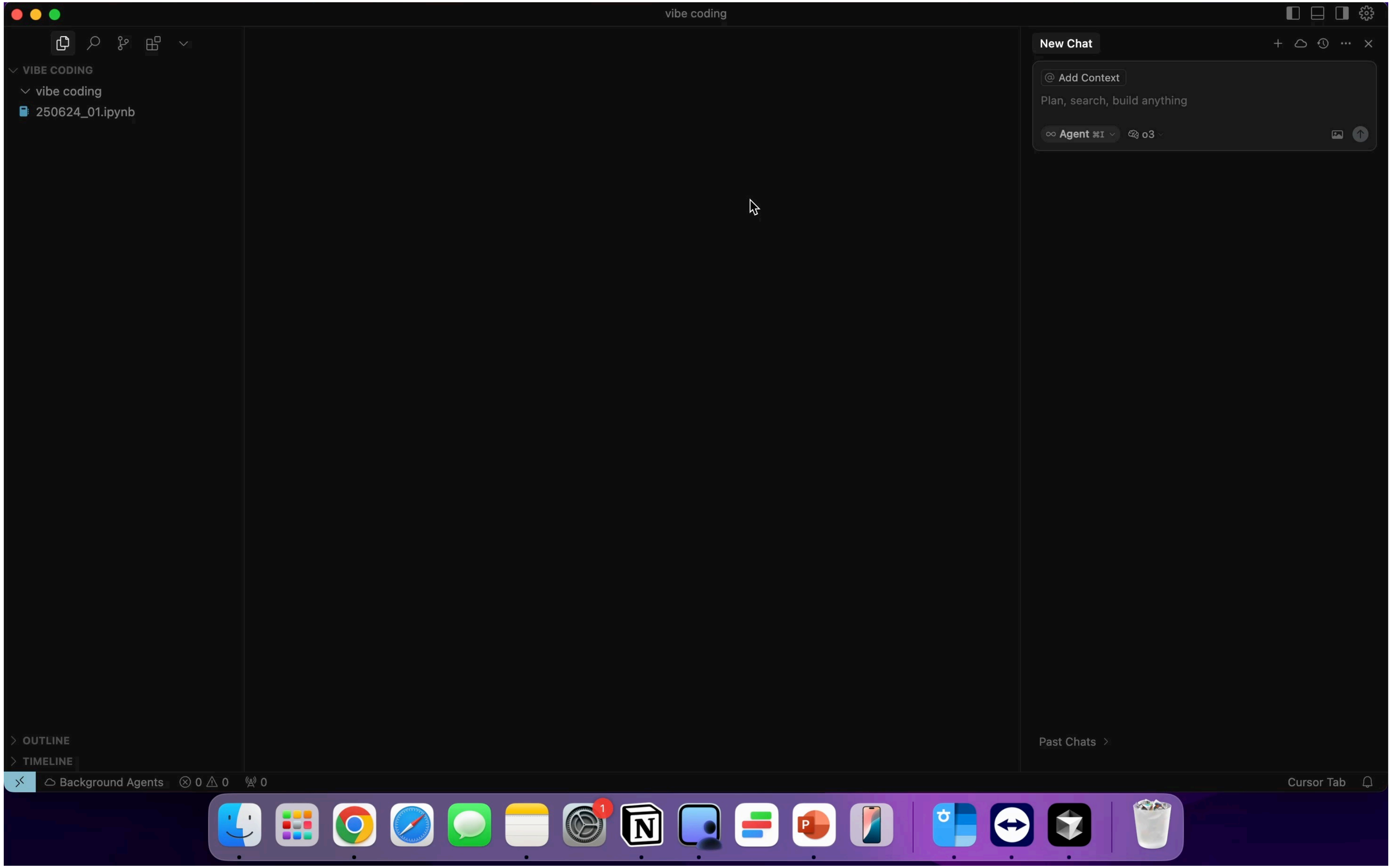
Math Vibe with AI

25.06.25

vibe coding

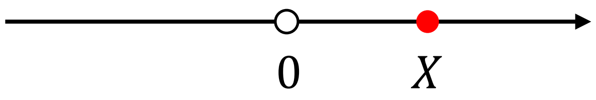

vibe coding
VIBE CODING
vibe coding
250624_01.ipynb
New Chat
Add Context
Plan, search, build anything
Agent
o3
Past Chats
OUTLINE
TIMELINE
Background Agents
Cursor Tab

Index

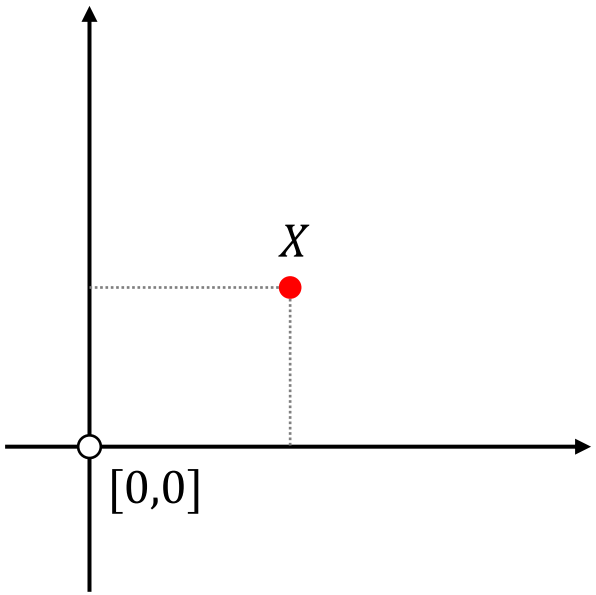
- vector
- matrix
- eigen decomposition

vector

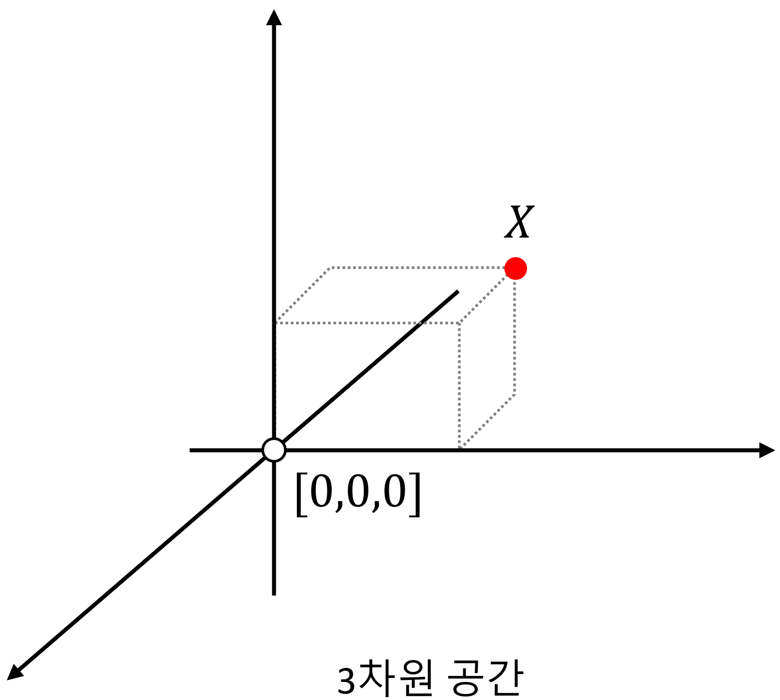
- vector 란, 숫자를 원소로 가지는 list 또는 array
- 벡터의 차원은 벡터가 가진 원소의 수
- 벡터의 방향은 벡터가 열 방향이면 열 벡터, 행 방향이면 행 벡터라고 부름



1차원 공간
(수직선)



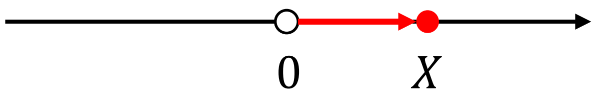
2차원 공간
(좌표평면)



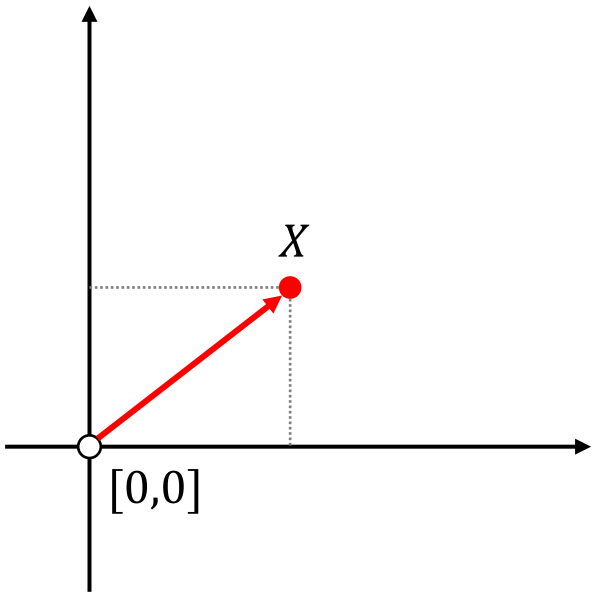
3차원 공간

vector

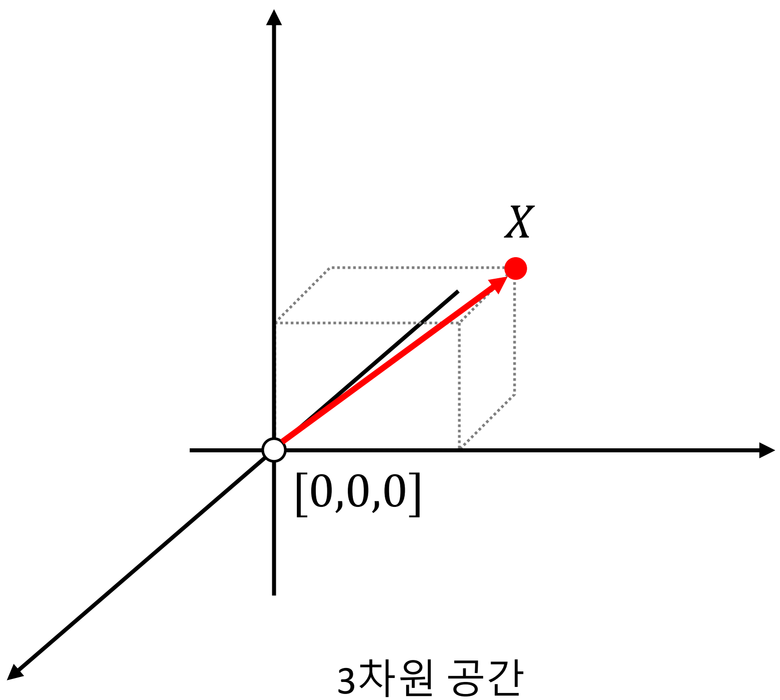
- vector 란, 숫자를 원소로 가지는 list 또는 array
- 벡터의 차원은 벡터가 가진 원소의 수
- 벡터의 방향은 벡터가 열 방향이면 열 벡터, 행 방향이면 행 벡터라고 부름



1차원 공간
(수직선)



2차원 공간
(좌표평면)



3차원 공간

vector

- vector 란, 숫자를 원소로 가지는 list 또는 array
- 벡터의 차원은 벡터가 가진 원소의 수
- 벡터의 방향은 벡터가 열 방향이면 열 벡터, 행 방향이면 행 벡터라고 부름

$$x = \begin{bmatrix} 1 \\ 4 \\ 5 \\ 6 \end{bmatrix} \qquad \mathrm{y} = \begin{bmatrix} 3 \\ -7 \end{bmatrix} \qquad \mathrm{z} = [1\ 4\ 5\ 6]$$

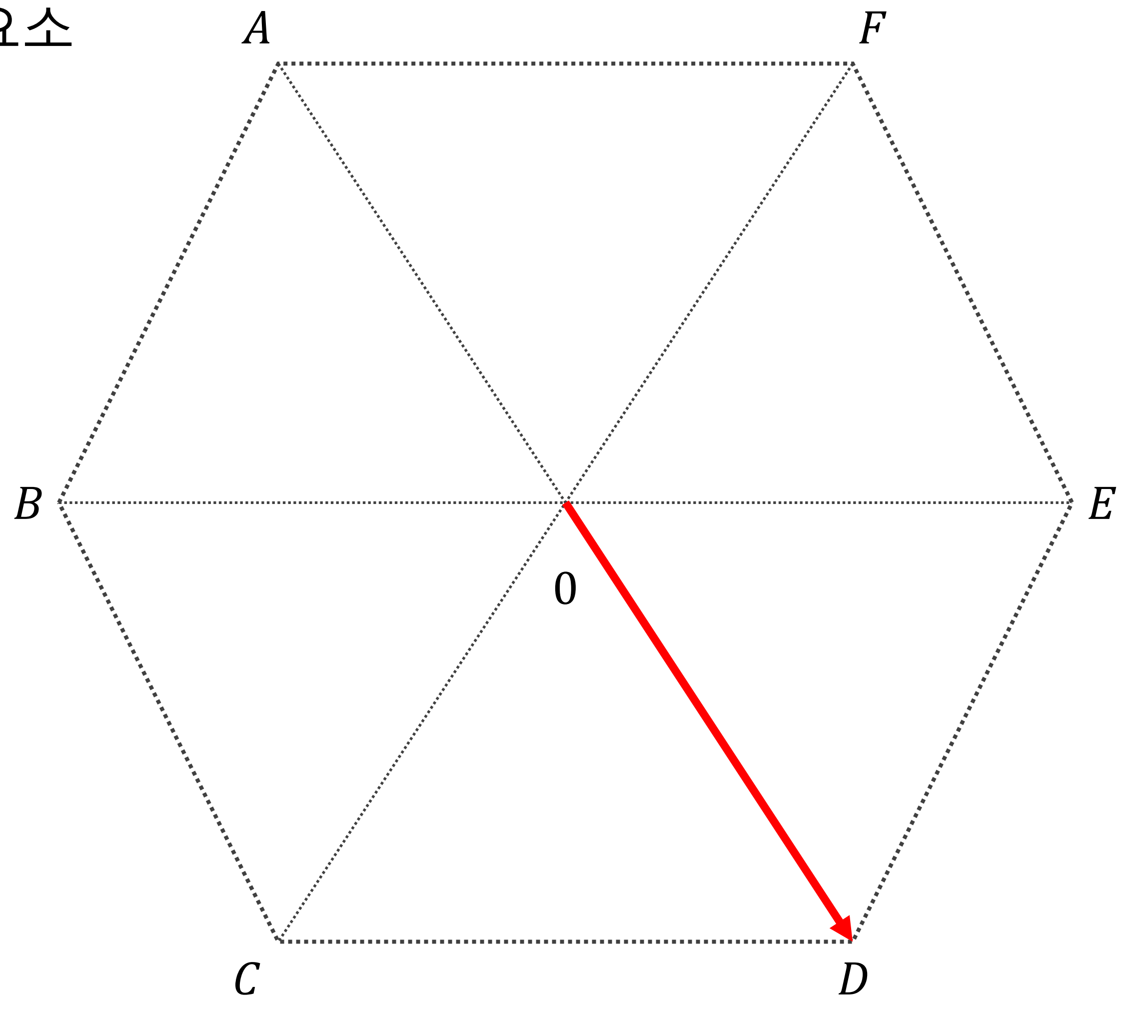
vector

- python 에서 vector 를 나타내는 방법

```
asList = [1,2,3] # 리스트
asArray = np.array([1,2,3]) # 방향 없는 배열, 1차원 리스트
rowVec = np.array([ [1,2,3] ]) # 행 벡터
colVec = np.array([ [1],[2],[3] ]) # 열 벡터
```

vector

- 기하학적으로 벡터는 '크기' 와 '방향' 을 가진 요소
- $\overrightarrow{OD} = \overrightarrow{AO} = \overrightarrow{FE} = \overrightarrow{BC}$
- $\overrightarrow{OD} = -\overrightarrow{OA}$



벡터의 연산

- 두 벡터의 덧셈/뺄셈은 서로 대응되는 원소끼리 더함/뺌

$$\begin{bmatrix}4\\5\\6\end{bmatrix}+\begin{bmatrix}10\\20\\30\end{bmatrix}=\begin{bmatrix}14\\25\\36\end{bmatrix}$$

$$\begin{bmatrix}4\\5\\6\end{bmatrix}-\begin{bmatrix}10\\20\\30\end{bmatrix}=\begin{bmatrix}-6\\-15\\-24\end{bmatrix}$$

```
v = np.array([1,2])
w = np.array([4,-6])
u = np.array([0,3,6,9])
print(v+w)
print(u+w)
```

벡터의 연산

✓ 열 벡터와 행 벡터를 더할 수 있을까?

$$[4\ 5\ 6] + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = ?$$

```
v = np.array([[4,5,6]])
w = np.array([[10],[20],[30]])
print(v+w)
```

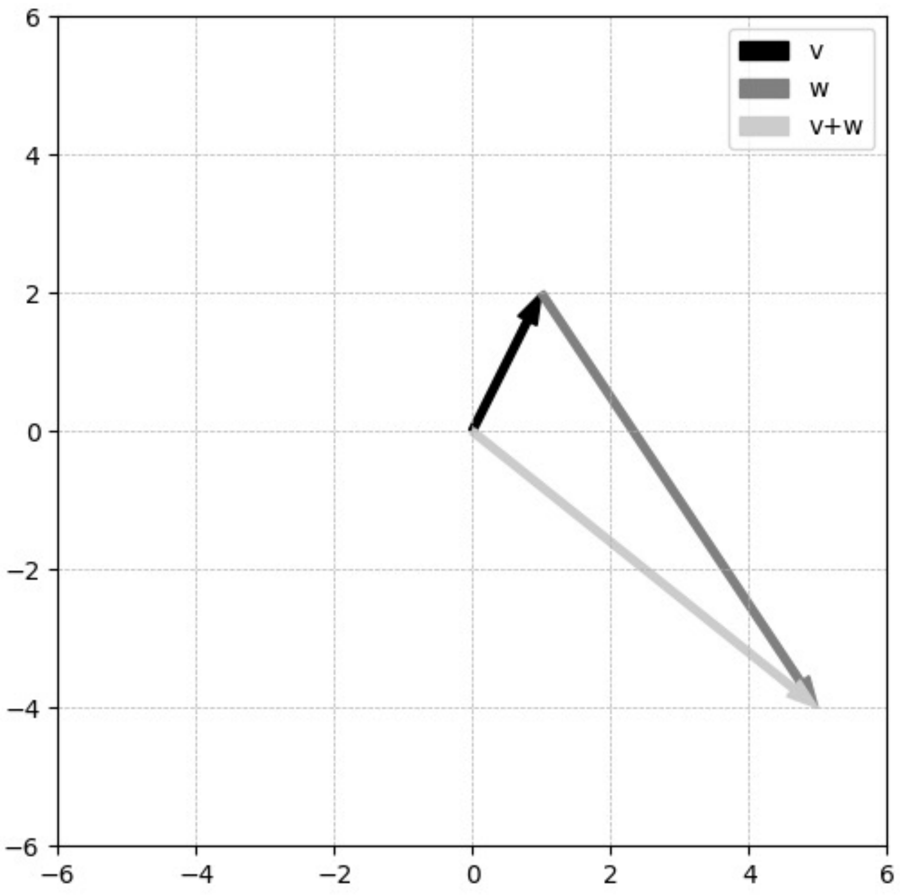
벡터의 연산

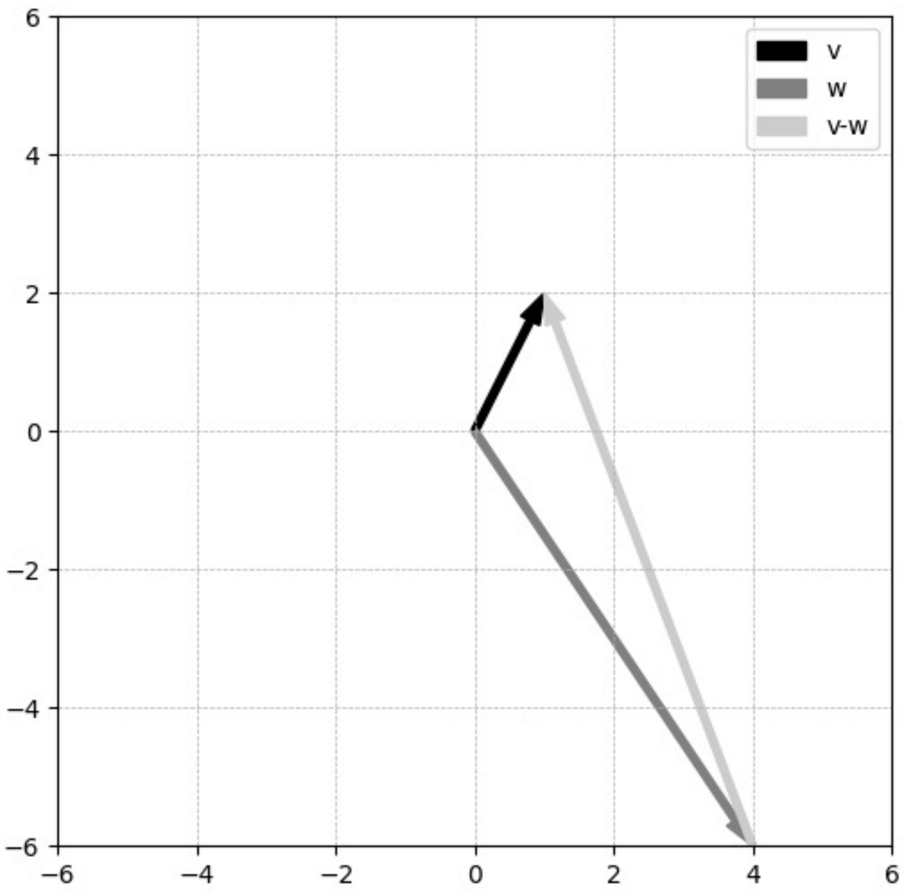
✓ python 에서는 한 벡터를 다른 벡터의 각 원소로 연산을 여러 번 반복하는 '브로드캐스팅' 이 가능

$$[4\ 5\ 6] + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{matrix} [\ 4\ 5\ 6\ ] \\ [\ 4\ 5\ 6\ ] \\ [\ 4\ 5\ 6\ ] \end{matrix} + \begin{matrix} [\ 10\ 10\ 10\ ] \\ [\ 20\ 20\ 20\ ] \\ [\ 30\ 30\ 30\ ] \end{matrix} = \begin{matrix} [14\ 15\ 16] \\ [24\ 25\ 26] \\ [34\ 35\ 36] \end{matrix}$$

- $[4\ 5\ 6]$ 을 열 벡터로, $\begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix}$ 을 행 벡터로 변경하여 실행해보자

벡터의 연산





위와 같이 벡터의 연산(덧셈, 뺄셈) 을 시각화하는 코드를 작성해보자 !

벡터의 연산

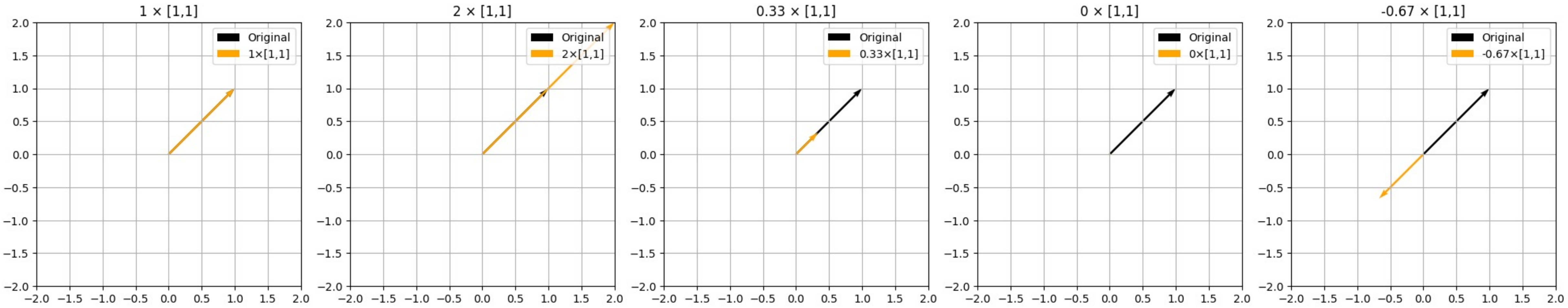
- 스칼라는 벡터나 행렬에 포함된 수가 아닌 수 그 자체

$$\lambda = 4, \qquad w = \begin{bmatrix} 9 \\ 4 \\ 1 \end{bmatrix}, \qquad \lambda w = \begin{bmatrix} 36 \\ 16 \\ 4 \end{bmatrix}$$

```
scalar = 4
w = np.array([[9,4,1]])
print(scalar*w)
```

벡터의 연산

- 스칼라는 벡터의 방향을 바꾸지 않고, 크기만 조정함
- 스칼라가 음수일 경우에는 어떻게 되는가 ?



벡터의 연산

- 전치 transpose 는 열 벡터를 행 벡터 또는 반대로 변환
- 예를 들어, 6차원 행 벡터에서 i 인덱스는 1로 고정이고, j 는 1~6 까지 존재
- 6차원 열 벡터는 i 인덱스가 1~6 까지 존재하고, j 는 1로 고정

$$m_{i,j}^{T} = m_{j,i}$$

- 벡터를 2번 전치할 경우, 원래 방향이 됨

$$a^{TT} = a$$

벡터의 연산

- 전치 transpose 는 열 벡터를 행 벡터 또는 반대로 변환

```
v = np.array([[1,2,3]])

v.T

np.transpose(v)
```

np.transpose() 또는 .T 와 같은 내장 함수나 메서드를 사용하지 않고 행 벡터를 열 벡터로 전치하는 for 루프를 작성해보자

벡터의 크기와 단위벡터

- 벡터의 크기 norm 는 벡터의 꼬리부터 머리까지의 거리이며, Euclidean Distance 로 구함

$$||v|| = \sqrt{\sum_{i=1}^{n} v_i^2}$$

- 3차원의 벡터 $v = [\, v_1, v_2, v_3]$ 은 $||v|| = \sqrt{v_1^2 + \; v_2^2 + \; v_3^2}$

```
v = np.array([1,2,3,7,8,9])
v_dim = len(v)
v_norm = np.linalg.norm(v)
print(v_norm)
```

벡터의 크기와 단위벡터

- 벡터의 크기는 벡터의 꼬리부터 머리까지의 거리이며, Euclidean Distance 로 구함

$$||v|| = \sqrt{\sum_{i=1}^{n} v_i^2}$$

- 3차원의 벡터 $v = [\, v_1, v_2, v_3]$ 은 $||v|| = \sqrt{v_1^2 + \, v_2^2 + \, v_3^2}$

위 식을 바탕으로, 벡터의 norm 을 직접 계산하는 함수를 작성해보자
작성한 함수가 np.linag.norm() 과 동일한 결과를 얻는지 확인해보자

벡터의 크기와 단위벡터

- 기하학적 길이가 1인 벡터를 단위 벡터 unit vector 라고 함

$$||v|| = 1$$

- 단위벡터가 아닌 비단위벡터와 같은 방향의 단위벡터를 만들기 위해서는

$$\hat{v} = \frac{1}{||v||} v$$

벡터를 입력으로 받고, 동일한 방향의 단위벡터를 출력하는 함수를 구현해보자

벡터의 내적

- 내적은 하나의 숫자로 두 벡터 사이의 관계를 나타냄
- 내적을 계산하기 위해서는 두 벡터에서 대응되는 원소끼리 곱한 다음 모든 결과를 더함

$$\delta = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

- 두 벡터 사이에 내적을 표기하는 방법을 여러 가지가 있음

$$a^T a \text{ or } a \cdot b \text{ or } < a, b >$$

벡터의 내적

- python 에서 내적을 구하는 방법

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

w = np.array([5,6,7,8])

np.dot(v,w)
```

벡터의 내적

- 벡터에 스칼라를 곱하면 내적도 그만큼 커짐

```
s = 10
v = np.array([1,2,3,4])
w = np.array([5,6,7,8])
np.dot(s*v, w)
```

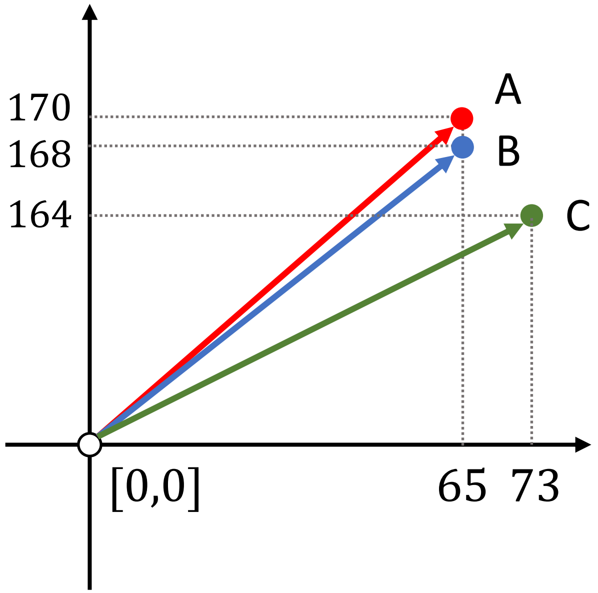
벡터의 내적

- 벡터에 음수의 스칼라를 곱한 후, 내적을 구해보자

```
s = -1
v = np.array([1,2,3,4])
w = np.array([5,6,7,8])
np.dot(s*v, w)
```

벡터의 내적

- 내적은 두 벡터 사이의 유사성 similarity 척도로 해석할 수 있음
- 3명의 키와 몸무게 데이터를 수집하고, 그 데이터를 2개의 벡터에 저장했다고 가정해보자



벡터의 내적

- 수학의 분배 법칙 $a(b+c) = ab + ac$ 은 벡터의 내적에도 적용됨

$$v \cdot (w + u) = v \cdot w + v \cdot u$$

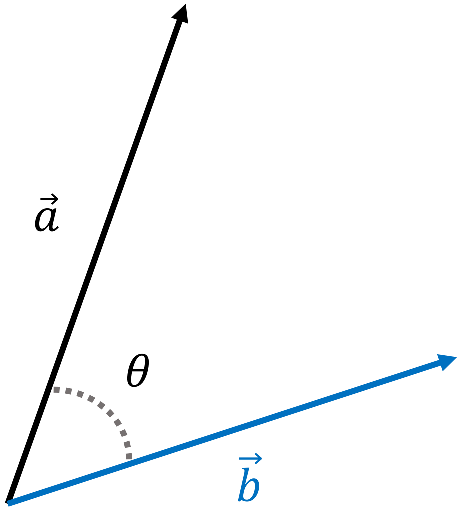
```
v = np.array([ 0,1,2 ])
w = np.array([ 3,5,8 ])
u = np.array([ 13,21,34 ])


res1 = np.dot( v, w+u )
res2 = np.dot( v,w ) + np.dot( v,u )
res1,res2
```

벡터의 내적

- 내적을 기하학적으로 정의하면, 두 벡터의 크기를 곱하고 두 벡터 사이의 각도에서 코사인값만큼 크기를 늘리는 것임

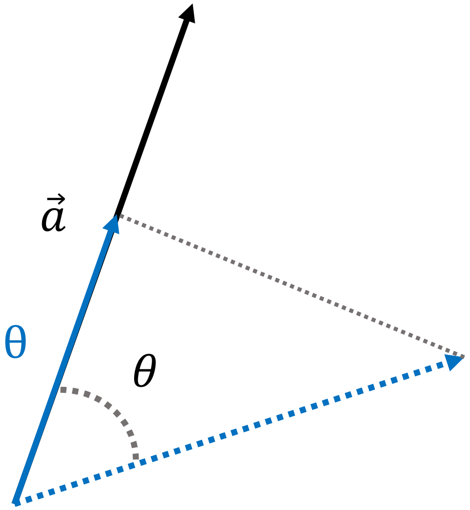
$$\alpha = \cos\left(\theta_{v,w}\right) ||v|| \, ||w||$$



벡터의 내적

- 내적을 기하학적으로 정의하면, 두 벡터의 크기를 곱하고 두 벡터 사이의 각도에서 코사인값만큼 크기를 늘리는 것임

$$\alpha = \cos(\theta_{v,w})\,||v||\,||w||$$

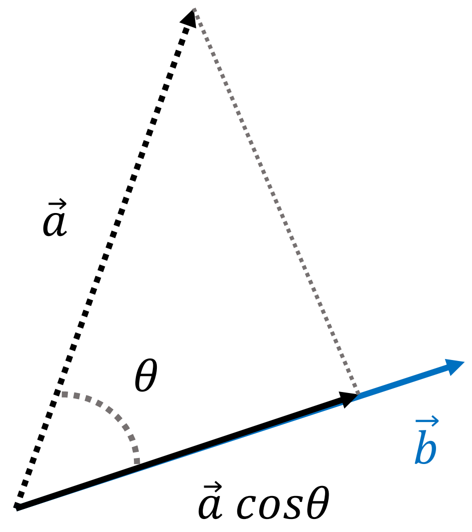


$\vec{b}$ 를 $\vec{a}$ 방향으로 정사영 = $\vec{b}$ cosθ

벡터의 내적

- 내적을 기하학적으로 정의하면, 두 벡터의 크기를 곱하고 두 벡터 사이의 각도에서 코사인값만큼 크기를 늘리는 것임

$$\alpha = \cos\left(\theta_{v,w}\right) ||v||\ ||w||$$



벡터의 내적

- 내적을 기하학적으로 정의하면, 두 벡터의 크기를 곱하고 두 벡터 사이의 각도에서 코사인값만큼 크기를 늘리는 것임

$$\alpha = \cos\left(\theta_{v,w}\right) ||v|| \, ||w||$$

내적의 교환 법칙이 성립하는지 살펴보고, 코드를 통해 구현해보자

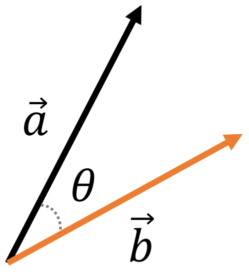
벡터의 내적

- 내적을 기하학적으로 정의하면, 두 벡터의 크기를 곱하고 두 벡터 사이의 각도에서 코사인값만큼 크기를 늘리는 것임

$$\alpha = \cos(\theta_{v,w})\,||v||\,||w||$$

예각

$\theta < 90^{\circ}$

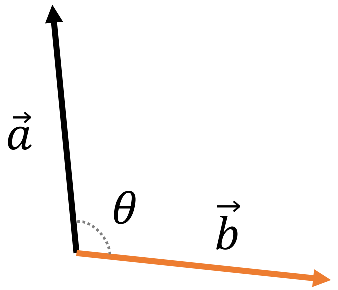


$\cos(\theta) > 0$

$\alpha > 0$

둔각

$\theta > 90^{\circ}$

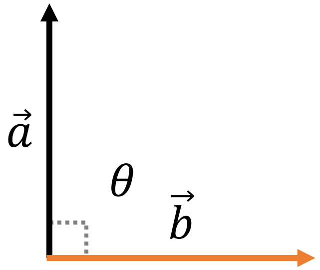


$\cos(\theta) < 0$

$\alpha < 0$

직각

$\theta < 90^{\circ}$

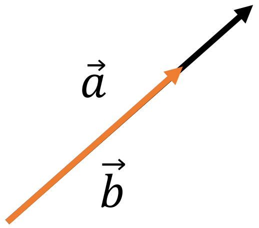


$\cos(\theta) = 0$

$\alpha = 0$

공선

$\theta = 0^{\circ}$

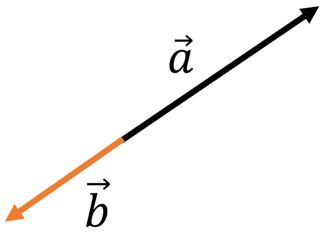


$\cos(\theta) = 1$

$\alpha = |a||b|$

공선

$\theta = 180^{\circ}$



$\cos(\theta) = -1$

$\alpha = -|a||b|$

벡터의 내적

- 어떤 벡터 $\vec{v}$ = $[v_1, v_2, v_3, \cdots, v_n]$ 의 크기는 다음과 같이 정의함

$$||\vec{v}|| = \sqrt{v_1^2 + v_2^2 + v_3^2 + \cdots + v_n^2}$$

$$||\vec{v}||^2 = v_1^2 + v_2^2 + v_3^2 + \cdots + v_n^2$$

$$||\vec{v}||^2 = \vec{v} \cdot \vec{v}$$

벡터의 크기 제곱을 벡터 그 자체의 내적으로 계산할 수 있는지 파이썬을 통해 확인해보자

벡터 집합

- 벡터들의 모음을 **집합** 이라고 하며, 벡터의 집합은 S 또는 V 와 같이 대문자 이탤릭체로 표시함

$$V = \{v_1, v_2, \cdots, v_n\}$$

- 예를 들어, 100개국의 COVID-19 양성 환자, 입원 및 사망자 수에 대한 데이터의 집합이 있다면,
- 각 국가의 데이터를 세 개의 원소를 가진 벡터에 저장하고 100개의 벡터가 포함된 벡터 집합을 생성할 수 있음

선형 가중 결합

- 선형 가중 결합은 여러 변수마다 가중치를 다르게 주어 정보를 혼합하는 방법
- 스칼라-벡터 곱셈을 한 다음에 합한 것으로,
- 벡터 집합에서 각 벡터에 스칼라를 곱한 다음 이들을 더해 하나의 벡터를 만듦

$$w = \lambda_1 v_1 + \lambda_2 v_2 + \cdots + \lambda_n v_n$$

모든 벡터 v_i 의 차원은 같다고 가정

선형 가중 결합

- $\lambda_1 = 1$, $\lambda_2 = 2$, $\lambda_3 = -3$, $v_1 = \begin{bmatrix} 4 \\ 5 \\ 1 \end{bmatrix}$, $v_2 = \begin{bmatrix} -4 \\ 0 \\ -4 \end{bmatrix}$, $v_3 = \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix}$ 에 대한 선형 가중 결합을 구해보자

```
v = np.array([ 0,1,2 ])
w = np.array([ 3,5,8 ])
u = np.array([ 13,21,34 ])


res1 = np.dot( v, w+u )
res2 = np.dot( v,w ) + np.dot( v,u )
res1,res2
```

선형 가중 결합

- 선형 가중 결합은 여러 방면에서 응용이 가능함
  - 통계 모델은 최소제곱법을 이용해, 독립변수에 계수를 곱한 선형 가중 결합으로 예측값을 생성함
  - 주성분 분석(PCA)과 같은 차원 축소 기법에서는, 변수 간의 선형 가중 결합을 통해 분산이 최대화되는 성분을 찾음
  - 인공 신경망에서 입력 데이터의 선형 가중 결합과 비선형 변환을 통해 목적 함수*를 최소화하도록 학습함

* 목적 함수 : 인공 신경망이 예측한 값과 실제 값의 차이

선형 독립성

- 벡터 집합에서 적어도 하나의 벡터를 집합의 다른 벡터들이 선형 가중 결합으로 나타낼 수 있을 때, 벡터 집합을 선형 종속 linear dependent 라고 함
- 반대로 집합에 있는 벡터들의 선형 가중 결합으로 집합의 아무런 벡터도 나타낼 수 없다면 해당 벡터 집합은 선형 독립 linear independent 임

* 목적 함수 : 인공 신경망이 예측한 값과 실제 값의 차이

선형 독립성

- 아래의 두 벡터 집합은 선형 독립일지, 선형 종속일지 생각해보자

$$V = \left\{ \begin{bmatrix} 1 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ 7 \end{bmatrix} \right\}, S = \left\{ \begin{bmatrix} 1 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ 6 \end{bmatrix} \right\}$$

선형 독립성

- 아래의 두 벡터 집합은 선형 독립일지, 선형 종속일지 생각해보자

$$V = \left\{ \begin{bmatrix} 1 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ 7 \end{bmatrix} \right\}, S = \left\{ \begin{bmatrix} 1 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ 6 \end{bmatrix} \right\}$$

- 벡터 집합 V는 선형 독립 → 집합의 한 벡터를 다른 벡터의 선형 배수로 나타낼 수 없음
- 집합 내 벡터들을 v_1, v_2 라고 했을 때, $v_1 = \lambda v_2$ 인 스칼라 λ 가 존재하지 않음
- 벡터 집합 S는 선형 종속 → 집합의 벡터들을 선형 가중 결합하여 집합의 다른 벡터를 만들 수 있음
- $s_1 = 0.5 s_2$ 하거나 $s_2 = 2 s_1$
- 선형 종속일 경우 무한히 많은 선형 결합 표현이 존재함

선형 독립성

- 아래의 벡터 집합 T 가 선형적으로 독립인지 종속인지 파악해보자

$$T = \left\{ \begin{bmatrix} 8 \\ -4 \\ 14 \\ 6 \end{bmatrix}, \begin{bmatrix} 4 \\ 6 \\ 0 \\ 3 \end{bmatrix}, \begin{bmatrix} 14 \\ 2 \\ 4 \\ 7 \end{bmatrix}, \begin{bmatrix} 13 \\ 2 \\ 9 \\ 8 \end{bmatrix} \right\}$$

위의 벡터 집합 T 가 선형 종속인지 독립인지 판별하는 코드를 구현해보자

선형 독립성

- 선형 종속일 경우, 집합의 벡터들이 선형 가중 결합으로 영벡터를 만들 수 있음

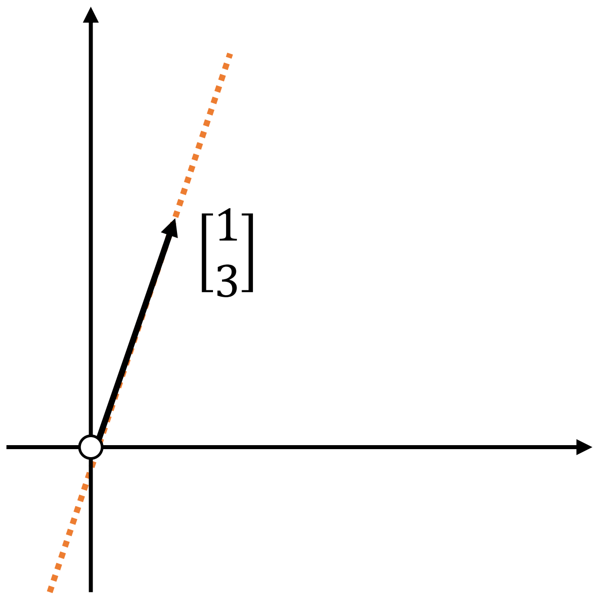
$$0 = \lambda_1 v_1 + \lambda_2 v_2 + \cdots + \lambda_n v_n , \qquad \lambda \in \mathbb{R}$$

- 반대로 벡터를 선형적으로 결합해서 영벡터를 생성할 수 있는 방법이 없다면 벡터의 집합은 선형 독립

부분공간과 생성

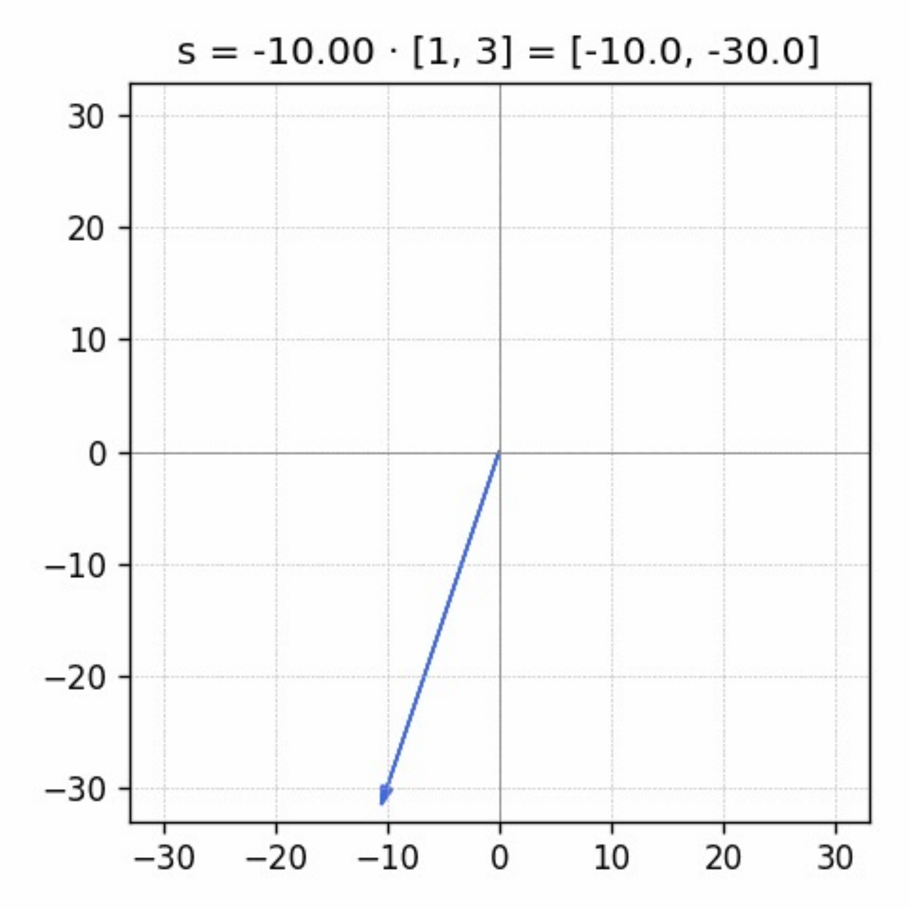
- (유한한) 벡터 집합의 벡터들과 가중치(계수, λ) 를 사용하여 무한히 선형 결합하는 방식으로 벡터의 부분공간 subspace 을 만듦
- 가능한 모든 선형 가중 결합을 구성하는 매커니즘을 벡터 집합의 생성 span 이라고 함
- 하나의 벡터를 가진 벡터 집합 V 을 예로 들면,

$$V = \left\{ \begin{bmatrix} 1 \\ 3 \end{bmatrix} \right\}$$



부분공간과 생성

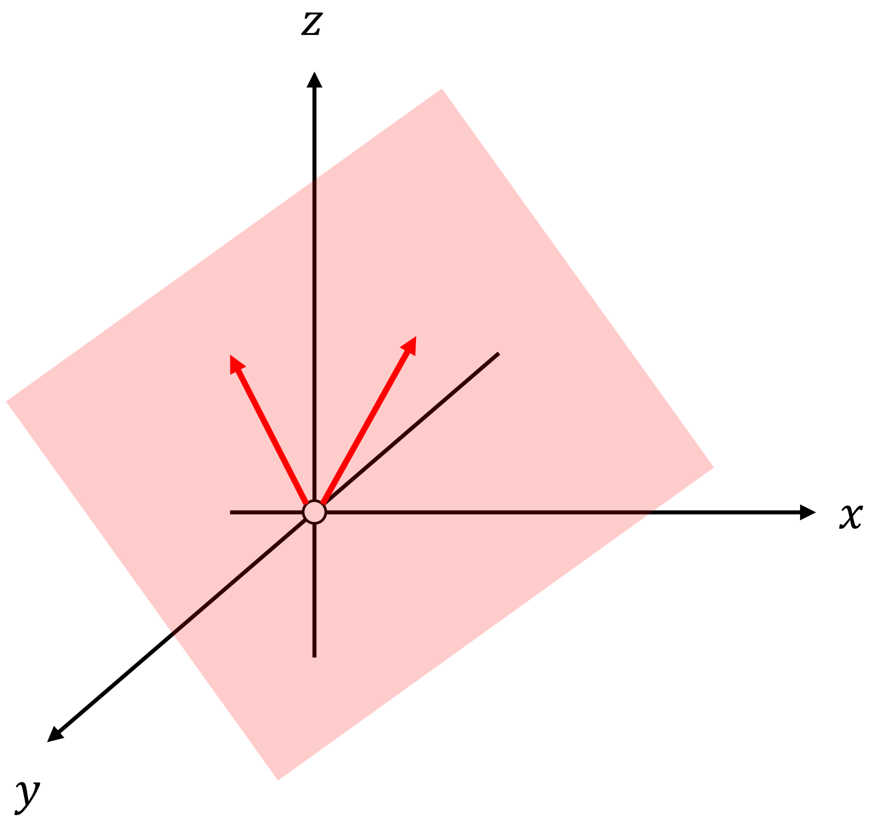
- 하나의 벡터를 가진 벡터 집합 V 을 예로 들면, $V = \left\{\begin{bmatrix}1\\3\end{bmatrix}\right\}$



부분공간과 생성

- 두 벡터를 가진 집합을 예로 들면, $V=\left\{\begin{bmatrix}1\\0\\2\end{bmatrix},\begin{bmatrix}-1\\1\\2\end{bmatrix}\right\}$

- span 은 2차원 부분공간



위 벡터 집합의 무한한 선형 결합의 결과를 이전 슬라이드와 같이 gif 파일로 만들어보자

기저

- 기저 basis 는 행렬의 정보를 설명하는데 사용하는 자 ruler 의 집합
- 가장 일반적인 기저 집합은, 데카르트 좌표계로, 익숙한 XY 평면을 의미함

$$S_2 = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}$$

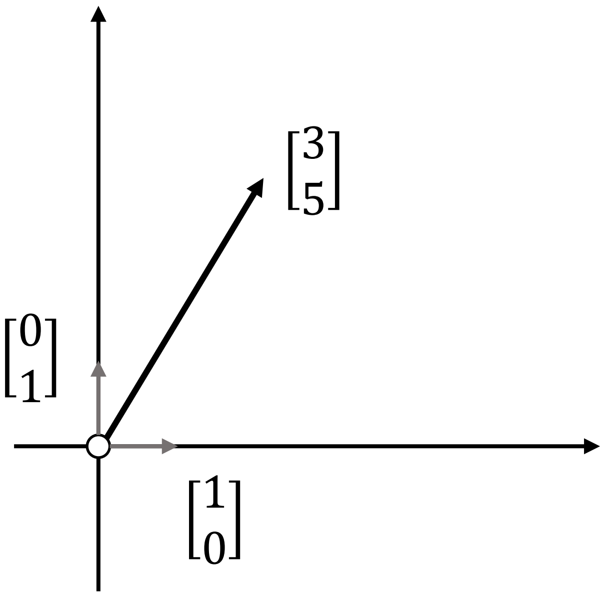
- 2차원 데카르트 그래프의 기저 집합은 위와 같으며, 서로 직교하고 단위 길이인 벡터로 이루어짐

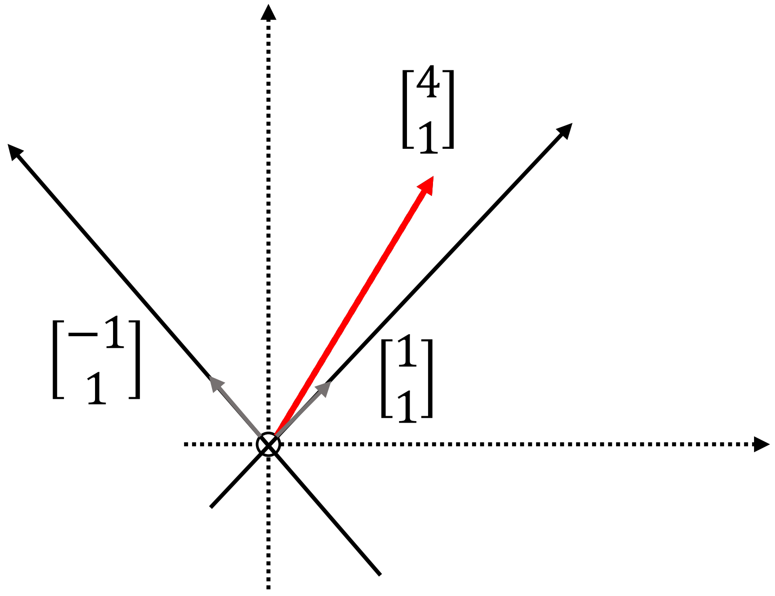
기저

- 아래의 벡터 $\vec{v}$ 를 데카르트 표준 좌표가 아닌 B_2 를 기저 벡터로 하여 시각화하는 코드를 구현해보자

$$\vec{v} = \begin{bmatrix} 3 \\ 5 \end{bmatrix}$$

$$B_2 = \left\{ \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right\}$$

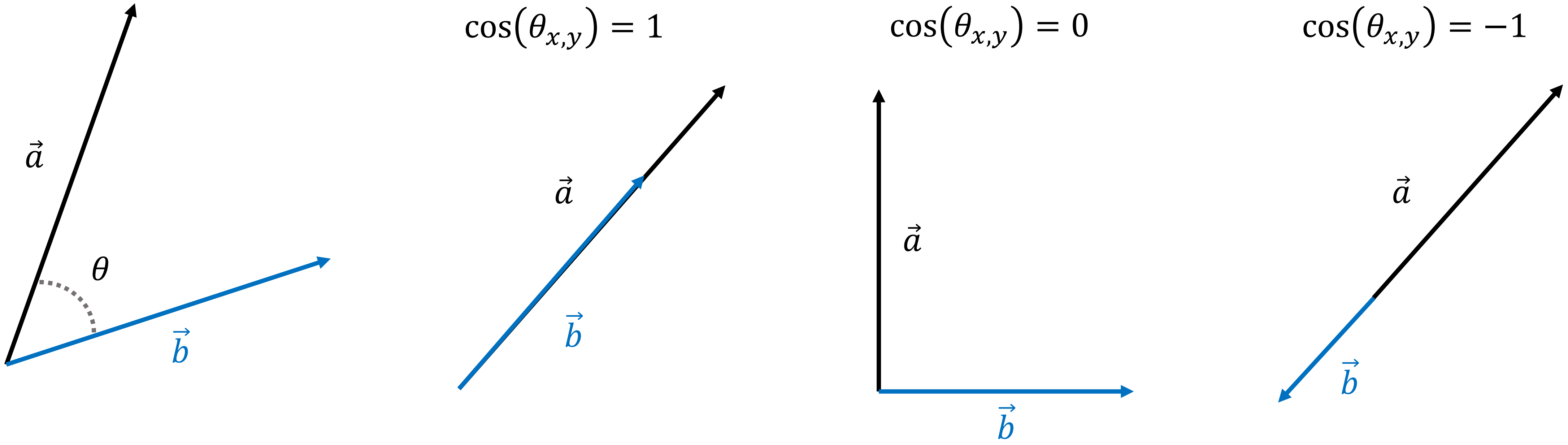




cosine similarity

- 두 벡터 간의 유사성을 평가하는 방법으로는 코사인 유사도 cosine similarity 가 있음

$$\cos(\theta_{x,y}) = \frac{\alpha}{||x||\,||y||}$$



Mini Project

- (vibe coding) cosine similarity 를 이용한 추천 시스템 구현하기
  - 데이터셋(kaggle 등의 사이트 참조)을 가지고, 영화 추천 혹은 상품 추천 시스템 등을 만들어보자
  - 각 조별로 사용한 데이터셋 및 구현한 코드 발표 진행

matrix

- 행렬은 벡터를 한 차원 더 끌어올린 것으로, 열 벡터의 집합, 행 벡터의 집합, 개별 행렬 원소가 정렬된 집합으로 표현할 수 있음
- 행렬의 크기는 (행, 열) 규칙으로 나타내며, 아래의 행렬은 행이 3개이고 열이 5개인 3x5 행렬임

$$A = \begin{bmatrix} 1 & 3 & 5 & 7 & 9 \\ 0 & 2 & 4 & 6 & 8 \\ 1 & 4 & 7 & 8 & 9 \end{bmatrix}$$

- 행렬 A 의 세 번째 행과 네 번째 열에 있는 원소는 $a_{3,4}$ 로 나타내며, 파이썬에서는 A[2,3] 으로 인덱싱함

matrix

- 0~59 까지의 수가 원소인 (6, 10) 크기의 행렬을 만들고,
- 행 2~4와 열 1~5의 부분 행렬을 추출하는 코드를 작성해보자

```
A = np.arange(60).reshape(6, 10)

submatrix = A[1:4, 0:5]
```

특수 행렬

- 난수 행렬 random number matrix
  - 가우스(정규) 와 같은 분포로부터 무작위로 추출된 숫자를 가진 행렬

```
A = np.random.rand(6, 10)
```

특수 행렬

- 정방 행렬 square matrix
    - 정방 행렬의 행 수 와 열의 수는 같음
    - 비정방 행렬은 행 수와 열의 수가 다른 것을 의미함

특수 행렬

- 대각 행렬 diagonal matrix
  - 행렬의 대각 (왼쪽 위에서 오른쪽 아래로 내려가는 원소들) 원소를 제외한, 모든 비대각 원소가 0임
  - 대각 원소는 0일 수도 있지만 0이 아닌 값을 가질 수 있는 유일한 원소임
  - np.diag() 함수에 행렬을 입력하면, 대각 원소의 벡터를 반환
  - np.diag() 함수에 벡터를 입력하면 대각선에 해당 벡터의 원소가 있는 행렬을 반환

np.diag() 함수에 행렬을 입력한 결과, 벡터를 입력한 결과를 비교해보자

특수 행렬

- 삼각 행렬 triangular matrix
  - 주 대각선의 위 또는 아래가 0 인 벡터
  - 0 이 아닌 원소가 대각선 위에 있으면 상삼각 행렬 이라고 하고,
  - 0 이 아닌 원소가 대각선 아래에 있으면 하삼각 행렬 이라고 함

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

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

np.triu() 와 np.tril() 를 이용하여 삼각 행렬을 출력해보자

특수 행렬

- 단위 행렬 identity matrix
  - 행렬 또는 벡터에 단위 행렬을 곱하면 동일한 행렬 또는 벡터가 됨
  - 단위 행렬은 모든 대각 원소가 1인 정방 대각 행렬
  - 문자 I 로 나타내며, I_5 는 5x5 크기의 단위 행렬을 의미함

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

np.eye() 를 이용하여 단위 행렬을 출력해보자

특수 행렬

- 영 행렬 zeros matrix
  - 모든 원소가 0인 행렬

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

np.zeros() 함수를 사용하여 영 행렬을 출력해보자

행렬의 연산

- 두 행렬을 더할 때는 대응되는 원소끼리 더하고, 뺄 때는 대응되는 원소끼리 뺌

$$\begin{bmatrix} 2 & 3 & 4 \\ 1 & 2 & 4 \end{bmatrix} + \begin{bmatrix} 0 & 3 & 1 \\ -1 & -4 & 2 \end{bmatrix} = \begin{bmatrix} (2+0) & (3+3) & (4+1) \\ (1-1) & (2-4) & (4+2) \end{bmatrix} = \begin{bmatrix} 2 & 6 & 5 \\ 0 & -2 & 6 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 3 & 4 \\ 1 & 2 & 4 \end{bmatrix} + \begin{bmatrix} 0 & 3 & 1 \\ -1 & -4 & 2 \end{bmatrix} = \begin{bmatrix} (2-0) & (3-3) & (4-1) \\ (1+1) & (2+4) & (4-2) \end{bmatrix} = \begin{bmatrix} 2 & 0 & 3 \\ 2 & 6 & 2 \end{bmatrix}$$

위 행렬의 덧셈 / 뺄셈을 코드로 구현해보자

행렬의 연산

- 벡터와 마찬가지로 행렬에서도 $\lambda + A$ 와 같이 스칼라를 더할 수 없으나,
- python 에서는 행렬의 요소에 스칼라를 추가하는 브로드캐스팅 연산(3+np.eye(2)) 이 가능함
- 정방 행렬에 스칼라를 더하기 위해서는 대각에 상숫값을 더하는 것과 같이 단위 행렬에 스칼라를 곱해서 더하는 방식인 행렬 이동 으로 구현함

$$\lambda = 6, A = \begin{bmatrix} 4 & 5 & 1 \\ 0 & 1 & 11 \\ 4 & 9 & 7 \end{bmatrix}$$

$$A + \lambda I \rightarrow \begin{bmatrix} 4 & 5 & 1 \\ 0 & 1 & 11 \\ 4 & 9 & 7 \end{bmatrix} + 6 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 10 & 5 & 1 \\ 0 & 7 & 11 \\ 4 & 9 & 3 \end{bmatrix}$$

행렬의 연산

$$\lambda = 6, A = \begin{bmatrix} 4 & 5 & 1 \\ 0 & 1 & 11 \\ 4 & 9 & 7 \end{bmatrix}$$

$$A + \lambda I \rightarrow \begin{bmatrix} 4 & 5 & 1 \\ 0 & 1 & 11 \\ 4 & 9 & 7 \end{bmatrix} + 6 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 10 & 5 & 1 \\ 0 & 7 & 11 \\ 4 & 9 & 3 \end{bmatrix}$$

```
A = np.array([ [4,5,1], [0,1,11], [4,9,7] ])
s = 6

A + s 와 A + s * np.eye(3) 의 결과값 비교
```

행렬의 연산

- 스칼라-행렬 곱셈은 행렬의 각 원소에 동일한 스칼라를 곱하는 것임

$$w\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} wa & wb \\ wc & wd \end{bmatrix}$$

- 아다마르곱은 두 행렬을 요소별로 곱하는 것(원소별 곱셈)임

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \odot \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} = \begin{bmatrix} 2a & 3b \\ 4c & 5d \end{bmatrix}$$

행렬의 연산

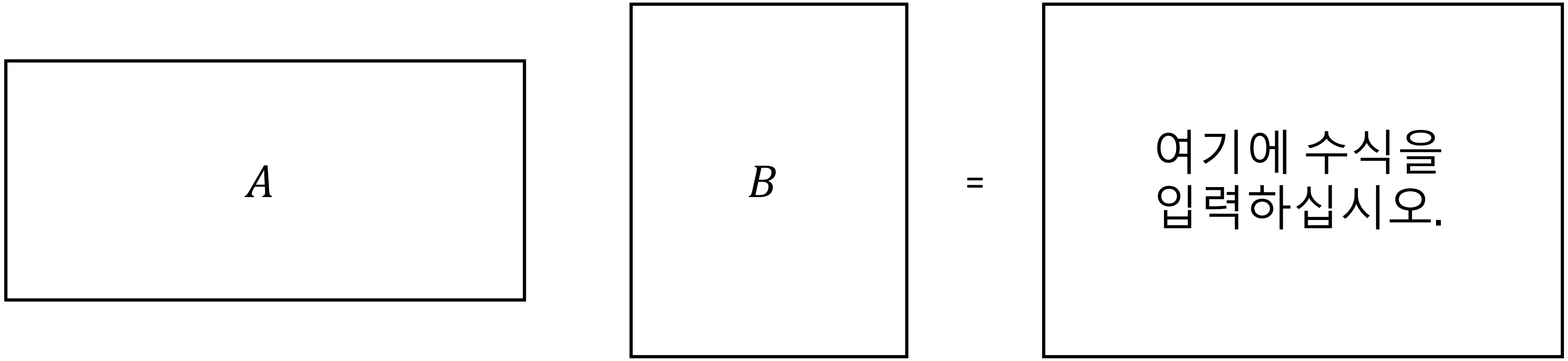
- python 을 이용해 아다마르 곱을 구현해보자

```
A = np.random.randn(3,4)
B = np.random.randn(3,4)

print(A*B)
print(np.multiply(A,B))
```

행렬 곱셈

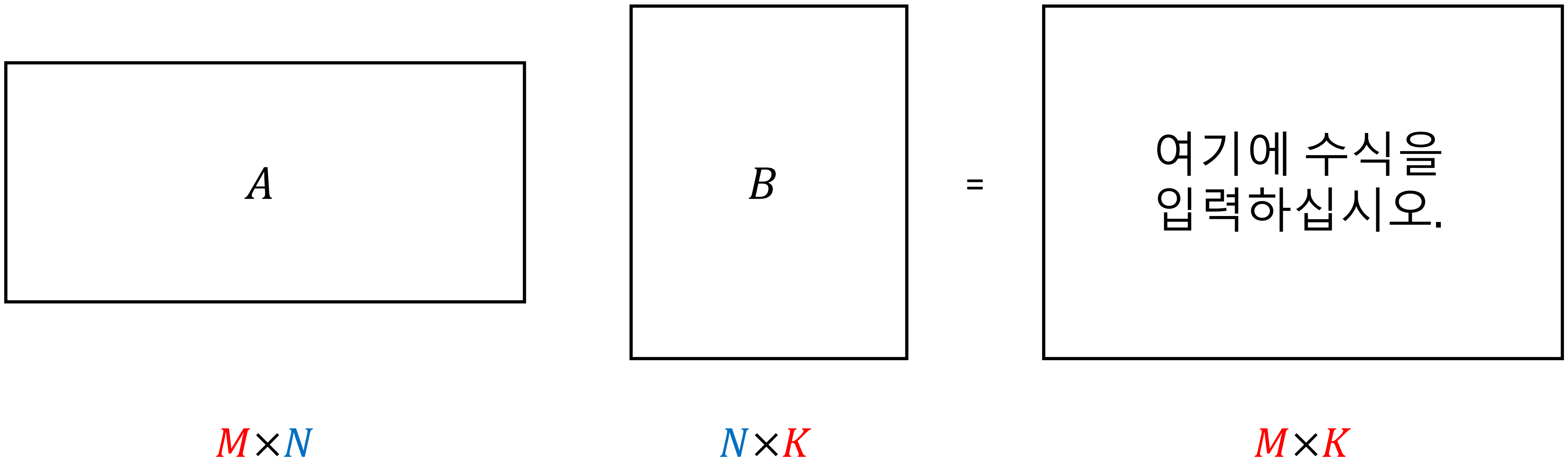
- 행렬 곱셈 matrix multiplication 은 원소별이 아닌 행과 열 단위로 동작함
- 한 행렬의 행과 다른 행렬의 열 사이 스칼라 곱셈의 집합으로 말할 수 있음
- 두 행렬을 곱하기 전에, 두 행렬을 곱하는 것이 가능한지 여부를 확인해야함



$M \times N$ $N \times K$ $M \times K$

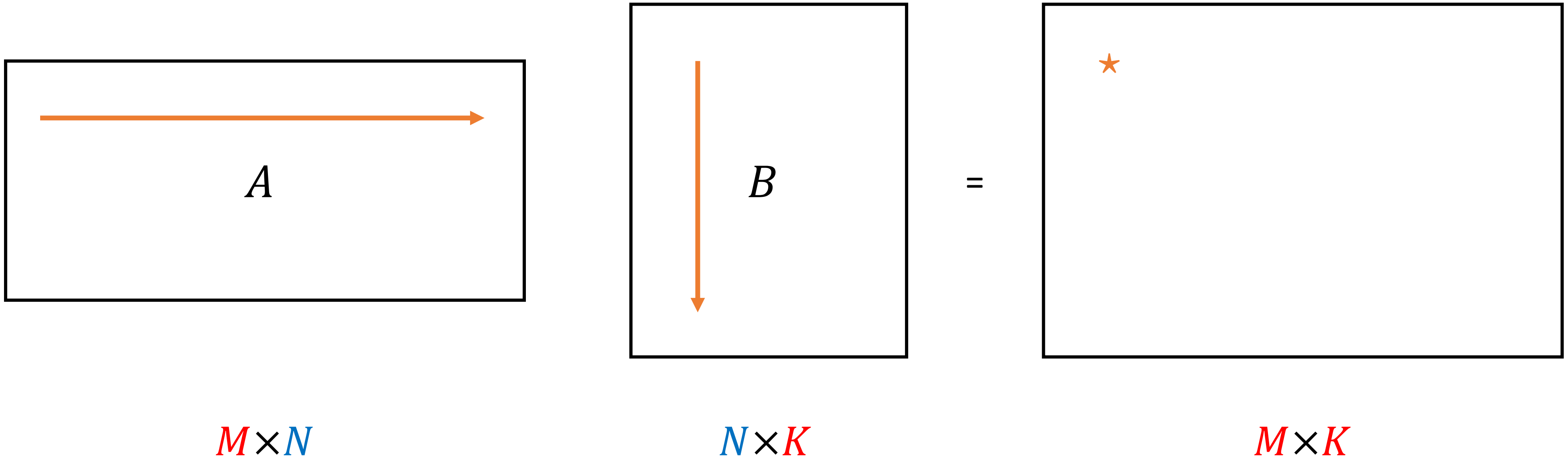
행렬 곱셈

- 행렬 곱셈은 내부 차원 수가 일치할 때만, 유효하고 곱셈 행렬의 크기는 외부 차원의 수로 정의함
- 즉, 왼쪽 행렬의 열 수가 오른쪽 행렬의 행 수와 같을 때 유효함
- 행렬 곱셈은 아다마르 곱(⊙) 과 같은 기호 없이 AB 와 같이 두 행렬을 나란히 적어 표기함



행렬 곱셈

- 왼쪽 행렬의 열 수가 오른쪽 행렬의 행 수와 일치하는 경우에만 행렬 곱셈이 유효한 이유는,
- 곱셈 행렬의 (i, j) 번째 원소가 왼쪽 행렬의 i 번째 행과 오른쪽 행렬의 j 번째 열 사이의 내적이기 때문임



행렬 곱셈

- 행렬-벡터의 곱셈
  - 행 벡터가 아닌 열 벡터만 행렬의 오른쪽에 곱할 수 있음
  - $M \times N$ 행렬의 왼쪽에 $1 \times M$ 행렬(행 벡터) 를 곱하거나, 오른쪽에 $N \times 1$ 행렬(열 벡터)를 곱할 수 있음
  - 행렬-벡터 곱셈의 결과는 항상 벡터임
  - 행렬에 행 벡터를 앞에 곱하면 다른 행벡터가 생성되지만, 행렬에 열벡터를 뒤에 곱하면 다른 열벡터가 생성됨

for 문을 이용하여 행렬의 곱셈을 구현해보자
@ 연산자를 사용해 구현한 내용과 결과를 비교해보자

행렬 곱셈

- 행렬-벡터의 곱셈
  - 스칼라와 벡터를 개별적으로 곱해서 선형 가중 결합을 계산했었으나,
  - 각 벡터를 행렬에 넣고 가중치(스칼라)를 벡터의 원소로 넣어 계산해보자

$$4\begin{bmatrix}3\\0\\6\end{bmatrix}+3\begin{bmatrix}1\\2\\5\end{bmatrix}=\begin{bmatrix}3&1\\0&2\\6&5\end{bmatrix}\begin{bmatrix}4\\3\end{bmatrix}=?$$

행렬 곱셈

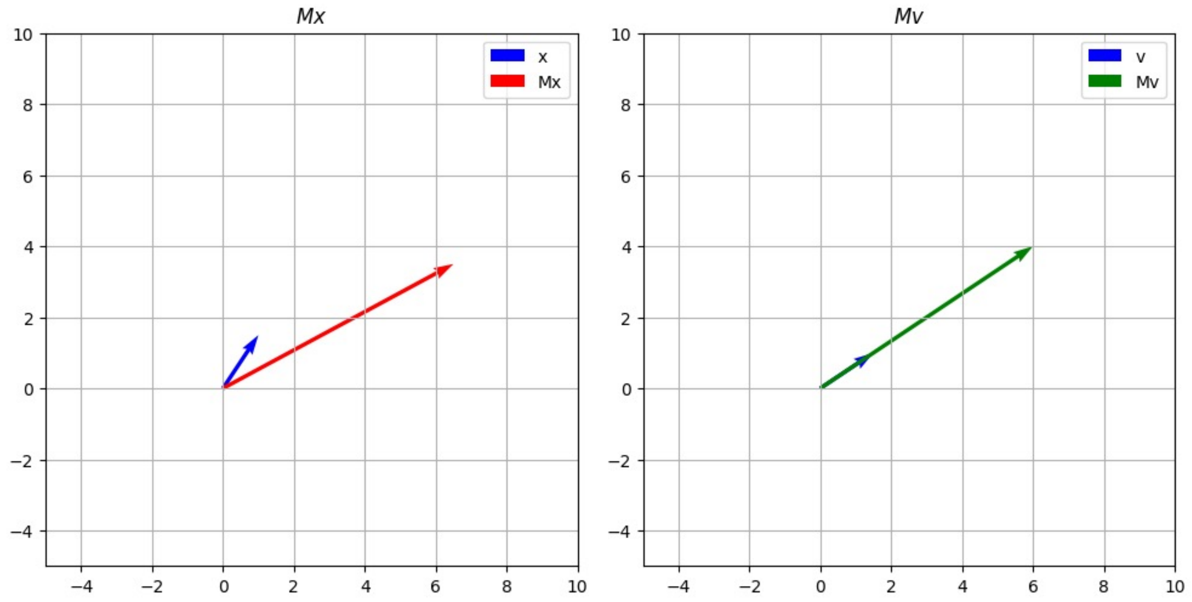
- 행렬-벡터의 곱셈
  - 스칼라와 벡터를 개별적으로 곱해서 선형 가중 결합을 계산 했었으나,
  - 각 벡터를 행렬에 넣고 가중치(스칼라)를 벡터의 원소로 넣어 계산해보자

앞선 슬라이드는 열 벡터-행렬의 선형 가중 결합이었다면,
행 벡터-행렬을 선형 가중 결합하는 코드를 구현해보자

행렬 곱셈

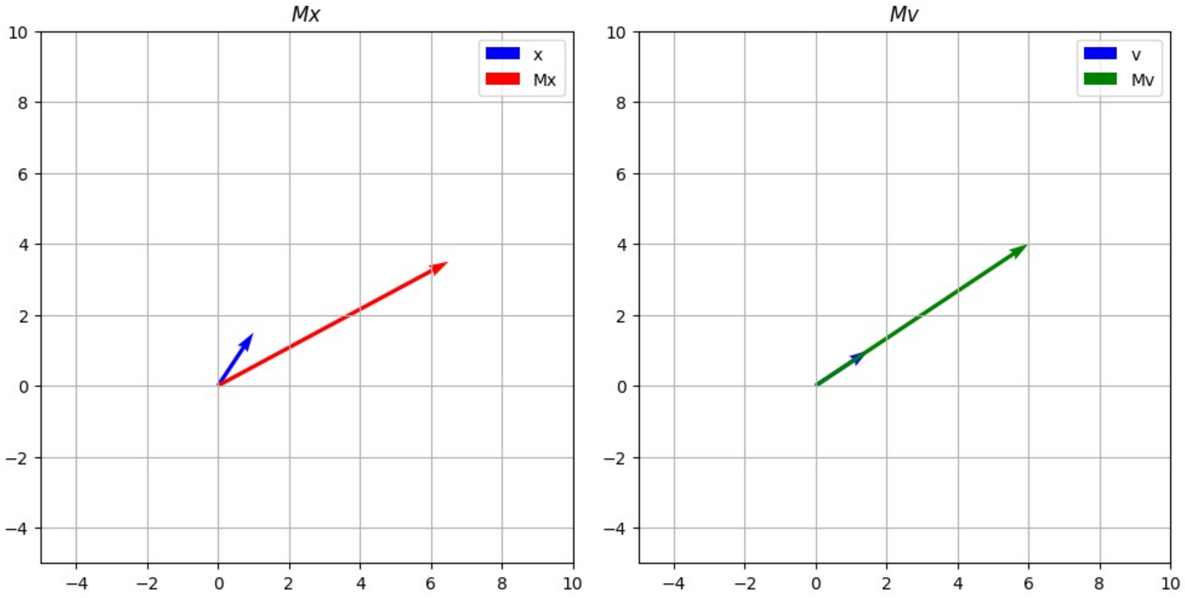
- 행렬-벡터의 곱셈을 기하학적 관점으로 생각하면, 해당 벡터를 회전하고 크기를 조정할 수 있음
- 스칼라-벡터 곱셈의 경우, 크기는 조정 가능하나 회전 시키지 않음

$$x = [1, 1.5], v = [1.5, 1], M = \begin{bmatrix} 2 & 3 \\ 2 & 1 \end{bmatrix}$$



행렬 곱셈

- 왼쪽 그래프의 경우, 행렬 M 이 벡터 x 를 회전하고 늘린 것을 확인할 수 있음
- 오른쪽 그래프의 경우, 행렬-벡터 곱이 크기는 조정되었으나, 방향은 유지됨
- (즉, 행렬-벡터 곱이 스칼라-벡터 곱인 것처럼 작동)
- 벡터 v 는 행렬 M 의 고유 벡터(eigen vector) 이고, M 이 v 의 크기를 키운 양이 고유값(eigen value) 가 됨



전치

- 전치는 벡터에 대한 전치와 마찬가지로, 단순히 말해 **행과 열을 바꾸는 것**
- 행렬을 이중 전치하면 원래 행렬이 됨 $M^{TT} = M$
- 전치 연산의 수학적 정의는

$$a_{i,j}^T = a_{j,i}$$

$$\begin{bmatrix} 3 & 0 & 4 \\ 9 & 8 & 3 \end{bmatrix}^T = \begin{bmatrix} 3 & 9 \\ 0 & 8 \\ 4 & 3 \end{bmatrix}$$

위 수식을 .T 혹은 np.transpose() 를 이용하여 출력해보자

전치

- 크기가 $M \times 1$ 인 두 열 벡터에서
- 첫 번째 벡터를 전치하면 크기가 $1 \times M$ 이고, 두 번째 벡터가 $M \times 1$ 이 됨
- 내부 차원은 일치하고, 외부 차원을 통해서 행렬 곱셈의 결과가 1×1 → 즉, 스칼라가 됨
- 따라서 내적을 $a^T b$ 로도 표시할 수 있음

행렬 연산

- 여러 행렬의 곱셈을 전치할 때는 개별 행렬을 전치하고 곱한 것과 동일하지만, **순서가 바뀜**
- L, I, V, E 가 모두 행렬이며, 곱셈이 가능하도록 행렬의 크기가 맞을 때

$$(LIVE)^T = E^T V^T I^T L^T$$

대칭 행렬

- 대칭 행렬 symmetric matrix 는 대응되는 행과 열이 같은 행렬을 의미함
- 즉, 행과 열을 바꿔도 행렬에는 변화가 없음 → 대칭 행렬은 자신의 전치 행렬과 같음

$$A^T = A$$

✓ 행의 수와 열의 수가 다른 비정방 행렬은 대칭이 될 수 있는가 ?

행렬의 대칭 여부를 확인하는 python 함수를 작성해보자
행렬을 입력받아, 그 행렬이 대칭이면 True, 비대칭이면 False 를 출력

대칭 행렬

- 어떤 행렬이든(비정방 행렬이라도) 자신의 전치를 곱하면 정방 대칭 행렬이 됨
- $A^T A = AA^T$ 모두 정방 대칭 행렬이 됨
- 만약 A 가 $M \times N$ 이라면, A^T 는 $N \times M$, $(M \times N)(N \times M) = M \times M$ 의 정방 행렬이 됨
- $(A^T A)^T = A^T A^{TT} = A^T A$ → $(A^T A)^T = A^T A$ 이 됨
- 결국 행렬이 전치 행렬과 같으므로 대칭 행렬이 됨

위 과정을 구현하여 증명해보자

행렬식

- 행렬식 determinant 은 (1) 정방 행렬에 대해서만 정의되고, (2) 특이 행렬에 대해서는 0 임
- 행렬식은 $\det(A)$ 또는 $|A|$ 로 나타냄
- 행렬과 벡터를 곱할 때, 행렬이 벡터를 얼마나 늘릴 것인가와 연관이 있음
- 음수의 행렬식은 변환 과정에서 하나의 축을 회전 시킴

$$\det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = aei + bfg + cdh - ceg - bdi - afh$$

행렬식

$$\det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = aei + bfg + cdh - ceg - bdi - afh$$

행렬식을 계산하는 python 함수를 구현해보고,
np.linalg.det() 또는 scipy.linaig.det() 과 결과를 비교해보자

역행렬

- 행렬 A의 역행렬은 A와 곱해서 단위 행렬을 만드는 행렬 A^{-1} 을 의미함
- 즉, $A^{-1}A = I$ 로, 행렬을 단위 행렬로 선형 변환하는 것으로 해석할 수 있음
- $Ax = b$ 형태의 문제에서 A와 b 를 이미 알고 있다면, x 를 구하기 위해서 행렬의 변환을 취소해야함

$$Ax = b$$

$$A^{-1}Ax = A^{-1}b$$

$$Ix = A^{-1}b$$

$$x = A^{-1}b$$

역행렬

- 역행렬을 계산하는 방법은 아래와 같음
- 2x2 행렬의 역을 구하려면, 대각 원소를 교환하고, 대각이 아닌 원소에 -1을 곱한 다음, 행렬식으로 나누어야 함

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

$$A^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

$$AA^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \frac{1}{ad - bc} \begin{bmatrix} ad - bc & 0 \\ 0 & ad - bc \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

역행렬

- 역행렬을 계산하는 방법은 아래와 같음
- 2x2 행렬의 역을 구하려면, 대각 원소를 교환하고, 대각이 아닌 원소에 -1을 곱한 다음, 행렬식으로 나누어야 함

$$\begin{bmatrix} 1 & 4 \\ 2 & 7 \end{bmatrix} \begin{bmatrix} 7 & -4 \\ -2 & 1 \end{bmatrix} \frac{1}{7-8} = \begin{bmatrix} (7-8) & (-4+4) \\ (14-14) & (-8+7) \end{bmatrix} \frac{1}{-1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

```
A = np.array([ [1,4], [2,7] ])

Ainv = np.linalg.inv(A)

A@Ainv
```

역행렬

- 역행렬의 역행렬은 원래 행렬이 되는지 Python 을 이용하여 증명해보자

$$(A^{-1})^{-1} = A$$

직교 행렬

- 직교 행렬은 orthogonal matrix 은 행렬의 모든 열은 서로 직교하며, 각 열의 norm 은 1 임

$$< q_i, q_j > = \begin{cases} 0, if\ i \neq j \\ 1, if\ i = j \end{cases}$$

- 모든 열은 자기자신과의 내적은 1이지만 다른 열과의 내적은 0 (직교)
- 행렬의 왼쪽으로 그 행렬의 전치를 곱하면 열들 사이의 모든 내적을 구할 수 있음
- Q^T의 행은 Q의 열과 같고 행렬 곱셈은 왼쪽 행렬의 모든 행과 오른쪽 행렬의 모든 열 사이 내적으로 이루어짐

직교 행렬

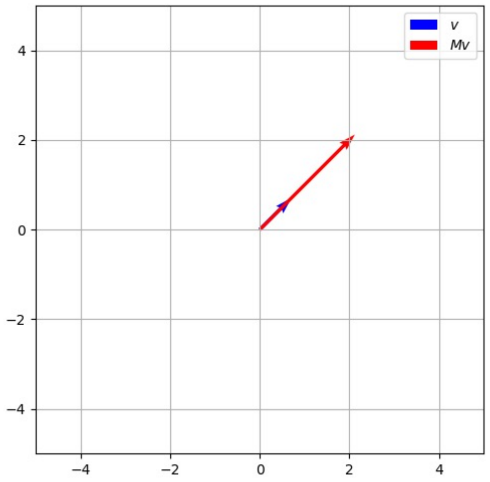
- 직교행렬의 정의인 각 열의 norm 이 1이고, 다른 열과 직교하는지 확인해보자

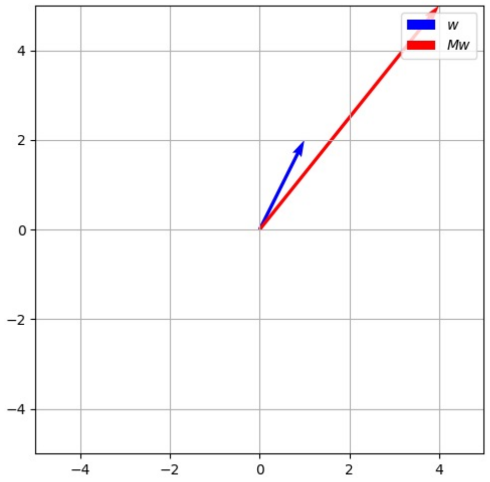
```
Q1 = np.array([[1,-1],[1,1]]) / np.sqrt(2)
Q2 = np.array([[1,2,2],[2,1,-2],[-2,2,-1]])/3


print(Q1.T@Q1)
print(Q2.T@Q2)
```

고유값 분해 eigen decomposition

- 고윳값 분해는 정방 행렬에 대해서만 정의되며, $M \times N$ 크기의 모든 정방 행렬에는 M개의 고유값(스칼라)과 M 에 대응되는 고유벡터가 존재함
- 행렬과 벡터가 특수하게 결합하면 행렬이 벡터의 크기는 증가시키지만, 방향을 바꾸지는 않음
- 이 벡터가 행렬의 고유 벡터이며, 늘어나는 양이 고유값을 의미함





고유값 분해 eigen decomposition

- 고유벡터는 행렬-벡터 곱셈이 스칼라-벡터 곱셈처럼 작동하는 것을 의미하였으며,
- 이를 수식으로 나타내면 $Av = \lambda v$

- 행렬의 고유값을 구하기 위해서는

$$Av = \lambda v$$

$$Av - \lambda v = 0$$

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

고유값 분해 eigen decomposition

- 고유값 찾기

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

$$\det\left(\begin{bmatrix} 4-\lambda & 2 \\ 1 & 3-\lambda \end{bmatrix}\right) = 0$$

$$(4-\lambda)(3-\lambda) - 2 \cdot 1 = 0$$

$$(4-\lambda)(3-\lambda) - 2 = 0$$

$$(12 - 7\lambda + \lambda^2) - 2 = 0$$

$$\lambda^2 - 7\lambda + 10 = 0$$

$$\therefore \lambda_1 = 5, \lambda_2 = 2$$

고유값 분해 eigen decomposition

- $\lambda_1 = 5$ 에 대한 고유벡터 찾기

$$(A - 5I)v = 0$$

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

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

$$-x + 2y = 0$$

$$x = 2y$$

$$v_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

고유값 분해 eigen decomposition

- $\lambda_2 = 2$ 에 대한 고유벡터 찾기

$$(A - 2I)v = 0$$

$$\left(\begin{bmatrix} 4 & 2 \\ 1 & 3 \end{bmatrix} - 2\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) = \begin{bmatrix} 2 & 2 \\ 1 & 1 \end{bmatrix} v = 0$$

$$2x + 2y = 0$$

$$x = -y$$

$$v_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

고유값 분해 eigen decomposition

- 정방 행렬을 고유값 분해하기 위해서는 고유값을 찾은 다음, 각 고유값을 사용하여 고유벡터를 찾음
- python 의 np.linalg.eig() 를 이용하여 행렬의 고유값을 찾아보자

```
M = np.array([[4, 2], [1, 3]])
np.linalg.eig(M)
```

Mini Project

- (vibe coding) 주어진 데이터(붓꽃 iris 데이터)를 PCA를 통해 분산을 최대화하는 새로운 축으로 변환, 정보 손실을 최소화하면서 차원을 축소하는 과정을 구현해보자

1. 데이터의 centering
2. 공분산 행렬 계산
3. 공분산 행렬의 고유값과 고유벡터 계산
4. 고유값이 큰 순서대로 고유 벡터를 정렬
5. 선택한 주성분(고유벡터) 방향으로 원본 데이터를 투영하여 차원을 축소
6. 원본 데이터와 축소된 데이터의 분포를 시각화