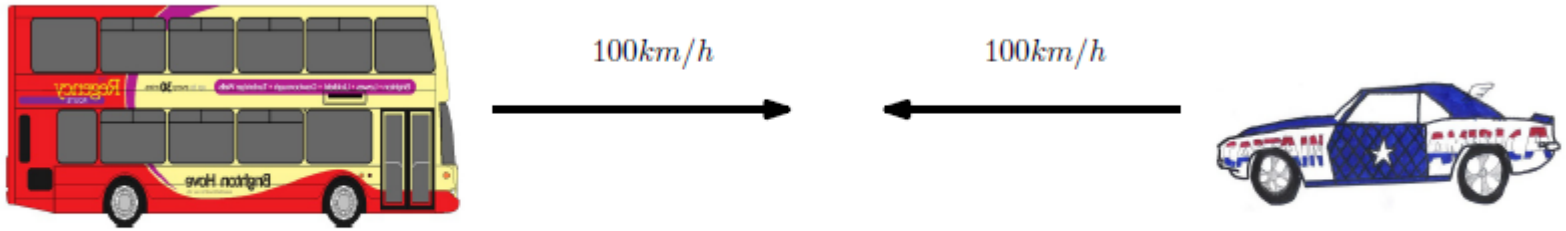
게임 수학 – 강의 2

동명대학교 게임공학과
강영민

다시 벡터 살펴보기

벡터의 표현

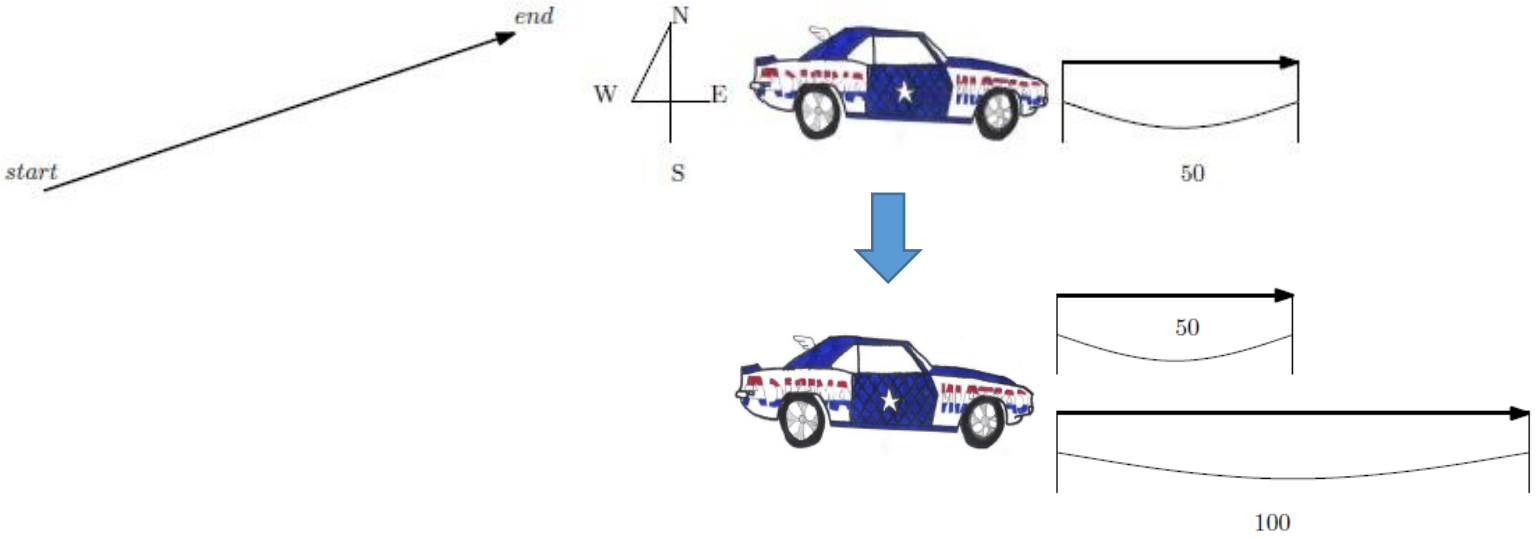
- 속도와 속력
  - 속력은 숫자로 표현하면 그만이지만, 속도는 숫자 하나로 표현 불가



동일한 속력으로 서로 마주 보며 달리는 차량의 속도

화살표를 이용한 벡터 표현

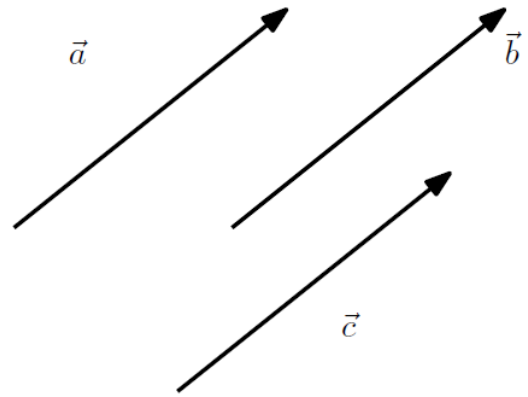
- 벡터를 표현하는 가장 직관적인 방법
  - 화살표: 시작하는 점과 끝나는 점으로 구성
    - 화살표의 방향은 벡터가 작용하는 방향
    - 화살표의 길이는 벡터가 가진 크기



속력이 두 배로 늘어난 자동차의 속도

동등벡터equivalent vector

- 벡터의 표기법 $\vec{a}$, $\mathbf{a}$
- 동등벡터
  - 크기와 방향이 같은 벡터들은 모두 동등한 벡터로 간주
    - 동등하다: 같은 양을 의미
      - 예) 물 5kg과 쌀 5kg은 같은 대상을 가리키는 것은 아니지만, “무게 ” 라는 양의 측면에서 동등
    - 두 벡터가 동등하다는 것은 두 벡터가 같은 존재라는 것이 아님
      - 벡터로 표현되는 양이 같다는 의미



동등벡터

벡터의 수학적 표현

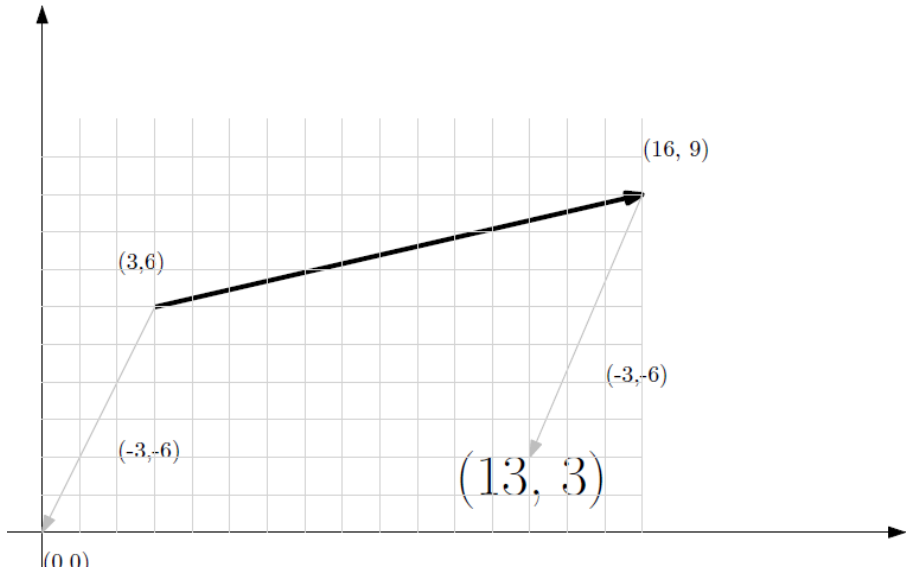
- 벡터를 조금 더 형식을 갖춰 표현하는 방법
  - 수학적으로 연산이 편리하도록 표현
- 벡터의 화살표 표현
  - 화살표가 그려지는 공간의 차원에 따라 좌표의 원소 개수 결정
  - 벡터의 차원 = 공간의 차원
  - 벡터가 n 차원이라면
    - 이 벡터는 n 개의 숫자로 표현되는 좌표 형태로 다룰 수 있을 것
    - n-튜플tuple

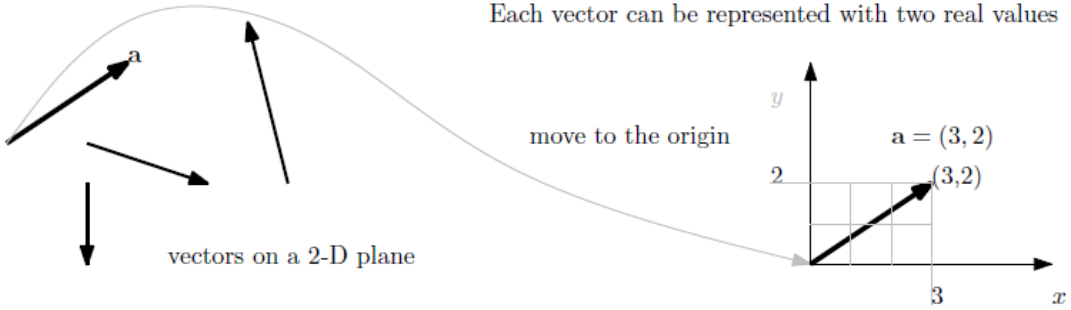
$$\mathbf{v} = (v_1, v_2, v_3, \cdots, v_n)$$

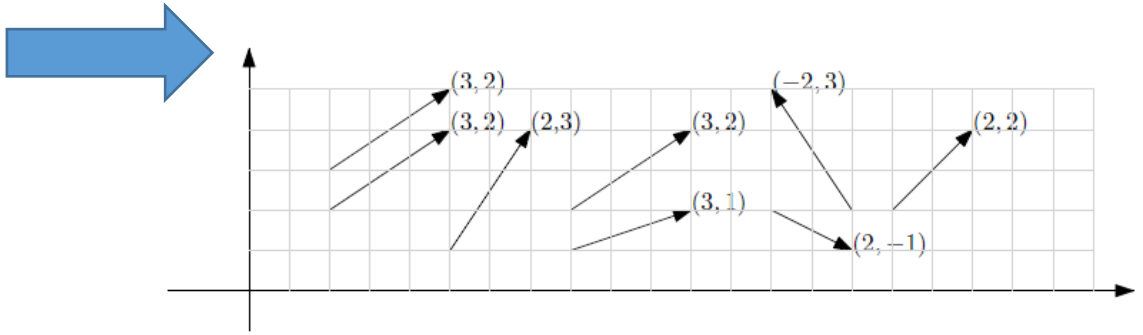
    - n 개의 차원을 가진 공간에서 그려지는 화살표 = n 차원 벡터 = n-튜플

2차원 벡터의 예

- 시작점 (xs, ys), 끝점 (xe, ye)
  - 네 개의 숫자로 표현할 수 있음
  - 동등 벡터는 모두 같은 형태로 표현되어야 함
  - 이를 위해서는 시작점을 원점으로 옮기면 동등 벡터는 모두 같은 끝점을 가짐
  - 이 끝점이 이들 동등벡터를 대표하는 양

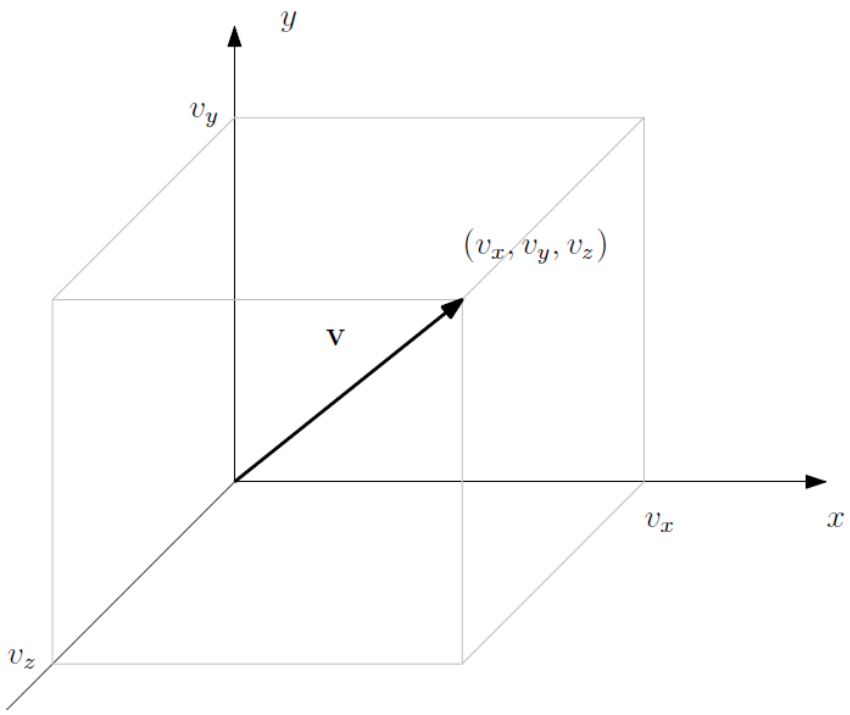






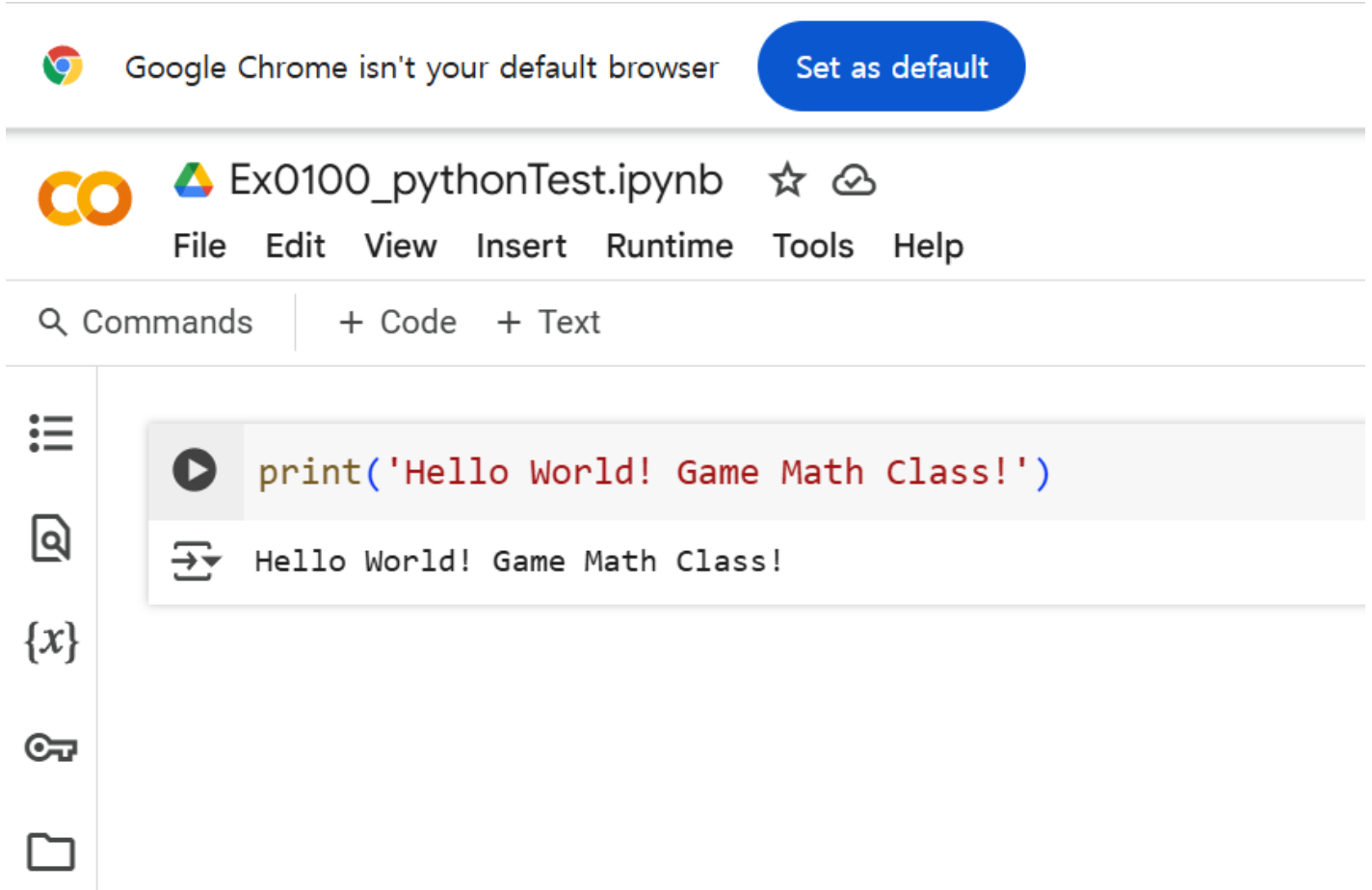
3차원 벡터

- 3차원 벡터는 2차원 벡터에 축을 하나 더하기만 하면 된다



벡터/벡터정규화 가시화

- 코랩 실행하기



벡터 가시화

- 가시화를 위해 만들어 놓은 함수들 사용하기

```
[ ] !wget https://raw.githubusercontent.com/dknife/linalg/main/tool/visualizer.py
    from visualizer import *
    !rm visualizer.py
```

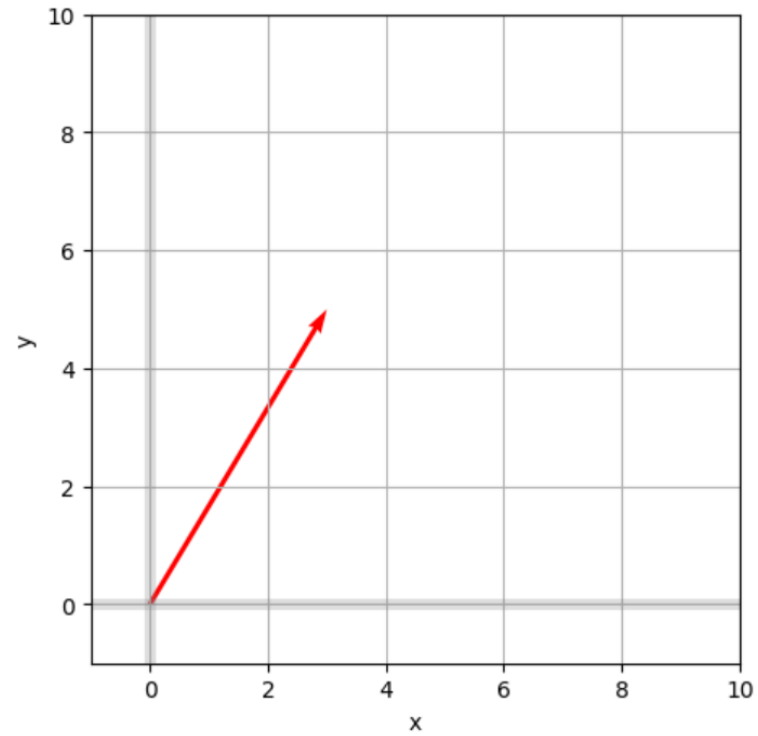
```
--2024-09-06 01:49:28--  https://raw.githubusercontent.com/dknife/linalg/main/tool/visualizer.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4814 (4.7K) [text/plain]
Saving to: ‘visualizer.py’

visualizer.py       100%[===================>]   4.70K  --.-KB/s    in 0s

2024-09-06 01:49:28 (49.6 MB/s) - ‘visualizer.py’ saved [4814/4814]
```

사용가능한 함수

- def axis2d(x=[-1, 1], y=[-1, 1], grid=True) :
- def draw_vec2d(axis, v, color='r', start_from = None, alpha = 1.0, label=None):
- def axis3d(x=[-1,1], y=[-1,1], z=[-1,1]) :
- def setCam(ax3d, cam_loc):
- def draw_vec3d(axis, v, color='r', start_from=None, alpha=1.0, label=None):
- def draw_points_in_matrix(axis, M, color='red') :
- def draw_points(my_axis, points_list, labels=None, color='red'):
- def draw_space_mat22(ax2d, M, label=None, color='gray'):
- def draw_polygons(ax, polygon_list, facecolors, edgecolors='black', alpha=0.8):
- def draw_mat22(ax2d, M, label=None):
- def draw_mat33(ax3d, M, label=None):
- def draw_circle(ax, center=[0,0], radius=1.0, color='blue'):

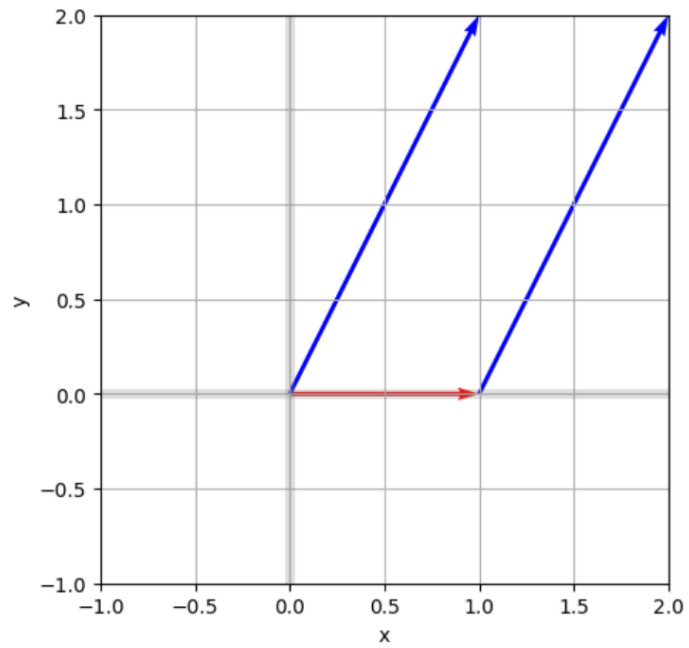
벡터 가시화

```
# (3,5) 벡터를 가시화 하자

my_axis = axis2d(x=[-1,10], y=[-1,10])
my_vector = np.array([3,5])
draw_vec2d(my_axis, my_vector)
```

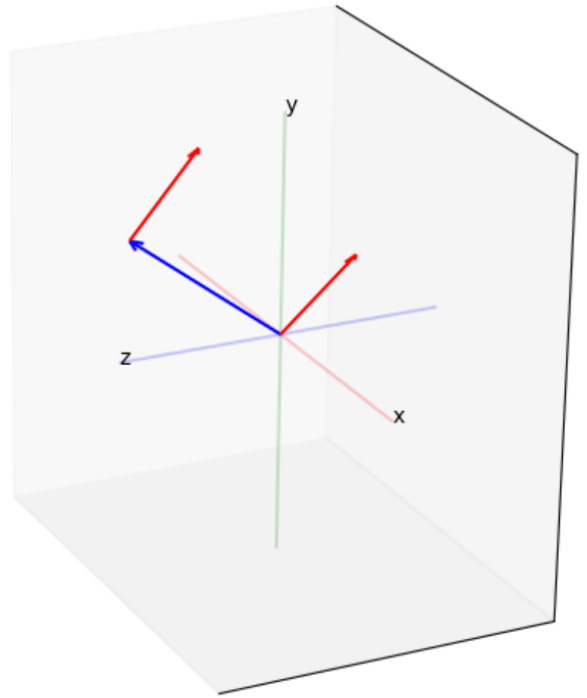


벡터 가시화

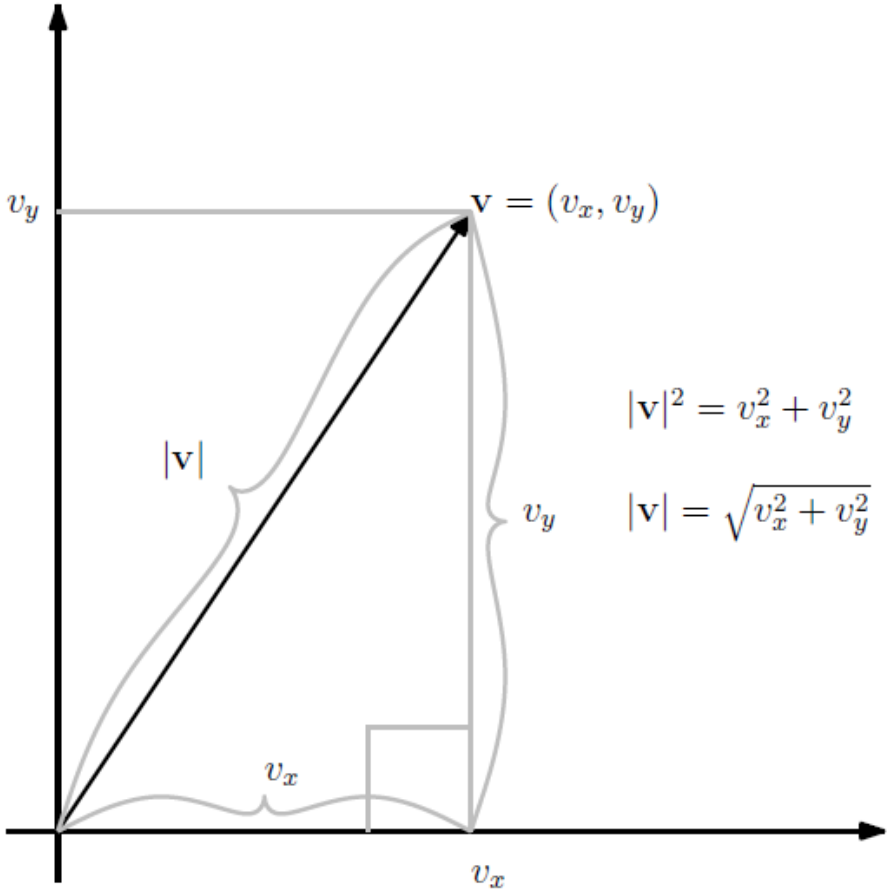
```
my_axis = axis2d(x=[-1,2], y=[-1,2])

u = np.array([1, 0])
v = np.array([1, 2])

draw_vec2d(my_axis, u, color='red')
draw_vec2d(my_axis, v, color='blue')
draw_vec2d(my_axis, v, color='blue', start_from = u)
```




```
[ ]  my_axis = axis3d(x=[-2,2], y=[-2,2], z=[-2,2])

     u = np.array([2.0, 1.5, 0.5])
     v = np.array([1.0, 1.5, 2.5])

     draw_vec3d(my_axis, u, color='red')
     draw_vec3d(my_axis, v, color='blue')
     draw_vec3d(my_axis, u, color='red', start_from=v)
```



2차원 벡터의 크기

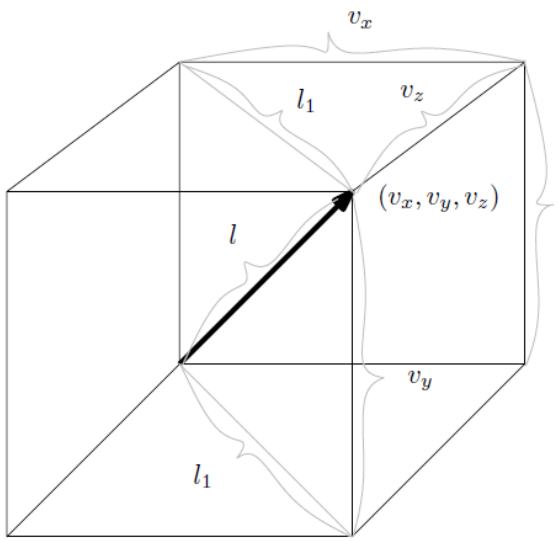


- 왼쪽에 보이는 벡터 v의 크기는?
  - 좌표값은 각 축으로의 길이
  - 2차원의 예에서
    - X좌표는 밑변
    - Y좌표는 높이
    - 벡터의 크기는 이러한 밑변과 높이를 가지는 직각삼각형의 빗변 길이
  - 피타고라스 정리로 간단히 계산
- 크기는 언제나 양의 값
- 이 값을 노름(norm, 놈)이라 함

3차원 벡터의 크기

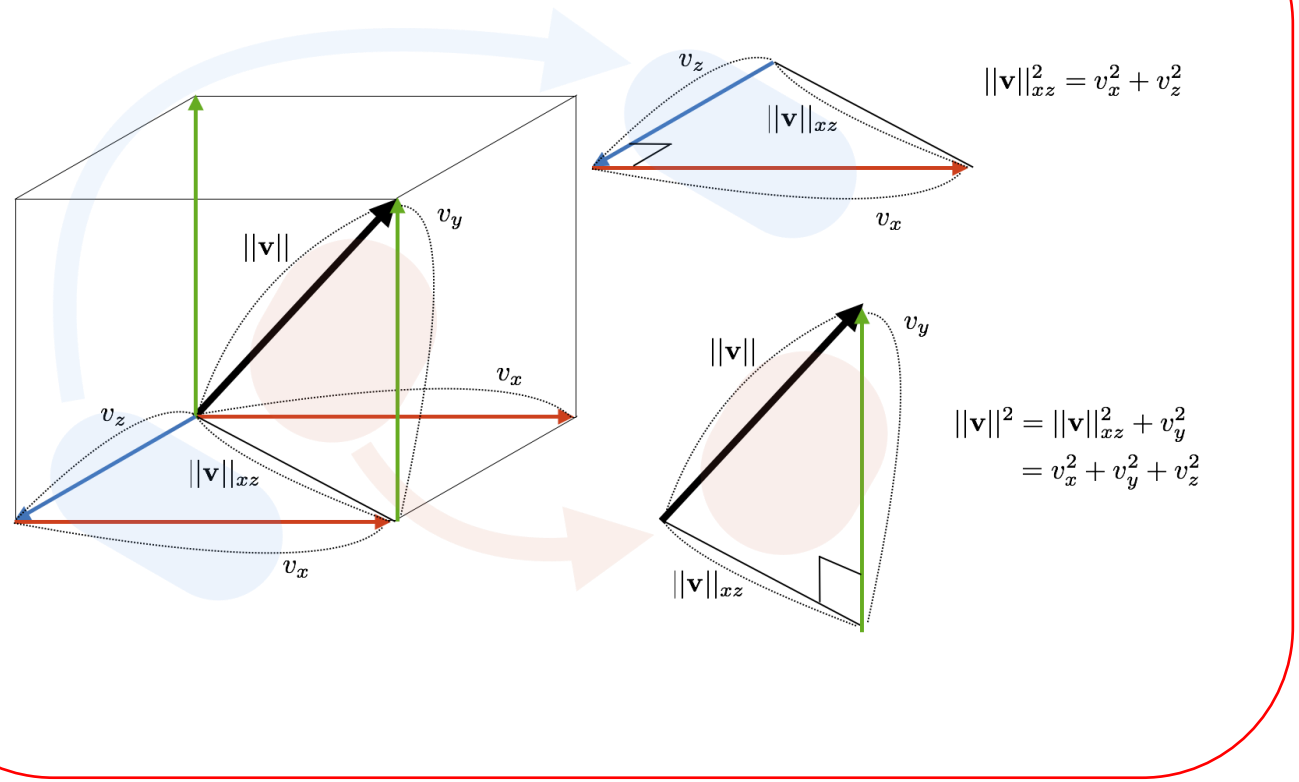
$$\mathbf{v} = (v_x, v_y, v_z)$$

$$||\mathbf{v}|| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$



$$l_1^2 = v_x^2 + v_z^2$$

$$l^2 = l_1^2 + v_y^2$$



$$||\mathbf{v}||_{xz}^2 = v_x^2 + v_z^2$$

$$||\mathbf{v}||^2 = ||\mathbf{v}||_{xz}^2 + v_y^2$$
$$= v_x^2 + v_y^2 + v_z^2$$

벡터 노름 – 일반화

- 피타고라스 정리를 이용하여 구한 벡터의 노름
  - 특별한 예: 2차 노름, l_2 노름
- 노름의 일반화

$$||\mathbf{x}||_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}}$$

  - p 값에 따라 l_p–노름이라 부름
  - p=1
    - 맨해튼 거리라고도 함
  - p가 무한대
    - 원소중에 제일 큰 값을 얻음 (max)

벡터 노름 계산해 보기

```python
import numpy as np          # numpy의 별명을 np로 함

v = np.array([1, 2, 3])     # 세 개의 원소를 가진 배열로 3차원 벡터 표현
print(v.ndim, v.shape)      # 넘파이 배열의 ndim과 shape 출력
```

```
1 (3,)
```

```python
def norm(v, p=2) :
    dim = v.shape[0]        # 원소의 개수를 담고 있음 = 벡터의 차원
    acc = 0
    for i in range(dim):    # 벡터의 차원만큼 모든 원소의 p 제곱 구해 누적
        acc += v[i]**p
    return acc**(1/p)       # 전체에 1/p 제곱을 적용하여 p-노름 계산
```

$$||\mathbf{x}||_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}}$$

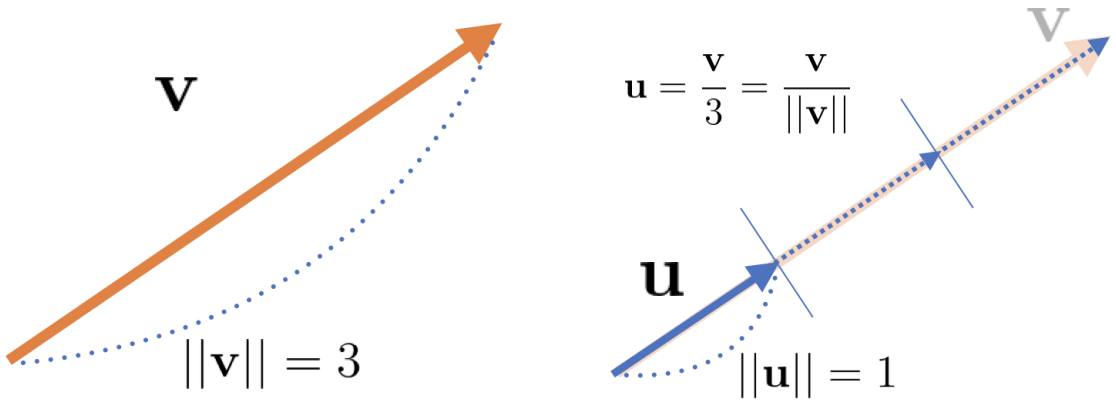
```python
v_norm_1 = norm(v, p=1)     # 맨해튼 노름 (L1)
v_norm_2 = norm(v, p=2)     # 유클리드 노름 (L2)
v_norm_3 = norm(v, p=3)     # 3-노름
v_norm_4 = norm(v, p=4)     # 4-노름
print(v_norm_1, v_norm_2, v_norm_3, v_norm_4)
```

```
6.0 3.7416573867739413 3.3019272488946263 3.1463462836457885
```

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.

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

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

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

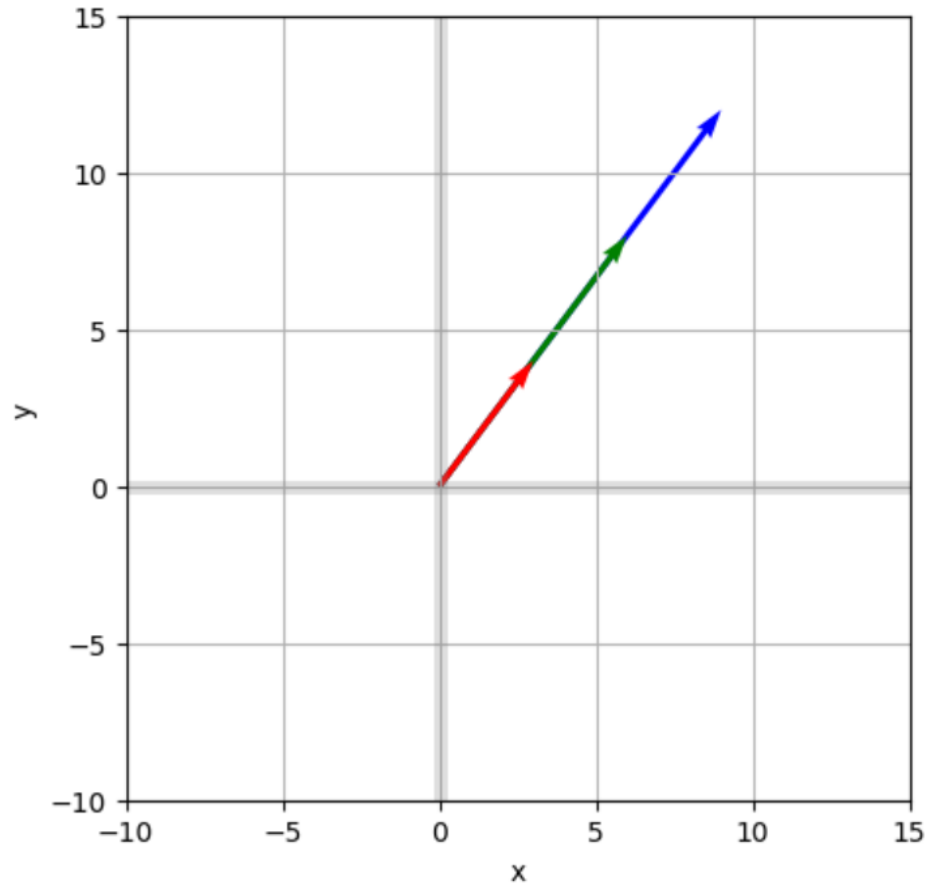
$$\hat{\mathbf{x}} = \mathbf{x}/l_{\mathbf{x}} \quad , \quad \mathbf{x} = l_{\mathbf{x}}\hat{\mathbf{x}}$$

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

벡터 노름과 정규화

- 벡터의 길이 변화

```
vec2x = vec * 2
vec3x = vec * 3
mySpace = axis2d(x=[-10, 15], y=[-10, 15])

draw_vec2d(mySpace, vec3x, color='blue')
draw_vec2d(mySpace, vec2x, color='green')
draw_vec2d(mySpace, vec, color='red')
```



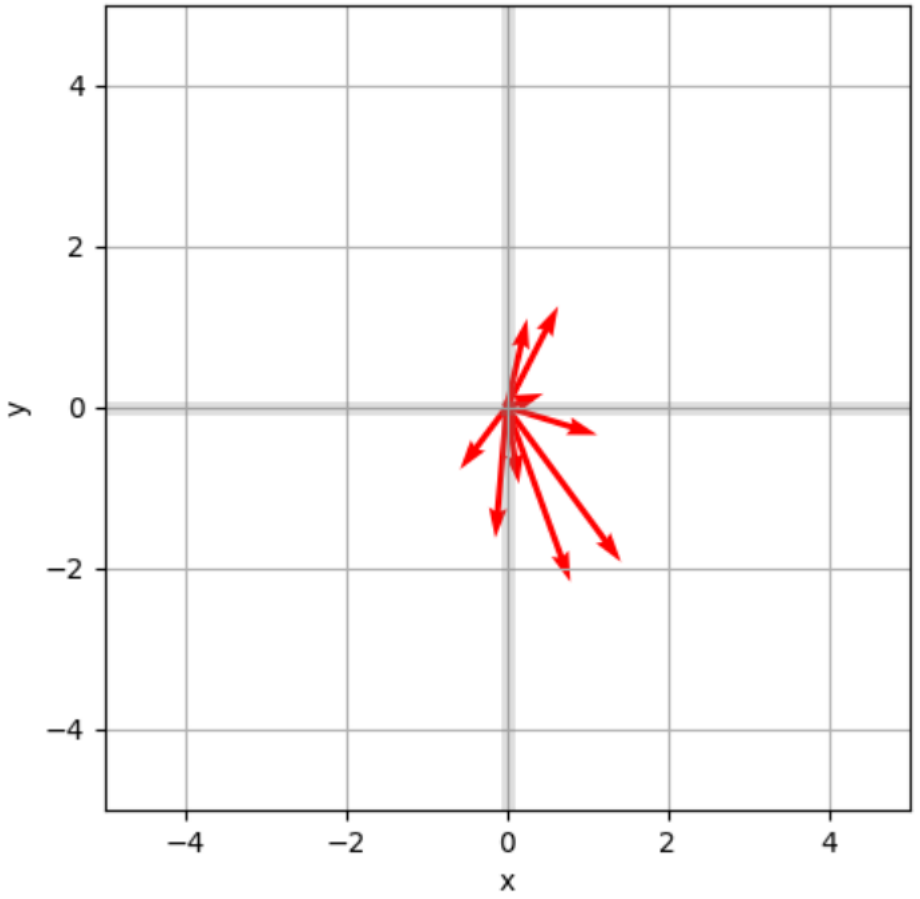
임의의 벡터

```
[ ]  n_vectors = 10
     vectors = np.random.randn(n_vectors, 2)
```

```
vectors
```

```
array([[ 0.45640603,  0.1702312 ],
       [ 0.79071949, -2.17405056],
       [ 1.42113044, -1.92574328],
       [-0.14206262, -1.62841086],
       [-0.58114353, -0.77515022],
       [ 0.04834487,  0.2114212 ],
       [ 1.12663021, -0.34543011],
       [ 0.15179819, -0.95508221],
       [ 0.25456136,  1.10134864],
       [ 0.63884106,  1.25418416]])
```

```
mySpace = axis2d(x=[-5, 5], y=[-5, 5])

for v in vectors:
    draw_vec2d(mySpace, v, color='red')
```



정규화

```
[ ]  def p_norm
         sum =
         for i
             su

         return
```

```
mySpace = axis2d(x=[-5, 5], y=[-5, 5])
```

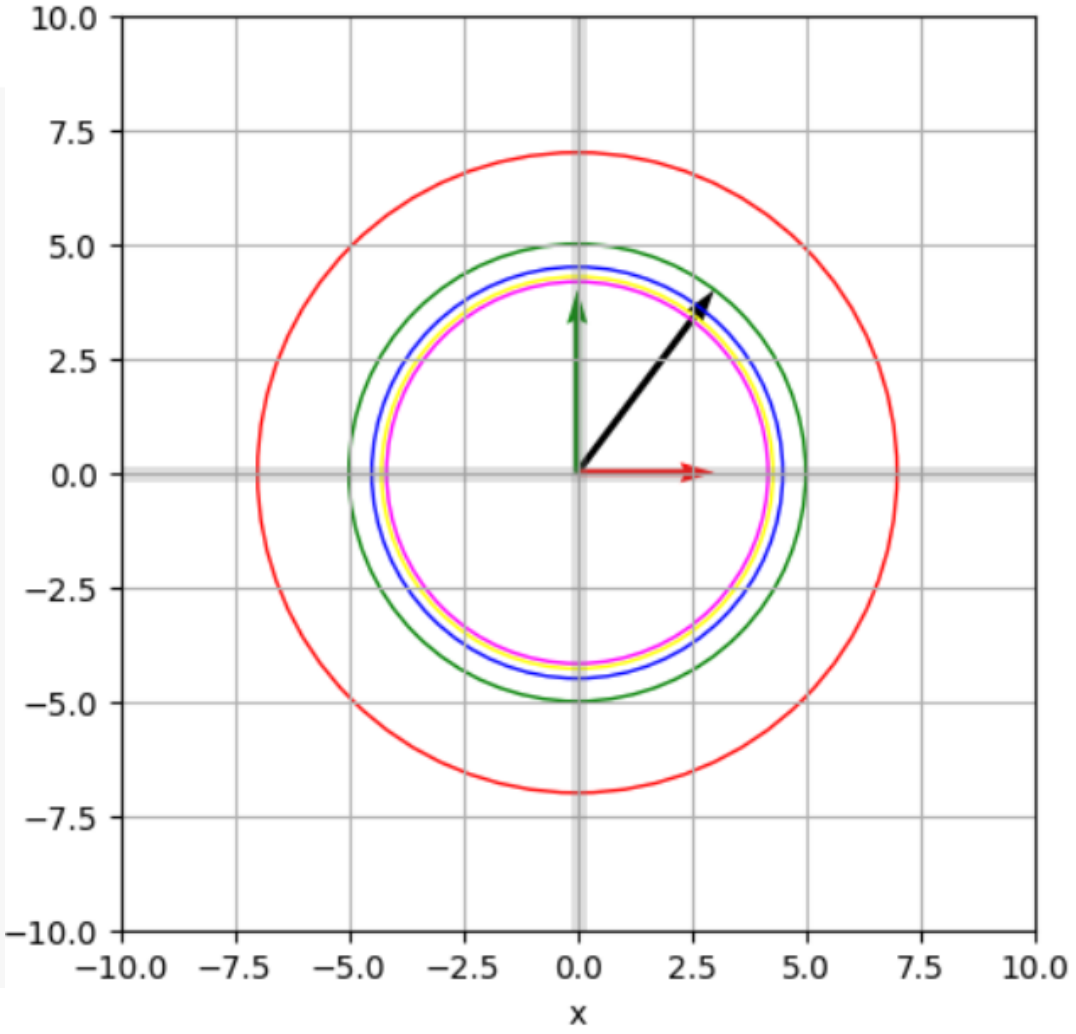
```
red')
```

```
ed, color='black')
```

```
[ ]  xVec = np.array([myVec[0], 0])
     yVec = np.array([0, myVec[1]])

     mySpace = axis2d(x=[-10,10], y=[-10,10])
     draw_vec2d(mySpace, myVec, color='black')
     draw_vec2d(mySpace, xVec,color = 'red')
     draw_vec2d(mySpace, yVec, color='green')
     p1_norm = p_norm(myVec, ord=1)
     p2_norm = p_norm(myVec, ord=2)
     p3_norm = p_norm(myVec, ord=3)
     p4_norm = p_norm(myVec, ord=4)
     p5_norm = p_norm(myVec, ord=5)
     draw_circle(mySpace, (0,0), p1_norm, color='red')
     draw_circle(mySpace, (0,0), p2_norm, color='green')
     draw_circle(mySpace, (0,0), p3_norm, color='blue')
     draw_circle(mySpace, (0,0), p4_norm, color='yellow')
     draw_circle(mySpace, (0,0), p5_norm, color='magenta')
```

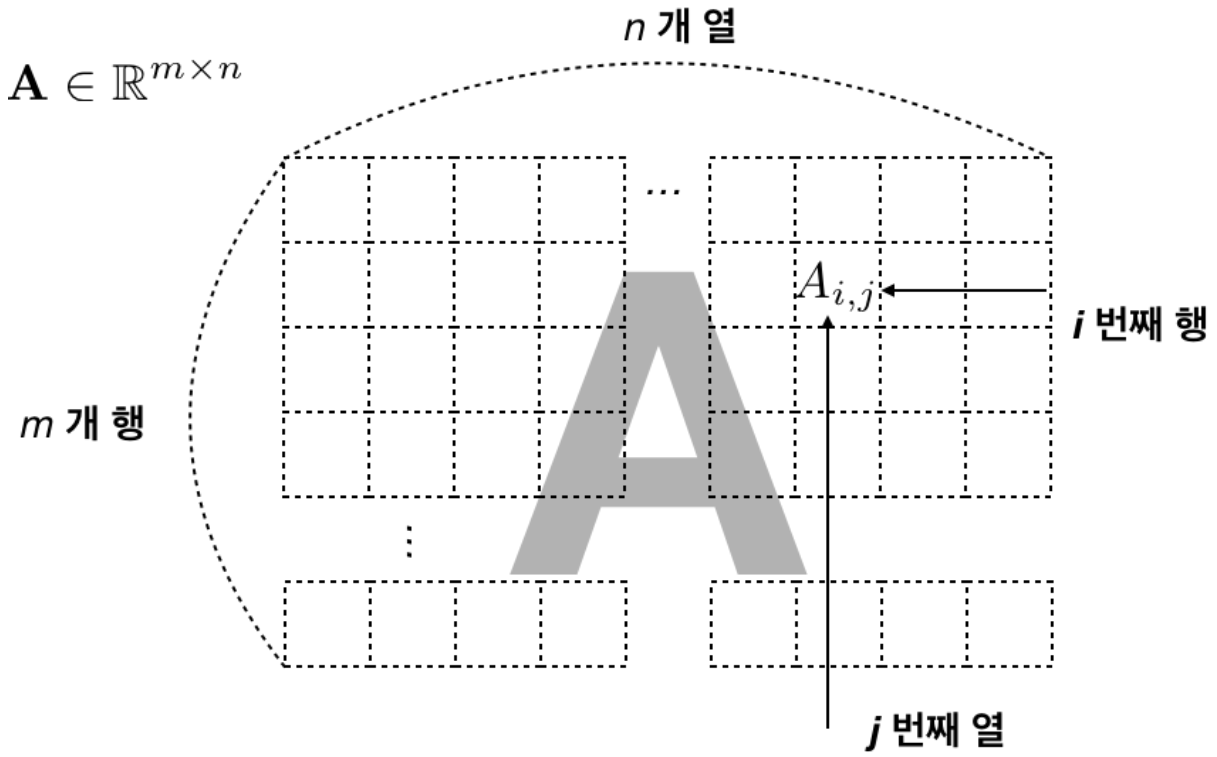
벡터 노름

```
[ ]  xVec = np.array([myVec[0], 0])
     yVec = np.array([0, myVec[1]])

     mySpace = axis2d(x=[-10,10], y=[-10,10])
     draw_vec2d(mySpace, myVec, color='black')
     draw_vec2d(mySpace, xVec,color = 'red')
     draw_vec2d(mySpace, yVec, color='green')
     p1_norm = p_norm(myVec, ord=1)
     p2_norm = p_norm(myVec, ord=2)
     p3_norm = p_norm(myVec, ord=3)
     p4_norm = p_norm(myVec, ord=4)
     p5_norm = p_norm(myVec, ord=5)
     draw_circle(mySpace, (0,0), p1_norm, color='red')
     draw_circle(mySpace, (0,0), p2_norm, color='green')
     draw_circle(mySpace, (0,0), p3_norm, color='blue')
     draw_circle(mySpace, (0,0), p4_norm, color='yellow')
     draw_circle(mySpace, (0,0), p5_norm, color='magenta')
```



행렬 데이터의 이해

- 행렬은 2차원으로 배열된 숫자



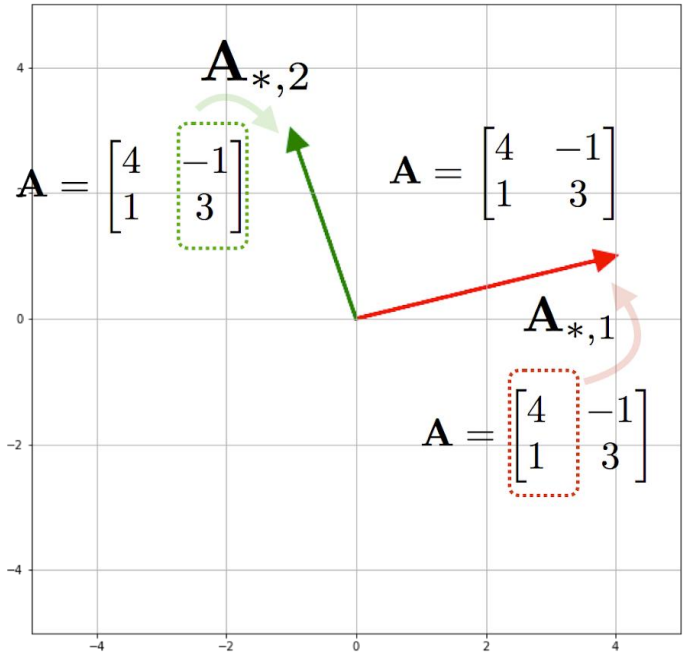
$$\mathbf{A} \in \mathbb{R}^{m \times n}$$

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)
```

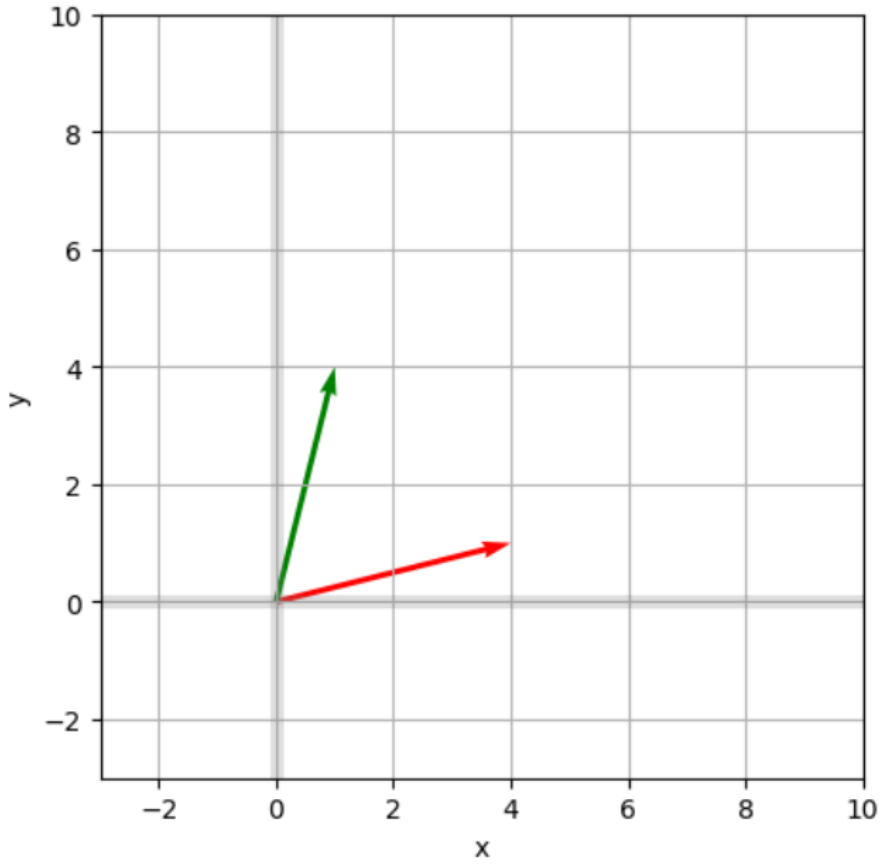
행렬의 가시화 – 행렬 생성 및 각 축 확인

```
[ ] mat
```

```
array([[4, 1],
       [1, 4]])
```

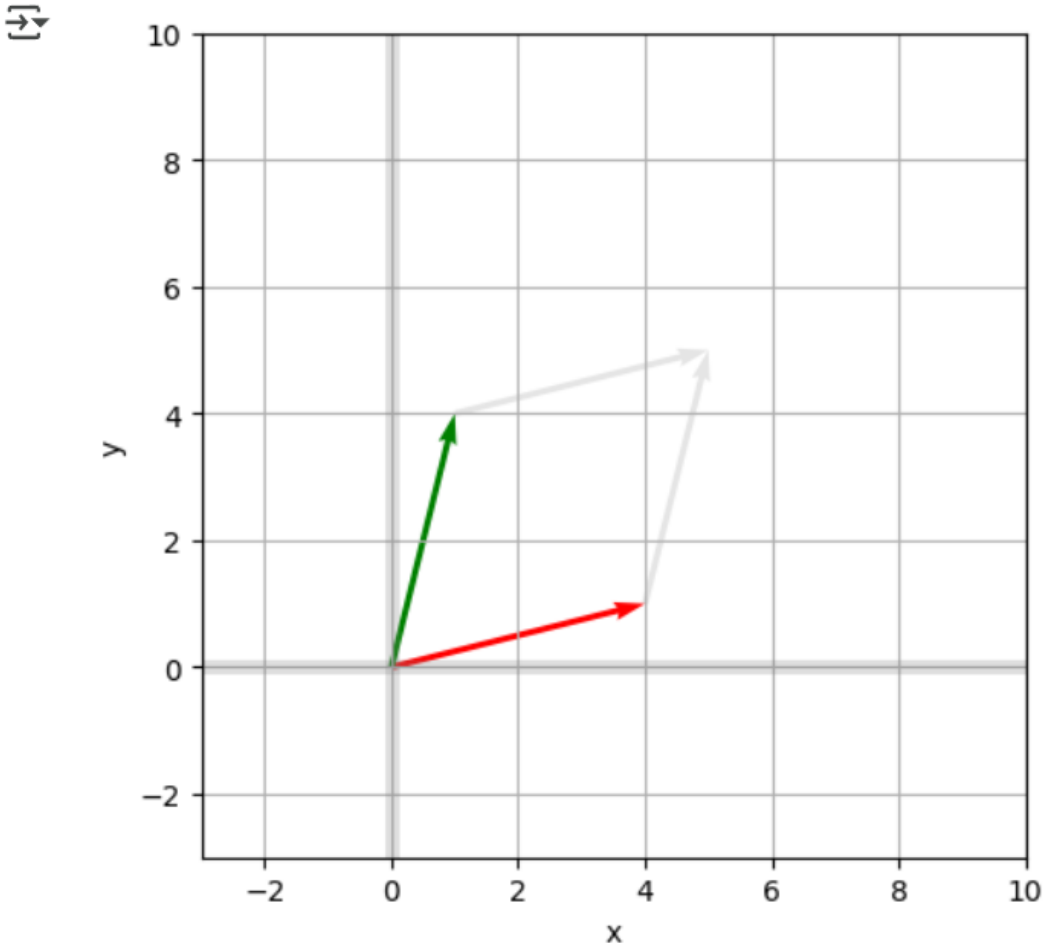
```
mySpace = axis2d(x=[-3, 10], y=[-3, 10])
x = mat[:, 0]
y = mat[:, 1]
print(x, y)
draw_vec2d(mySpace, x, color='red')
draw_vec2d(mySpace, y, color='green')
```

```
[4 1] [1 4]
```



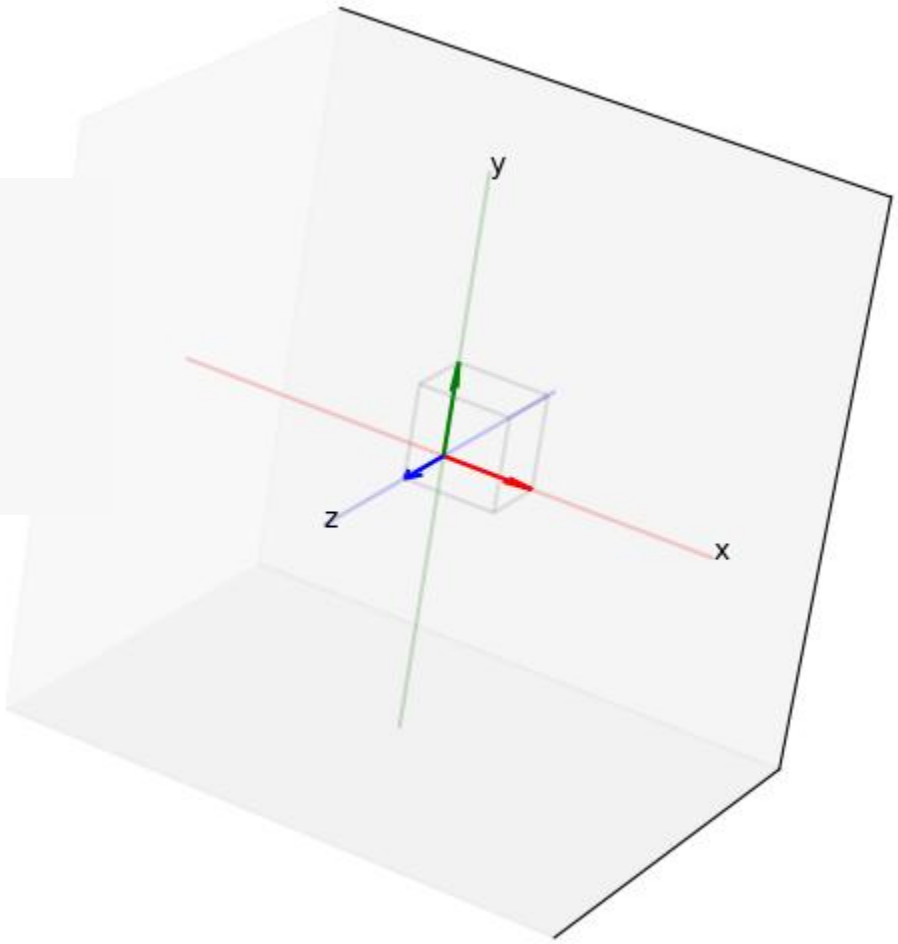
행렬의 가시화

```
[ ] mySpace = axis2d(x=[-3, 10], y=[-3, 10])
    draw_mat22(mySpace, mat)
```



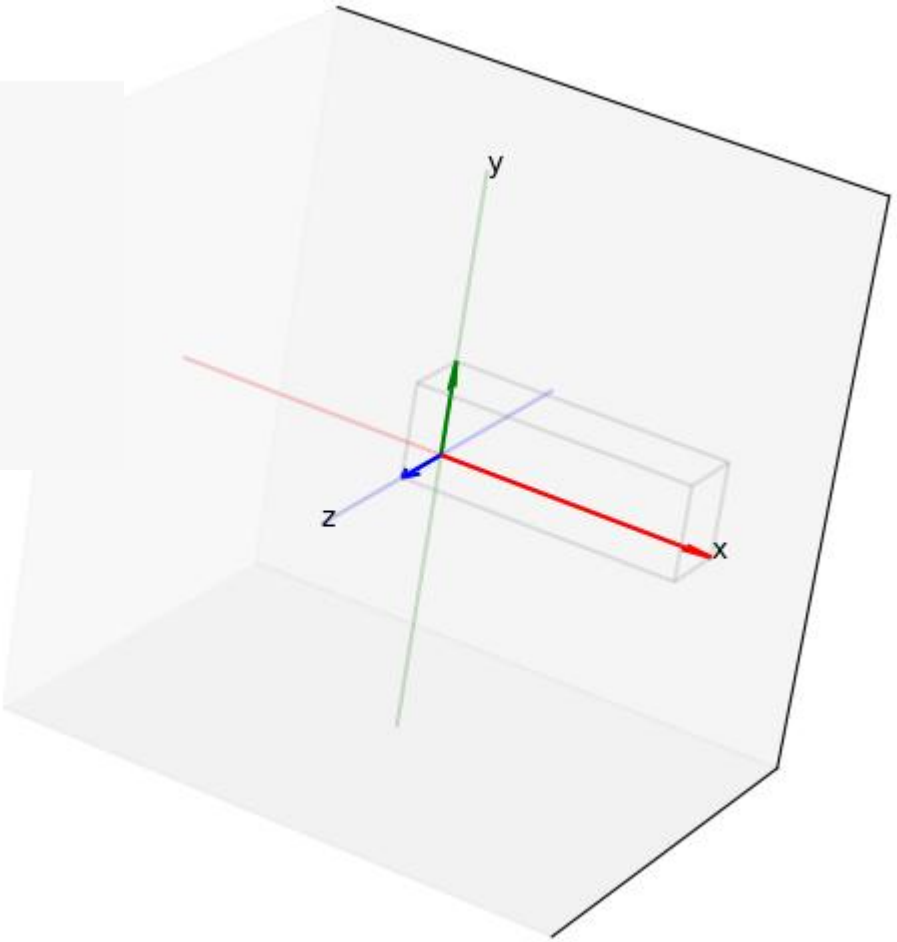
행렬의 가시화 – 3x3 행렬

```
mat33 = np.array([[1, 0, 0],
                  [0, 1, 0],
                  [0, 0, 1]])

mat33
```



행렬의 가시화 – 3x3 행렬

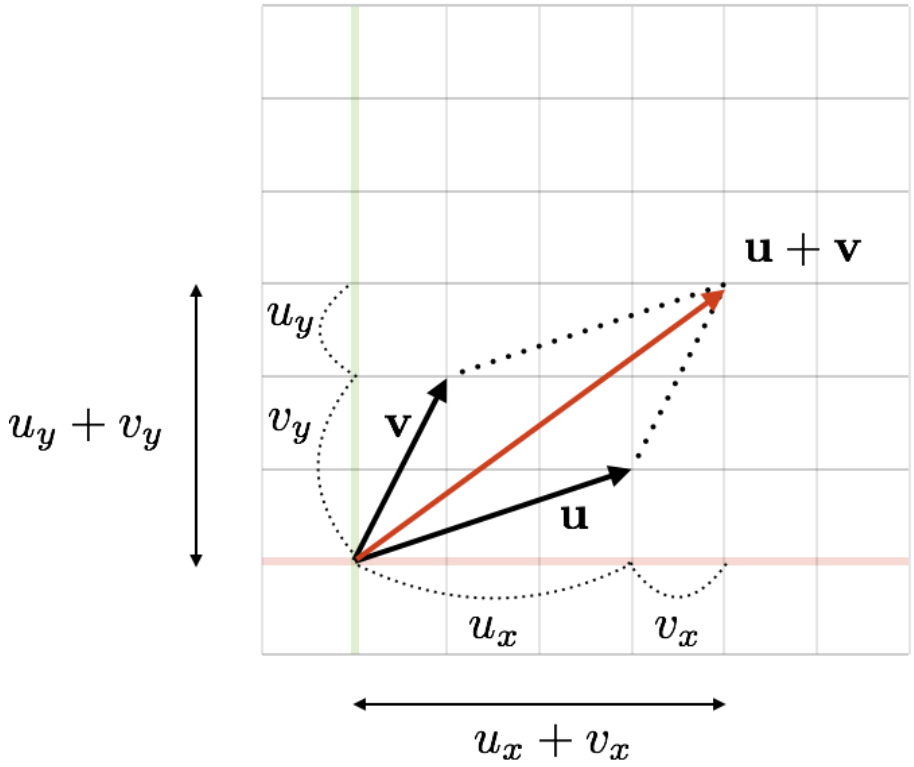
```
mat33 = np.array([[3, 0, 0],
                  [0, 1, 0],
                  [0, 0, 1]])
mySpace = axis3d(x=[-3, 3], y=[-3, 3], z=[-3, 3])
setCam(mySpace, np.array([1, 1, 2]))
draw_mat33(mySpace, mat33)
```



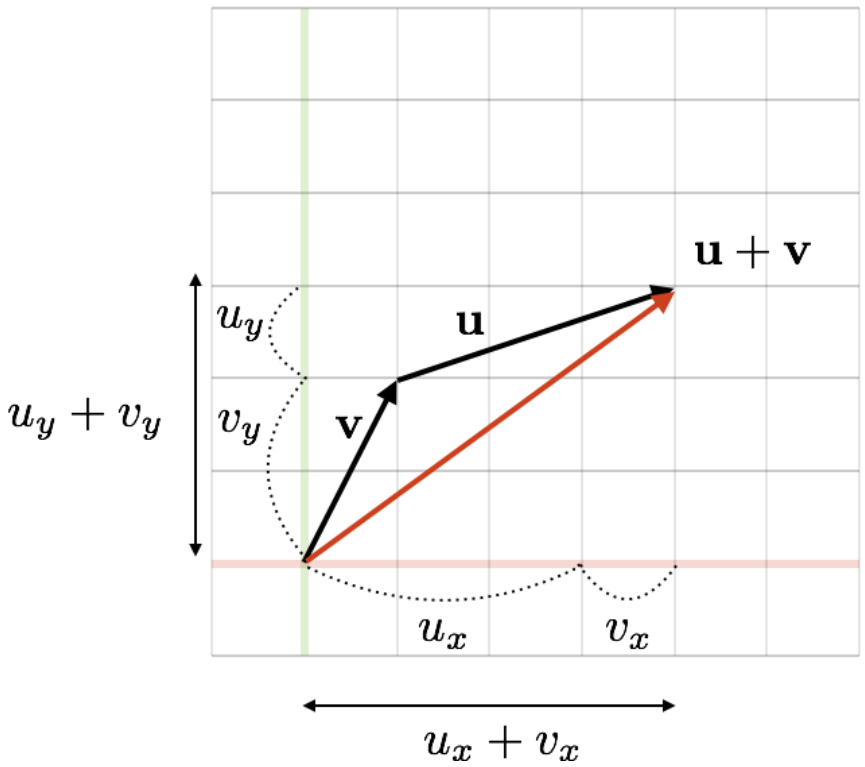
벡터의 연산 – 덧셈 $\mathbf{w} = \mathbf{u} + \mathbf{v}$

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

평행사변형을 이용한 벡터 합 가시화

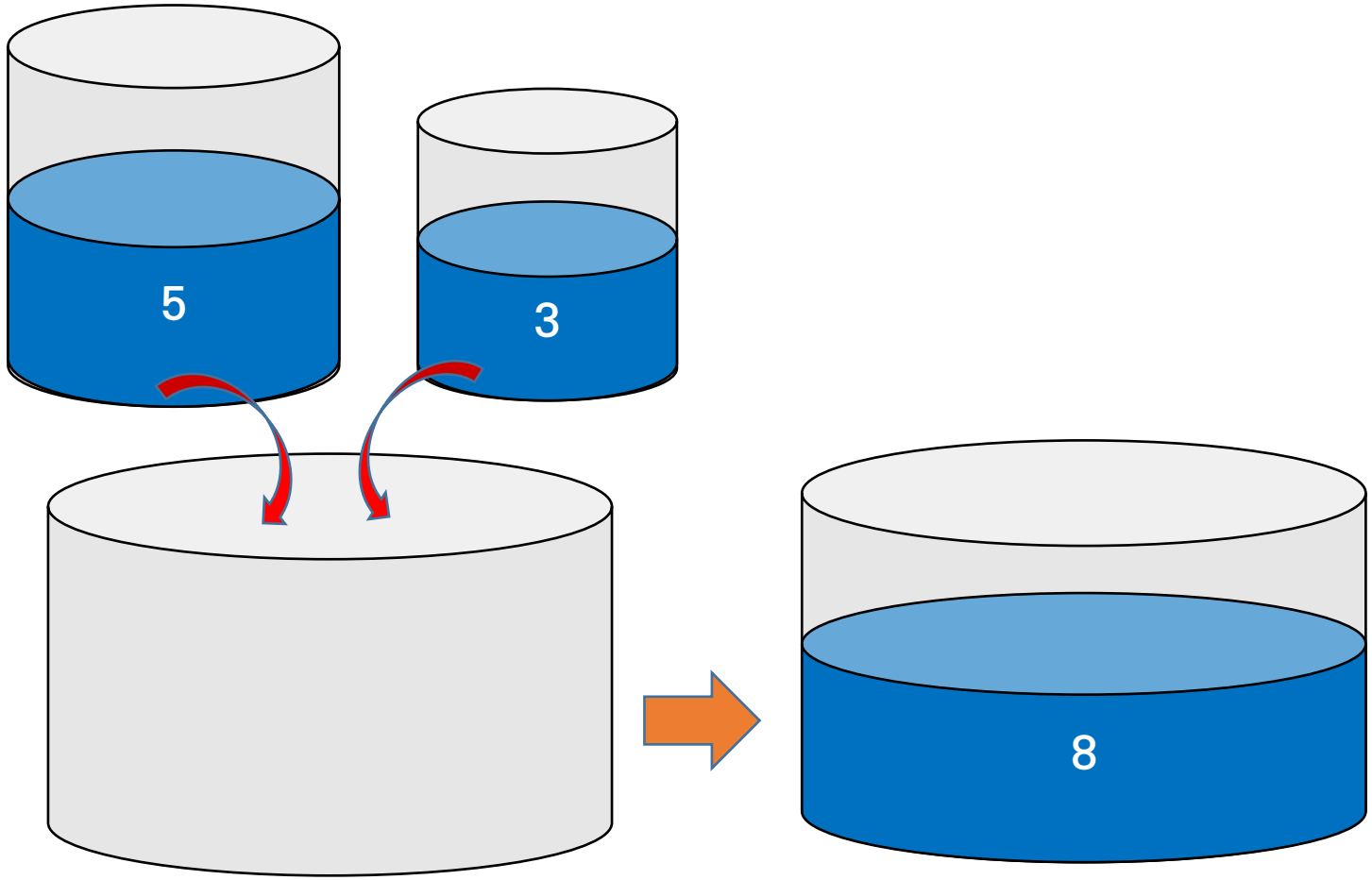


이어지는 벡터를 이용한 벡터 합 가시화



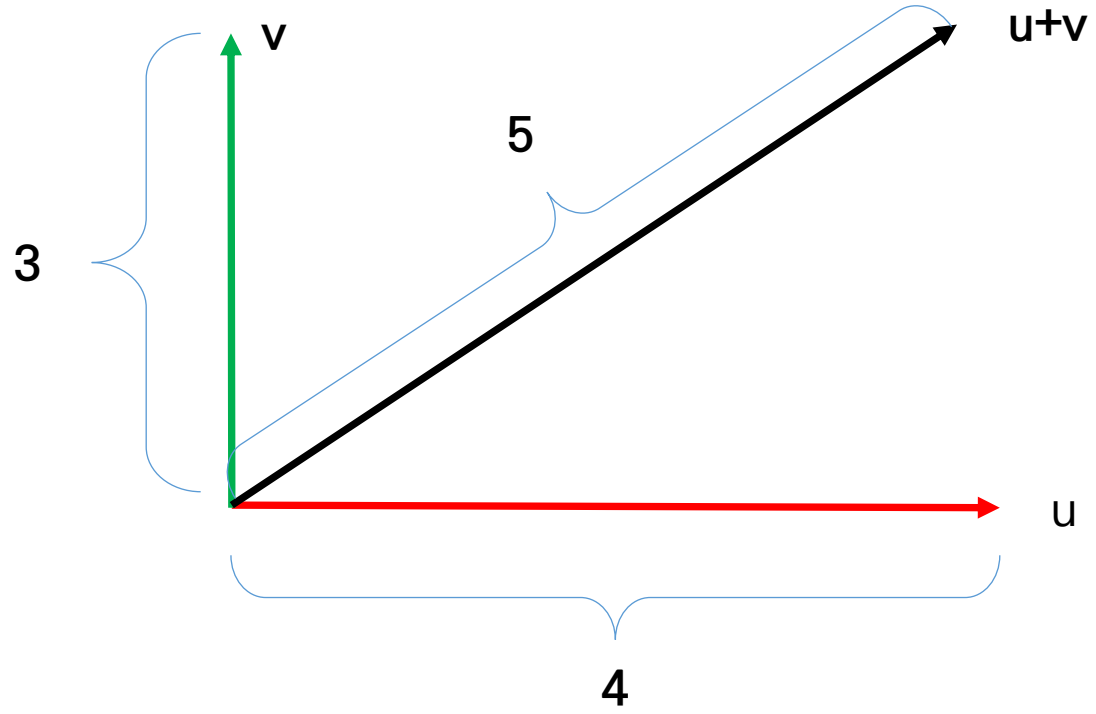
스칼라 덧셈과 벡터 덧셈의 차이

- 스칼라 덧셈
  - 크기의 모아짐
    - 5 + 3 = 8



스칼라 덧셈과 벡터 덧셈의 차이

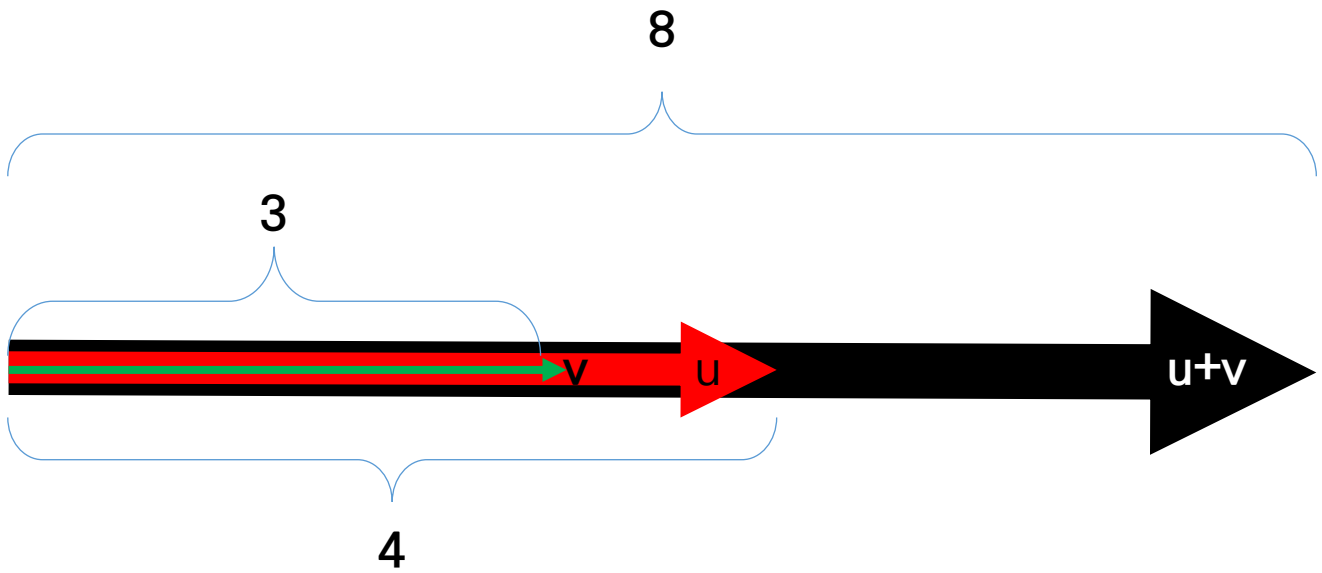
- 벡터의 덧셈
  - 크기가 모아지지 않음



- 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

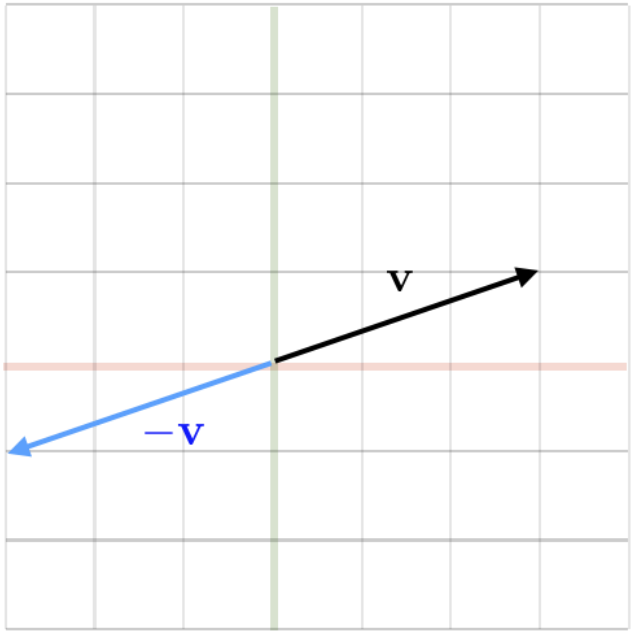


- 같은 방향이면 크기가 그대로 모아짐
- 스칼라 덧셈
  - 같은 방향을 가진 벡터의 덧셈으로 이해할 수 있음

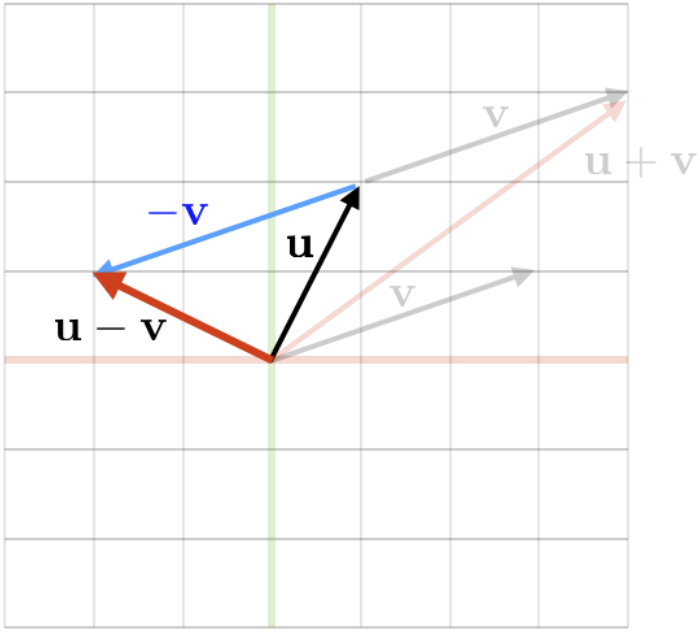
벡터의 연산 – 뺄셈 $\mathbf{w} = \mathbf{u} - \mathbf{v}$

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

음수 부호가 붙은 벡터

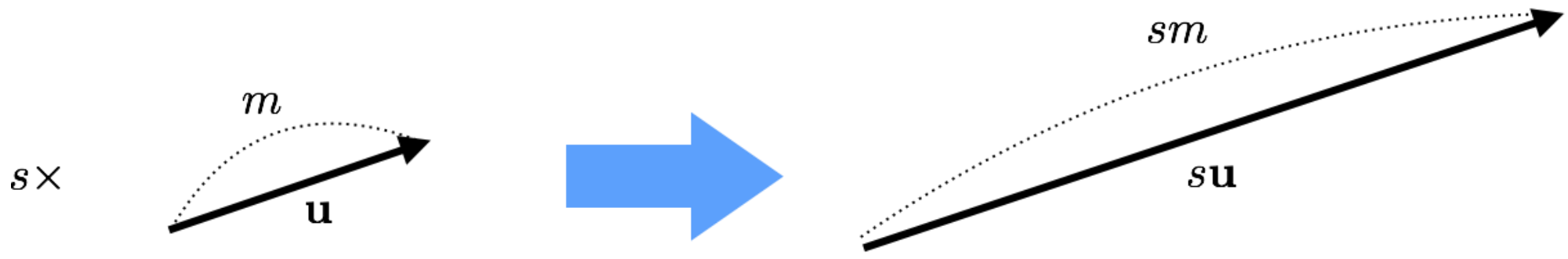


벡터의 뺄셈



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

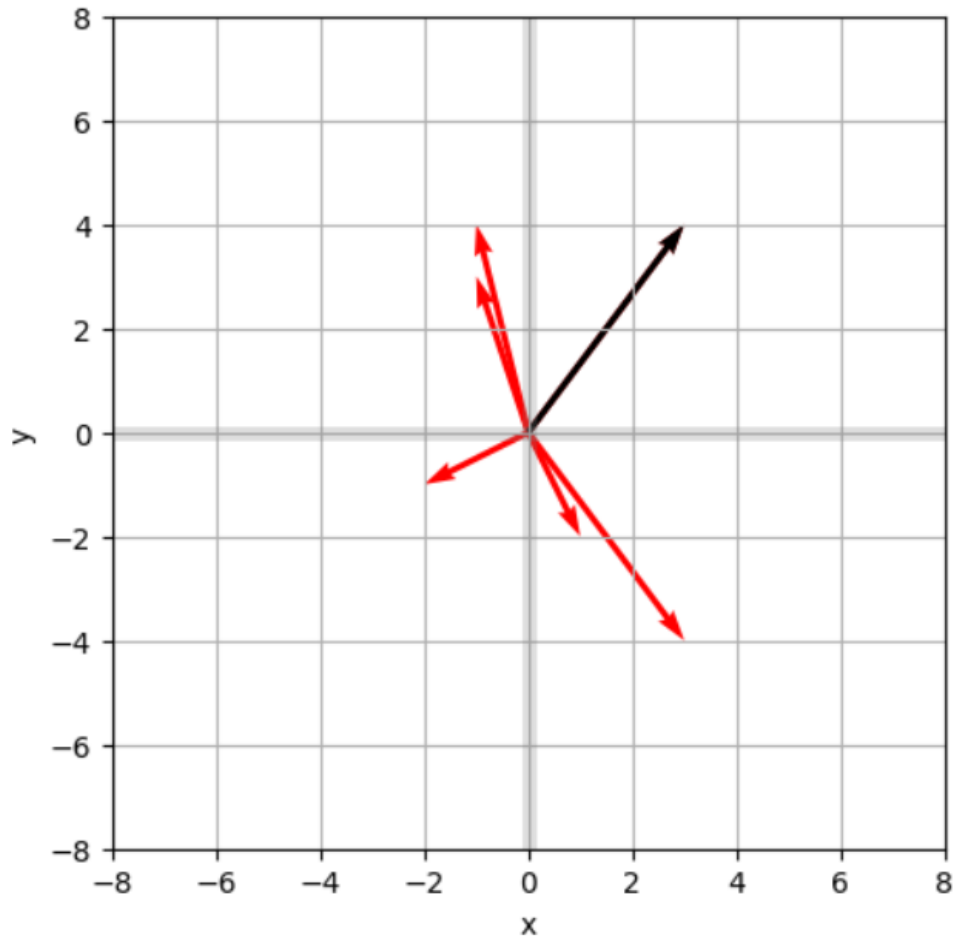
여러 벡터의 합

```
v1 = np.array([-1, 3])
v2 = np.array([3,4])
v3 = np.array([-2, -1])
v4 = np.array([1,-2])
v5 = np.array([-1, 4])
v6 = np.array([3,-4])
w = v1 + v2 + v3 + v4 + v5 + v6
w
```

```
array([3, 4])
```

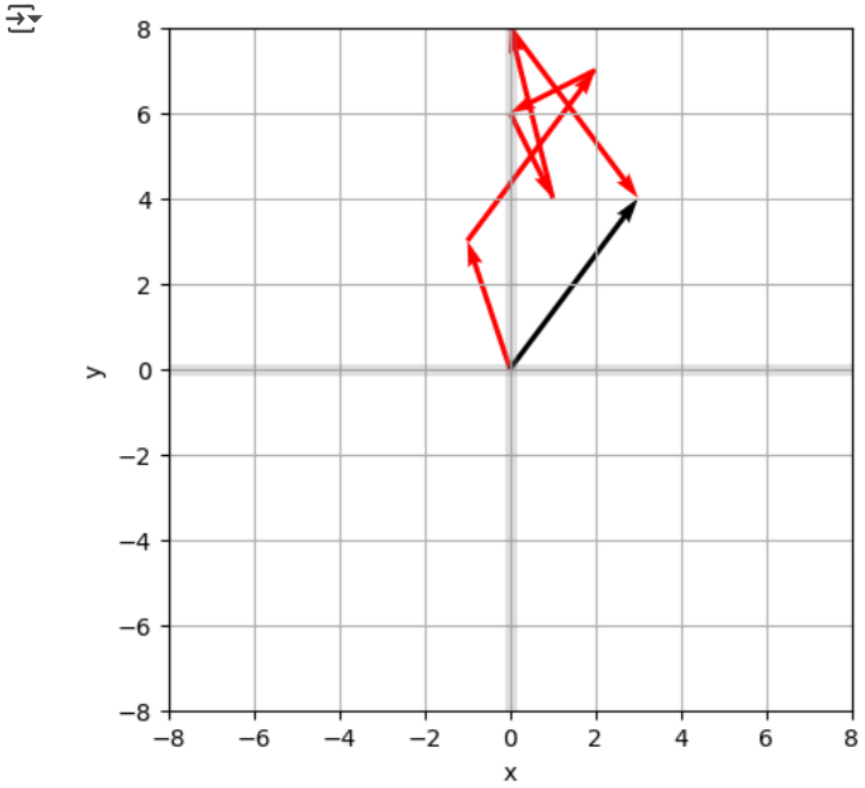
여러 벡터의 합

```
end = np.array([0,0])
mySpace = axis2d(x=[-8, 8], y=[-8, 8])
draw_vec2d(mySpace, v1, start_from=end, color='red')
draw_vec2d(mySpace, v2, start_from=end, color='red')
draw_vec2d(mySpace, v3, start_from=end,color='red')
draw_vec2d(mySpace, v4, start_from=end,color='red')
draw_vec2d(mySpace, v5, start_from=end,color='red')
draw_vec2d(mySpace, v6, start_from=end, color='red')
draw_vec2d(mySpace, w,  color='black')
```



여러 벡터의 합

```
end = np.array([0,0])
mySpace = axis2d(x=[-8, 8], y=[-8, 8])
draw_vec2d(mySpace, v1, start_from=end, color='red')
end += v1
draw_vec2d(mySpace, v2, start_from=end, color='red')
end += v2
draw_vec2d(mySpace, v3, start_from=end,color='red')
end += v3
draw_vec2d(mySpace, v4, start_from=end,color='red')
end += v4
draw_vec2d(mySpace, v5, start_from=end,color='red')
end += v5
draw_vec2d(mySpace, v6, start_from=end, color='red')
end += v6
draw_vec2d(mySpace, w,  color='black')
```



벡터의 곱

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

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

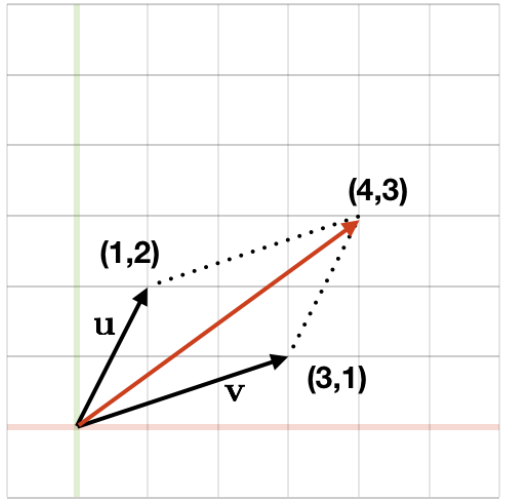
$$w_i = u_i v_i$$

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

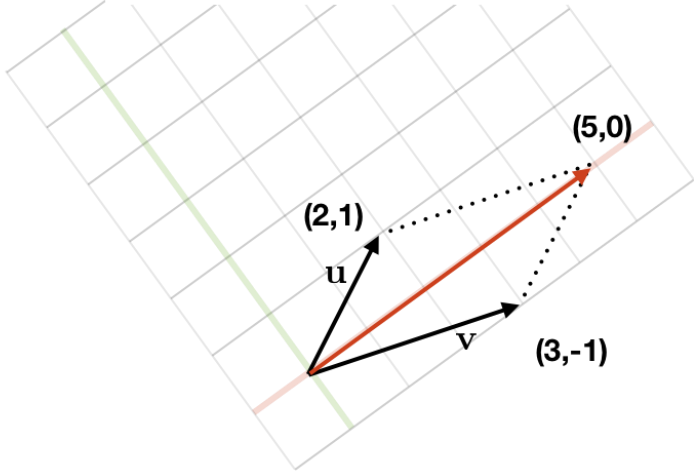
벡터의 아다마르 곱

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

어떤 좌표 공간에서 두 벡터의 합



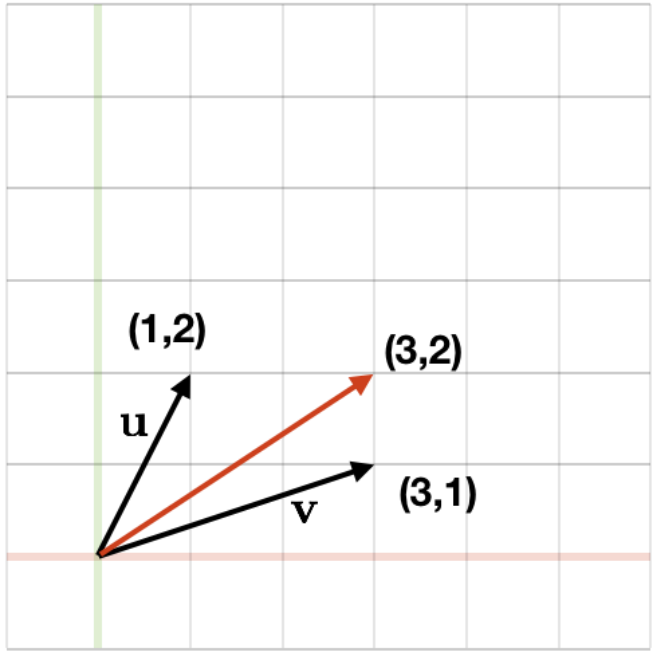
변경된 좌표계에서 두 벡터의 합



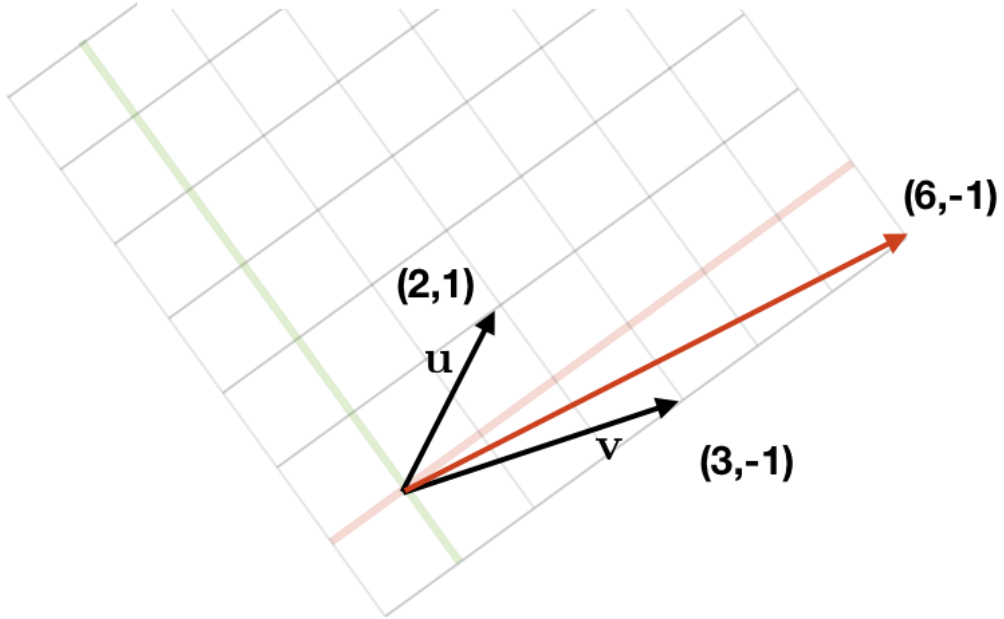
벡터의 아다마르 곱

- 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

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

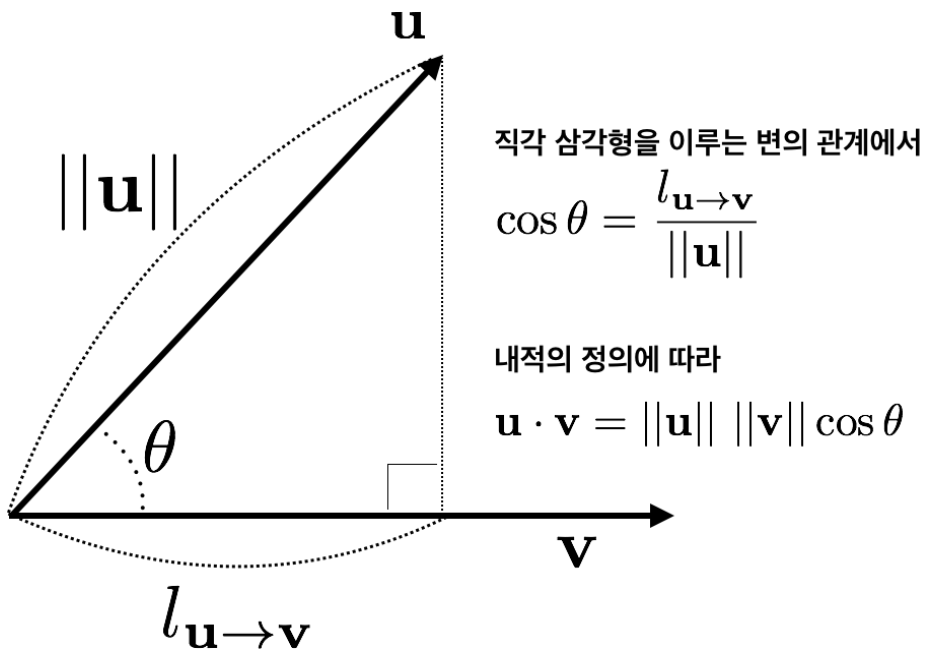
- 가위곱
  - 외적, 벡터곱이라고도 함
  - 두벡터를 곱해 새로운 벡터를 얻을 수 있음
  - 좌표계가 변해도 크기와 방향이 동일한 결과

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

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