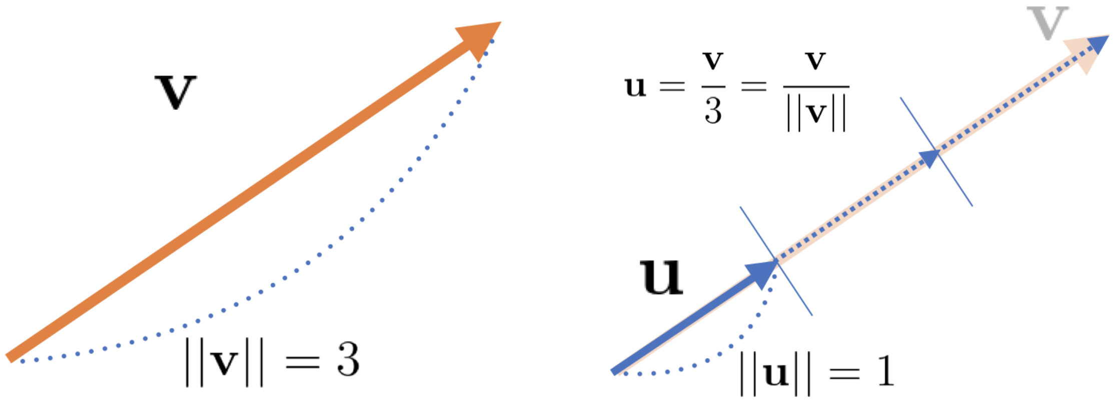
게임 수학 – 강의 2

동명대학교 게임공학과
강영민

벡터의 정규화

- 단위벡터unit vector
  - 길이가 1인 벡터
  - 서로 다른 단위벡터는 방향만이 중요하다
    - 단위벡터 = 방향벡터

- 정규화normalization
  - 벡터를 단위벡터로 만드는 일
    - 벡터의 방향만을 찾는 일
    - 벡터의 길이를 1로 만드는 것과 같다



벡터의 정규화

- 연산

$\mathbf{x}$를 단위 벡터 $\tilde{\mathbf{x}}$로 만드는 정규화는 다음과 같다.

$$\tilde{\mathbf{x}} = \frac{\mathbf{x}}{||\mathbf{x}||}$$

$$\mathbf{x} = (x_1, x_2, \cdots, x_n)$$

$$l_{\mathbf{x}} = ||\mathbf{x}||_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

$$\hat{\mathbf{x}} = \mathbf{x}/l_{\mathbf{x}} \quad , \quad \mathbf{x} = l_{\mathbf{x}}\hat{\mathbf{x}}$$

$$\hat{\mathbf{x}} = \left(\frac{x_1}{\sqrt{\sum_{i=1}^{n} x_i^2}}, \frac{x_2}{\sqrt{\sum_{i=1}^{n} x_i^2}}, \cdots, \frac{x_n}{\sqrt{\sum_{i=1}^{n} x_i^2}} \right) = \left(\frac{x_1}{l_{\mathbf{x}}}, \frac{x_2}{l_{\mathbf{x}}}, \cdots, \frac{x_n}{l_{\mathbf{x}}} \right)$$

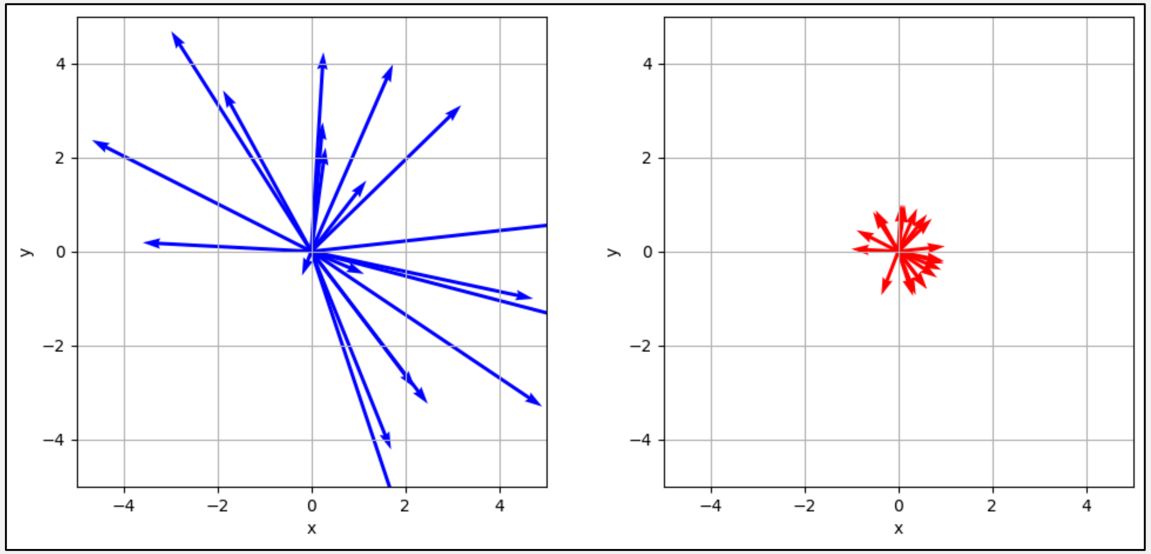
벡터/벡터정규화 가시화

- **벡터 생성하기**

```
import numpy as np
import matplotlib.pyplot as plt
n_vectors = 20 # 총 30 개의 벡터를 생성할 예정
vectors = np.random.randn(n_vectors, 2) * 3 # 생성된 난수에 3을 곱해 키움
```

벡터/벡터 정규화 가시화

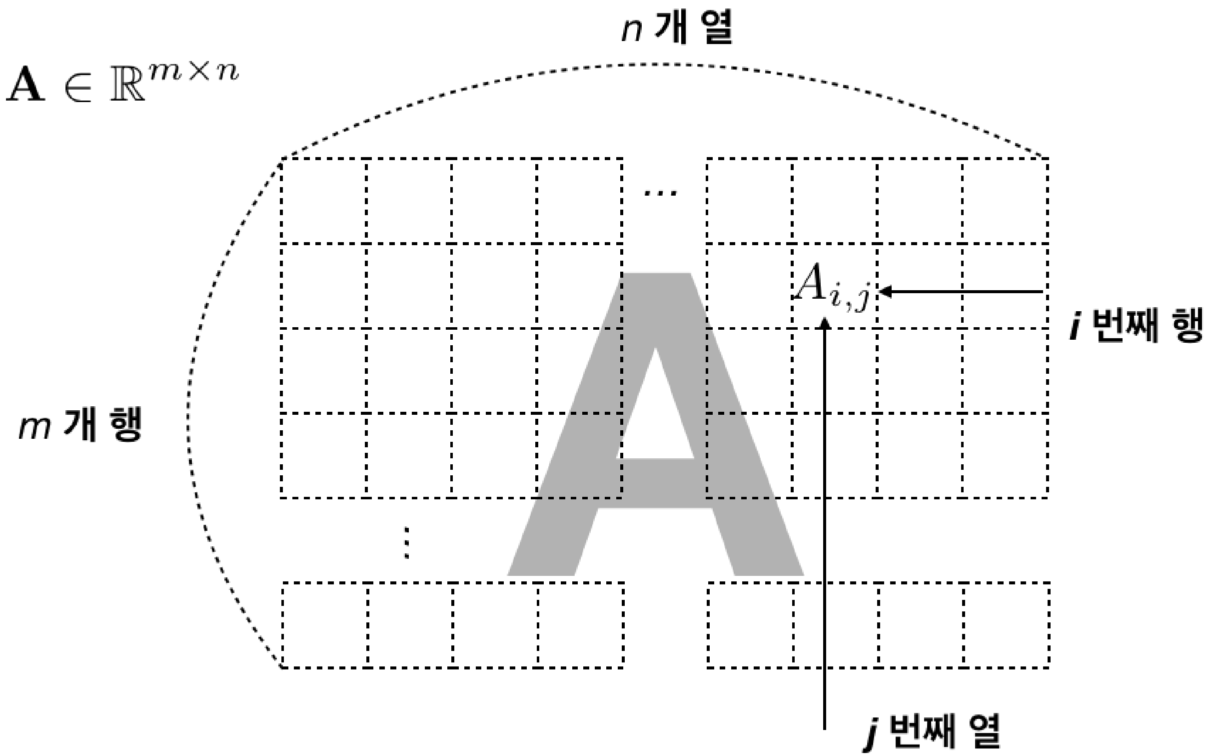
```
# 벡터 시각화
origin = np.zeros(2)  # 원점 (0, 0)

fig = plt.figure(figsize = (10, 5))      # 그림의 크기가 가로 10인치, 세로 5인치
ax1 = fig.add_axes([0.1, 0.1, 0.4, 0.8]) # ax1은 그림 내의 (0.1, 0.1)에서 가로 0.4, 세로 0.8 크기 차지
ax2 = fig.add_axes([0.6, 0.1, 0.4, 0.8]) # ax1은 그림 내의 (0.6, 0.1)에서 가로 0.4, 세로 0.8 크기 차지

ax1.set_xlim([-5, 5])
ax1.set_ylim([-5, 5])
ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.grid(True)

ax2.set_xlim([-5, 5])
ax2.set_ylim([-5, 5])
ax2.set_xlabel('x')
ax2.set_ylabel('y')
ax2.grid(True)
```




```
for v in vectors:
    ax1.quiver(*origin, *v, angles='xy', scale_units='xy', scale=1, color='b')
    v_normalized = v / np.linalg.norm(v)
    ax2.quiver(*origin, *v_normalized, angles='xy', scale_units='xy', scale=1, color='r')
```

행렬 데이터의 이해

- 행렬은 2차원으로 배열된 숫자



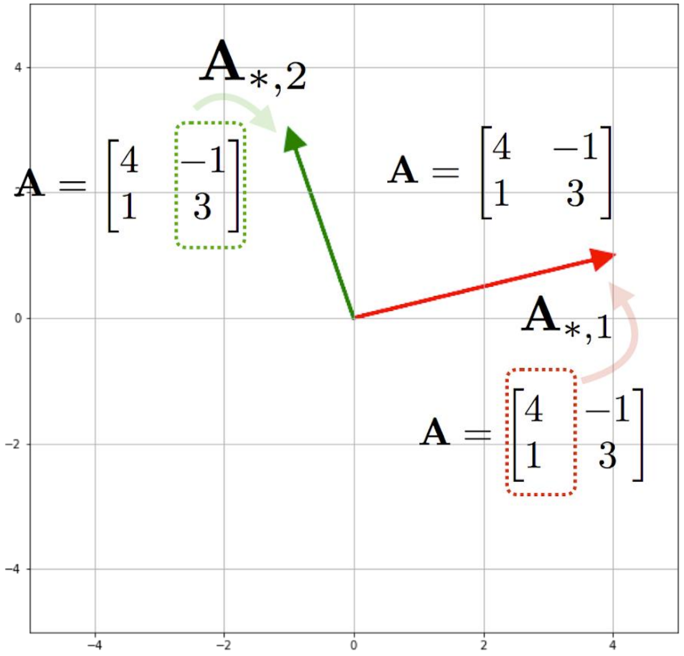
$$\mathbf{A} \in \mathbb{R}^{m \times n}$$

행렬은 언제부터 사용했나?

- 행렬은 1차 방정식의 풀이에 아주 오래 전부터 사용
  - 그 특성이 정확히 파악되지 않았음
- 1800년대까지는 배열array이라는 이름으로 알려짐
- 구장산술九章算術– 기원전 10세기~ 기원전 2세기에 걸쳐 쓰여짐
  - 연립 방정식을 풀기 위해 배열을 적용하는 예가 처음으로 소개
  - 판별식 개념도 등장
  - 유럽에서는 1545년에 알려짐
    - 이탈리아 수학자 지롤라모 카르다노Girolamo Cardano
    - 위대한 기술Ars Magna를 통해 이 기법을 유럽에 전함

행렬의 가시화

- 가시화하기 쉬운 행렬
  - 2차원 공간에 그려질 수 있는 행렬
    - 2x2 행렬
    - 2개의 2차원 벡터가 존재
- 행렬의 가시화
  - 2차원 벡터들을 2차원 공간에 그림



행렬은
여러 벡터가 모여 있는 것으로
이해할 수 있다.

행렬의 모습을 가시화하는 것은
이들 벡터를 각각 그리면 된다.

왼쪽의 두 화살표가 바로
우리가 처음으로
행렬의 모양을 눈으로
확인할 수 있는 이미지이다.

**이런 행렬이 무슨 놀라운 일을 하는지는
나중에 알아보자**

행렬의 가시화 – 행렬 생성

```
mat = np.array([ [3,-1],
                 [5,4]  ])

mat[0,:], mat[1,:], mat[:,0], mat[:,1]
```

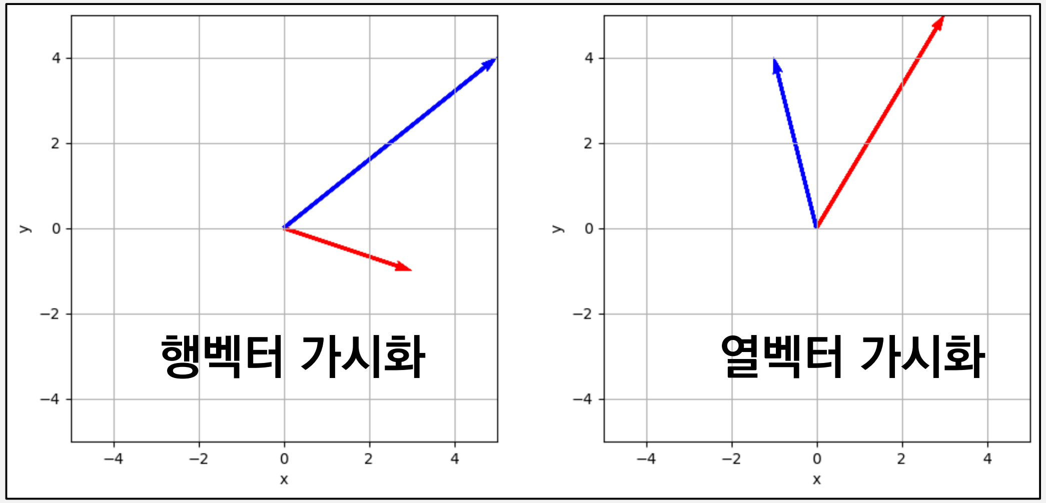
3	−1
5	4

```
(array([ 3, -1]), array([5, 4]), array([3, 5]), array([-1, 4]))
```

```
mat[0,:]   0행 벡터   (3,-1)
mat[1,:]   1행 벡터   (5,4)
mat[:,0]   0열 벡터   (3,5)
mat[:,1]   1열 벡터   (-1,4)
```

행렬의 가시화

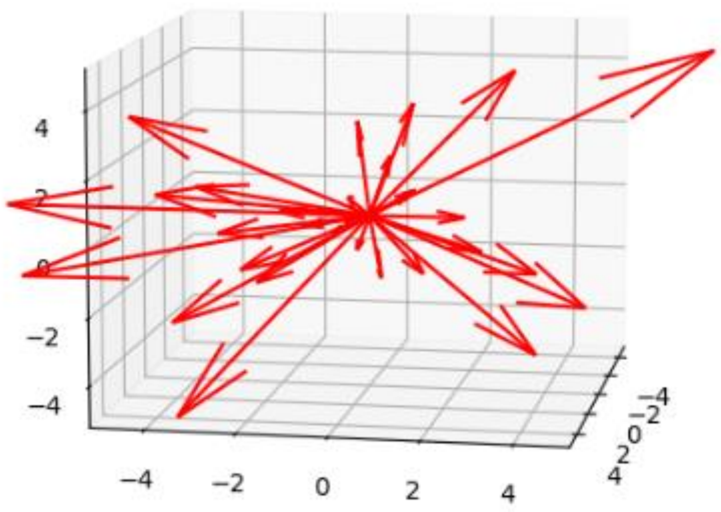
```
# 행렬 시각화
origin = np.zeros(2)  # 원점 (0, 0)

fig = plt.figure(figsize = (10, 5))      # 그림의 크기가 가로 10인치, 세로 5인치
ax1 = fig.add_axes([0.1, 0.1, 0.4, 0.8]) # ax1은 그림 내의 (0.1, 0.1)에서 가로 0.4, 세로 0.8 크기 차지
ax2 = fig.add_axes([0.6, 0.1, 0.4, 0.8]) # ax1은 그림 내의 (0.6, 0.1)에서 가로 0.4, 세로 0.8 크기 차지
ax1.set_xlim([-5, 5])
ax1.set_ylim([-5, 5])
ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.grid(True)

ax2.set_xlim([-5, 5])
ax2.set_ylim([-5, 5])
ax2.set_xlabel('x')
ax2.set_ylabel('y')
ax2.grid(True)

for v in vectors:
    # 행벡터 가시화
    ax1.quiver(*origin, *mat[0,:], angles='xy', scale_units='xy', scale=1, color='r')
    ax1.quiver(*origin, *mat[1,:], angles='xy', scale_units='xy', scale=1, color='b')

    # 열벡터 가시화
    ax2.quiver(*origin, *mat[:,0], angles='xy', scale_units='xy', scale=1, color='r')
    ax2.quiver(*origin, *mat[:,1], angles='xy', scale_units='xy', scale=1, color='b')
```



3차원 벡터/벡터정규화 가시화

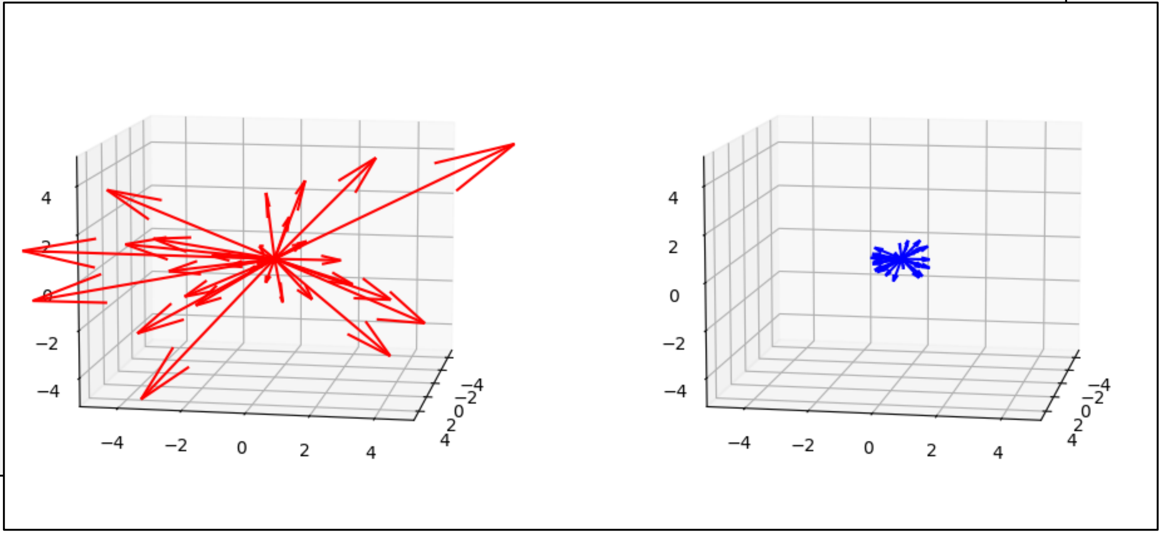
- 벡터 생성하기

```
Vec3Ds = np.random.randn(30, 3) * 3
```



벡터/벡터 정규화 가시화

```
# 3D 벡터 시각화

origin3d = np.zeros(3) # 원점 (0, 0)
fig = plt.figure(figsize = (10, 5))      # 그림의 크기가 가로 10인치, 세로 5인치
ax1 = fig.add_axes([0.1, 0.1, 0.4, 0.8], projection='3d')
ax2 = fig.add_axes([0.6, 0.1, 0.4, 0.8], projection='3d')

ax1.set_xlim([-5, 5]);ax1.set_ylim([-5, 5]);ax1.set_zlim([-5, 5])
ax1.grid(True)

ax2.set_xlim([-5, 5]);ax2.set_ylim([-5, 5]);ax2.set_zlim([-5, 5])
ax2.grid(True)


for v in Vec3Ds:
    # 3D 벡터 가시화
    ax1.quiver(*origin3d, *v, color='r')
    vNormalized = v / np.linalg.norm(v)
    ax2.quiver(*origin3d, *vNormalized, color='b')

ax1.view_init(10,10)
ax2.view_init(10,10)
```



3x3 행렬의 가시화

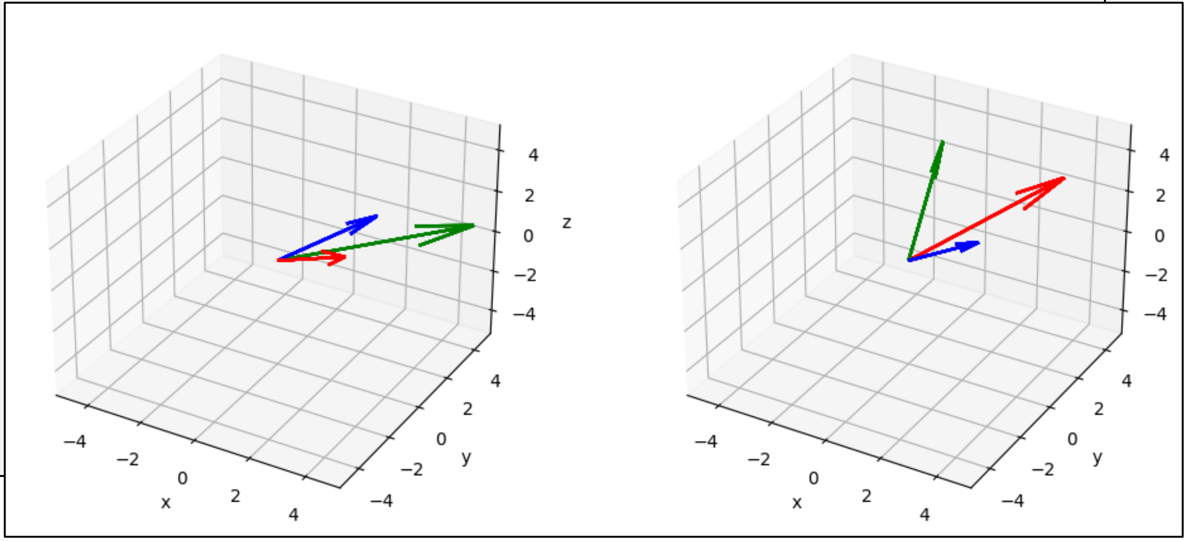
```
mat = np.array([ [3,-1, 2],
                 [5, 4, 1],
                 [2, 3, 1] ])

mat[0,:], mat[1,:], mat[2,:], mat[:,0], mat[:,1], mat[:,2]
```

3	−1	2
5	4	1
2	3	1

```
(array([ 3, -1, 2]),
 array([5, 4, 1]),
 array([2, 3, 1]),
 array([3, 5, 2]),
 array([-1, 4, 3]),
 array([2, 1, 1]))
```

3x3 행렬의 가시화

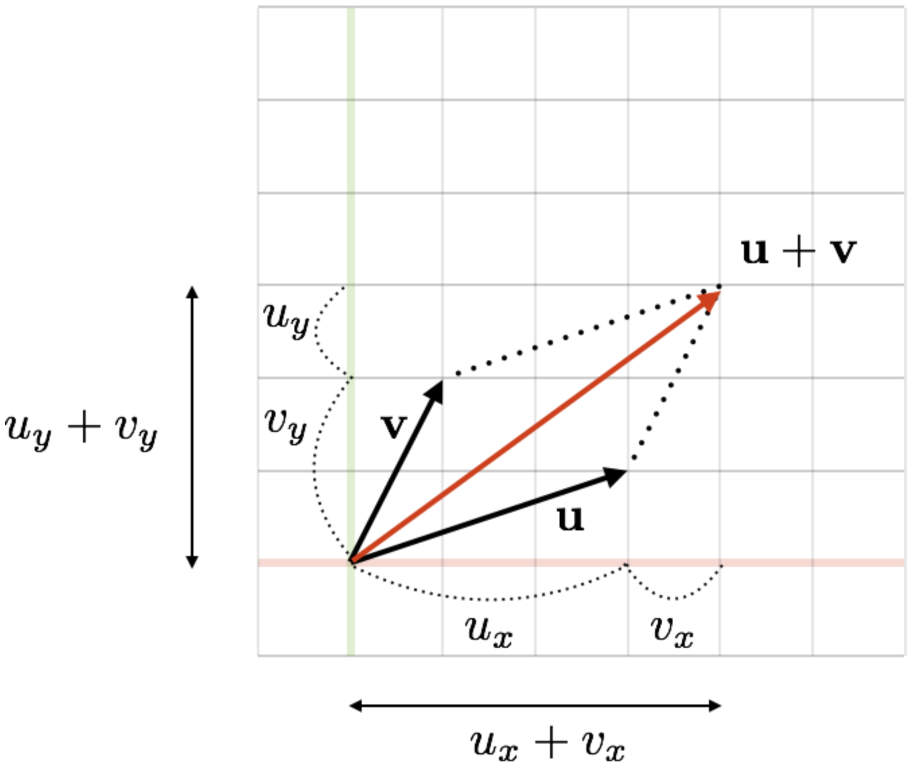
```
# 3x3 행렬 시각화

origin3d = np.zeros(3) # 원점 (0, 0)

fig = plt.figure(figsize = (10, 5))      # 그림의 크기가 가로 10인치, 세로 5인치
ax1 = fig.add_axes([0.1, 0.1, 0.4, 0.8], projection='3d')
ax2 = fig.add_axes([0.6, 0.1, 0.4, 0.8], projection='3d')

ax1.set_xlim([-5, 5]); ax1.set_ylim([-5, 5]); ax1.set_zlim([-5, 5]);
ax1.set_xlabel('x'); ax1.set_ylabel('y'); ax1.set_zlabel('z');
ax1.grid(True)

ax2.set_xlim([-5, 5]); ax2.set_ylim([-5, 5]); ax2.set_zlim([-5, 5]);
ax2.set_xlabel('x'); ax2.set_ylabel('y'); ax2.set_zlabel('z');
ax2.grid(True)

for v in vectors:
    # 행벡터 가시화
    ax1.quiver(*origin3d, *mat[0,:], color='r')
    ax1.quiver(*origin3d, *mat[1,:], color='g')
    ax1.quiver(*origin3d, *mat[2,:], color='b')

    # 열벡터 가시화
    ax2.quiver(*origin3d, *mat[:,0], color='r')
    ax2.quiver(*origin3d, *mat[:,1], color='g')
    ax2.quiver(*origin3d, *mat[:,2], color='b')
```



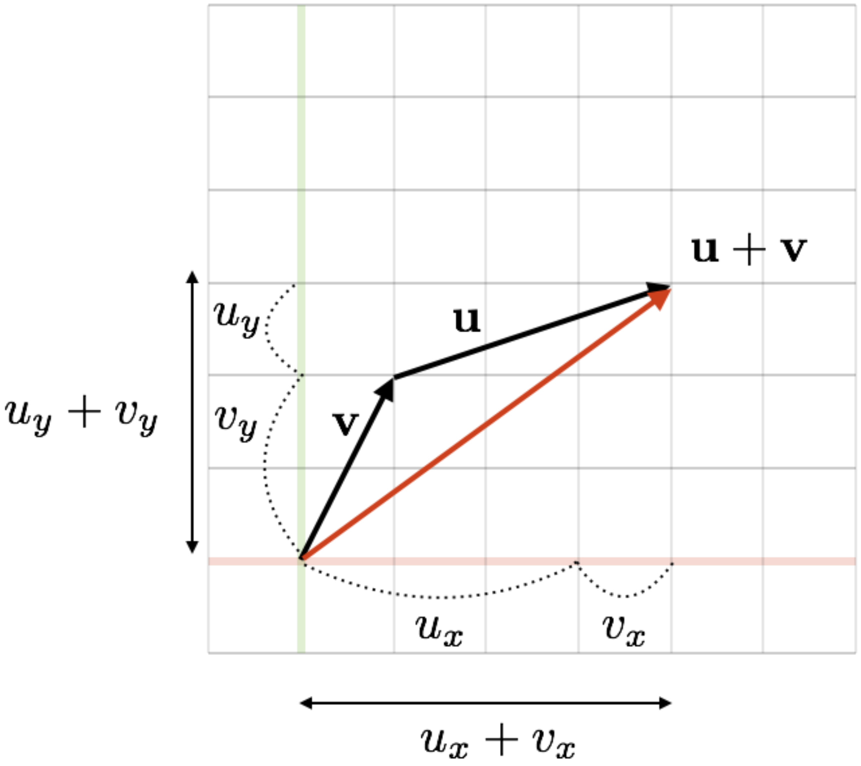
벡터의 연산 – 덧셈 $\mathbf{w} = \mathbf{u} + \mathbf{v}$

$$\mathbf{w} = (u_x + v_x,\ u_y + v_y,\ u_z + v_z)$$

평행사변형을 이용한 벡터 합 가시화

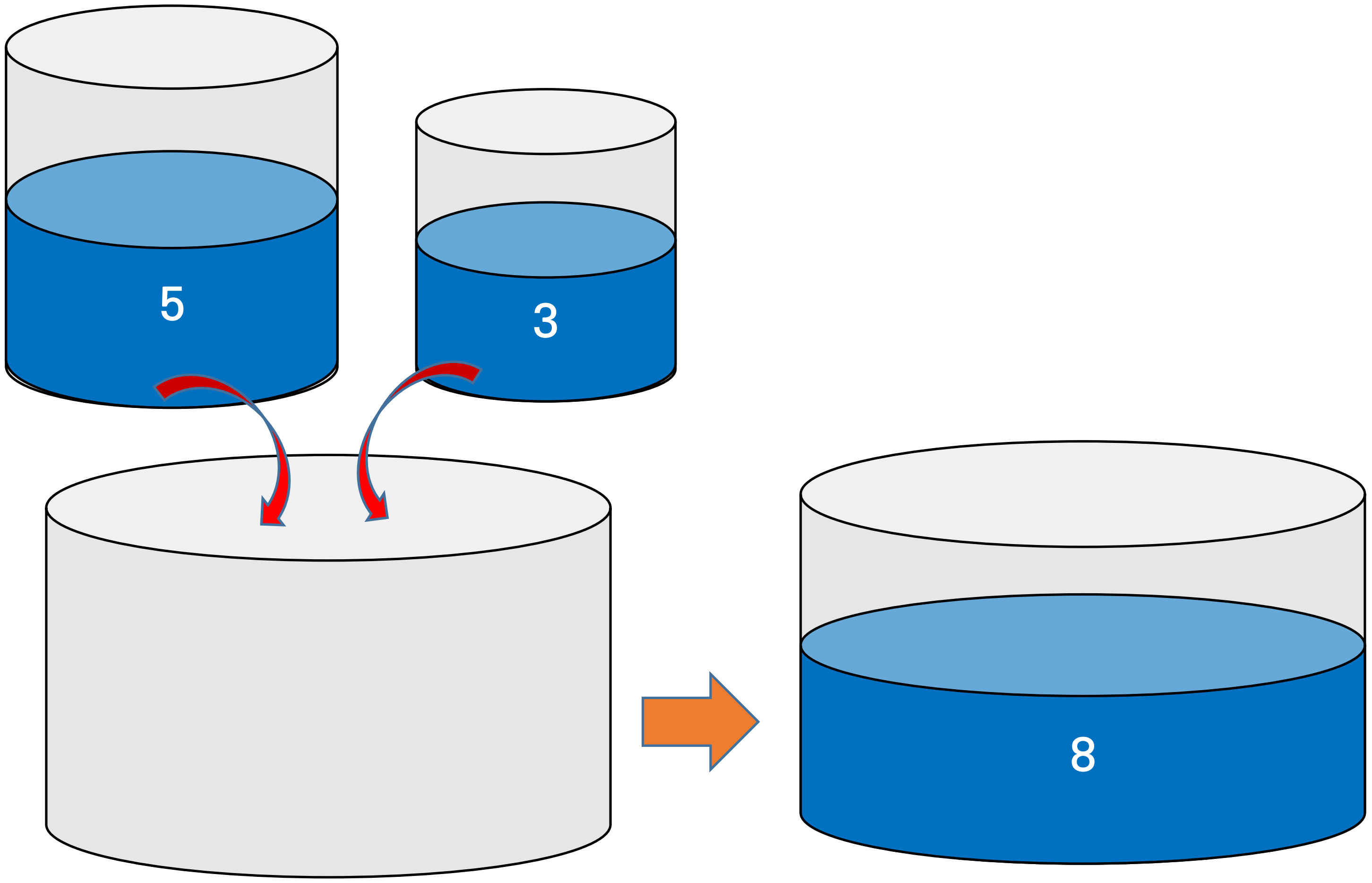


이어지는 벡터를 이용한 벡터 합 가시화



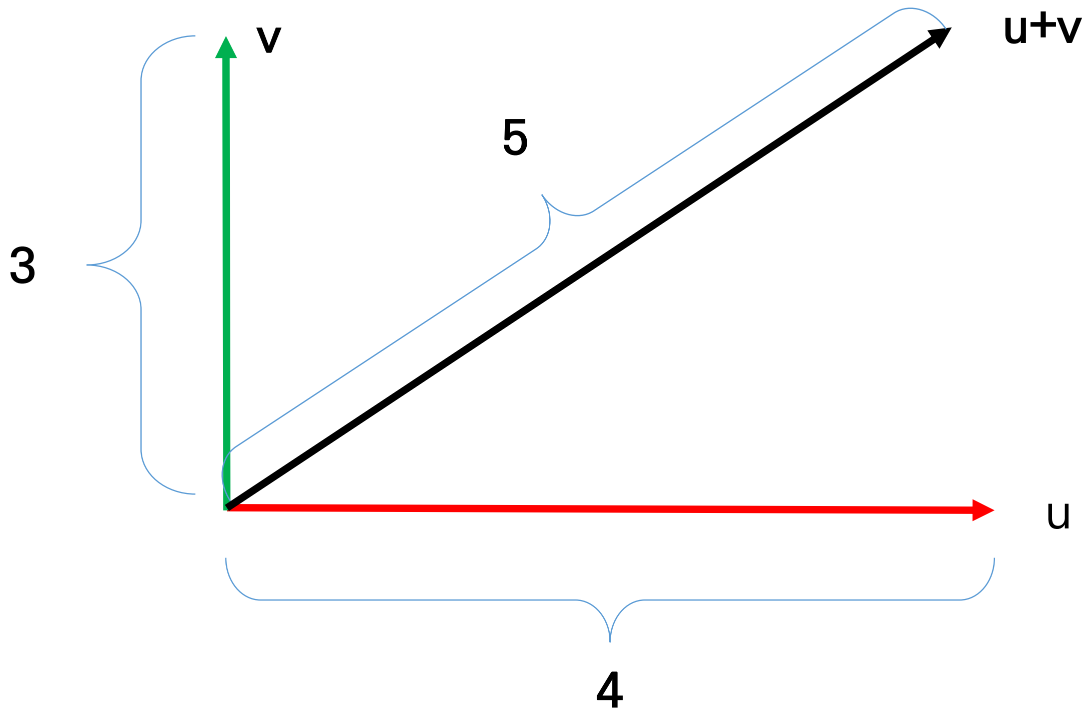
스칼라 덧셈과 벡터 덧셈의 차이

- 스칼라 덧셈
  - 크기의 모아짐
    - 5 + 3 = 8



스칼라 덧셈과 벡터 덧셈의 차이

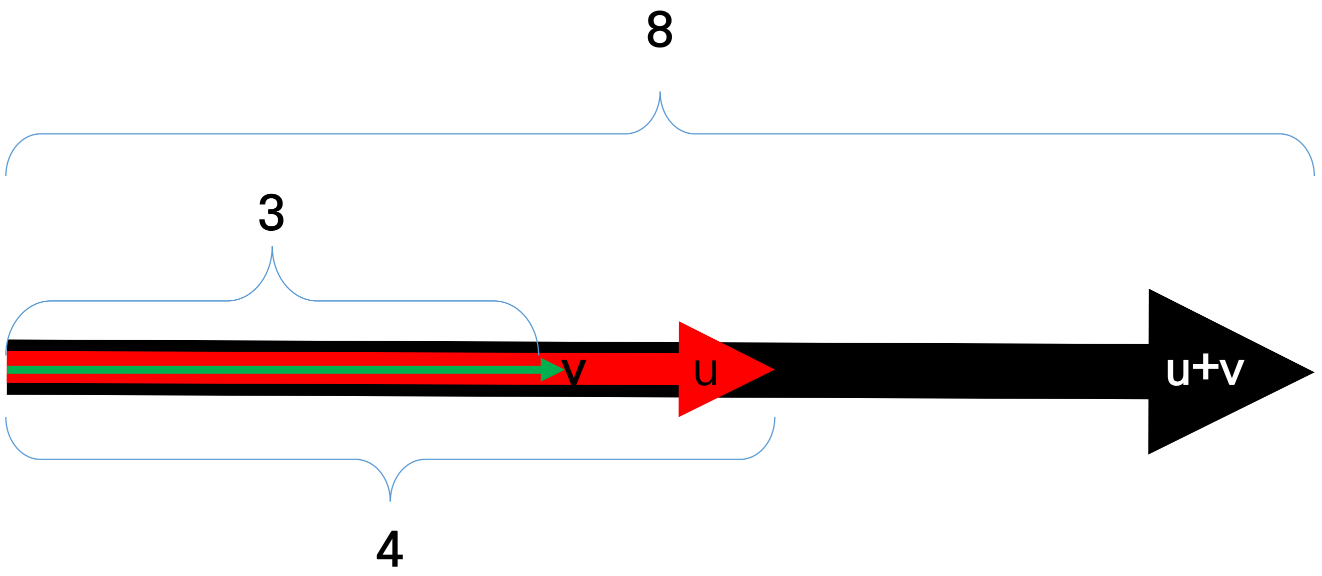
- 벡터의 덧셈
  - 크기가 모아지지 않음



- 벡터는 크기 뿐만 아니라 방향이 존재
- 방향이 덧셈에 영향을 미침
  - 크기가 모아지는 데에 영향을 미침
  - 어떤 방향으로 크기가 다 모아지고
  - 어떤 방향으로는 서로 상쇄되기도 함

스칼라 덧셈과 벡터 덧셈의 차이

- 벡터의 덧셈에서 크기를 최대로 모을 수 있는 경우

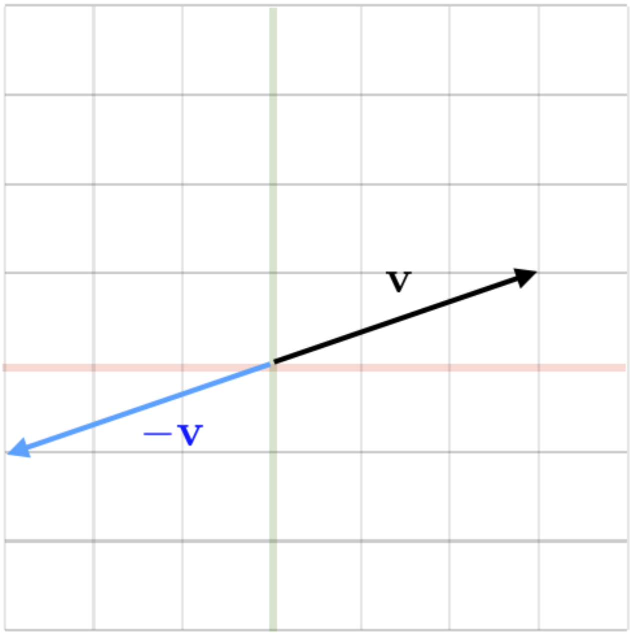


- 같은 방향이면 크기가 그대로 모아짐
- 스칼라 덧셈
  - 같은 방향을 가진 벡터의 덧셈으로 이해할 수 있음

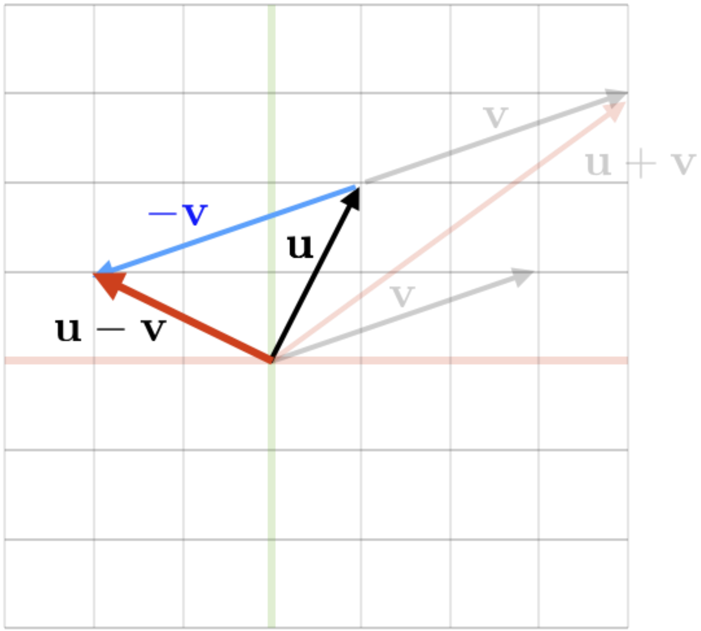
벡터의 연산 – 뺄셈 $\mathbf{w} = \mathbf{u} - \mathbf{v}$

$$\mathbf{w} = (u_x - v_x,\ u_y - v_y,\ u_z - v_z)$$

음수 부호가 붙은 벡터

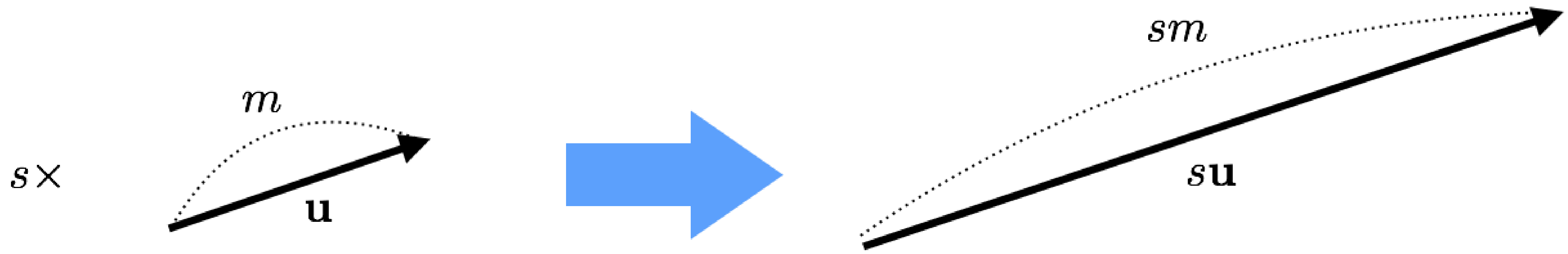


벡터의 뺄셈



벡터의 연산 – 스칼라 곱

$$\mathbf{u} = (u_1, u_2, \cdots, u_n) \qquad s\mathbf{u} = (su_1, su_2, \cdots, su_n)$$



벡터의 기본적인 연산 규칙

$\mathbf{u}+\mathbf{v}=\mathbf{v}+\mathbf{u}$	덧셈과 뺄셈에 대한 교환 법칙
$(\mathbf{u}+\mathbf{v})+\mathbf{w}=\mathbf{u}+(\mathbf{v}+\mathbf{w})$	덧셈과 뺄셈에 대한 결합 법칙
$\mathbf{u}+\mathbf{0}=\mathbf{u}$	덧셈과 뺄셈에 대한 항등원인 0 벡터
$\mathbf{u}+(-\mathbf{u})=\mathbf{0}$	벡터에서 자기 자신을 빼면 0 벡터
$(k+l)\mathbf{u}=k\mathbf{u}+l\mathbf{u}$	스칼라 덧셈에 대해 벡터 곱의 분배 법칙
$(kl)\mathbf{u}=k(l\mathbf{u})$	스칼라들의 곱과 벡터 곱하기 사이의 결합 법칙
$1\mathbf{u}=\mathbf{u}$	벡터에 스칼라 1을 곱하면 자기 자신
$0\mathbf{u}=\mathbf{0}$	벡터에 스칼라 0을 곱하면 0 벡터
$(-1)\mathbf{u}=-\mathbf{u}$	벡터에 스칼라 -1을 곱하면 반대 방향의 벡터

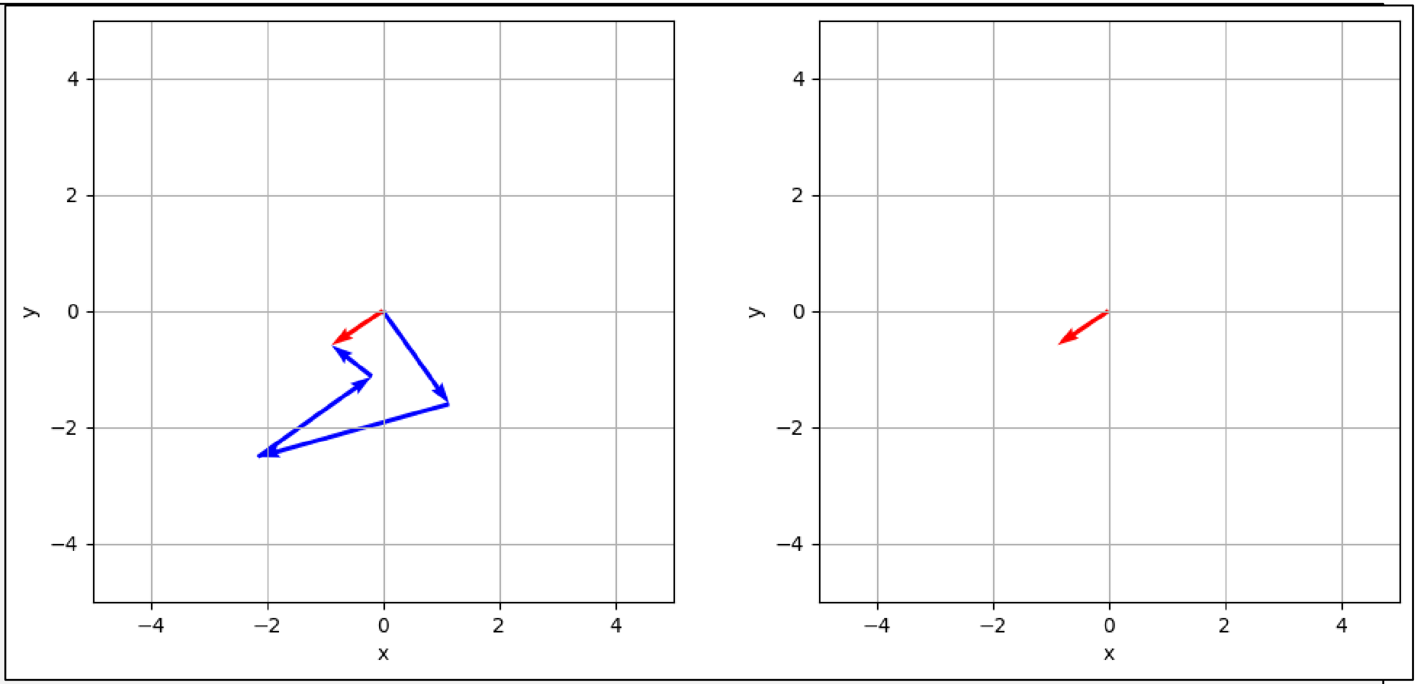
여러 벡터의 합

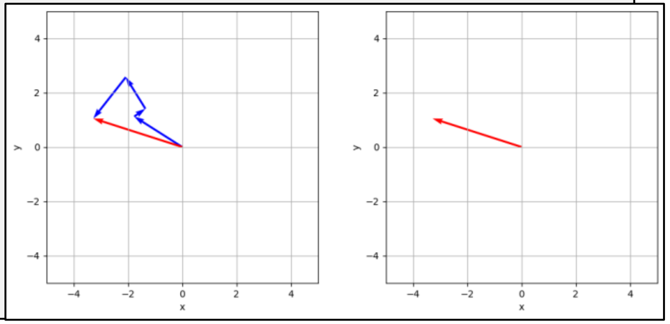
```python
import numpy as np
import matplotlib.pyplot as plt

vectors = np.random.randn(4, 2)
print(vectors)
```

```
[[ 1.14338362 -1.60154362]
 [-3.29528862 -0.91225096]
 [ 1.95917493 1.38633679]
 [-0.69085441 0.53807552]]
```

여러 벡터의 합

```python
# 벡터합 시각화
origin = np.zeros(2)  # 원점 (0, 0)

fig = plt.figure(figsize = (10, 5))
ax1 = fig.add_axes([0.1, 0.1, 0.4, 0.8])
ax2 = fig.add_axes([0.6, 0.1, 0.4, 0.8])
ax1.set_xlim([-5, 5]); ax1.set_ylim([-5, 5])
ax1.set_xlabel('x'); ax1.set_ylabel('y')
ax1.grid(True)

ax2.set_xlim([-5, 5]); ax2.set_ylim([-5, 5])
ax2.set_xlabel('x'); ax2.set_ylabel('y')
ax2.grid(True)

start = origin
v_sum = np.zeros(2)
for v in vectors:
    ax1.quiver(*start, *v, angles='xy', scale_units='xy', scale=1, color='b')
    start = start + v

    v_sum = v_sum + v

ax1.quiver(*origin, *v_sum, angles='xy', scale_units='xy', scale=1, color='r')
ax2.quiver(*origin, *v_sum, angles='xy', scale_units='xy', scale=1, color='r')
```





벡터의 곱

- 덧셈과 뺄셈은 성분끼리 더하거나 빼면 되었음
  - 벡터들 사이의 곱하기도 그렇게 할 수 있는가?
  - 이러한 곱하기를 아다마르Hadamar 곱이라고 한다

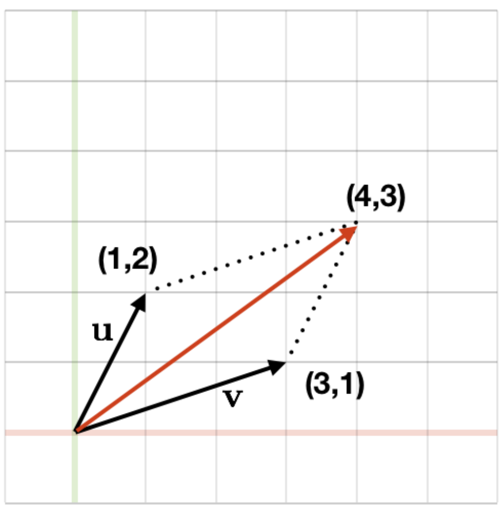
$$\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$$

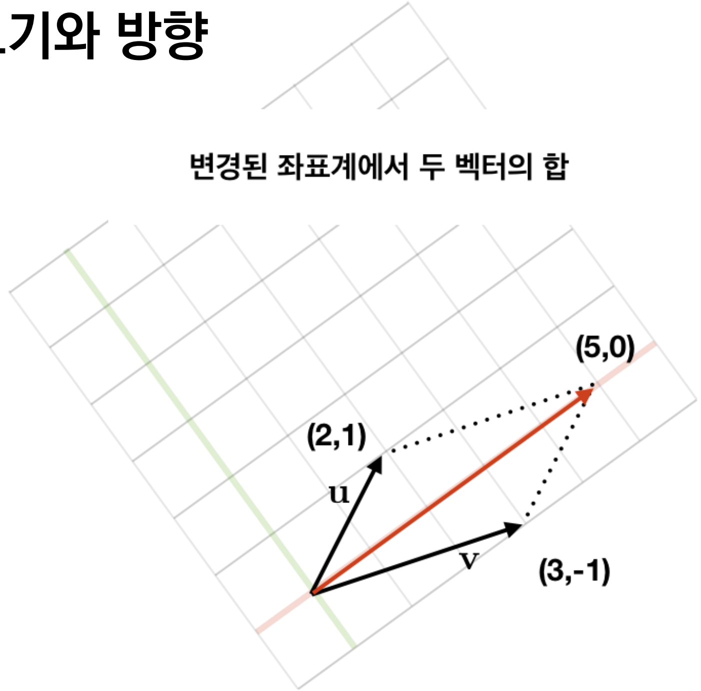
$$w_i = u_i v_i$$

$$\mathbf{w} = (u_1 v_1, u_2 v_2, u_3 v_3, \cdots, u_n v_n)$$

벡터의 아다마르 곱

- 곱셈으로서의 성질이 부족
  - 좌표계 의존적 성질
    - 불변성이 유지되지 않는다
- 불변성이란 무엇일까?
  - 덧셈을 다시 보자
    - 좌표계 변경에도 덧셈 결과 벡터는 동일한 크기와 방향

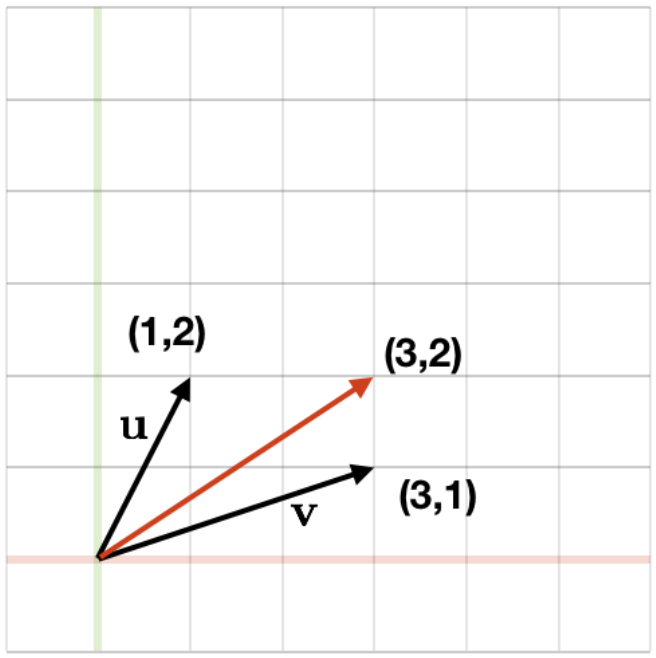




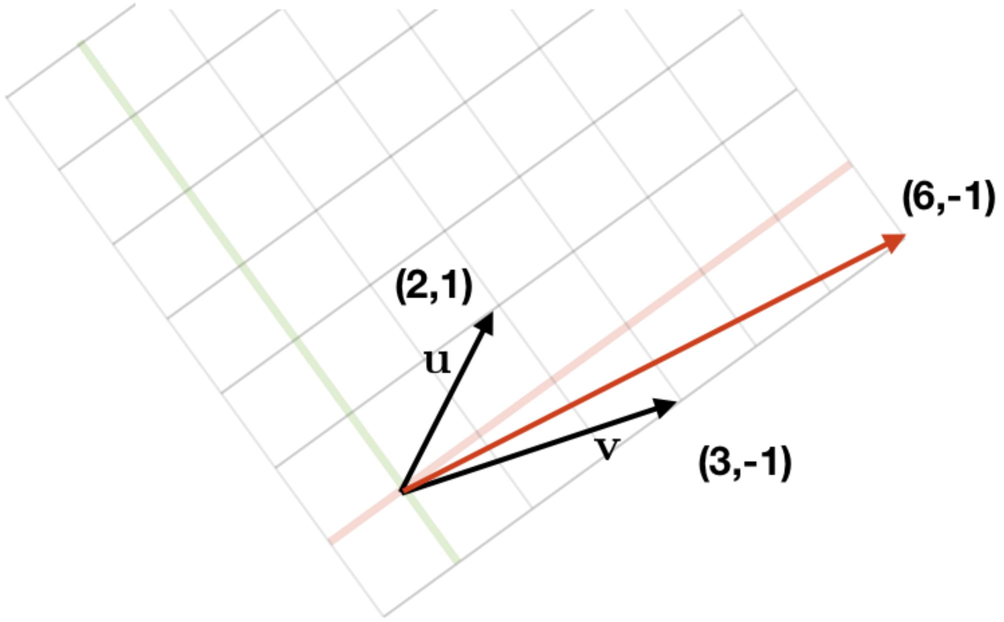
벡터의 아다마르 곱

- 서로 다른 좌표계에서 두 벡터의 아다마르 곱
    - 다른 크기와 방향을 가진 벡터를 만들어낼 수 있다
    - (약속에 불과한) 좌표계 의존적인 연산

어떤 좌표 공간에서 두 벡터의 아다마르 곱



변경된 좌표계에서 두 벡터의 아다마르 곱



좌표계에 의존적이지 않은 벡터 곱

- 점곱
  - 내적, 스칼라곱이라고도 함
  - 두 벡터를 곱해 스칼라 값을 얻을 수 있음
  - 좌표계가 변해도 동일한 스칼라 결과

$$s = \mathbf{u} \cdot \mathbf{v} = \mathbf{u}^{\mathrm{T}}\mathbf{v}$$

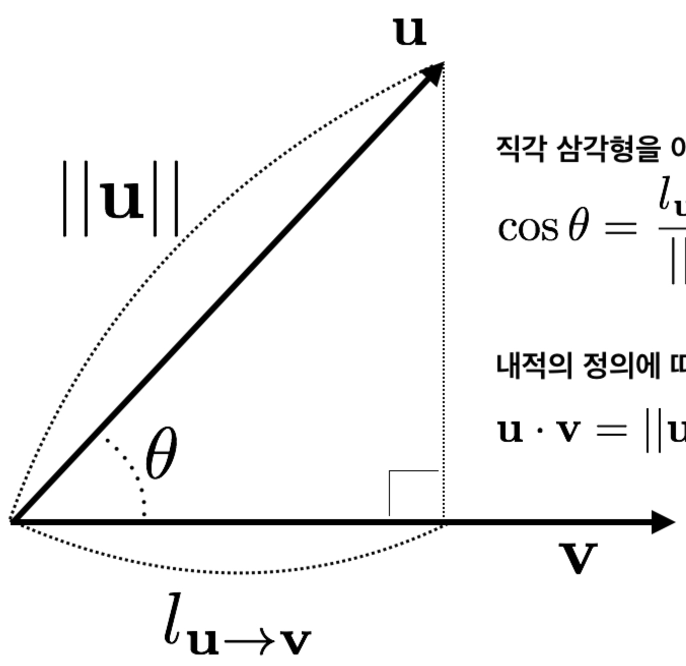
- 가위곱
  - 외적, 벡터곱이라고도 함
  - 두벡터를 곱해 새로운 벡터를 얻을 수 있음
  - 좌표계가 변해도 크기와 방향이 동일한 결과

$$\mathbf{w} = \mathbf{u} \times \mathbf{v}$$

다음 주제는 벡터의 곱

- 벡터 곱이 가지는 불변적 성질과 그 기하적 의미를 이해

점곱



직각 삼각형을 이루는 변의 관계에서

$$\cos\theta = \frac{l_{\mathbf{u}\to\mathbf{v}}}{||\mathbf{u}||}$$

내적의 정의에 따라

$$\mathbf{u}\cdot\mathbf{v} = ||\mathbf{u}||\ ||\mathbf{v}||\cos\theta$$

가위곱