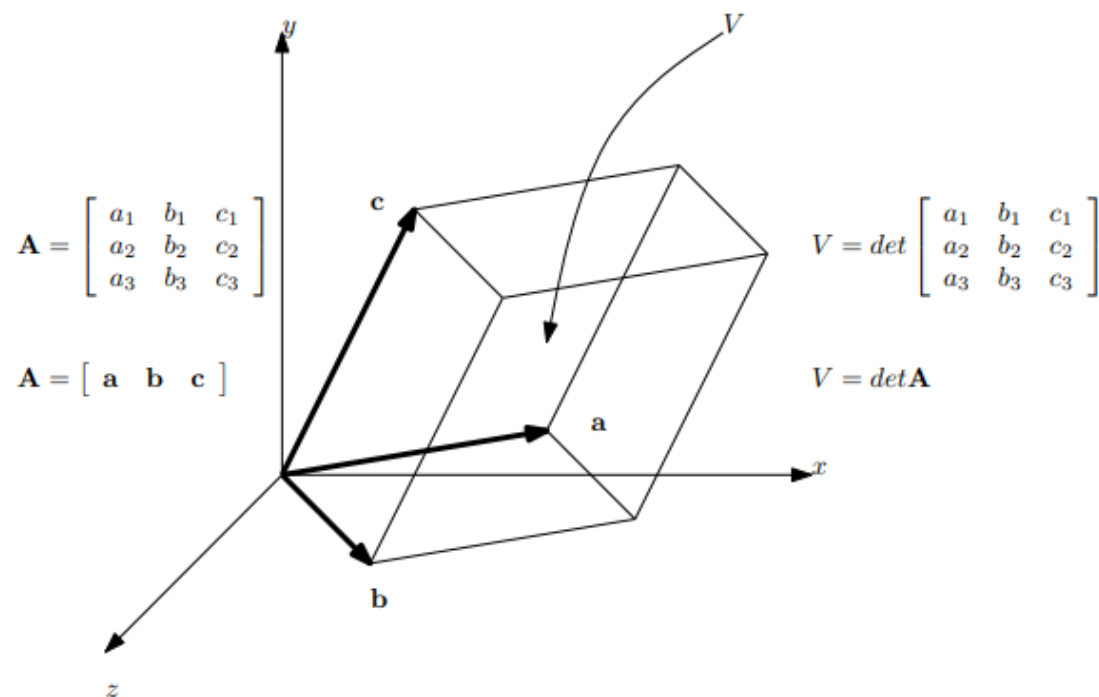


## 3x3 행렬의 기하적 이해

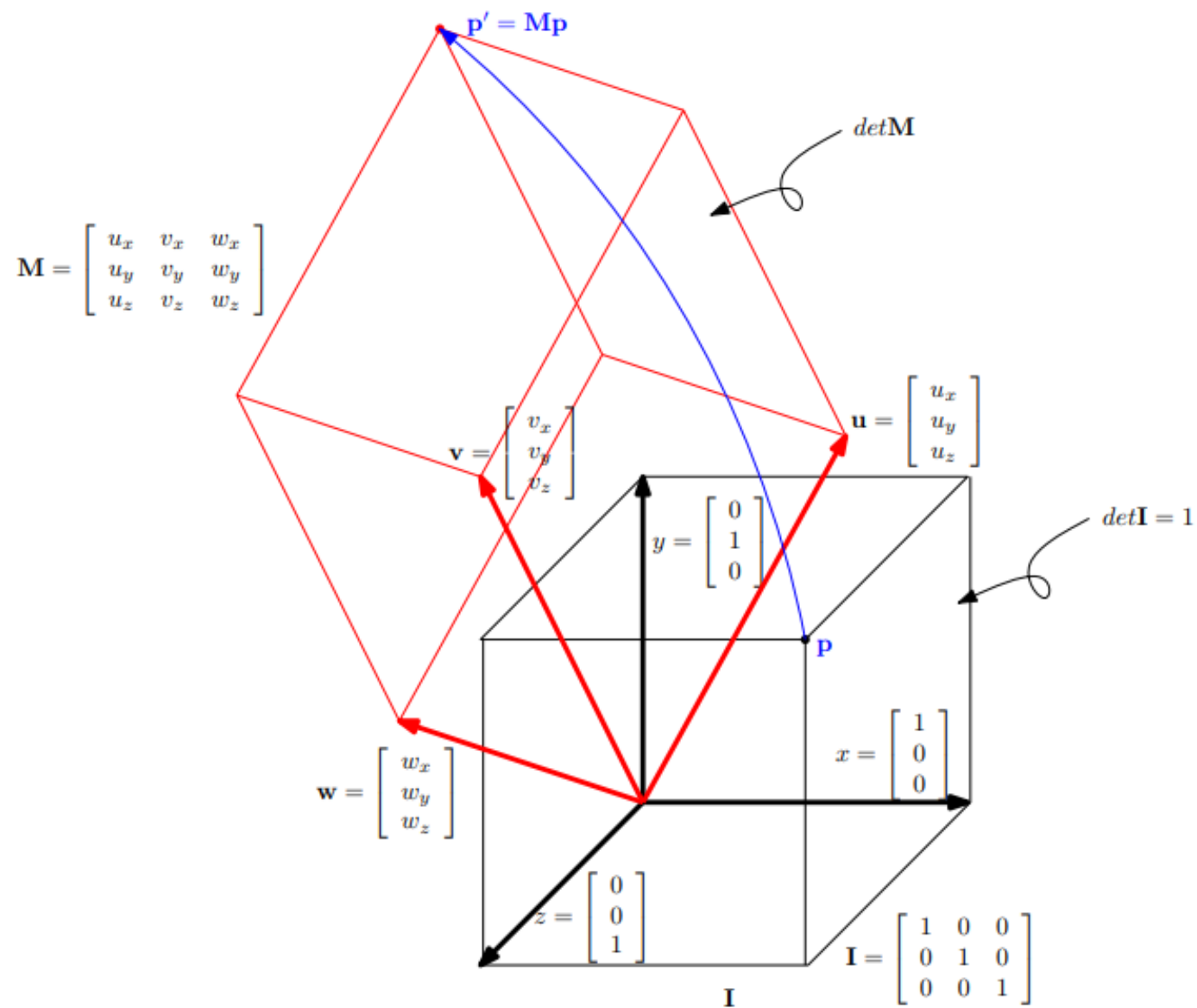
$\mathbf{A} \in \mathbb{R}^{3 \times 3}$ 는 세 개의 벡터  $\mathbf{a}, \mathbf{b}, \mathbf{c}$ 를 포함

$$\mathbf{A} = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} = [\mathbf{a} \quad \mathbf{b} \quad \mathbf{c}]$$

이 세 개의 벡터들이 만드는 평행육면체의 크기가 세 개의 벡터들로 구성된 행렬의 행렬식



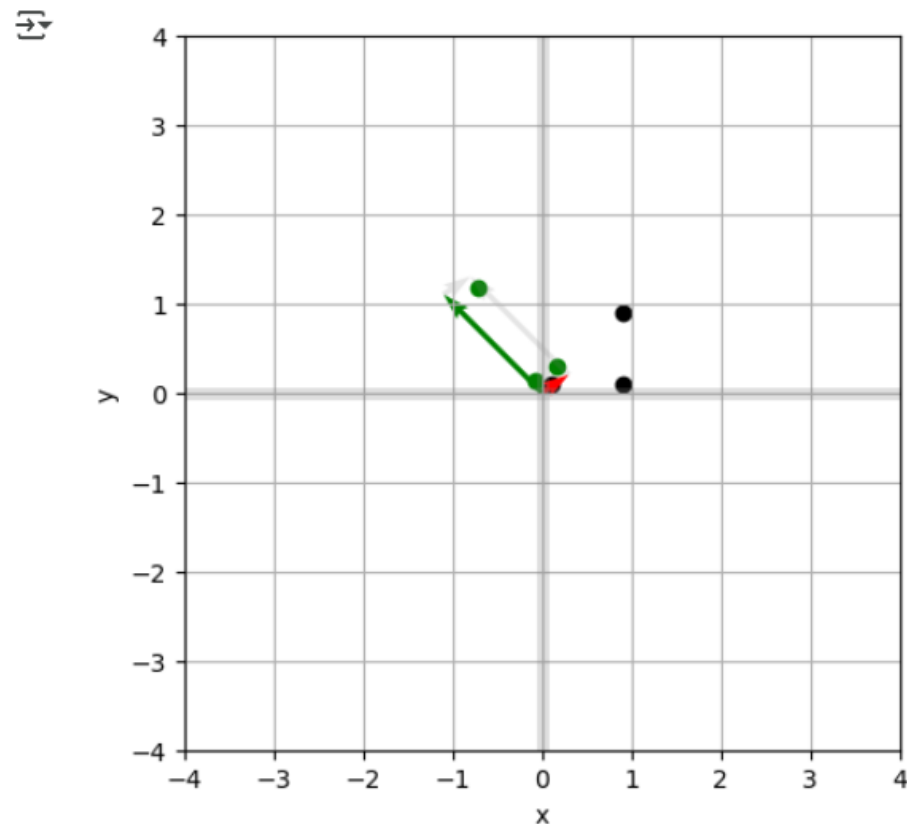
## 변환으로서의 3x3 행렬과 행렬식의 이해



# 행렬식의 기하적 의미

변형된 공간이 원래의 공간과 비교할 때  
어느 정도로 확대/축소 되었나

```
Space2D = axis2d(x=[-4, 4], y=[-4, 4])
M = np.array([
    [0.3, -1.1],
    [0.2, 1.1]
])
points = np.array([
    [0.1, 0.9, 0.9],
    [0.1, 0.1, 0.9]
])
transformed = M @ points
draw_points_in_matrix(Space2D, points, color='black')
draw_points_in_matrix(Space2D, transformed, color='green')
draw_mat22(Space2D, M)
```



# 행렬식의 기하적 의미

```
▶ p0 = points[:, 0] # point(*, 0)
  p1 = points[:, 1]
  p2 = points[:, 2]
  p0, p1, p2
```

```
↗ (array([0.1, 0.1]), array([0.9, 0.1]), array([0.9, 0.9]))
```

```
[ ] def area(p0, p1, p2):
    u = p1-p0
    v = p2-p0
    uxv = np.cross(u, v)
    return 0.5 * np.linalg.norm(uxv) # area of the triangle
```

```
[ ] original = area(p0, p1, p2)
  original
```

```
↗ <ipython-input-5-ded147881eec>:4: DeprecationWarning: Arrays of 2-dimensional vectors are deprecated. Use arrays of 3-dimensional vectors instead. (deprecated in NumPy 2.0)
  uxv = np.cross(u, v)
  np.float64(0.32000000000000006)
```

```
[ ] p0 = transformed[:, 0] # point(*, 0)
  p1 = transformed[:, 1]
  p2 = transformed[:, 2]
  transformed_area = area(p0, p1, p2)
  transformed_area
```

```
↗ <ipython-input-5-ded147881eec>:4: DeprecationWarning: Arrays of 2-dimensional vectors are deprecated. Use arrays of 3-dimensional vectors instead. (deprecated in NumPy 2.0)
  uxv = np.cross(u, v)
  np.float64(0.17600000000000002)
```

```
[ ] transformed_area / original
```

```
↗ np.float64(0.5499999999999999)
```

```
[ ] detM = np.linalg.det(M)
  detM
```

```
↗ np.float64(0.5499999999999999)
```

```
[ ] original * detM
```

```
↗ np.float64(0.17600000000000002)
```

```
[ ] expected_area = detM * original
  expected_area
```

```
↗ np.float64(0.17600000000000002)
```

# 행렬식의 특성

- 몇 가지 기억해 둘 행렬식의 특성

$$|\mathbf{A}| = |\mathbf{A}^T|$$

$$\mathbf{A} \in \mathbb{R}^{n \times n} \Rightarrow |k\mathbf{A}| = k^n |\mathbf{A}|$$

$$|\mathbf{AB}| = |\mathbf{A}||\mathbf{B}|$$



# 행렬식 계산

- 행렬  $\mathbf{A} \in \mathbb{R}^{n \times n}$ 의  $i$  행,  $j$  열 여인자  $C_{ij}$ 를 이용한 행렬식 계산

$$\begin{aligned} \det \mathbf{A} = |\mathbf{A}| &= \sum_{j=1}^n A_{1j}C_{1j} = \sum_{j=1}^n A_{2j}C_{2j} = \cdots = \sum_{j=1}^n A_{nj}C_{nj} \\ &= \sum_{i=1}^n A_{i1}C_{i1} = \sum_{i=1}^n A_{i2}C_{i2} = \cdots = \sum_{i=1}^n A_{in}C_{in} \end{aligned}$$

- $\mathbf{A}$ 의 임의의 행 벡터  $\mathbf{A}_{i,*}$ 와  $\mathbf{C}$ 의 동일 위치 행 벡터  $\mathbf{C}_{i,*}$ 의 내적
- $\mathbf{A}$ 의 임의의 열 벡터  $\mathbf{A}_{*,j}$ 와  $\mathbf{C}$ 의 동일 위치 열 벡터  $\mathbf{C}_{*,j}$ 의 내적

$$\det \mathbf{A} = |\mathbf{A}| = \mathbf{A}_{i,*} \mathbf{C}_{i,*}^T = \mathbf{A}_{*,j}^T \mathbf{C}_{*,j}$$

여인자를 구하면  
행렬식을 구할 수 있다

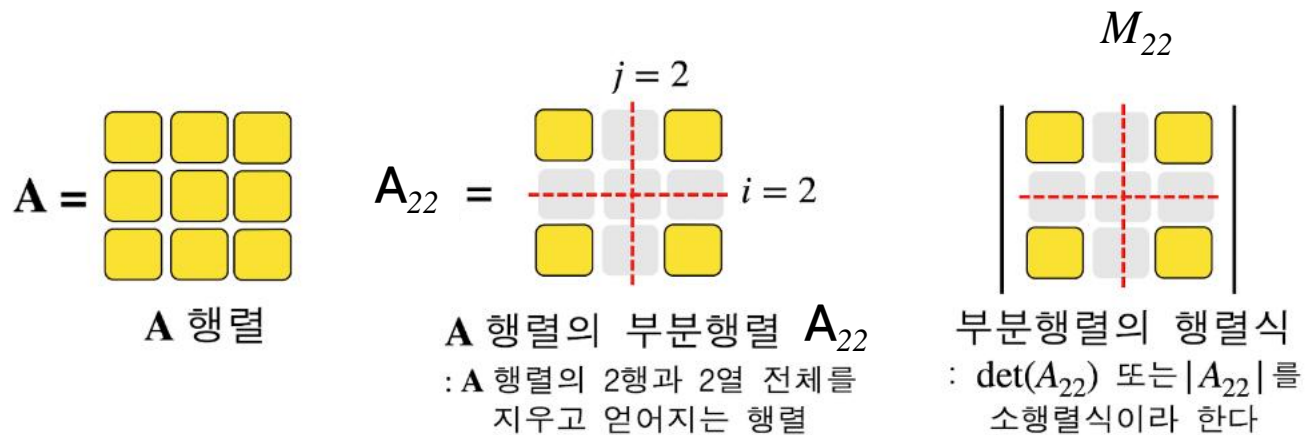
# 소행렬식 구하기

- 부분행렬

- A의 부분행렬  $A_{ij}$ 는 행렬 A의  $i$ 행과  $j$ 열을 제거하고 얻는 행렬
- A의  $i$ 행과  $j$ 열 성분을 의미하는  $A_{ij}$ 와 혼동하지 않을 것

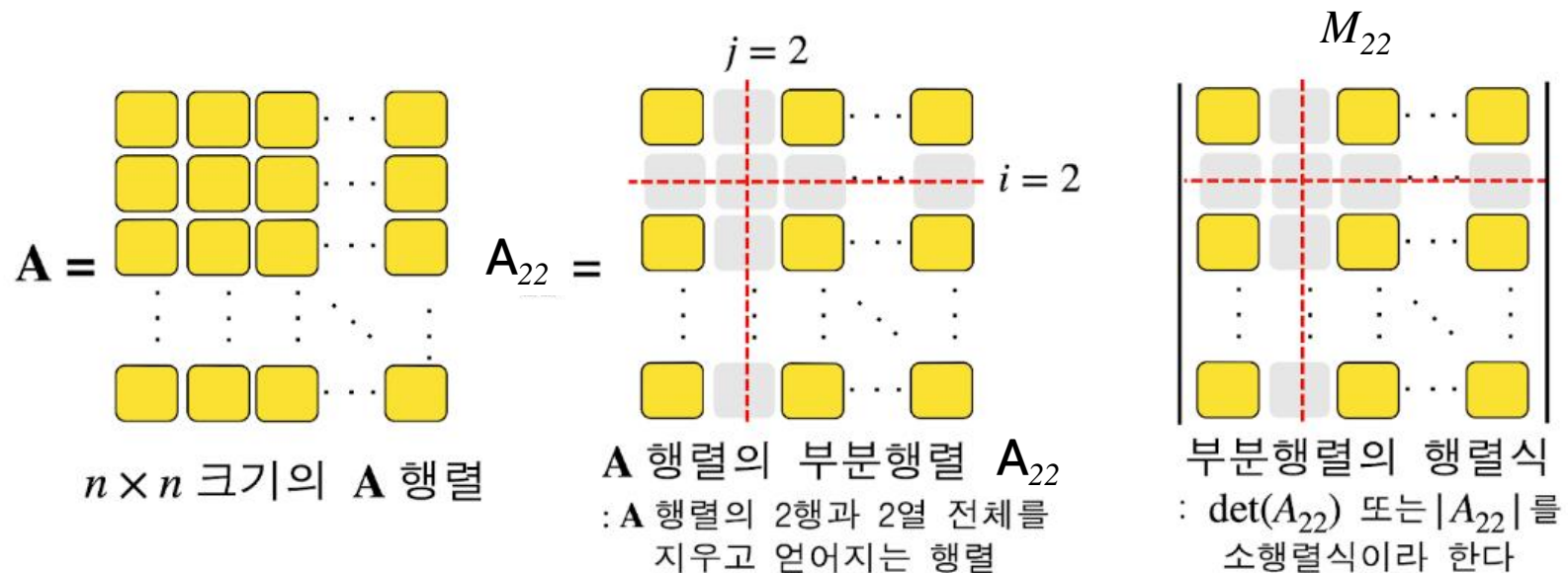
- 소행렬식

- $M_{ij}$ :  $A_{ij}$ 의 행렬식



# 소행렬식의 일반화

- $n \times n$  행렬



# 여인수

$$C_{ij} = (-1)^{i+j} \det(\mathbf{A}_{ij}) \iff C_{ij} = (-1)^{i+j} M_{ij}$$

여인수<sup>cofactor</sup>는 소행렬식이 구해지는 위치마다 결정되며 그 값이 그림과 같이 소행렬식과 같거나 위치에 따라 부호만 달라질 수 있다.

$$\begin{bmatrix} + & - \\ - & + \end{bmatrix}$$

2 × 2 크기 행렬에 대한  
여인수 부호

$$\begin{bmatrix} + & - & + \\ - & + & - \\ + & - & + \end{bmatrix}$$

3 × 3 크기 행렬에 대한  
여인수 부호

...

$$\begin{bmatrix} + & - & + & \cdots \\ - & + & - & \cdots \\ + & - & + & \cdots \\ - & + & - & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$


$(-1)^{i+j}$ 에 따라 결정되는 여인수의 부호

# 행렬식의 계산 – 여인수 전개

$$\det(\mathbf{A}) = \sum_{j=1}^n a_{1j} C_{1j}$$


행렬이 3차 정사각 행렬일 때:

2x2 부분행렬로 구함


$$|\mathbf{A}| = a_{11}C_{11} + a_{12}C_{12} + a_{13}C_{13}$$

행렬이 4차 정사각 행렬일 때:


3x3 부분행렬로 구함


$$|\mathbf{A}| = a_{11}C_{11} + a_{12}C_{12} + a_{13}C_{13} + a_{14}C_{14}$$

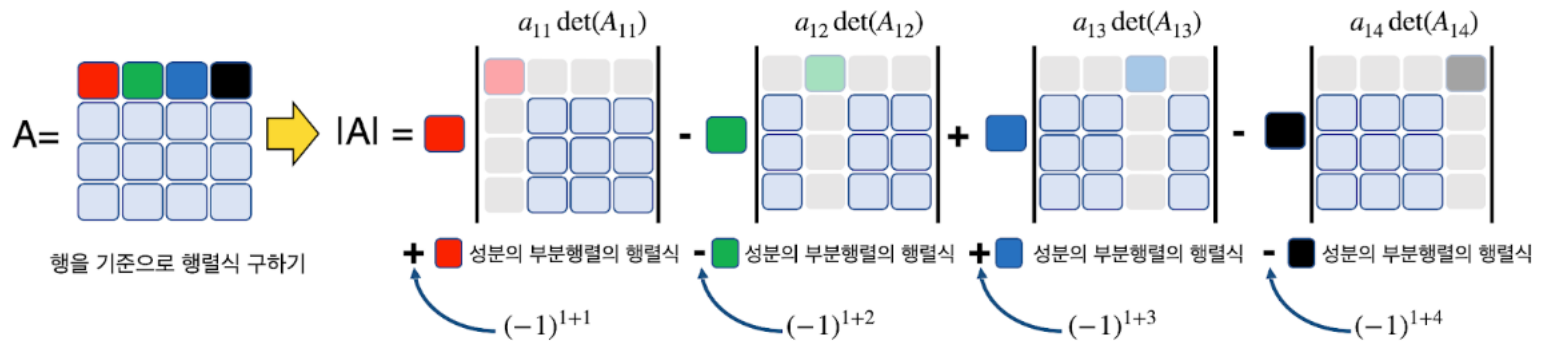
재귀 호출을 통해 구현 가능  
: 2x2 행렬의 행렬식은?

행렬이 n차 정사각 행렬일 때:

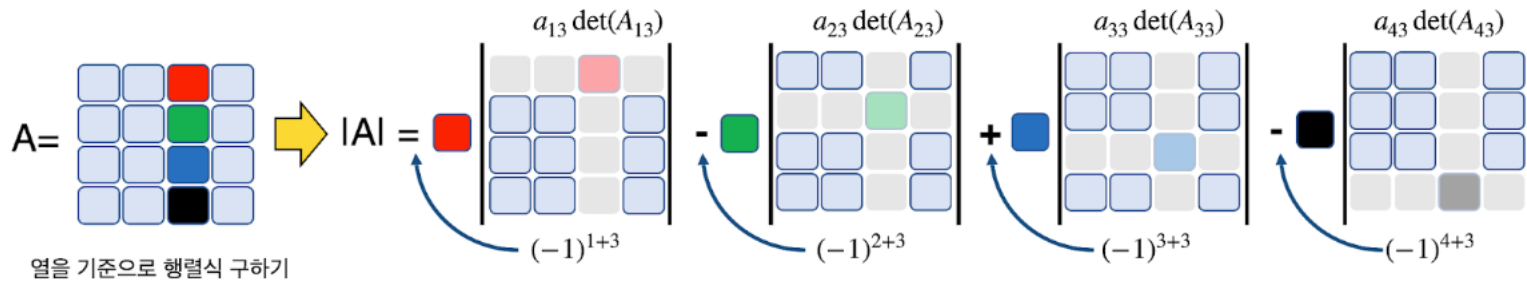
(n-1)x(n-1) 부분행렬로 구함


$$|\mathbf{A}| = a_{11}C_{11} + a_{12}C_{12} + a_{13}C_{13} + \cdots + a_{1n}C_{1n}$$

# 행렬식의 계산 – 여인수 전개



**임의의 행이나 열을 선택하여 여인수 전개 적용**



# 2x2 예제

- 행렬

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

- 소행렬식의 행렬

$$\mathbf{M} = \begin{bmatrix} \det[d] & \det[c] \\ \det[b] & \det[a] \end{bmatrix} = \begin{bmatrix} d & c \\ b & a \end{bmatrix}$$



- 여인자의 행렬

$$\mathbf{C} = \begin{bmatrix} (-1)^{1+1}d & (-1)^{1+2}c \\ (-1)^{2+1}b & (-1)^{2+2}a \end{bmatrix} = \begin{bmatrix} d & -c \\ -b & a \end{bmatrix}$$

- 행렬식

$$\det(\mathbf{A}) = ad - bc$$

# 3x3 예제

행렬

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

행렬식

$$\begin{aligned} \det(\mathbf{A}) &= \begin{vmatrix} 3 & 1 & 0 \\ -4 & -2 & 2 \\ 5 & 4 & -1 \end{vmatrix} = 3 \cdot \begin{vmatrix} -2 & 2 \\ 4 & -1 \end{vmatrix} - 1 \cdot \begin{vmatrix} -4 & 2 \\ 5 & -1 \end{vmatrix} + 0 \cdot \begin{vmatrix} -4 & -2 \\ 5 & 4 \end{vmatrix} \\ &= 3 \cdot (2 - 8) - 1 \cdot (4 - 10) + 0 \cdot (-16 - (-10)) \\ &= -12 \end{aligned}$$



# 행렬식 계산 효율

- 다음 행렬의 행렬식을 구하는 빠른 방법은?

$$\mathbf{A} = \begin{bmatrix} 2 & 0 & 30 & 15 \\ 3 & 1 & 3 & 1 \\ 0 & 0 & 4 & 5 \\ 0 & 0 & 3 & 1 \end{bmatrix}$$

- 0이 가장 많은 행이나 열을 찾는다

$$\mathbf{A} = \begin{bmatrix} 2 & 0 & 30 & 15 \\ 3 & 1 & 3 & 1 \\ 0 & 0 & 4 & 5 \\ 0 & 0 & 3 & 1 \end{bmatrix}$$

해당 행이나 열을 기준으로 여인수 전개

$$\det \mathbf{A} = 1 \begin{vmatrix} 2 & 30 & 15 \\ 0 & 4 & 5 \\ 0 & 3 & 1 \end{vmatrix} = 1 \cdot 2 \begin{vmatrix} 4 & 5 \\ 3 & 1 \end{vmatrix} = -22$$

# 삼각행렬의 행렬식

- 잘 살펴 보자

$$\det(\mathbf{A}) = \begin{vmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} \end{vmatrix} = a_{11} \begin{vmatrix} a_{22} & 0 & 0 \\ a_{32} & a_{33} & 0 \\ a_{42} & a_{43} & a_{44} \end{vmatrix} = a_{11} \cdot a_{22} \begin{vmatrix} a_{33} & 0 \\ a_{43} & a_{44} \end{vmatrix} = a_{11} \cdot a_{22} \cdot a_{33} \cdot a_{44}$$

## 정리 6.1 - 삼각행렬의 행렬식

어떤 행렬  $\mathbf{A}$ 가  $n \times n$  크기의 상삼각행렬, 하삼각행렬, 또는 대각행렬일 경우 이 행렬의 행렬식  $\det(\mathbf{A})$ 은 다음과 같이 주대각성분의 곱이다.

$$\det(\mathbf{A}) = a_{11} \cdot a_{22} \cdots a_{nn}$$

# 행렬식 계산 구현 – 재귀호출

- **Det(A):**

- Select a row or a column  $\rightarrow v$
- Compute the vector of cofactors along  $v \rightarrow c$ 
  - Computation of cofactors
    - Recursive call of **Det(A<sub>ij</sub>)**
- Compute the inner product of  $v$  and  $c$  and return the result

```
M = np.random.rand(4,4)
print(M)

M12 = np.delete(np.delete(M, 1, axis=0), 2, axis=1)
print(M12)
```

submatrix 구하기 예시

```
[[0.09173146 0.31002178 0.00114925 0.56483156]
 [0.6702642  0.10518936 0.04424309 0.99360292]
 [0.7855713  0.83524755 0.4896947  0.92187119]
 [0.69052721 0.19332984 0.19512489 0.84153646]]
[[0.09173146 0.31002178 0.56483156]
 [0.7855713  0.83524755 0.92187119]
 [0.69052721 0.19332984 0.84153646]]
```

# 행렬식 계산 전에 행렬 다루기

```
▶ M = np.random.rand(3,3)  
M
```

```
↻ array([[0.24858822, 0.32907555, 0.16754728],  
        [0.94455687, 0.66185563, 0.47999908],  
        [0.82532319, 0.13824988, 0.63444805]])
```

```
▶ M.shape
```

```
↻ (3, 3)
```

```
[ ] M = np.random.rand(5, 5)  
    print(M)  
    print(M.shape)
```

```
↻ [[0.89812269 0.16785124 0.1499851  0.33925657 0.61741416]  
   [0.61799352 0.40039423 0.07492019 0.58110507 0.49804987]  
   [0.68767253 0.44146367 0.53363926 0.26298161 0.98774626]  
   [0.61694095 0.99045415 0.36492197 0.23580411 0.74822636]  
   [0.62436915 0.05789441 0.53895975 0.31702081 0.11576815]]  
(5, 5)
```

```
[ ] M.T
```

```
↻ array([[0.89812269, 0.61799352, 0.68767253, 0.61694095, 0.62436915],  
        [0.16785124, 0.40039423, 0.44146367, 0.99045415, 0.05789441],  
        [0.1499851 , 0.07492019, 0.53363926, 0.36492197, 0.53895975],  
        [0.33925657, 0.58110507, 0.26298161, 0.23580411, 0.31702081],  
        [0.61741416, 0.49804987, 0.98774626, 0.74822636, 0.11576815]])
```

```
[ ] np.linalg.det(M)
```

```
↻ np.float64(0.07735830086916867)
```

# 행렬식 계산 전에 행렬 다루기

```
[ ] def 부분행렬(행렬, 지울_행, 지울_열):  
    행삭제결과 = np.delete(행렬, 지울_행, axis=0)  
    부분행렬 = np.delete(행삭제결과, 지울_열, axis=1)  
    return 부분행렬
```

```
▶ M00 = 부분행렬(M, 0, 0)  
print(M00)  
print(M00.shape)
```

```
↻ [[0.40039423 0.07492019 0.58110507 0.49804987]  
   [0.44146367 0.53363926 0.26298161 0.98774626]  
   [0.99045415 0.36492197 0.23580411 0.74822636]  
   [0.05789441 0.53895975 0.31702081 0.11576815]]  
(4, 4)
```

```
[ ] M = np.random.rand(5, 5)  
print(M)  
print(M.shape)  
  
M1 = 부분행렬(M, 0, 0)  
print(M1.shape)  
M2 = 부분행렬(M1, 0, 0)  
print(M2.shape)  
M3 = 부분행렬(M2, 0, 0)  
print(M3.shape)  
M4 = 부분행렬(M3, 0, 0)  
print(M4.shape)  
M5 = 부분행렬(M4, 0, 0)  
print(M5.shape)  
print(M5)
```

# 행렬식 계산

```
def 행렬식(행렬) :  
  
    행렬꼴 = 행렬.shape  
  
    # 정사각행렬인지 검사한다  
    if 행렬꼴[0] != 행렬꼴[1] :  
        # 정사각행렬이 아니다.  
        return 0  
  
    차원 = 행렬꼴[0]  
    # 재귀호출을 하지 않고 끝낼 수 있는 종료 조건 검사-처리  
    if 차원 == 0:  
        return 0  
    if 차원 == 1:  
        return 행렬[0][0]  
    if 차원 == 2:  
        return 행렬[0][0]*행렬[1][1] - 행렬[0][1] * 행렬[1][0]  
  
    # 재귀호출을 이용한 행렬식의 계산  
    # 행[0] : u  
    # 행[0]을 기준으로 계산된 여인자들의 벡터: v  
    # 행렬식은 u.dot(v)  
    행렬식_결과 = 0  
  
    for i in range(차원):  
  
        행0의_i번째_원소 = 행렬[0][i]  
  
        소행렬식 = 행렬식 ( 부분행렬(행렬, 0, i) )  
        여인자 = (-1)**(i)*소행렬식  
  
        행렬식_결과 += 행0의_i번째_원소 * 여인자  
  
    return 행렬식_결과
```

# 내가 만든 행렬식의 비효율성

```
[ ] import time
```

```
▶ start = time.time()  
result = 행렬식(myMat)  
eTime = time.time() - start  
print(f"나의 행렬식 계산기의 소요시간: {eTime}, 행렬식: {result}")
```

```
start = time.time()  
result = np.linalg.det(myMat)  
eTime = time.time() - start  
print(f"나의 행렬식 계산기의 소요시간: {eTime}, 행렬식: {result}")
```

```
↔ 나의 행렬식 계산기의 소요시간: 30.035696268081665, 행렬식: 0.016220476945112316  
   나의 행렬식 계산기의 소요시간: 0.00017261505126953125, 행렬식: 0.016220476945112344
```

# 역행렬

- 역행렬은 정방행렬에만 존재
- $\mathbf{A}$ 의 역행렬이 존재한다면, 이 역행렬을  $\mathbf{A}^{-1}$ 로 표현
- 역행렬  $\mathbf{A}^{-1}$ 은 다음과 같은 조건을 만족
  - $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$
  - $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$
- 역행렬이 존재하는 행렬을 가역행렬(invertible matrix)
- 역행렬이 존재하지 않는 행렬은 특이행렬(singular matrix)
- 의사 역행렬(pseudo-inverse)
  - 행렬  $\mathbf{A}$ 가 정방행렬이 아니고  $\mathbb{R}^{m \times n}$ 에 속한다고 하자. 다른 어떤 행렬  $\mathbf{B}$ 가  $\mathbb{R}^{n \times m}$ 에 속하면, 두 행렬의 곱  $\mathbf{AB}$ 는  $\mathbb{R}^{m \times m}$ 에 속하는 정방행렬이 된다. 만약  $\mathbf{AB} = \mathbf{I} \in \mathbb{R}^{n \times n}$ 이라면,  $\mathbf{B}$ 를  $\mathbf{A}$ 의 의사 역행렬(pseudo-inverse)라고 한다.



# 역행렬의 계산

- 역행렬의 계산은 수반행렬(adjoint matrix)를 이용하여 쉽게 정의
  - 행렬  $\mathbf{A}$ 의 수반행렬: 여인자  $C_{ij}$ 를 성분으로 하는 행렬  $\mathbf{C}$ 의 전치(transpose)

$$\text{adj } \mathbf{A} = \begin{pmatrix} C_{11} & C_{21} & \cdots & C_{n1} \\ C_{12} & C_{22} & \cdots & C_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ C_{1n} & C_{2n} & \cdots & C_{nn} \end{pmatrix} = \mathbf{C}^T$$

- 수반행렬을 행렬의 행렬식으로 나누면 역행렬이 된다.

$$\mathbf{A}^{-1} = \frac{\text{adj } \mathbf{A}}{|\mathbf{A}|} = \frac{1}{|\mathbf{A}|} \begin{pmatrix} C_{11} & C_{21} & \cdots & C_{n1} \\ C_{12} & C_{22} & \cdots & C_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ C_{1n} & C_{2n} & \cdots & C_{nn} \end{pmatrix} = \mathbf{C}^T$$

식은 간단하지만, 여인자를 구하는 재귀호출이 매우 많은 계산을 요구

# 역행렬의 계산

```
▶ def 소행렬식(행렬):  
    소행렬식_행렬 = np.zeros(행렬.shape)  
    차원 = 행렬.shape[0]  
    for 행 in range(차원):  
        for 열 in range(차원):  
            소행렬식_행렬[행][열] = 행렬식(부분행렬(행렬, 행, 열))  
  
    return 소행렬식_행렬  
  
def 여인자행렬(행렬):  
    여인자_행렬 = np.zeros(행렬.shape)  
    차원 = 행렬.shape[0]  
    for 행 in range(차원):  
        for 열 in range(차원):  
            여인자_행렬[행][열] = (-1)**(행+열)*소행렬식(부분행렬(행렬, 행, 열))  
  
    return 여인자_행렬
```

```
[ ] myMat = np.random.rand(4,4)  
myMat
```

```
↗ array([[0.39131086, 0.54604703, 0.20808678, 0.39031833],  
         [0.1793318 , 0.69345465, 0.99866051, 0.98324583],  
         [0.27855457, 0.33872312, 0.6627936 , 0.18943569],  
         [0.43406961, 0.53768489, 0.69193101, 0.50775741]])
```

```
[ ] 소행렬식(myMat), 여인자행렬(myMat)
```

```
↗ (array([[-0.04751704, -0.11565177, -0.0257832 , 0.0467119 ],  
         [ 0.04997898, 0.03101484, -0.00381629, 0.00468243],  
         [-0.08017037, -0.11870148, 0.03438351, 0.10401718],  
         [-0.16321874, -0.19324684, 0.00039814, 0.10692669]]),  
 array([[-0.04751704, 0.11565177, -0.0257832 , -0.0467119 ],  
         [-0.04997898, 0.03101484, 0.00381629, 0.00468243],  
         [-0.08017037, 0.11870148, 0.03438351, -0.10401718],  
         [ 0.16321874, -0.19324684, -0.00039814, 0.10692669]]))
```

# 역행렬의 계산

```
def 수반행렬(행렬) :  
    return 여인자행렬(행렬).T
```

```
def 역행렬(행렬):  
    return 수반행렬(행렬)/행렬식(행렬)
```

```
[ ] 어떤행렬 = np.random.rand(4,4)  
    그역행렬 = 역행렬(어떤행렬)
```

어떤행렬, 그역행렬

```
(array([[0.7586104 , 0.47925658, 0.68931091, 0.78515887],  
        [0.68917454, 0.66351434, 0.28964592, 0.07591528],  
        [0.48852502, 0.38595339, 0.2170957 , 0.54452426],  
        [0.53971455, 0.06467792, 0.18810643, 0.86994122]]),  
 array([[-0.62655588,  2.97119197, -4.89465254,  3.36993351],  
        [-0.48812261, -1.10761073,  5.59848771, -2.9670671 ],  
        [ 2.64764438, -0.65537343, -1.97758538, -1.09458591],  
        [-0.14748912, -1.61927908,  3.04803719, -0.48394048]]))
```

```
[ ] 항등행렬 = 어떤행렬 @ 그역행렬  
    print(항등행렬)
```

```
[[ 1.00000000e+00 -4.62012674e-16 -5.27806138e-16 -8.57046887e-16]  
 [-3.16368511e-17  1.00000000e+00 -2.61269393e-16 -1.36554073e-16]  
 [-1.35734878e-16 -2.51516237e-16  1.00000000e+00 -3.60969924e-16]  
 [-2.82718208e-16 -7.24998971e-17 -2.41005742e-17  1.00000000e+00]]
```

```
[ ] v = np.array([1, 2, 3, 4])  
    변환결과 = 어떤행렬.dot(v)  
  
    print(변환결과)  
    print(그역행렬.dot(변환결과))
```

```
[6.92569177 3.18880208 4.0898159  4.71315458]  
[1. 2. 3. 4.]
```