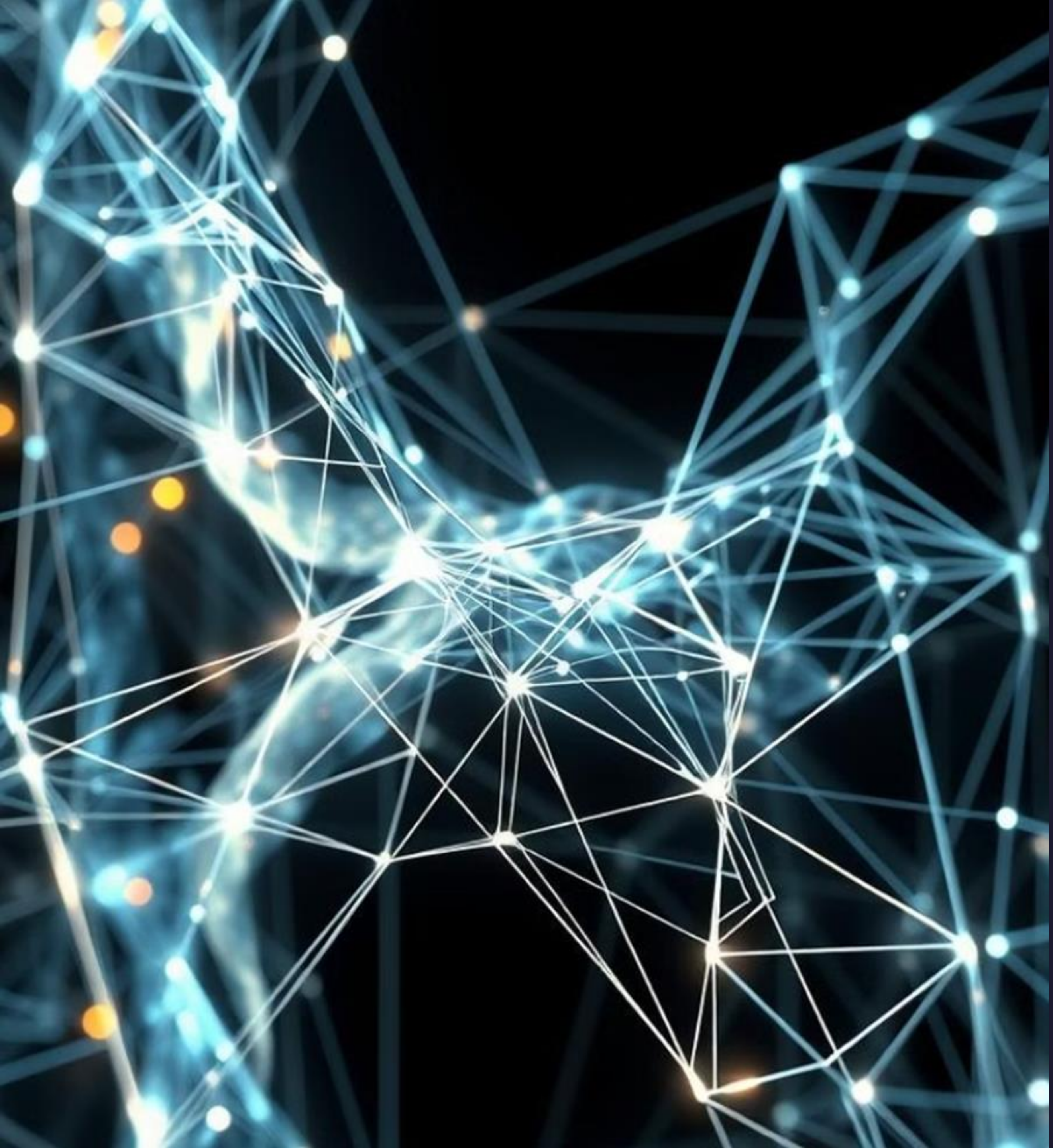


Linear Algebra for ML

Hossam Ahmed

Ziad Waleed

Mario Mamdouh

NumPy

IEEE FCI - Helwan Student Branch

Agenda

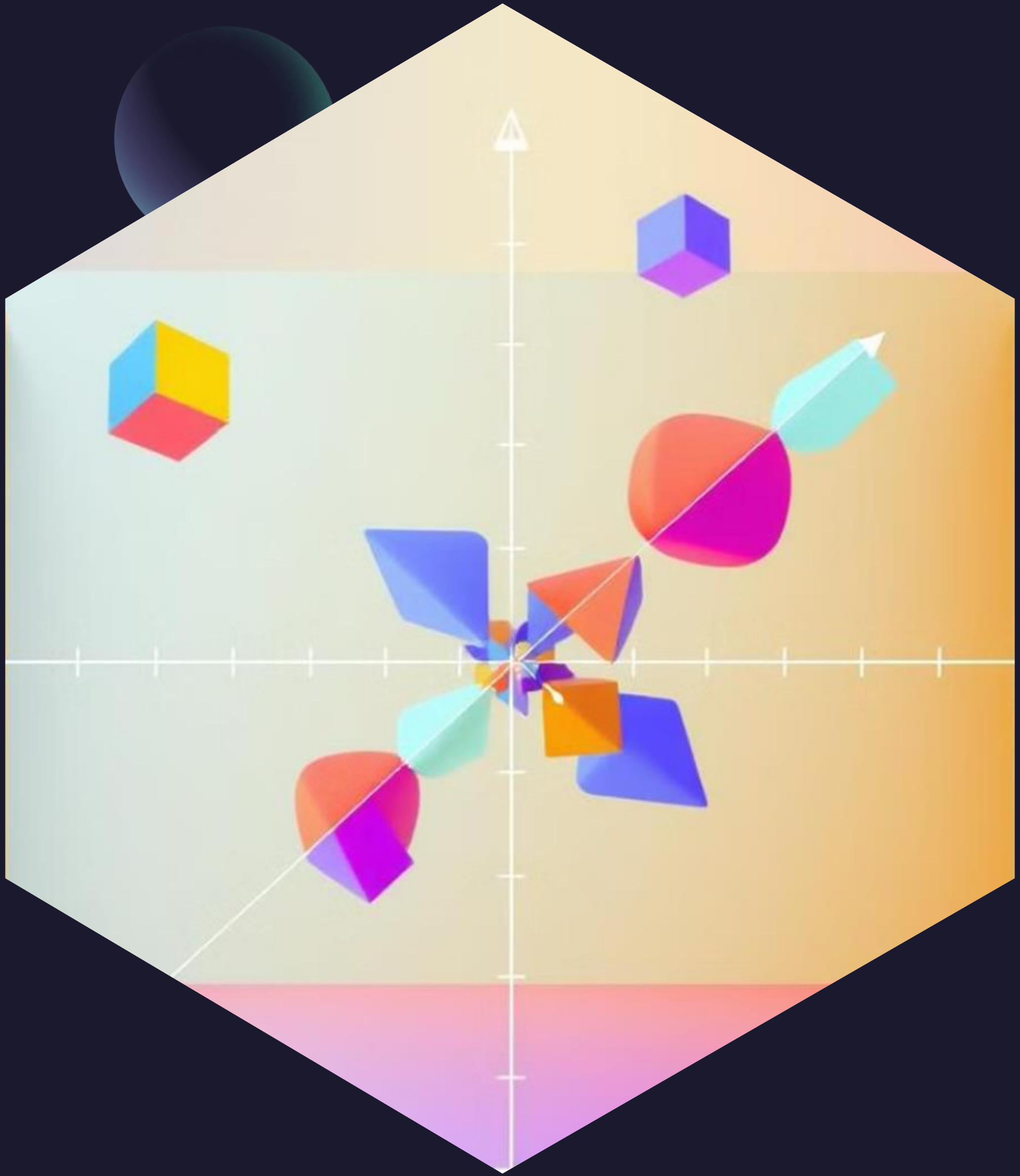
- Vectors
- Linear Combination
- Linear Transformation
- Composition of Linear Transformation
- Matrix multiplications
- Norms
- Norms and distance
- Dot product
- Dot product and similarity
- Normalization of vectors
- Determinants

Motivation

- Linear Algebra, is not an isolated mathematical field.

- It’s one of the most applicable fields of math.
  - Anything involves space and dimensions would involve linear algebra.
    - Even games.

- And data points have dimensions, data is some sort of space that we can transform.



Vectors

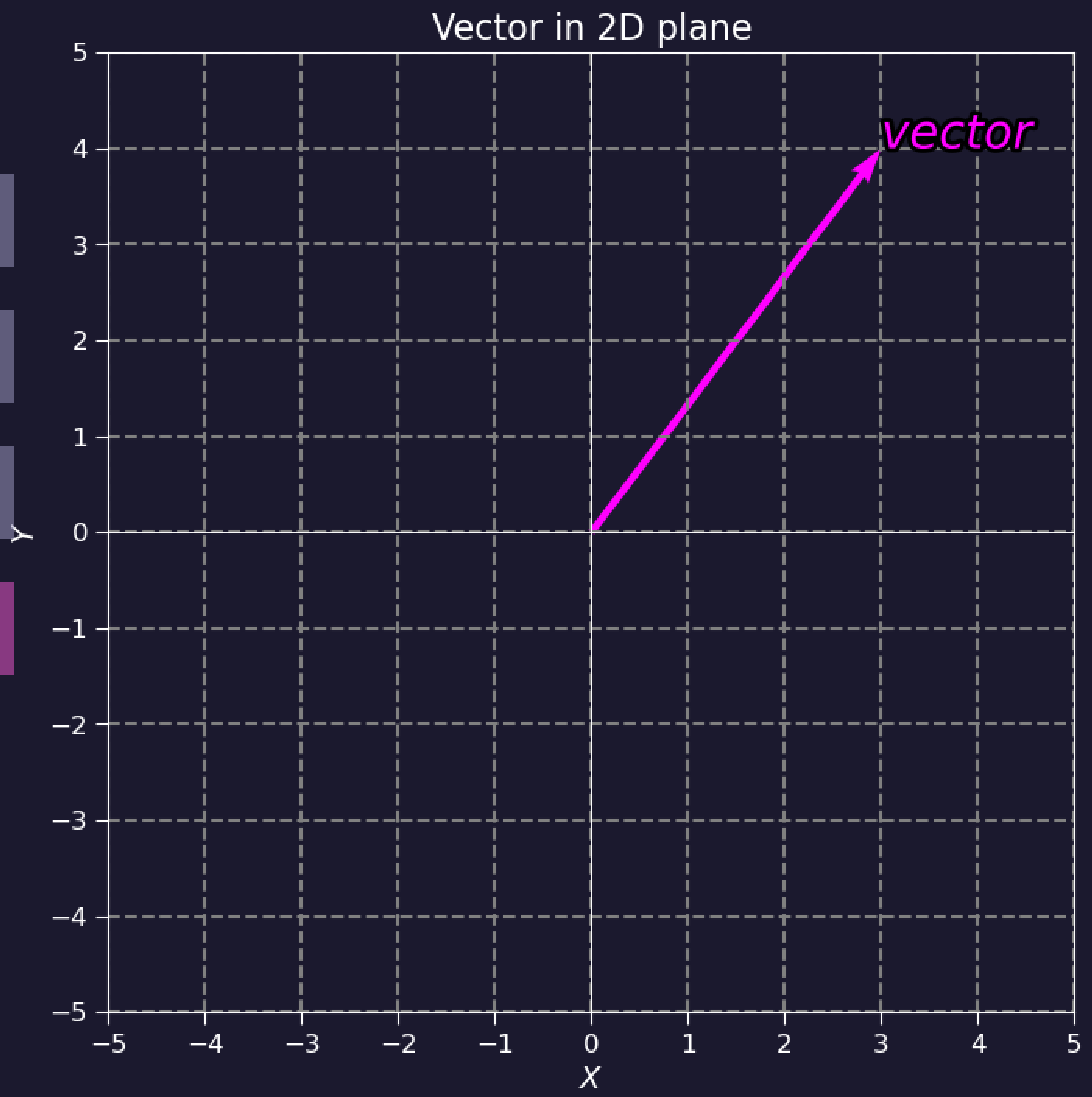
A **vector** is a tuple of one or more values called scalars.

A **vector** has both **magnitude** and **direction**.

We can present any 2D vector with 2 elements.

Can you present the vector with 2 elements?

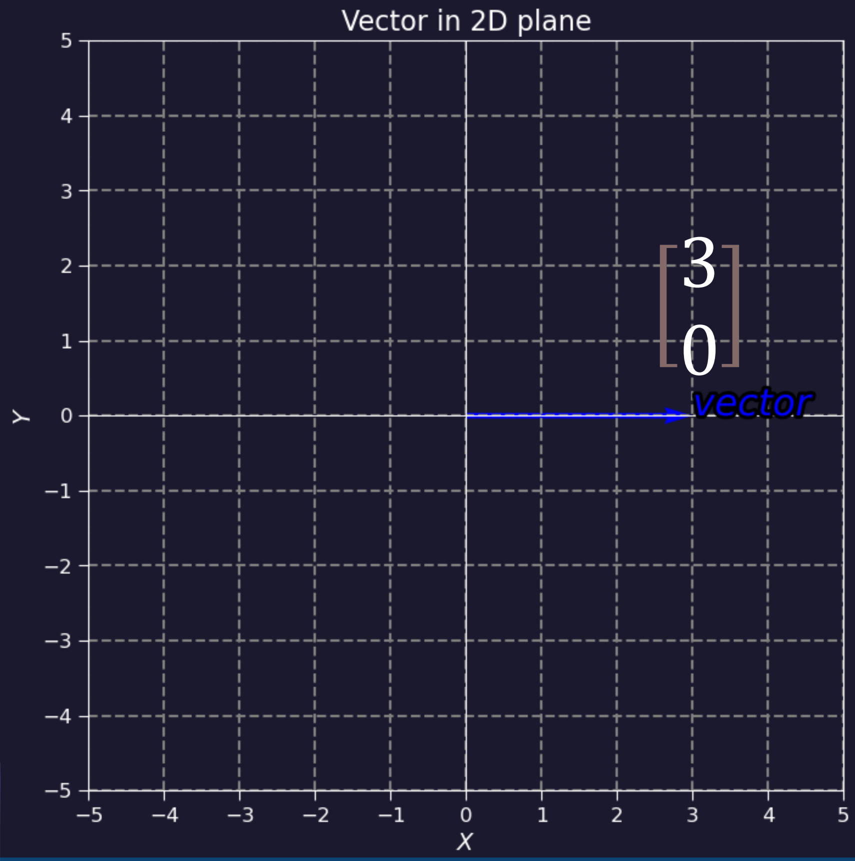
$$\begin{bmatrix} 3 \\ 4 \end{bmatrix}$$



Vectors

Because you shifted from the origin in the x-axis by 3

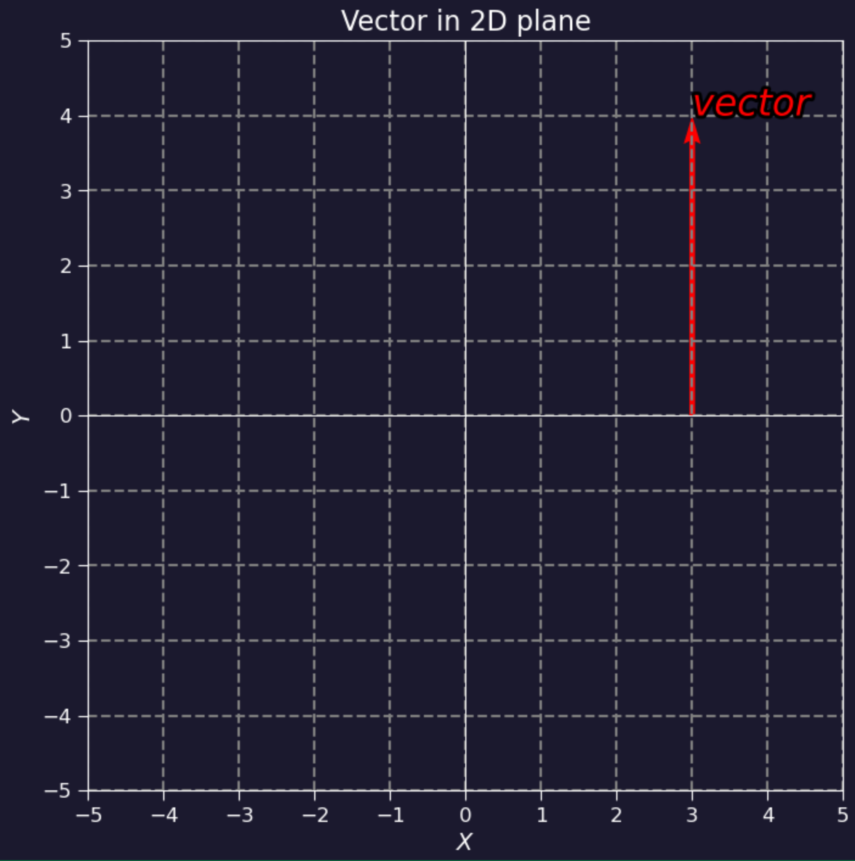
What about this one ?



$$\begin{bmatrix}3\\0\end{bmatrix}$$

Draw another vector where the blue one ended.

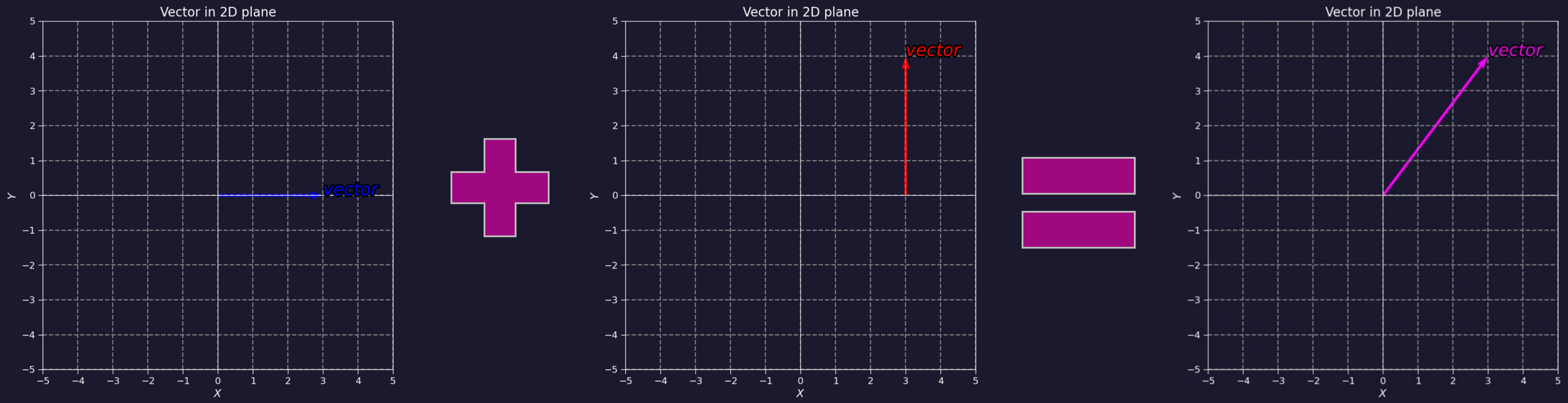
What about this one ? $\cancel{\begin{bmatrix}0\\4\end{bmatrix}}$?



$$\begin{bmatrix}3\\4\end{bmatrix}$$

consider the vector from its tip relative to the origin

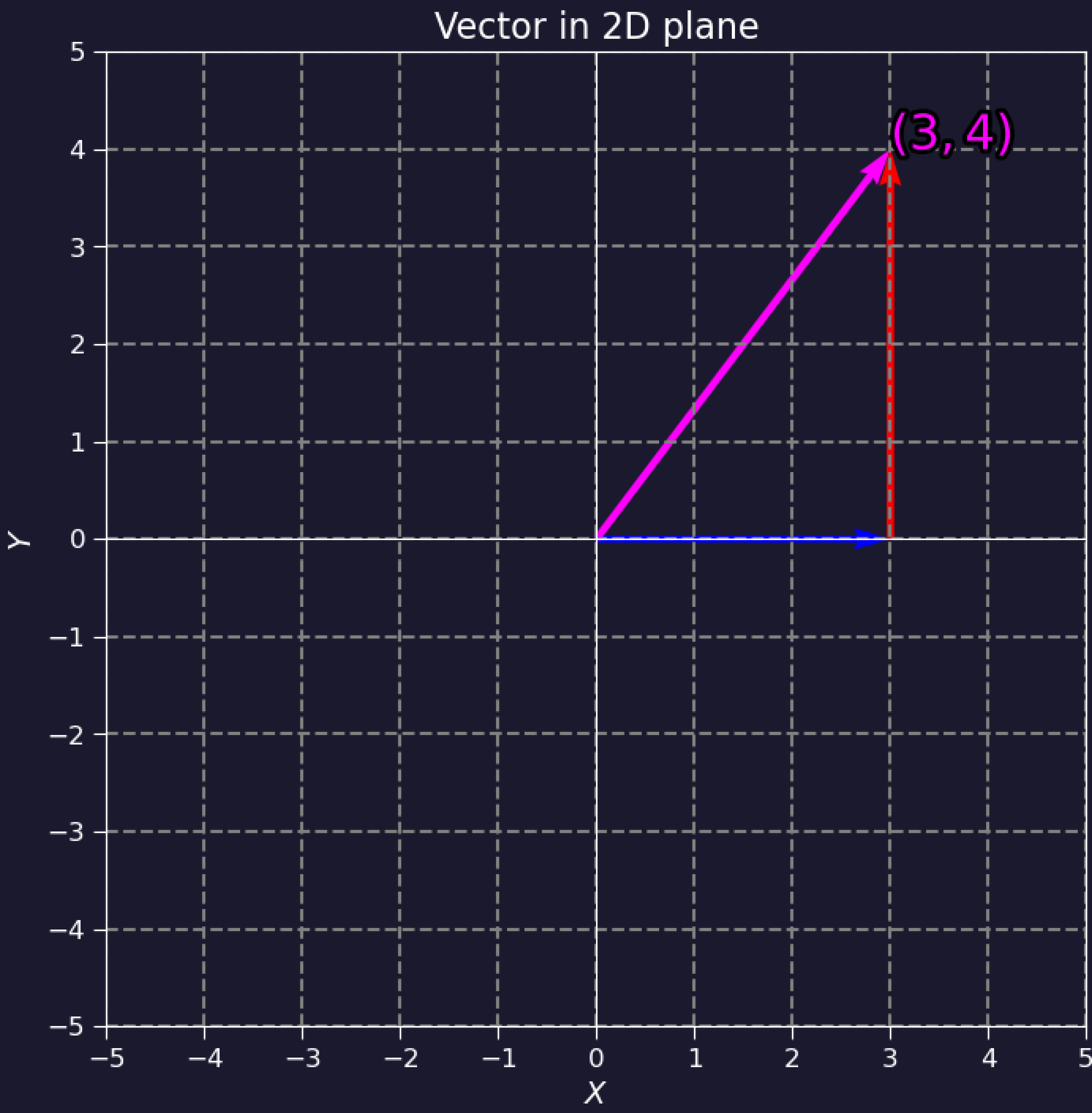
Vectors



$$\begin{bmatrix} 3 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

Vectors

$$\begin{bmatrix} 3 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$



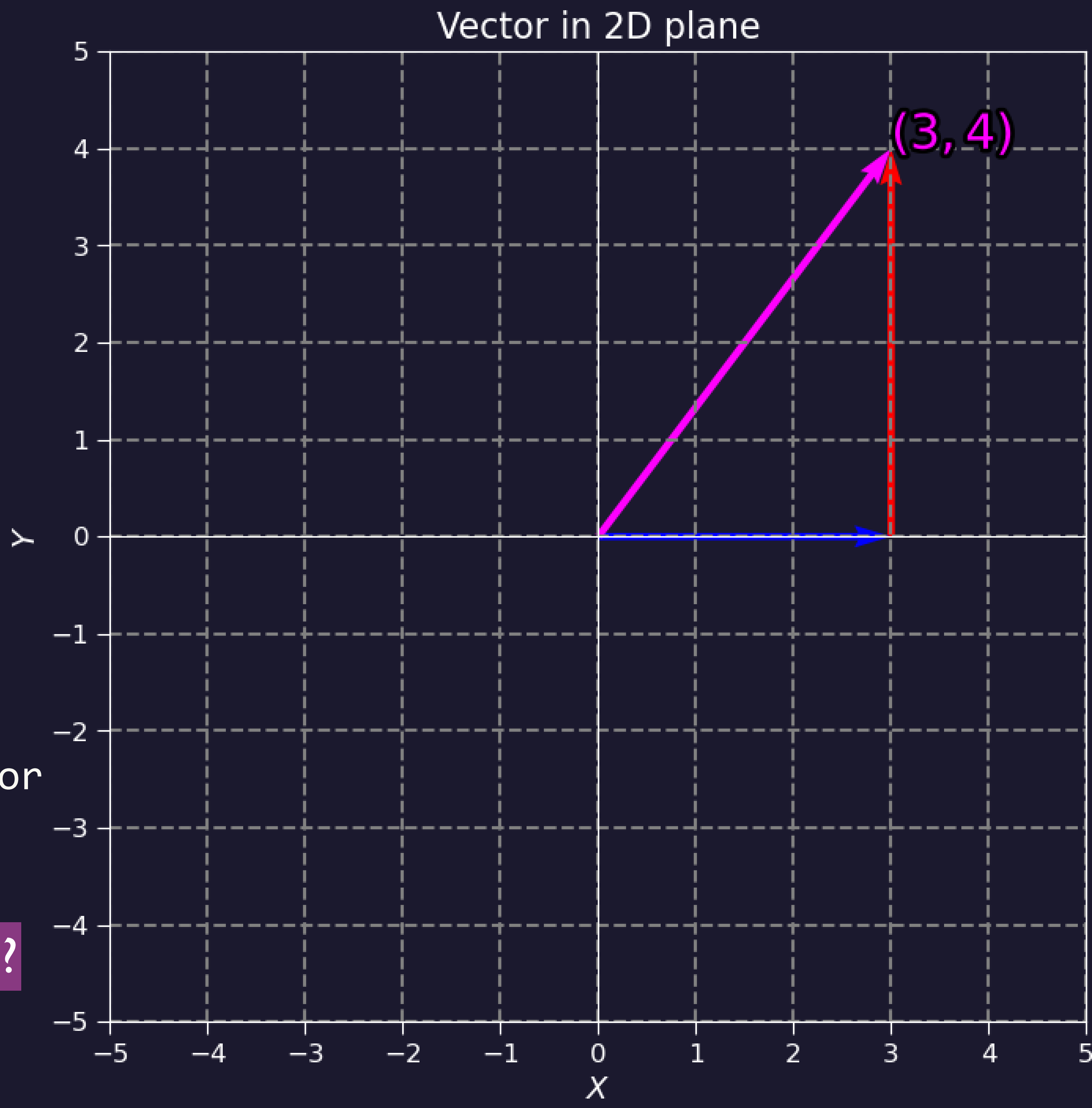
IEEE FCI - Helwan Student Branch

Vectors NumPy

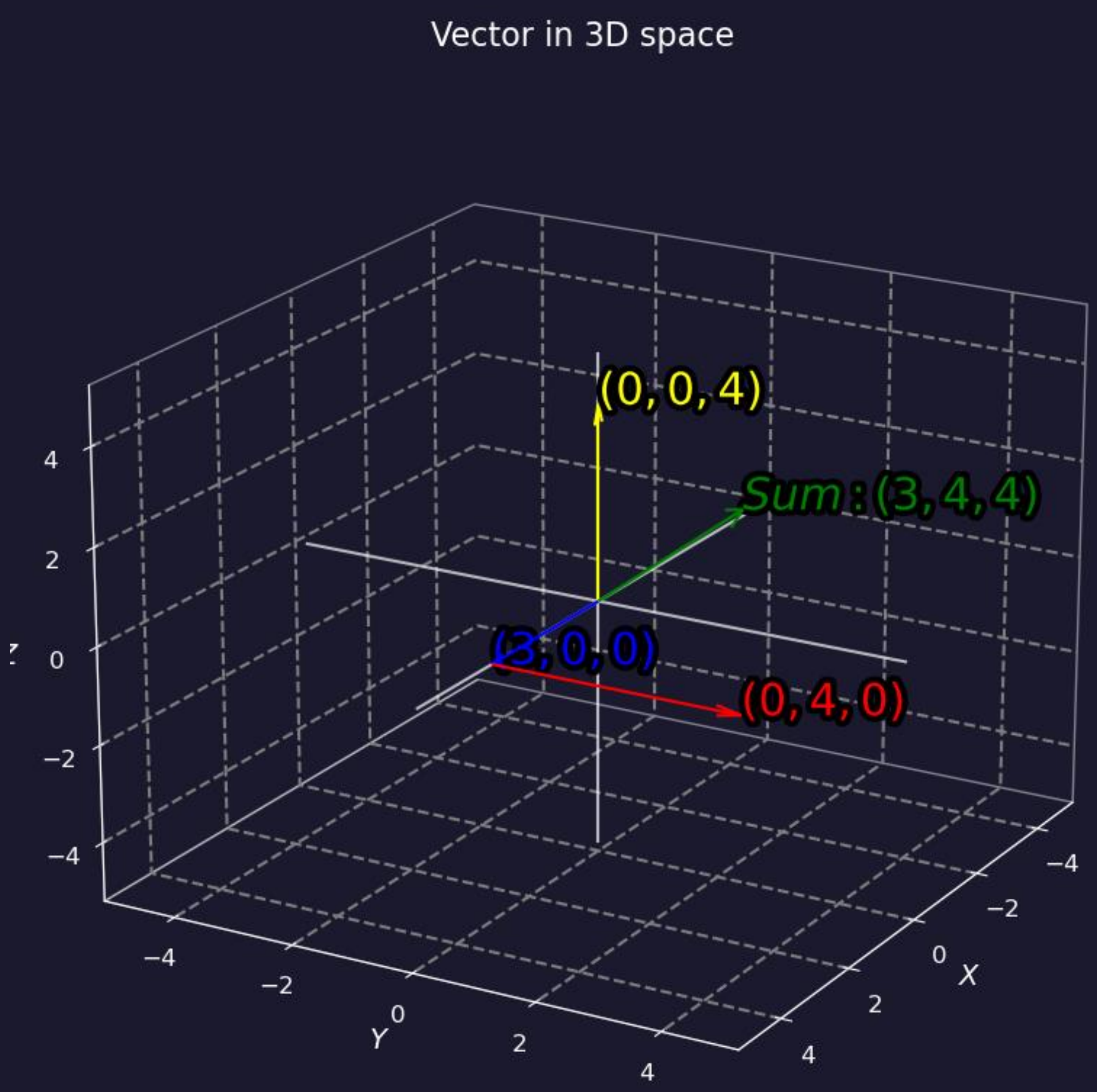
$$\begin{bmatrix}3\\0\end{bmatrix} + \begin{bmatrix}0\\4\end{bmatrix} = \begin{bmatrix}3\\4\end{bmatrix}$$

```python
import numpy as np
blue_vector = np.array([3,0])
red_vector = np.array([0,4])
purple_vector = blue_vector + red_vector
print(purple_vector) # [3 4]
```

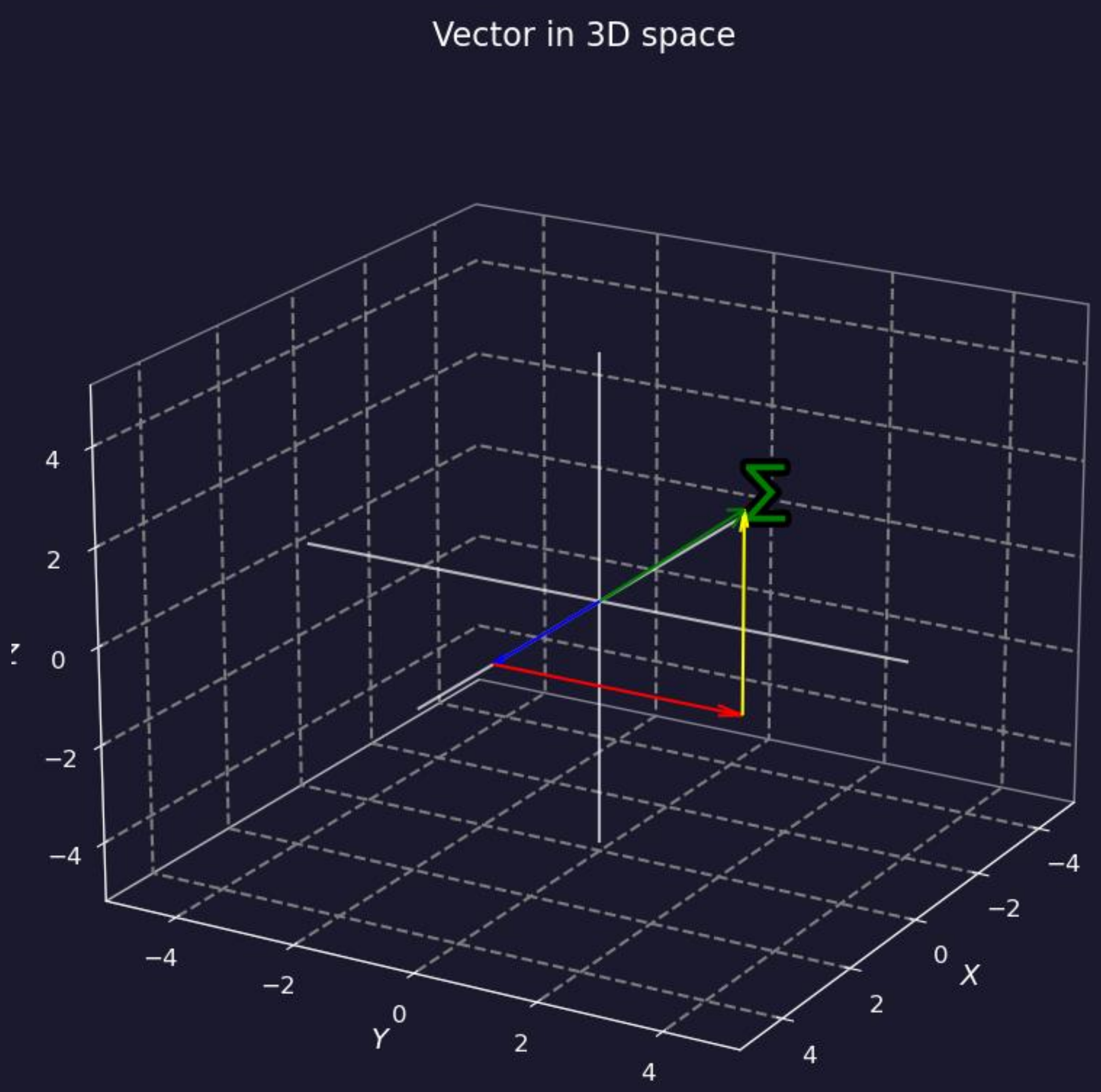
What is the different between a list & array ?



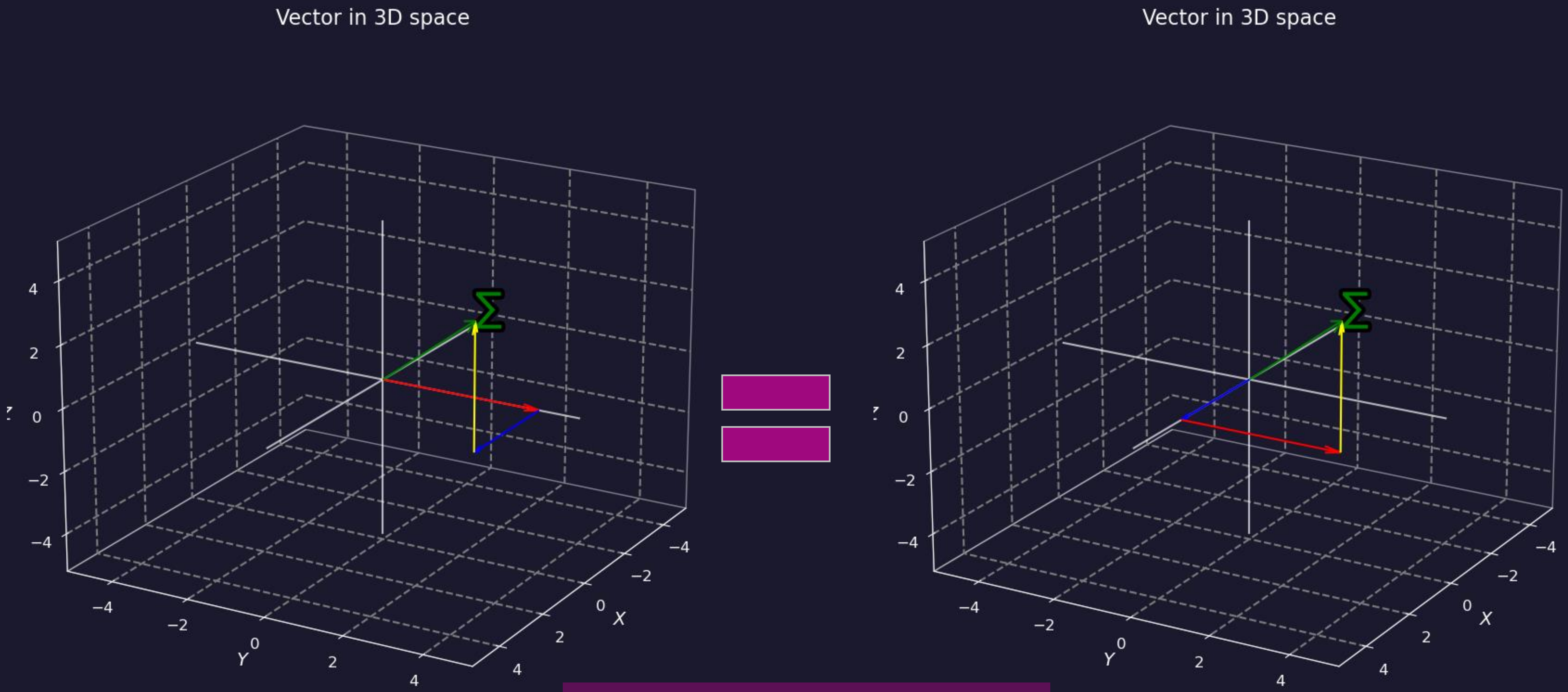
Vectors



Vector in 3D space
(0, 0, 4)
Sum : (3, 4, 4)
(3, 0, 0)
(0, 4, 0)
X
Y



Vector in 3D space
Σ
X
Y



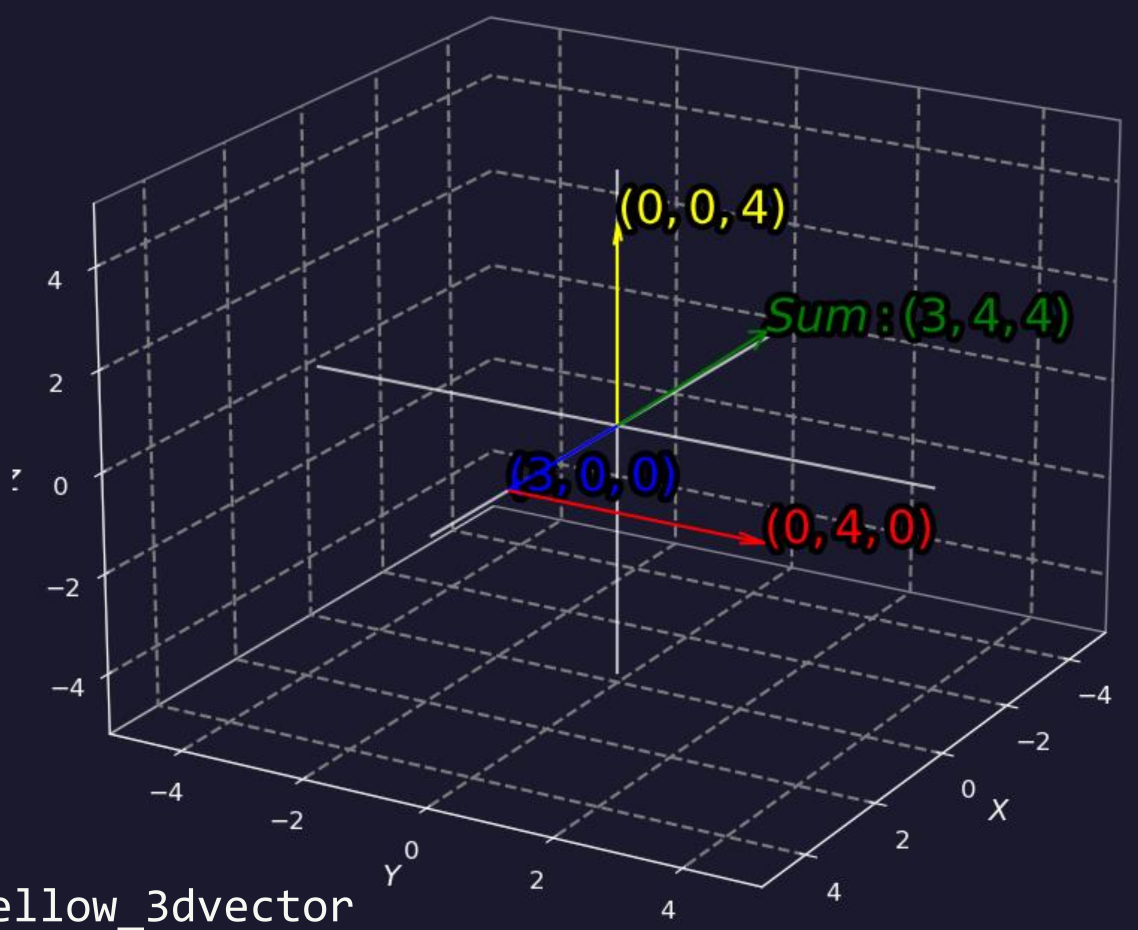
Doesn’t matter where you start !

Vectors NumPy

$$\begin{bmatrix}0\\0\\4\end{bmatrix}+\begin{bmatrix}3\\0\\0\end{bmatrix}+\begin{bmatrix}0\\4\\0\end{bmatrix}=\begin{bmatrix}3\\4\\4\end{bmatrix}$$

```
import numpy as np
blue_3dvector = np.array([3,0,0])
red_3dvector = np.array([0,4,0])
yellow_3dvector = np.array([0,0,4])
sum_3dvector = blue_3dvector + red_3dvector + yellow_3dvector
print(sum_3dvector) # [3 4 4]
```

Vector in 3D space



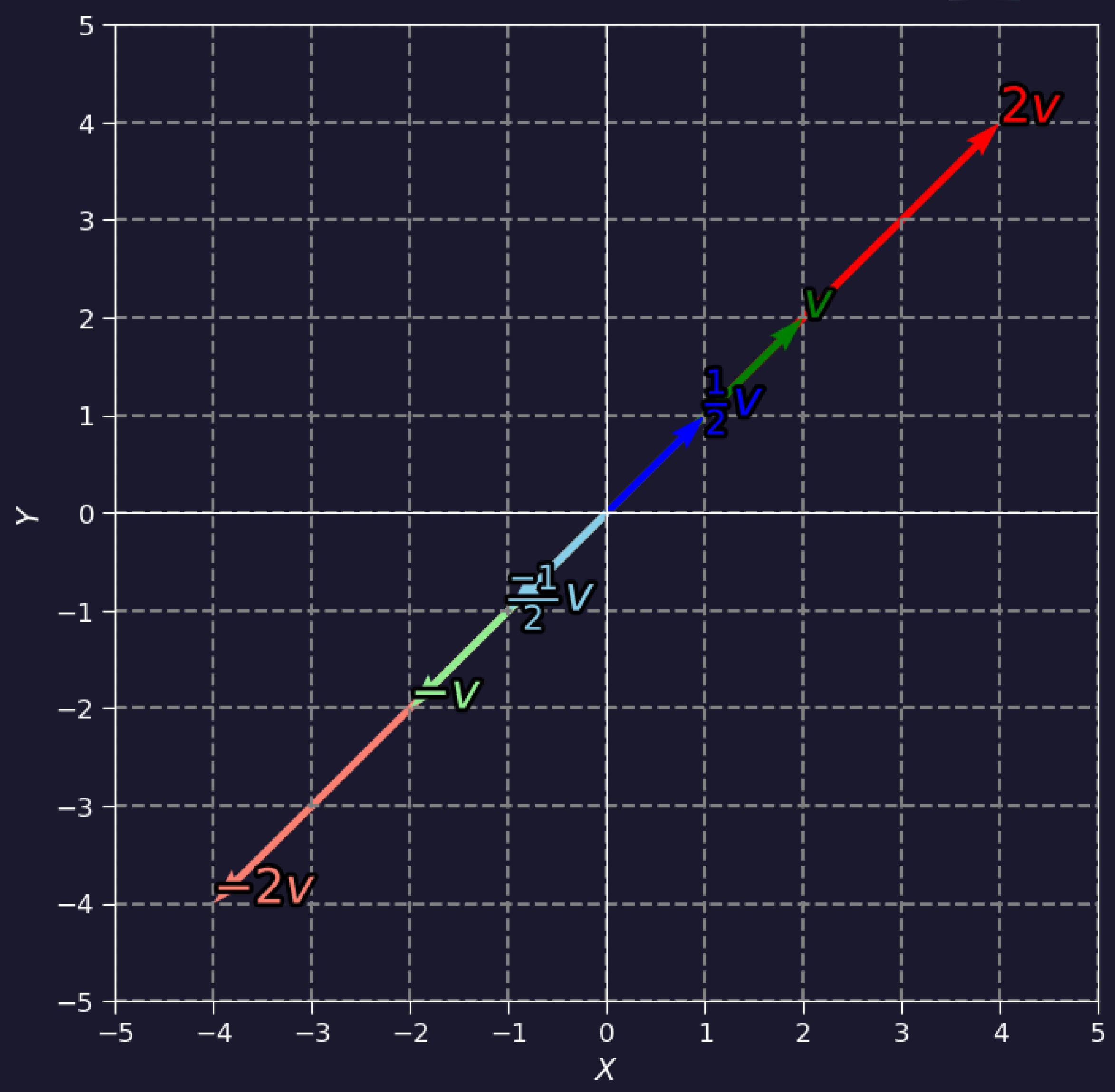
Vectors NumPy

$$v = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

$$, \frac{1}{2}v = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, 2v = \begin{bmatrix} 4 \\ 4 \end{bmatrix}$$

```python
import numpy as np
v = np.array([2,2])
v2 = 2*v
v1_2 = 0.5*v
print(v, v2, v1_2) # [2 2] [4 4] [1. 1.]
```

Why *[1. 1.]* nor *[1 1]*?



Vectors NumPy

```
v = np.array([4, 2])
print(v*0.5*v) #[8. 2.]
```

```
v = np.array([4, 2])
print((v*v)//2) # same output as the above ❓
```

$$v = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$$

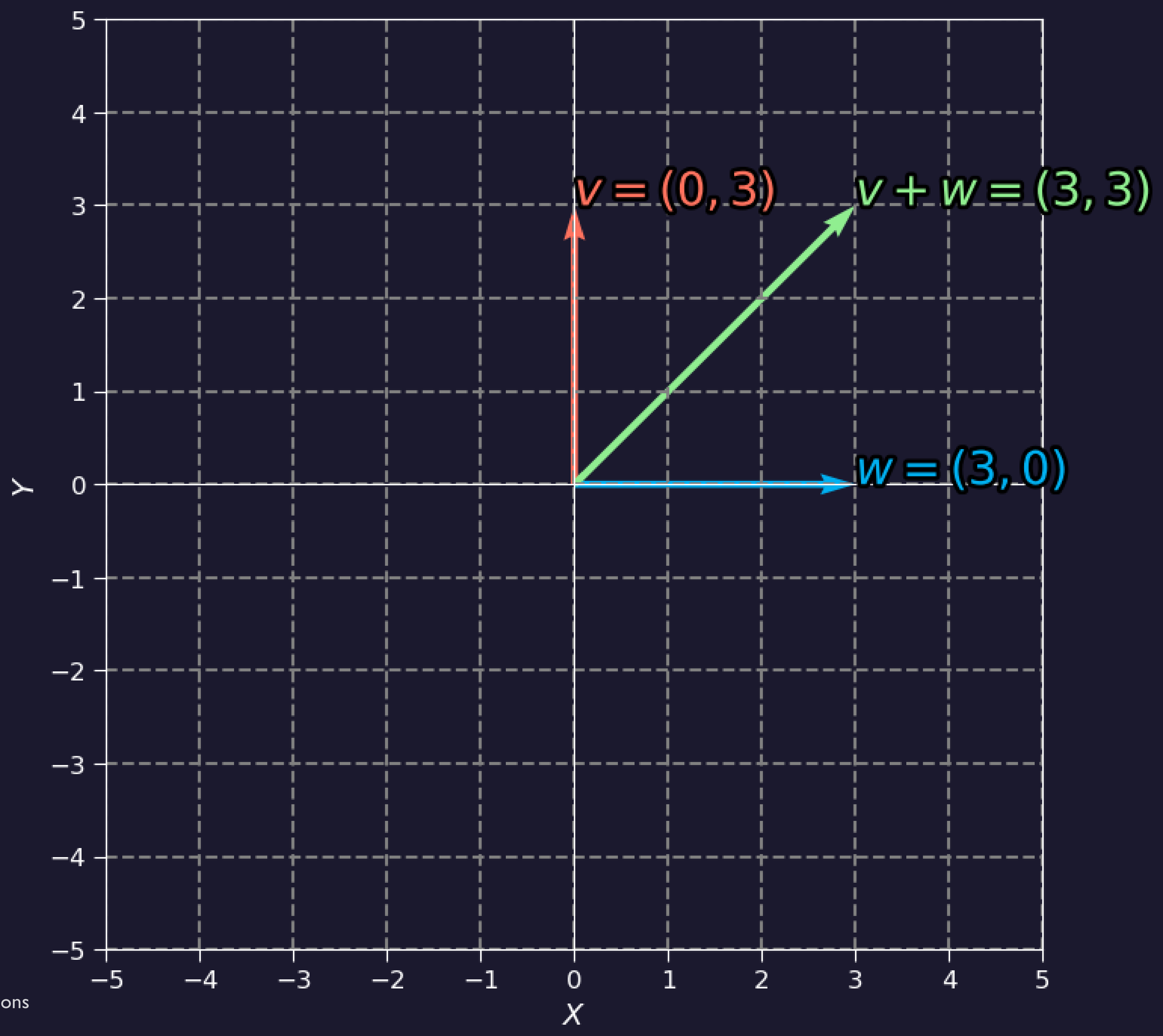
$$v * \frac{1}{2} v = \frac{1}{2} v^2 = \frac{1}{2} \begin{bmatrix} 4 \\ 2 \end{bmatrix}^2 = \frac{1}{2} \begin{bmatrix} 16 \\ 4 \end{bmatrix} = \begin{bmatrix} 8 \\ 2 \end{bmatrix}$$

Two vectors of equal length can be multiplied together, c = a × b , as with addition and subtraction, this operation is performed **element-wise** to result in a new vector of the same length.

IEEE FCI - Helwan Student Branch

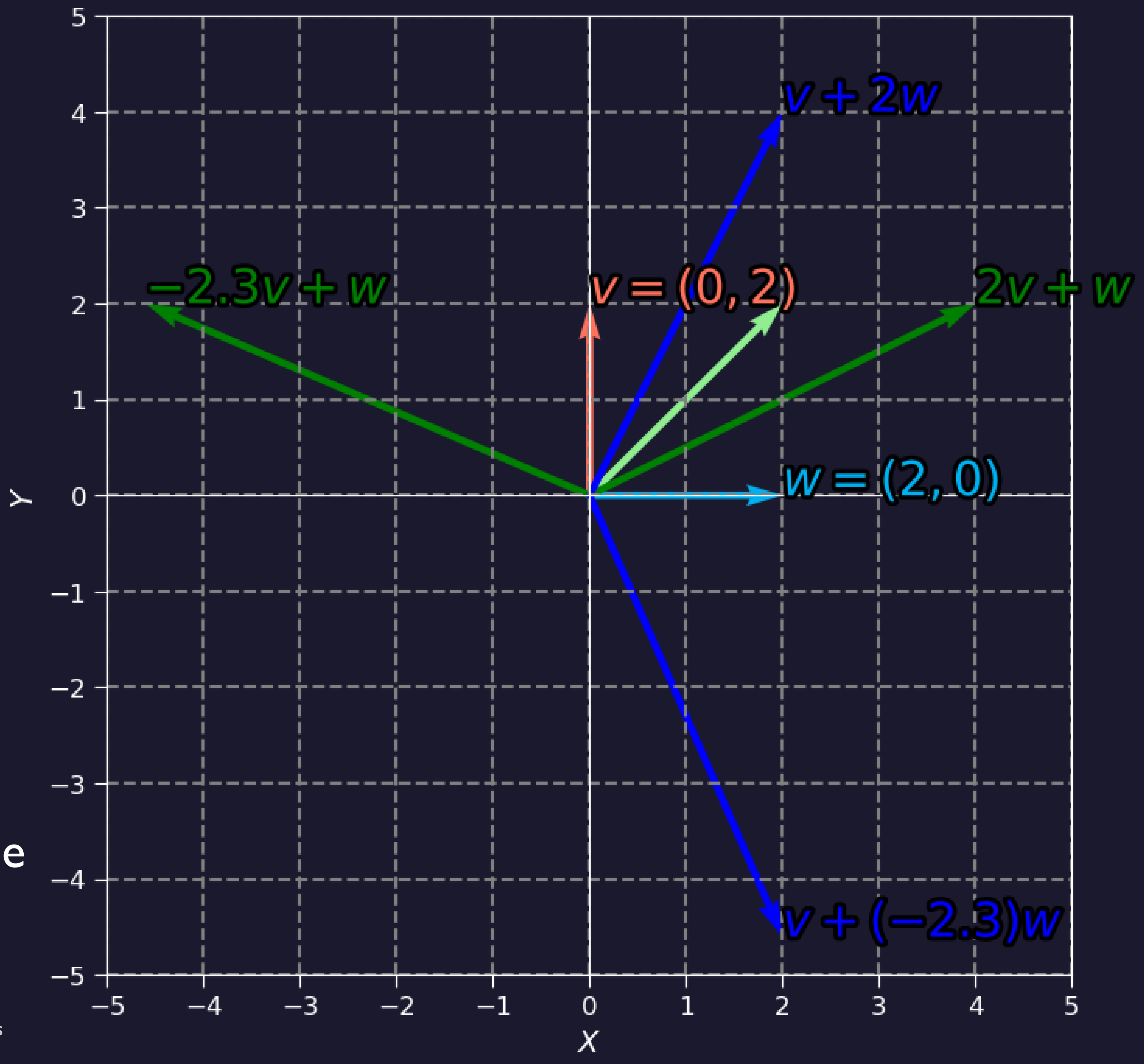
Linear Combination

- We have two vectors v and w.
- Add v and w.
- Scale only v and fix w then add them.
- Reverse the previous step.



Linear Combination

- We have two vectors v and w.
- Add v and w.
- We consider v and w constants that would be multiplied with a varied scalers to be capable of spanning the entire plane.
- v and w shouldn't be linear independent to be capable of spanning the entire plane.
- Scaling produce vectors on the same line, only addition can rotate.



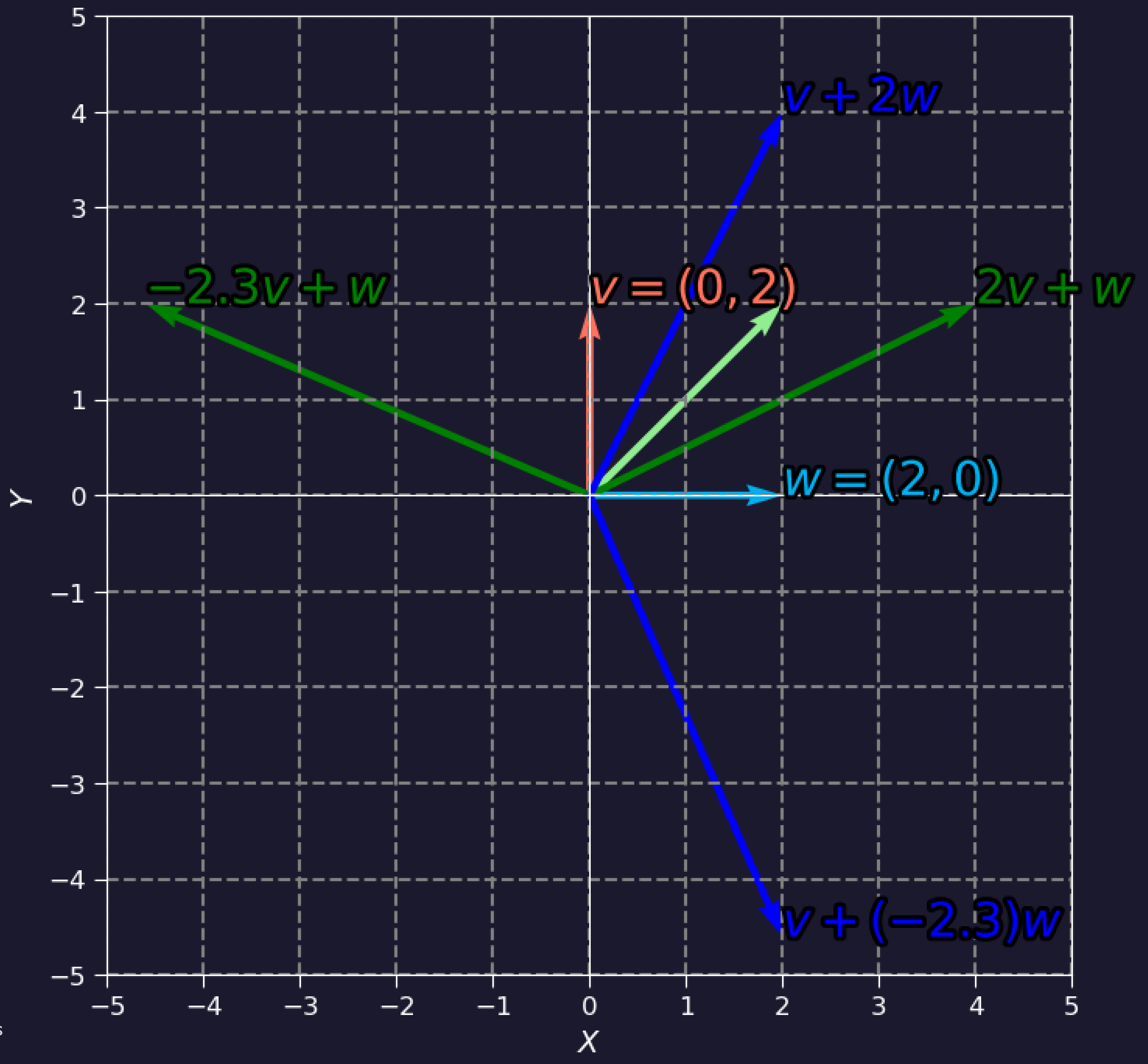
Linear Combination

- **A linear combination** is an expression formed by multiplying each term in a set by a constant and adding the results.

- The span of v and w is the set of all their linear combination.

- $a\vec{v} + b\vec{w}$ Let a and b varies over the real numbers.

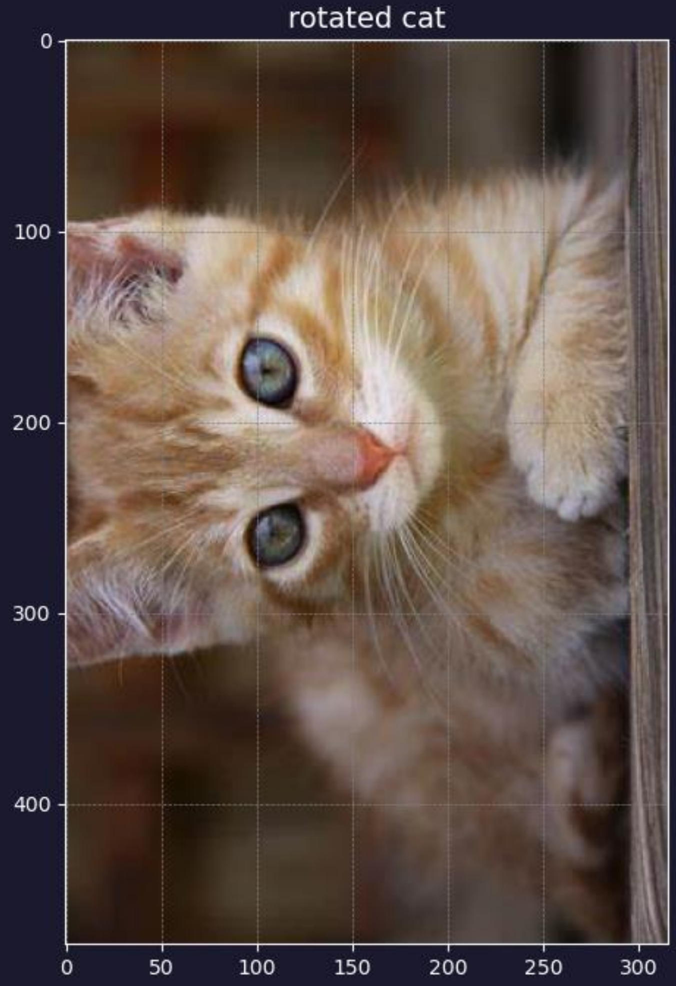
- Linear independence



Linear Transformation

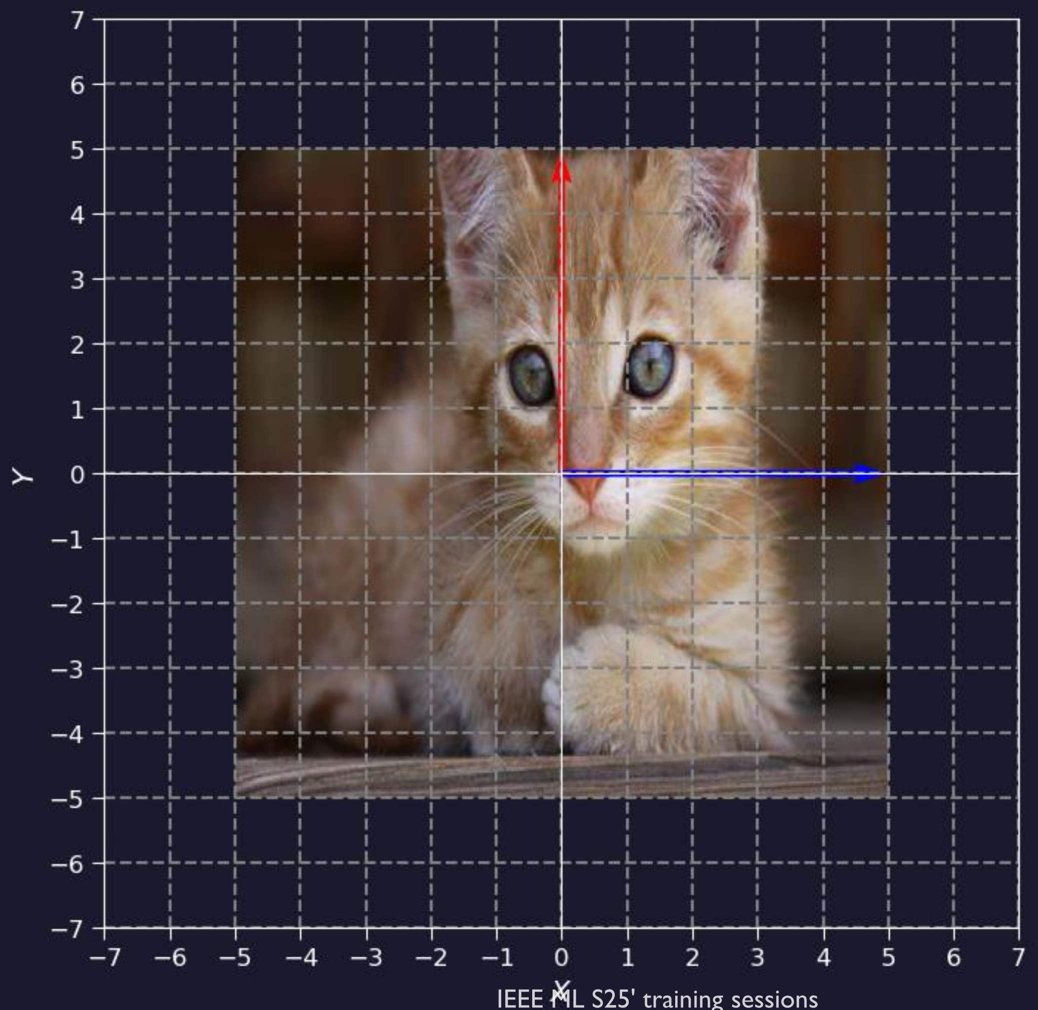
- **In many cases we need to apply some function or transformation on the data to change our perspective.**

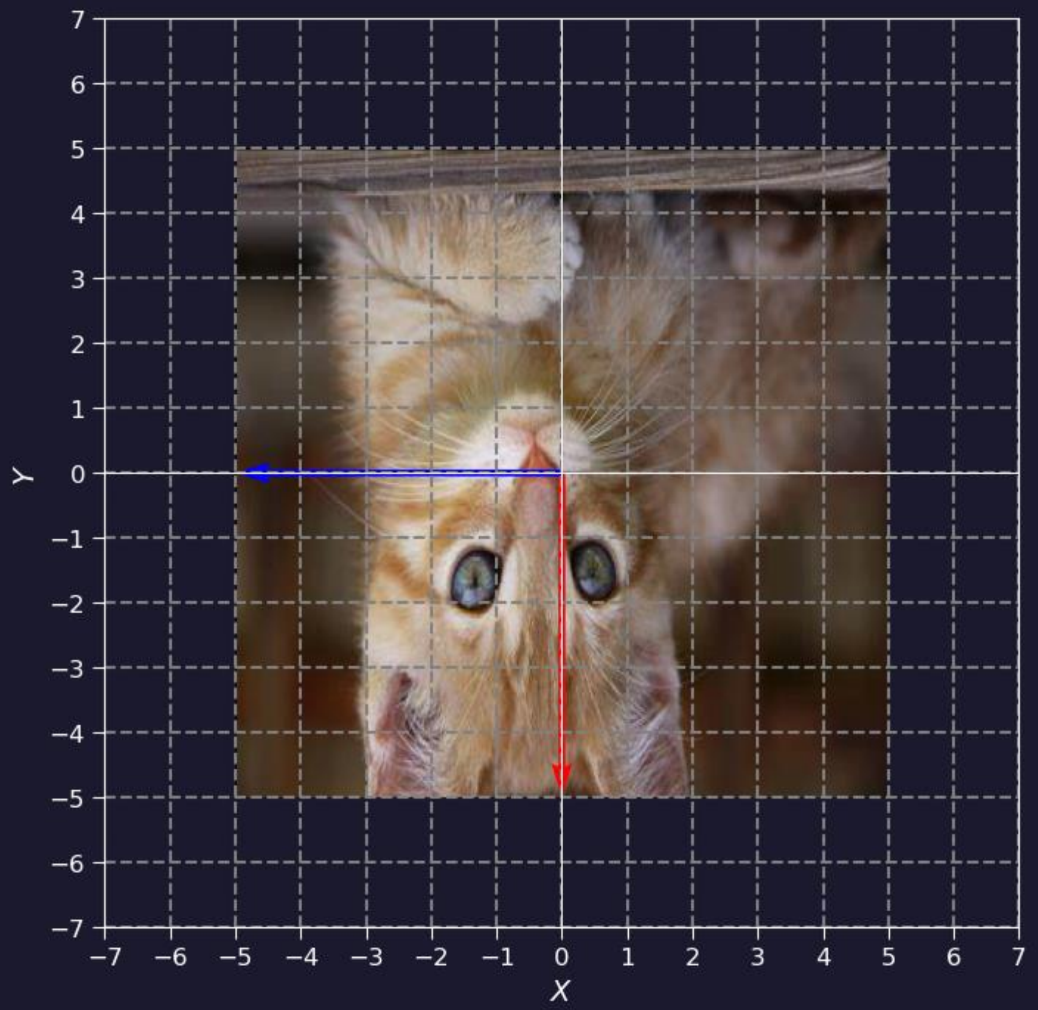
Transformation



Linear Transformation

- **What did change in the image or shall we say the space?**

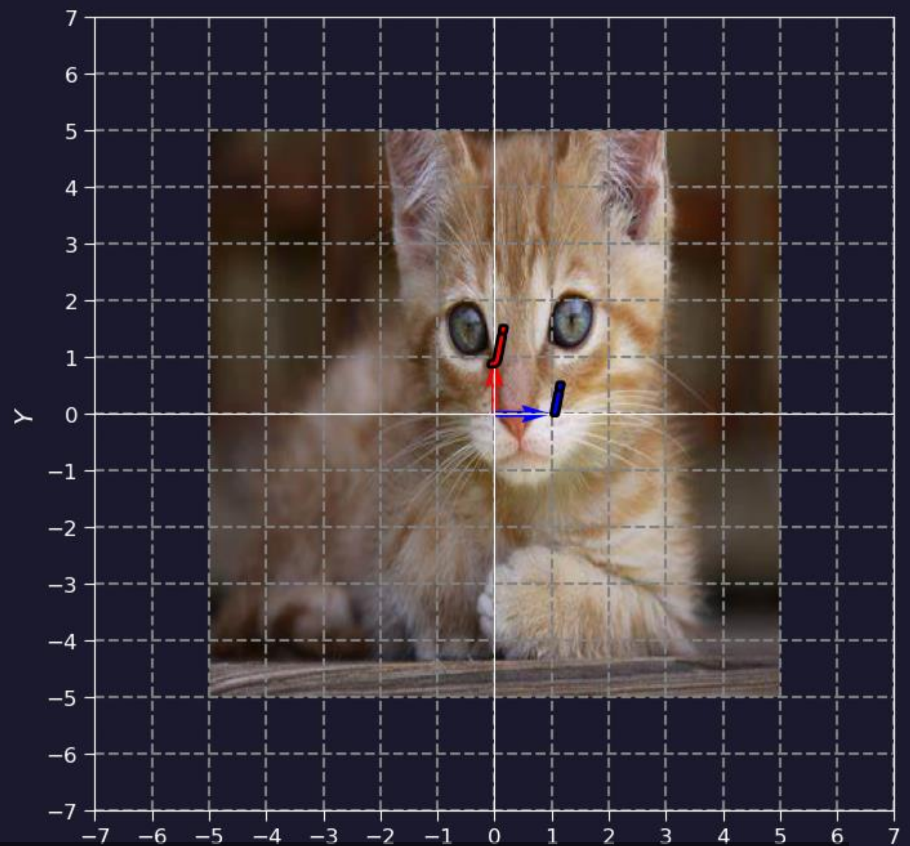


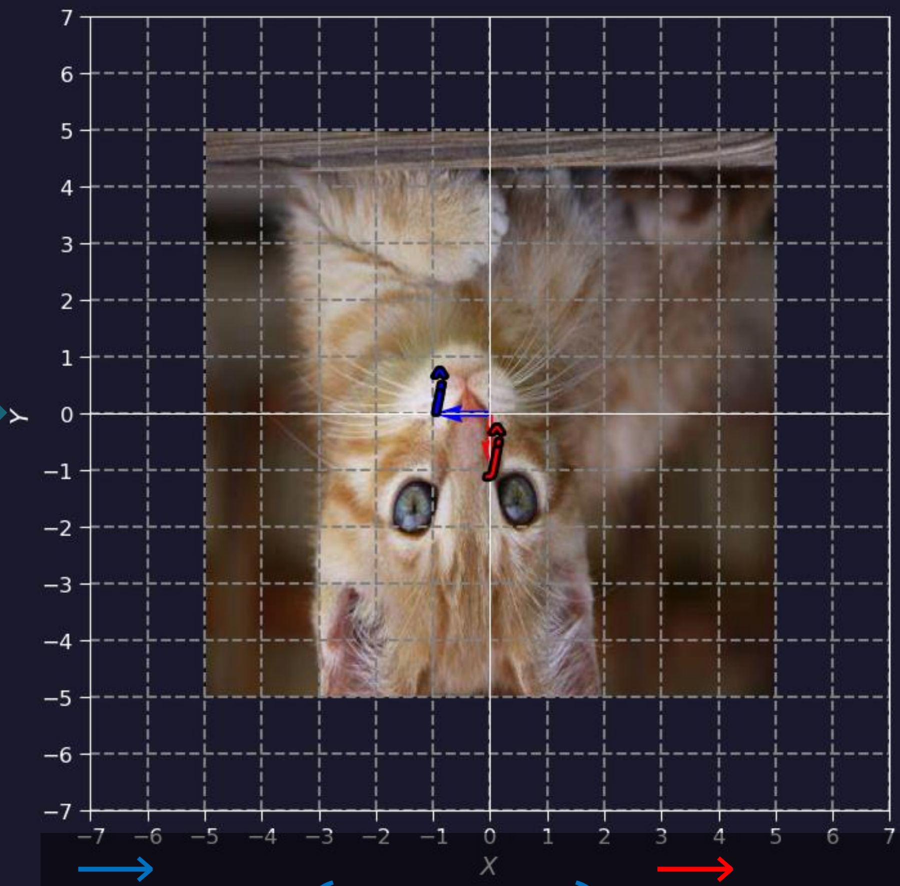


$\vec{v} = a\vec{i} + b\vec{j}$

Linear Transformation

- **If you know the basis vectors we can know where each point will go after the rotation of 180 degree.**





$\vec{i} = (0,1), \vec{j} = (1,0)$

$\overrightarrow{i_t} = (0,-1), \overrightarrow{j_t} = (-1,0)$

$$\vec{v} = a\vec{i} + b\vec{j}$$

Linear Transformation

After we rotate the image where this vector would go?

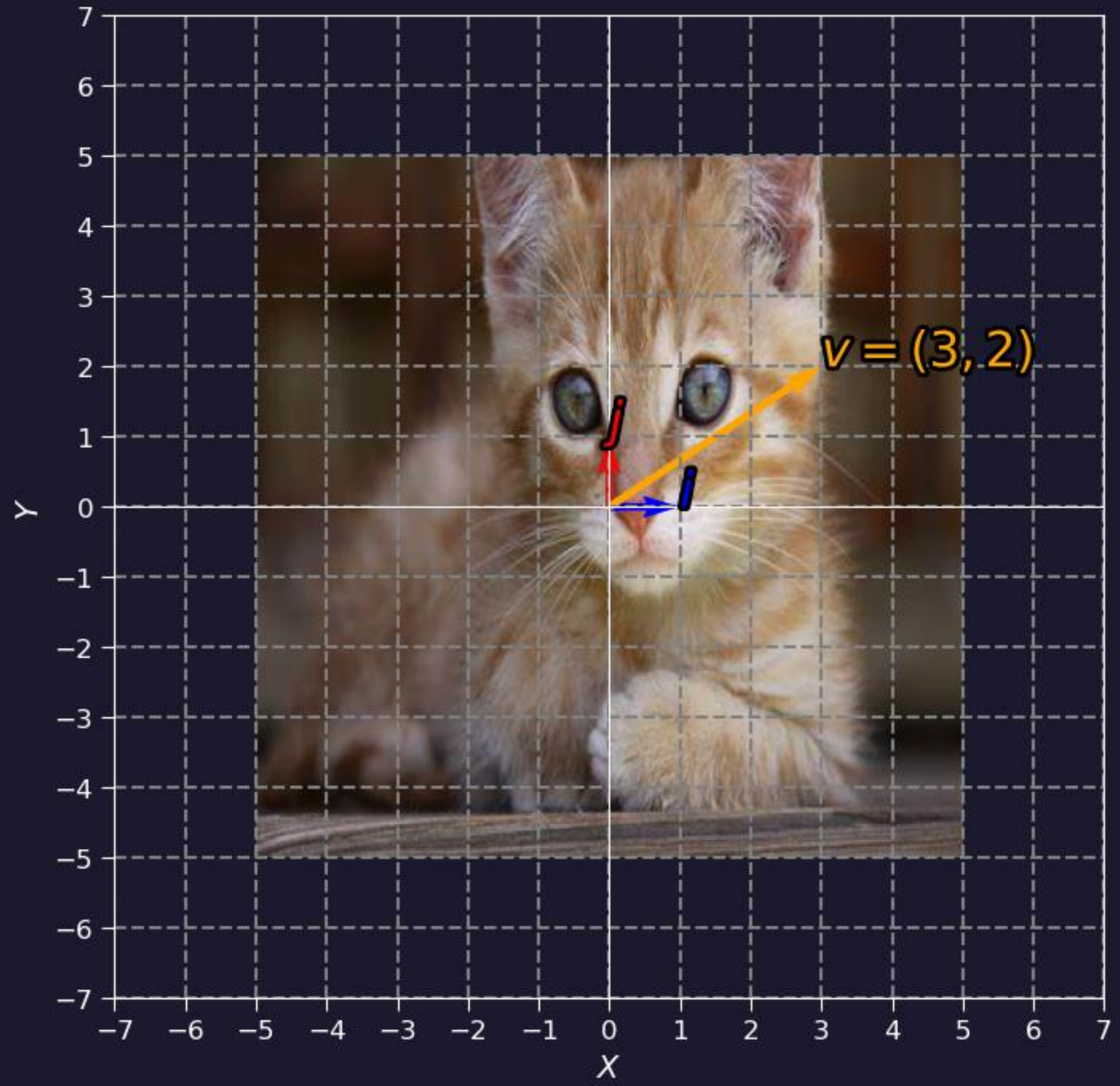
$$\vec{v} = (3,2) \qquad \vec{v} = 3\vec{i} + 2\vec{j}$$

$$\overrightarrow{i_t} = (0,-1), \overrightarrow{j_t} = (-1,0)$$

$$\overrightarrow{v_t} = 3\overrightarrow{i_t} + 2\overrightarrow{j_t}$$

$$\overrightarrow{v_t} = 3\begin{bmatrix} 0 \\ -1 \end{bmatrix} + 2\begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

$$\overrightarrow{v_t} = \begin{bmatrix} 0 \\ -3 \end{bmatrix} + \begin{bmatrix} -2 \\ 0 \end{bmatrix} = \begin{bmatrix} -2 \\ -3 \end{bmatrix}$$



IEEE FCI - Helwan Student Branch

$$\vec{v} = a\vec{i} + b\vec{j}$$

Linear Transformation

After we rotate the image where this vector would go?

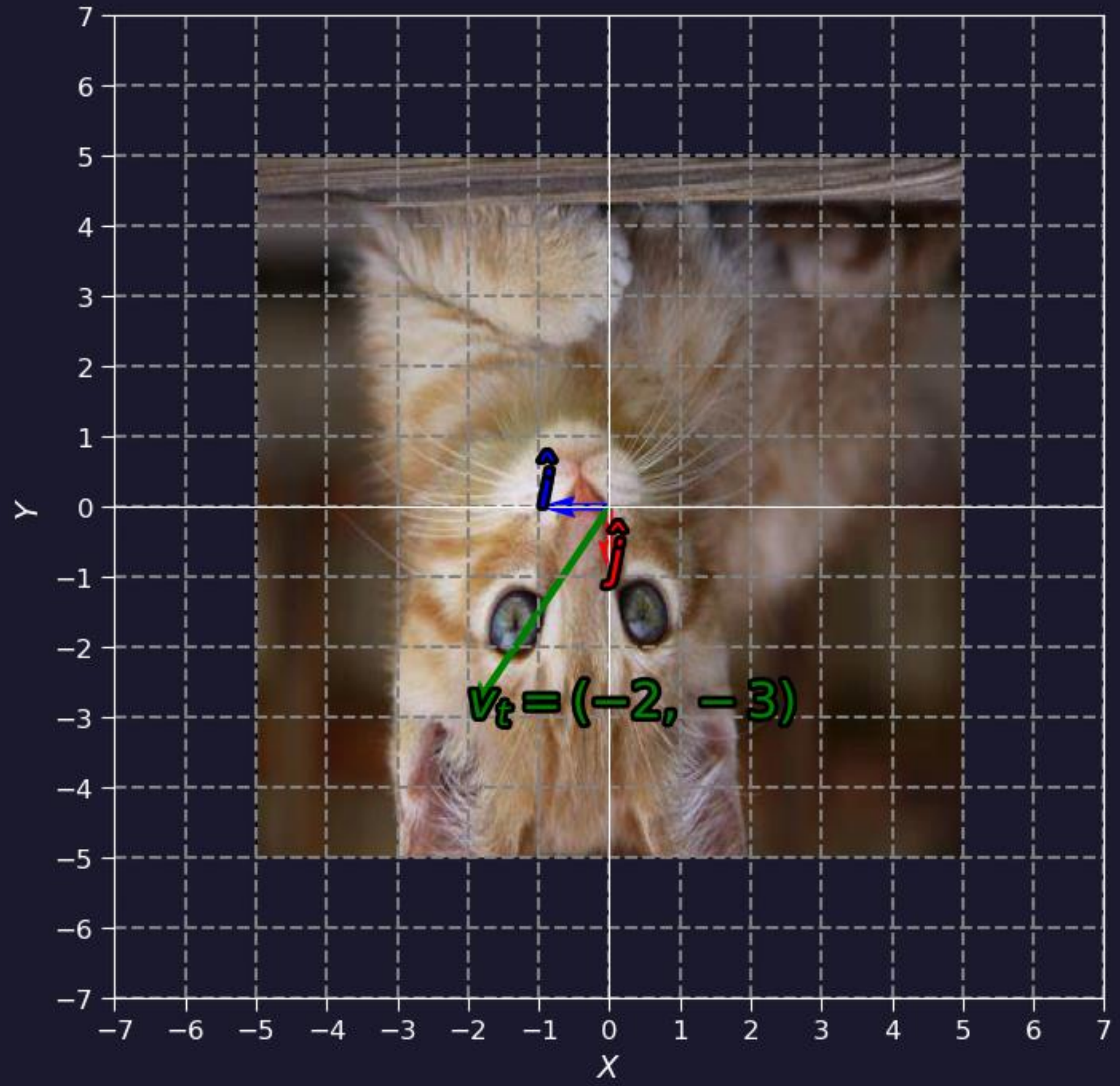
$\vec{v} = (3,2)$ $\vec{v} = 3\vec{i} + 2\vec{j}$

$$\overrightarrow{i_t} = (0,-1), \overrightarrow{j_t} = (-1,0)$$

$$\overrightarrow{v_t} = 3\overrightarrow{i_t} + 2\overrightarrow{j_t}$$

$$\overrightarrow{v_t} = 3\begin{bmatrix}0\\-1\end{bmatrix} + 2\begin{bmatrix}-1\\0\end{bmatrix}$$

$$\overrightarrow{v_t} = \begin{bmatrix}0\\-3\end{bmatrix} + \begin{bmatrix}-2\\0\end{bmatrix} = \begin{bmatrix}-2\\-3\end{bmatrix}$$



Linear Transformation

- Linear transformations preserve **straight lines** and **parallelism** in the geometry of an image.
  - The **origin** must **remain fixed** that mean the origin is the same before and after the transformation.

- For a transformation $T: \mathbb{R}^n \to \mathbb{R}^m$ it must satisfy 2 conditions.

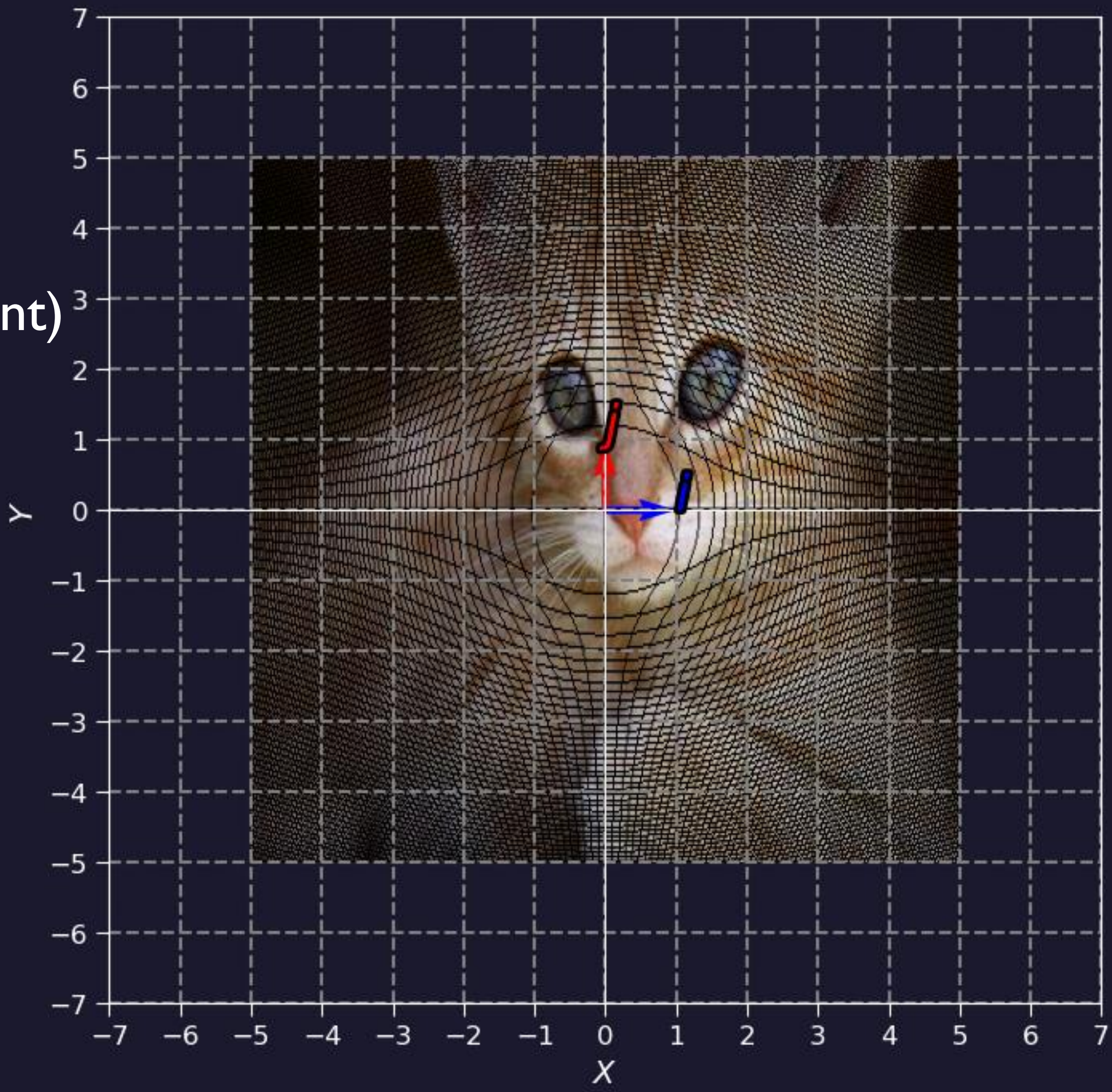
- **Additivity** (Preservation of vector addition)
  - For all vectors $v, \mathrm{u} \in \mathbb{R}^n$, the transformation T must satisfy
    - $\boldsymbol{T(v,u) = T(u) + T(v)}$

- **Homogeneity** (Preservation of scalar multiplication):
  - For any vector $\mathrm{u} \in \mathbb{R}^n$ and any scaler $\mathrm{c} \in \mathbb{R}$, the transformation T must satisfy
    - $\boldsymbol{T(cu) = cT(u)}$

Linear Transformation

- Linear transformation on image like
  - Translation (Shifts the image)
  - Rotation (Rotates the image around a point)
  - Scaling (Resize the image)
  - Shearing (Skews the image along an axis)
  - Reflection (Flips the image across an axis)

- Non-Linear Transformation
  - Barrel distortion is a lens effect that makes straight lines curve outward in wide-angle photos.
  - Radial Transformation ,twists the image in a radial or spiral manner



$$\vec{v} = a\vec{i} + b\vec{j}$$

Linear Transformation matrix

$$\vec{v} = \begin{bmatrix} a \\ b \end{bmatrix} \quad \vec{i} = \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} \quad \vec{j} = \begin{bmatrix} j_1 \\ j_2 \end{bmatrix} \quad \vec{i_t} = \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} \quad \vec{j_t} = \begin{bmatrix} j_1 \\ j_2 \end{bmatrix}$$

- Transformation matrix

$$\begin{bmatrix} i_1 & j_1 \\ i_2 & j_2 \end{bmatrix}$$

- Transform any vector to the new space.

Vector matrix multiplication

$$\begin{bmatrix} i_1 & j_1 \\ i_2 & j_2 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = a \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} + b \begin{bmatrix} j_1 \\ j_2 \end{bmatrix} = \begin{bmatrix} i_1 a & j_1 b \\ i_2 a & j_2 b \end{bmatrix}$$

Composition of Linear Transformation

- Transform any vector to the new space.

$$\begin{bmatrix} i_1 & j_1 \\ i_2 & j_2 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = a \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} + b \begin{bmatrix} j_1 \\ j_2 \end{bmatrix} = \begin{bmatrix} i_1 a & j_1 b \\ i_2 a & j_2 b \end{bmatrix} = \begin{bmatrix} a_t \\ b_t \end{bmatrix}$$

- What if we need to apply another transformation on the transformed vector ?
  - Like how you may rotate an image then flip or scale it.

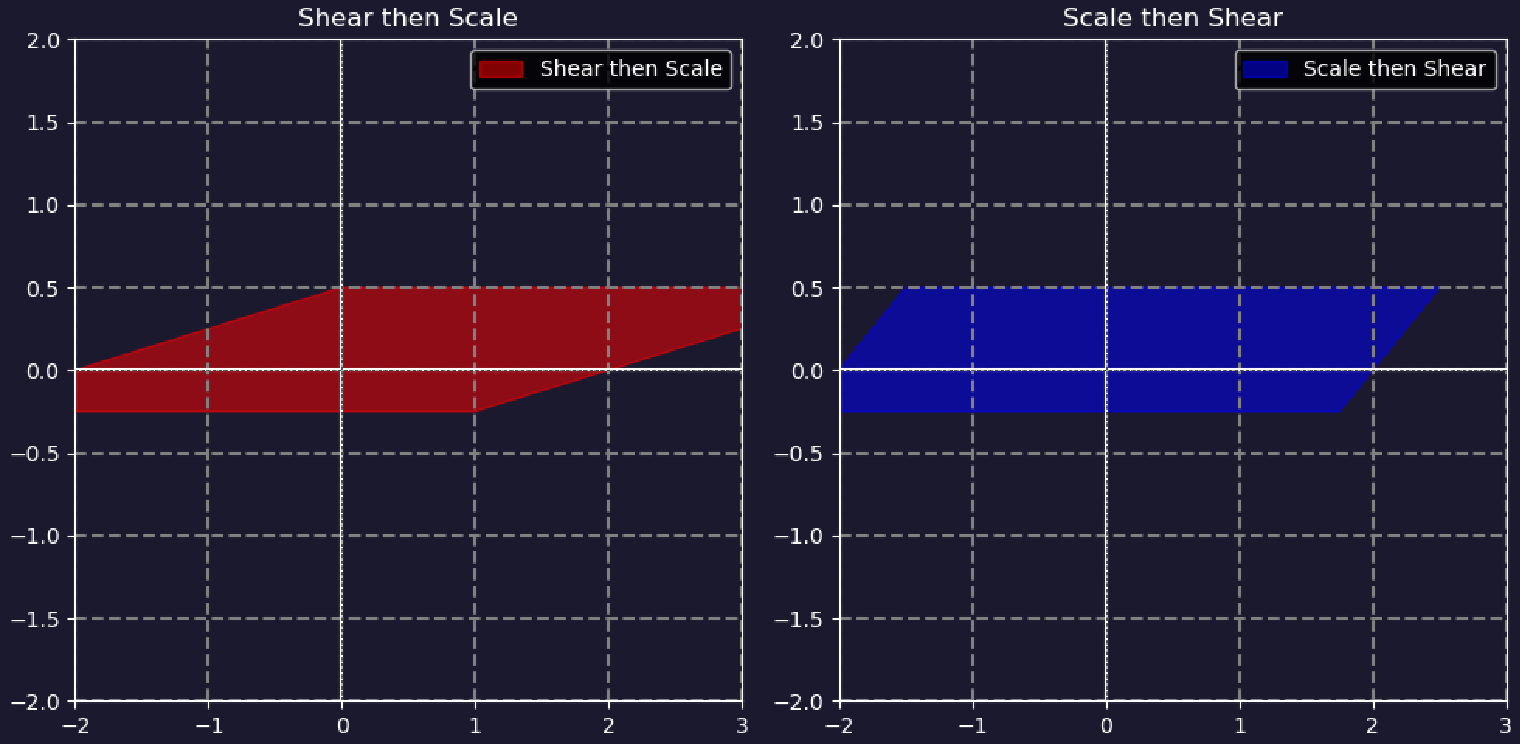
$$\underbrace{\begin{bmatrix} i_1 & j_1 \\ i_2 & j_2 \end{bmatrix}}_{\text{Second transformation}} \left(\underbrace{\begin{bmatrix} i_1 & j_1 \\ i_2 & j_2 \end{bmatrix}}_{\text{First transformation}} \begin{bmatrix} a \\ b \end{bmatrix} \right) \rightarrow \begin{bmatrix} i_1 & j_1 \\ i_2 & j_2 \end{bmatrix} \begin{bmatrix} a_t \\ b_t \end{bmatrix} \rightarrow \begin{bmatrix} i_1 a_t & j_1 b_t \\ i_2 a_t & j_2 b_t \end{bmatrix}$$

Composition of Linear Transformation

- What if we need to apply another transformation on the transformed vector ?
  - Like how you may rotate an image then flip or scale it.

$$\begin{bmatrix} i_1 & j_1 \\ i_2 & j_2 \end{bmatrix}\left(\begin{bmatrix} i_1 & j_1 \\ i_2 & j_2 \end{bmatrix}\begin{bmatrix} a \\ b \end{bmatrix}\right) \rightarrow \begin{bmatrix} i_1 & j_1 \\ i_2 & j_2 \end{bmatrix}\begin{bmatrix} a_t \\ b_t \end{bmatrix} \rightarrow \begin{bmatrix} i_1 a_t & j_1 b_t \\ i_2 a_t & j_2 b_t \end{bmatrix}$$

- Order of these transformations matters.
  - Shearing then scaling isn't like scaling then shearing.



Composition is matrix multiplication

- What if we need to apply another transformation on the transformed vector ?
  - Like how you may rotate an image then flip or scale it.

$$\begin{bmatrix} i_1 & j_1 \\ i_2 & j_2 \end{bmatrix} \left(\begin{bmatrix} i_1 & j_1 \\ i_2 & j_2 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \right) \rightarrow \begin{bmatrix} i_1 & j_1 \\ i_2 & j_2 \end{bmatrix} \begin{bmatrix} a_t \\ b_t \end{bmatrix} \rightarrow \begin{bmatrix} i_1 a_t & j_1 b_t \\ i_2 a_t & j_2 b_t \end{bmatrix}$$

$$\begin{bmatrix} i_1 & j_1 \\ i_2 & j_2 \end{bmatrix} \left(\begin{bmatrix} i_1 a & j_1 b \\ i_2 a & j_2 b \end{bmatrix} \right) \rightarrow \begin{bmatrix} i_1 a_t & j_1 b_t \\ i_2 a_t & j_2 b_t \end{bmatrix}$$

Composition of two matrix transformation

The product of two matrices

Matrix multiplications

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dg \end{bmatrix}$$

- Matrix multiplications condition COR (#cols = #Rows)
- Matrix multiplications ? ROC (Row×Cols), (Rows, Cols)
- For matrices A of size $m \times n$ and B of size $n \times p$, the element at position (i, j) in the resulting matrix C is given by:
- $$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Matrix multiplications

$O(n^3)$

```
# Define two 2x2 matrices
A = [[1, 2],
     [3, 4]]

B = [[5, 6],
     [7, 8]]

# Initialize result matrix
result = [[0, 0],
          [0, 0]]
```

```
# Matrix multiplication using nested for loops
for i in range(2):   # Loop over rows of A
    for j in range(2):   # Loop over columns of B
        for k in range(2):   # Loop over rows of B
            result[i][j] += A[i][k] * B[k][j]
```

```
print("Matrix multiplication using for loops:")
for row in result:
    print(row)
```

```
Matrix multiplication using for loops:
[19, 22]
[43, 50]
```

Matrix multiplications NumPy

$O(n^3)$

```
import numpy as np

# Define two 2x2 matrices
A = np.array([[1, 2],
              [3, 4]])

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

```
# Matrix multiplication using numpy
result = np.dot(A, B) # or use A @ B or np.matmul(A,B)
print("Matrix multiplication using numpy:")
print(result)
```

Why NumPy is more optimized ?

Eliminating Python Loops, as python loops introduce significant overhead because of **dynamic typing** and interpreter-related inefficiencies, NumPy's operations are implemented in compiled C.

Efficient Memory Management, NumPy arrays are stored in **contiguous blocks of memory** (like C arrays), which enables fast memory access. This contrasts with Python's native lists, which store objects in a scattered way.

Multithreading and Parallelism, NumPy can leverage multi-core processors by executing certain operations in parallel, optimized use of **CPU** cores.

Norms

- A norm is a **function** works on **vectors** and output **non-negative real numbers**.
  - It's like the distance from the origin.
  - Conditions of norms, given a function $f: X \to \mathbb{R}$ it should follow
    - **Triangle inequality** $f(x+y) \leq f(x) + f(y), \forall\, x, y \in X$
    - **Absolute homogeneity** $f(sx) = |s| f(x)$
    - **Positive definiteness** $\forall\, x \in X$, if $f(x) = 0, then\ x = 0$
      - Means zero only at the origin.

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

L_p norm equation.

IEEE FCI - Helwan Student Branch

Norms and distance

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

$$L_1 = \|X\|_1 = \sum_{i=1}^{n} |x_i|$$

$$L_2 = \|X\|_2 = \left(\sum_{i=1}^{n} |x_i|^2\right)^{\frac{1}{2}}$$

$$\left\|\begin{bmatrix}2\\3\\4\end{bmatrix}\right\|_1 = |2| + |3| + |4| = 9$$

$$\left\|\begin{bmatrix}2\\3\\4\end{bmatrix}\right\|_2 = \sqrt{|2|^2 + |3|^2 + |4|^2} = 5.38$$

Manhattan norm

Manhattan distance

Euclidean norm

Euclidean distance

Norms and distance

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

$$L_1 = \|X\|_1 = \sum_{i=1}^{n} |x_i|$$

Manhattan distance

$$L_2 = \|X\|_2 = \left(\sum_{i=1}^{n} |x_i|^2\right)^{\frac{1}{2}}$$

Euclidean distance

$$d(x, y) = \|X - Y\|_p$$

- The distance is just the **norm** of the **difference** between the vectors/points.

Dot product

- Dot product, an algebraic operation that takes **two vectors** and return a **single scaler** number.
  - Dot product also called scaler product, **projection product**.
  - It's the **sum of the product** of **corresponding entries** of the two vectors.

$$A.B = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} = \sum_{i=1}^{n} a_i b_i$$

Dot product

- What do we mean by **projection product?**

$$\cos(\theta) = \frac{||project(B)||}{||B||}$$

$$||project(B)|| = \cos(\theta)||B||$$

$$A \,.\, B = ||project(B)||\ ||A||$$

$$A \,.\, B = \cos(\theta)||B||||A||$$

Dot product and similarity

- **Cosine similarity,** is a measure of similarity between two non-zero vectors, it always belongs between [-1,1], it depend on the angle between the vectors.
  - **Cheap to compute** (low complexity), especially for sparse vectors.
  - Used to **compare vector representations of data**.
  - The smaller the angle, the more similar the vectors are in direction.
  - Cosine similarity **doesn't consider the magnitude (length)** of the vectors.

$$A \,.\, B = \cos(\theta)\,||B||\,||A||$$

$$\cos(\theta) = \frac{A \,.\, B}{||B||\,||A||}$$

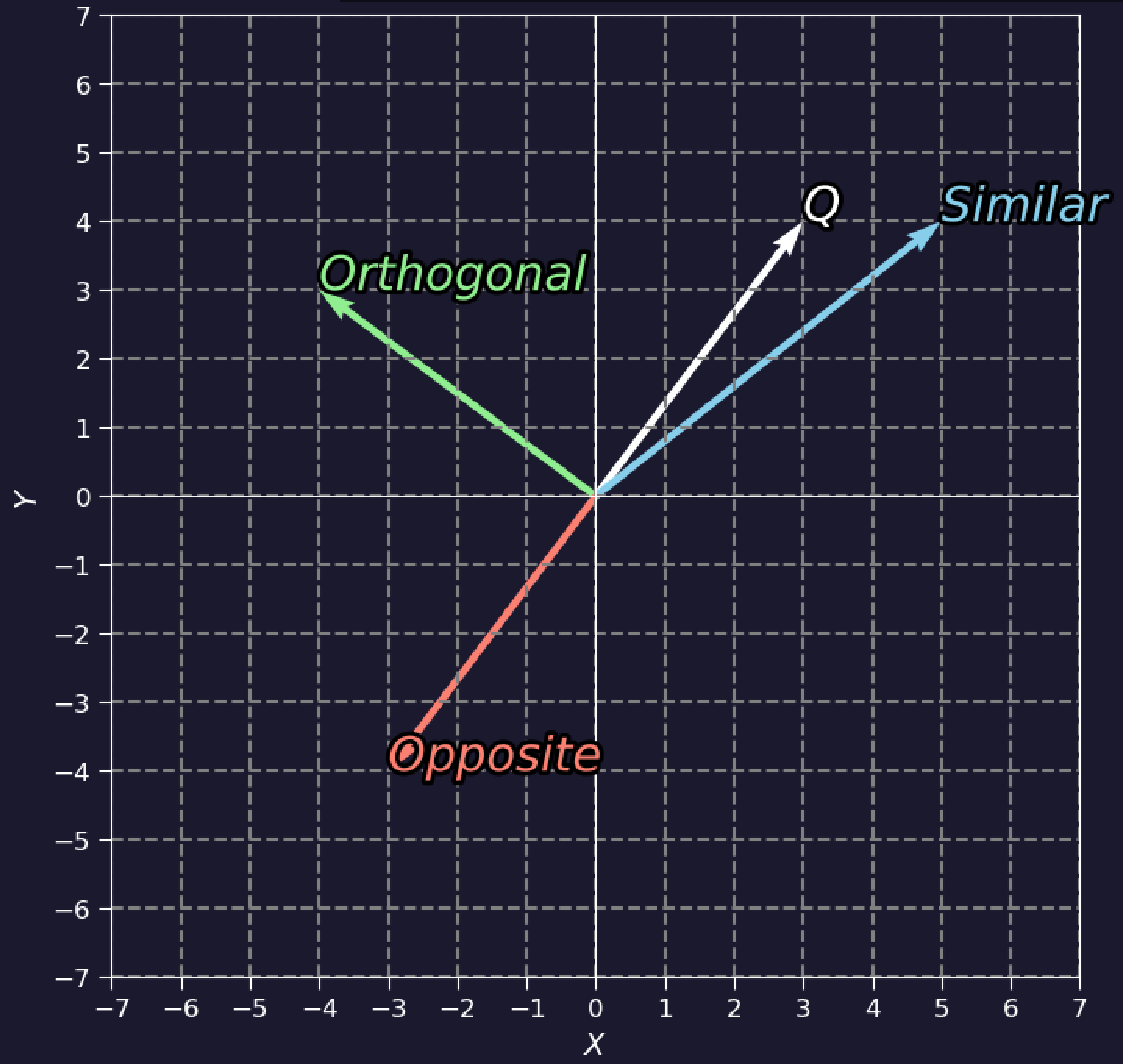
Dot product and similarity

$$cos(\theta) = \frac{A \,.\, B}{||B||\,||A||}$$

- $Q = (3,4), A = (5,4),$
- $B = (-4,3), C = (-3,-4)$

- $S_{cos}(Q, A) = 0.969$ **(similar)**
- $S_{cos}(Q, B) = 0$ **(Not similar)**
- $S_{cos}(Q, C) = -1$ **(Not similar)**

Can you do the math your self?



Dot product and similarity

$$\cos(\theta) = \frac{A \cdot B}{||B||\,||A||}$$

- How can we present data in a vector to compare it?

- There are many ways to do that:
  - **Machine Learning model** to present the information in the data to a vector, with a representation that maximize the similarity between data points from the same class and minimize the similarity between different classes representations.
  - **Mathematical model** that can reduce the data points to a vector.
  - **Reshaping the data**, like reshaping the grid of pixels that present an image to be a single vector of pixels.
  - **Encoding**
    - One-Hot Encoding : converts categorical data into binary vectors.
    - Bag of Words (BoW): represents text by counting the frequency of each word within a document.

Dot product and similarity

$$\cos(\theta) = \frac{A \cdot B}{||B||\,||A||}$$

- How can we present data in a vector to compare it?
- There are many ways to do that:
  - **Machine Learning model** to present the information in the data to a vector, with a representation that maximize the similarity between data points from the same class and minimize the similarity between different classes representations.
  - **Mathematical model** that can reduce the data points to a vector.
  - **Reshaping the data**, like reshaping the grid of pixels that present an image to be a single vector of pixels.
  - **Encoding**
    - One-Hot Encoding : converts categorical data into binary vectors.
    - Bag of Words (BoW): represents text by counting the frequency of each word within a document.

Dot product and similarity

$$\cos(\theta) = \frac{A \cdot B}{||B||\,||A||}$$

- Let’s represent words meaning using three dimensions
  - **Positivity** ➕ (x-axis): Measures how positive or negative a word is.
  - **Intensity** 💪 (y-axis): Captures the strength of the word.
  - **Formality** 💼 (z-axis): Indicates the level of formality or informality of the word
  - **From 1 to 10 how do you see these words in each dimension?**

  Joy, Excited, Grateful, Calm, Upset, Assistance, Help

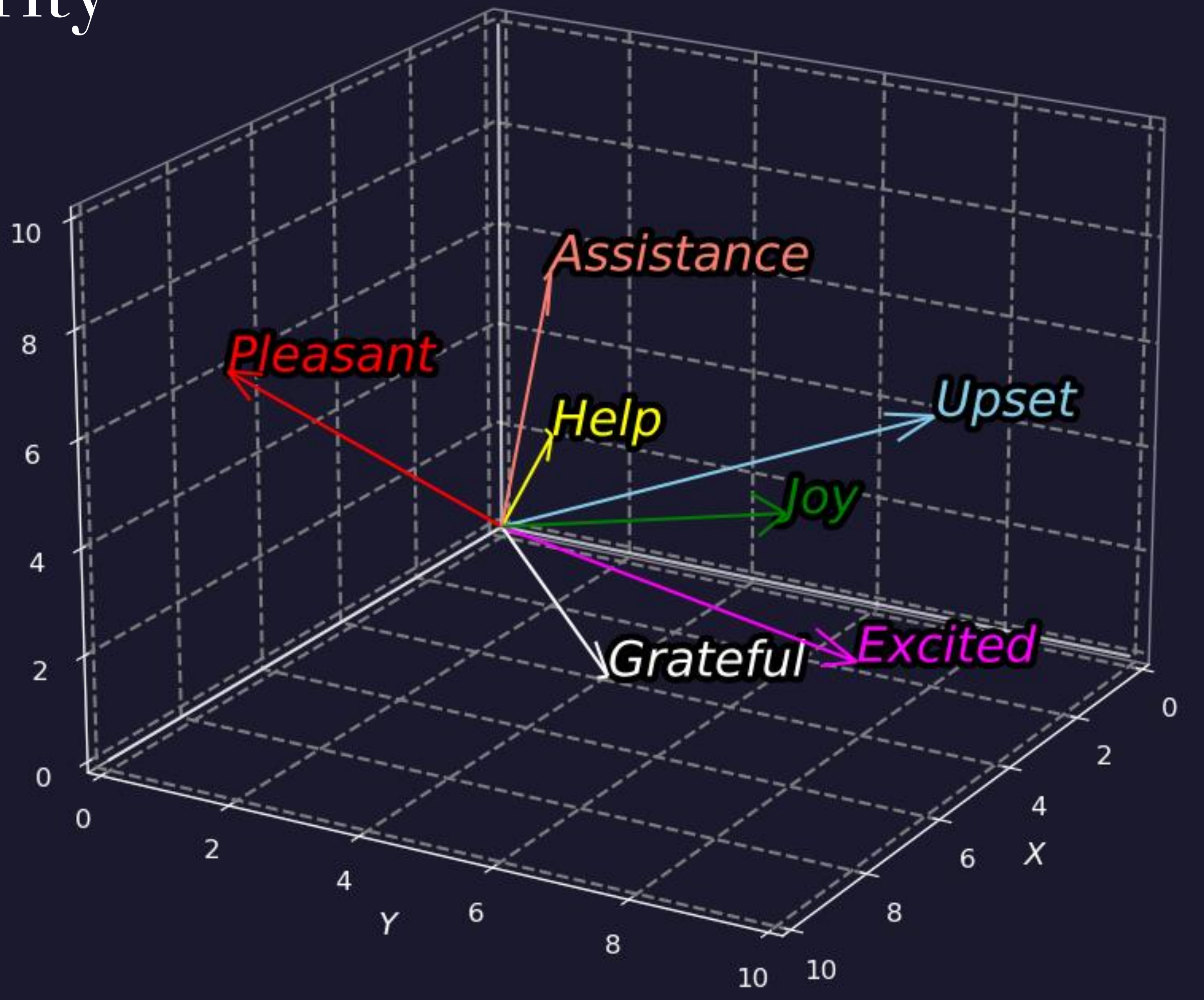
Dot product and similarity

$$cos(\theta) = \frac{A \cdot B}{||B||\,||A||}$$

Word	Positivity ➕	Intensity 💪	Formality 💼	Vector Representation
Joy	8	9	6	(8, 9,6)
Excited	7	9.5	3	(7,9.5,3)
Grateful	9	7	3	(9,7,3)
Pleasant	7	5	9	(7,5,9)
Upset	2	8	5	(2,8,5)
Assistance	7	5	9	(7,5,9)
Help	7	5	6	(7,5,6)

Dot product and similarity

Word	Vector Representation
Joy	(8, 9,6)
Excited	(7,9.5,3)
Grateful	(9,7,3)
Pleasant	(7,5,9)
Upset	(2,8,5)
Assistance	(7,5,9)
Help	(7,5,6)



Normalization of vectors

- **Normalization**, refers to the process of making something "standard" or, well, "normal."
- Take a **vector of any length** and, keeping it pointing in the same direction, **change its length to 1**, turning it into what is called a **unit vector**.

$$\hat{u} = \frac{\vec{u}}{\|\vec{u}\|}$$

- To normalize a vector, simply **divide each component by its magnitude (L2)**.

$$\hat{u} = \left(\frac{u_x}{\|\vec{u}\|}, \frac{u_y}{\|\vec{u}\|}, \frac{u_z}{\|\vec{u}\|}, \cdots, \frac{u_n}{\|\vec{u}\|}\right)$$

Normalization of vectors

$$cos(\theta) = \frac{A \cdot B}{||B|| \, ||A||}$$

- To normalize a vector, simply **divide each component by its magnitude (L2)**.

$$\hat{u} = (\frac{u_x}{||\vec{u}||}, \frac{u_y}{||\vec{u}||}, \frac{u_z}{||\vec{u}||}, \dots, \frac{u_n}{||\vec{u}||})$$

- Recall , how to calculate the L2 norm

$$||\vec{u}|| = \sqrt{u_x^2 + u_y^2 + u_z^2 + \cdots + u_n^2}$$

- What is the magnitude (L2 norm) of $\hat{u}$?

$$||\hat{u}|| = 1$$

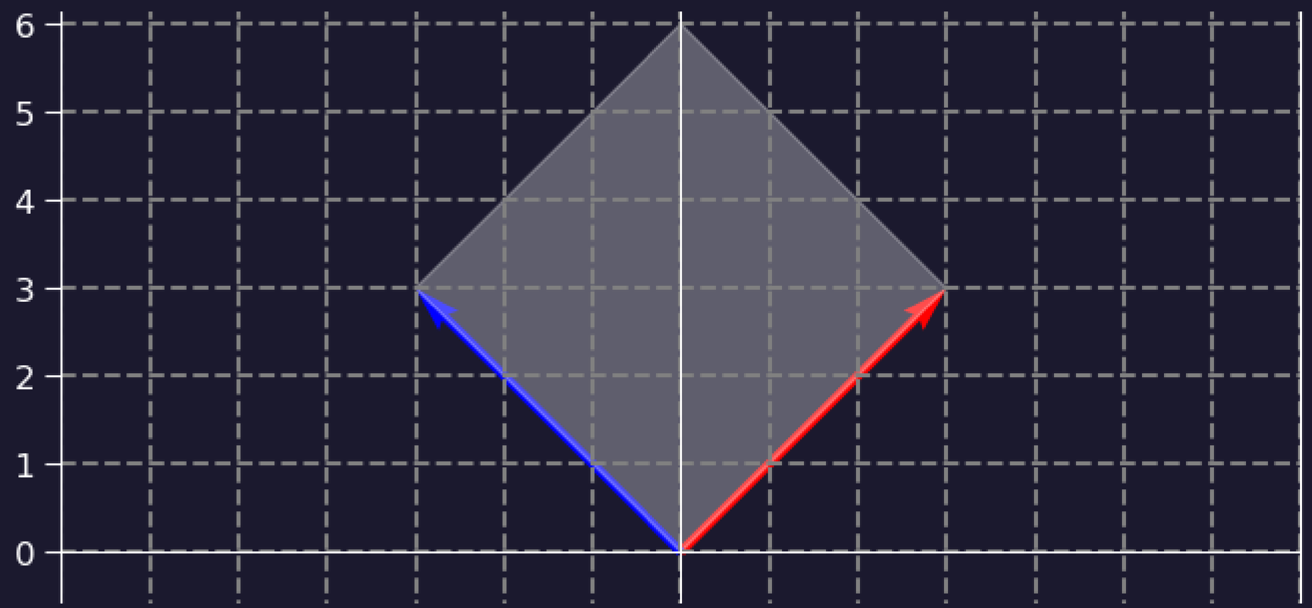
If we normalized A and B vectors, the **cosine similarity** would be the dot product $A \cdot B$

Determinants

- **Determinant** is a function that takes a squared matrix and return a scaler.

- **Determinant** it's how much the area is transformed.
  - the area between two vectors that form a parallelogram.

- It's the effect of the transformation over the area, or the grid.
  - And like the linear transformation
  - if you know **what happen to the unit vectors**, you can know what happen to any two transformed vectors.



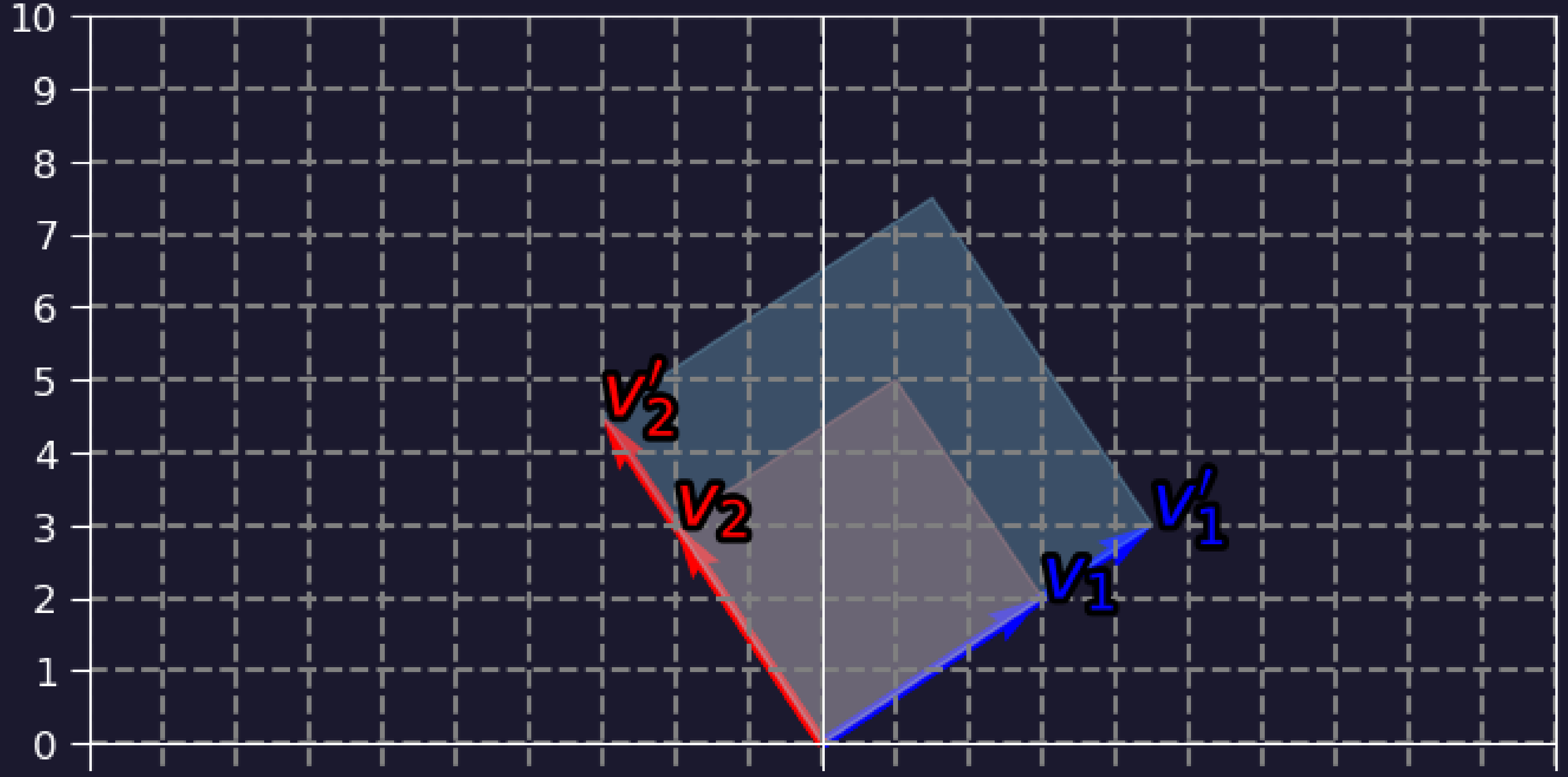
Determinants

$$v_1 = \begin{bmatrix} 3 \\ 2 \end{bmatrix}, v_2 = \begin{bmatrix} -2 \\ 3 \end{bmatrix}$$

- **Scaling transformation**

- **Transformation matrix**

$$\begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$



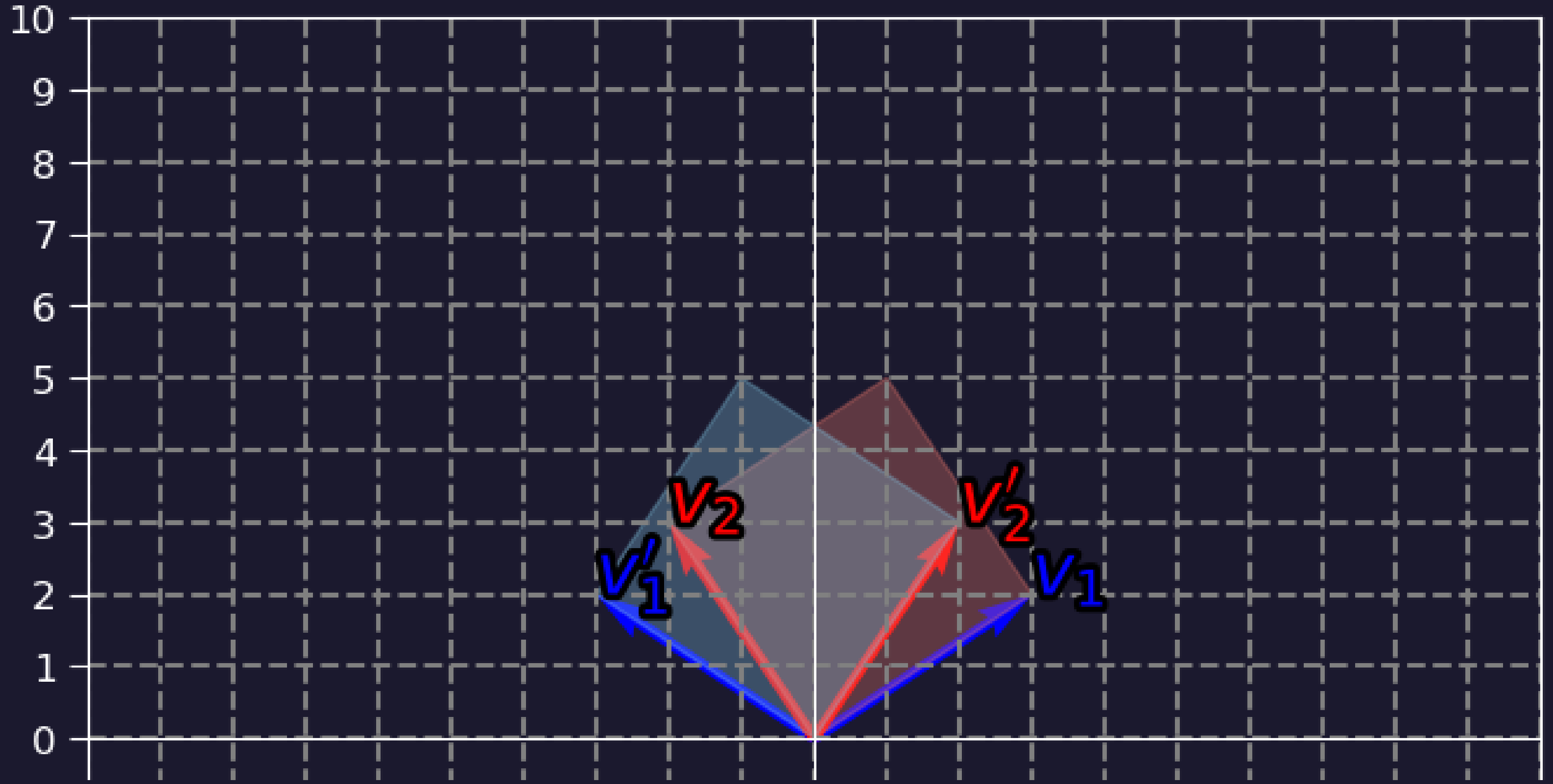
Determinants

$$v_1 = \begin{bmatrix} 3 \\ 2 \end{bmatrix}, v_2 = \begin{bmatrix} -2 \\ 3 \end{bmatrix}$$

- **Reflection transformation**

- **Transformation matrix**

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$



Determinants

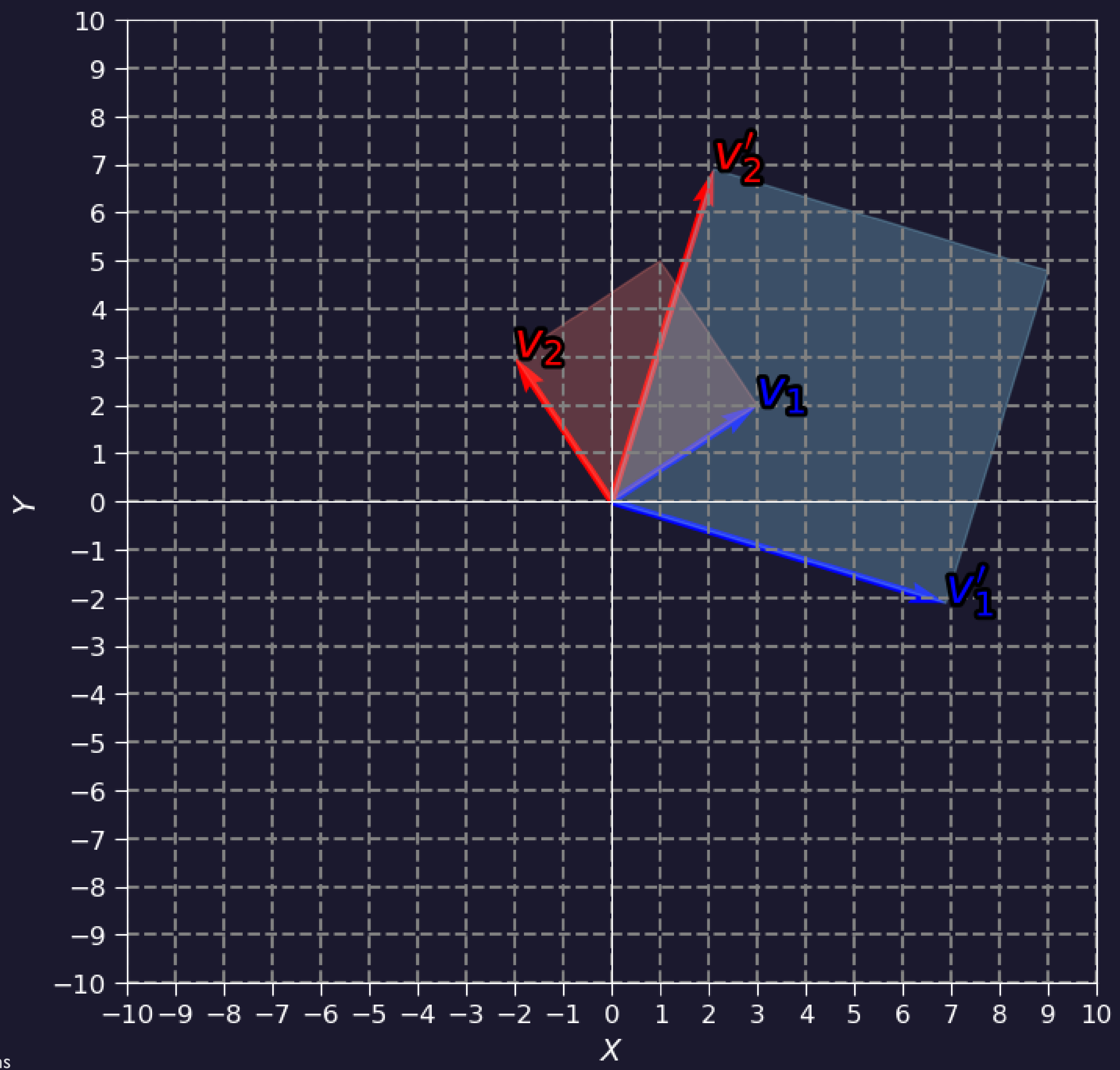
- **Rotation transformation**

- **Transformation matrix**

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

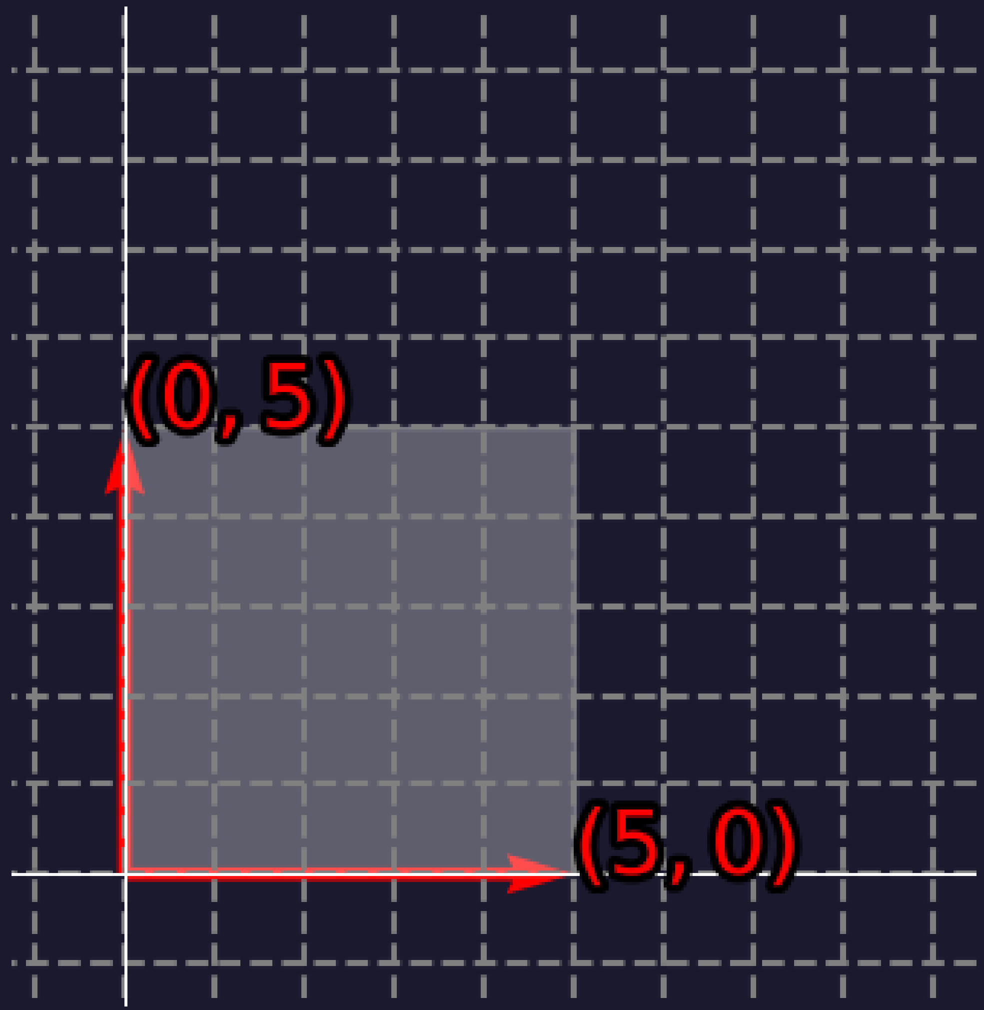
- **We can also scale it**

$$a * \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$



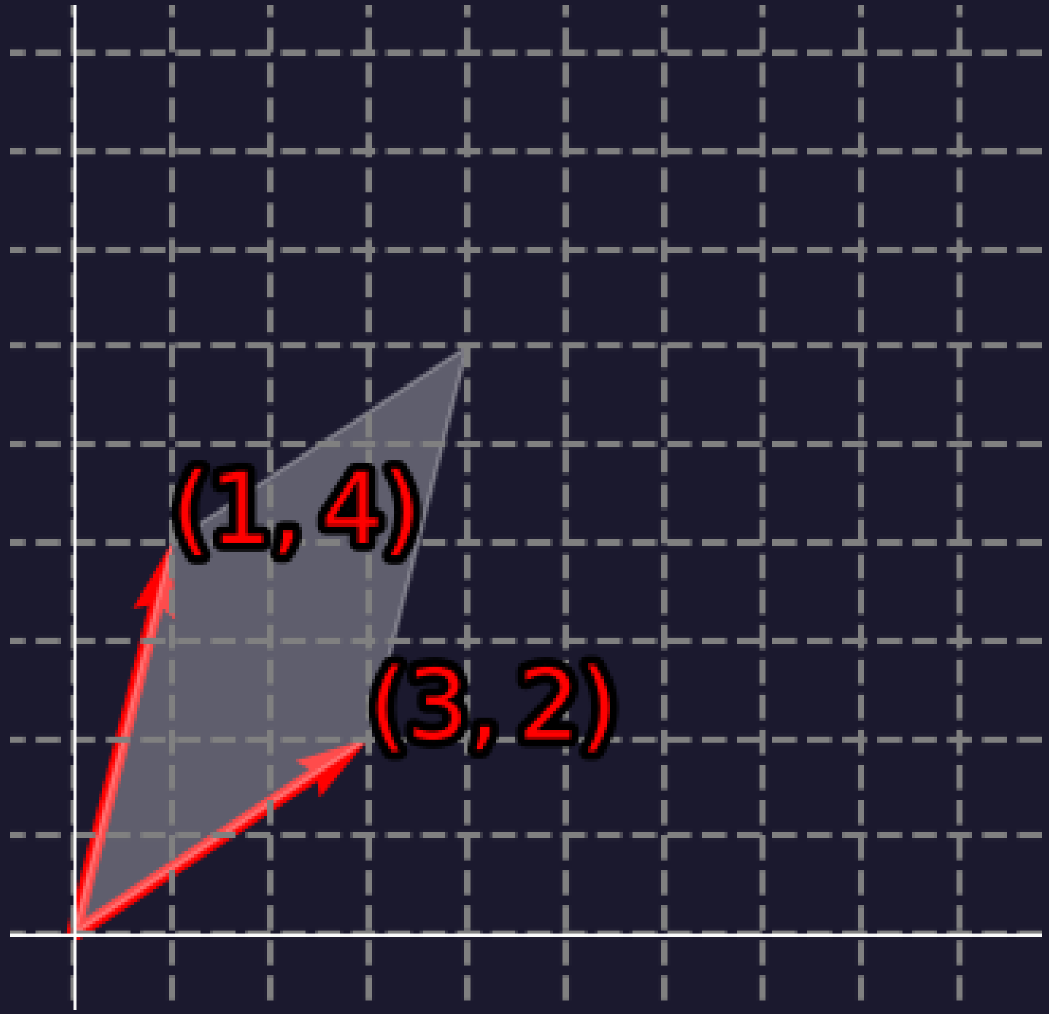
Determinants NumPy

```
matrix = np.array([[5, 0], [0, 5]])
determinant = np.linalg.det(matrix)
print("Determinant:", determinant)
# 24.999999999999996
```



Determinants NumPy

```
matrix = np.array([[3, 2], [1, 4]])
determinant = np.linalg.det(matrix)
print("Determinant:", determinant)
#10.000000000000002
```



Determinants

NumPy

3D

```python
import numpy as np

# Define a 3x3 matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Calculate the determinant
determinant = np.linalg.det(matrix)
print("Determinant:", determinant)
# -9.51619735392994e-16
```

Determinants

4D

```python
import numpy as np

# Define a 4x4 matrix
matrix_4d = np.array([
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12],
    [13, 14, 15, 16]
])
# Calculate the determinant
determinant_4d = np.linalg.det(matrix_4d)
print("Determinant:", determinant_4d)
#-1.820448242817726e-31
```

See 👀

- https://articulatedrobotics.xyz/tutorials/coordinate-transforms/rotation-matrices-2d/[2D Rotations with a simulation]
- https://youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab&si=1eGYH0ELkA4EVftr [playlist 📺]
- https://www.khanacademy.org/math/linear-algebra [Khan Academy]
- https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors
- https://en.wikipedia.org/wiki/Rotation_matrix
- https://shad.io/MatVis/
- https://visualize-it.github.io/linear_transformations/simulation.html
- https://nitsan.itch.io/linear-algebra-visualizer