Date

13] Vector operations (programming)

Scalar Multiplication and Sum of Vectors

1] Visualizing of vector $v \in \mathbb{R}^2$

$$v = [1 \quad 3]^T$$

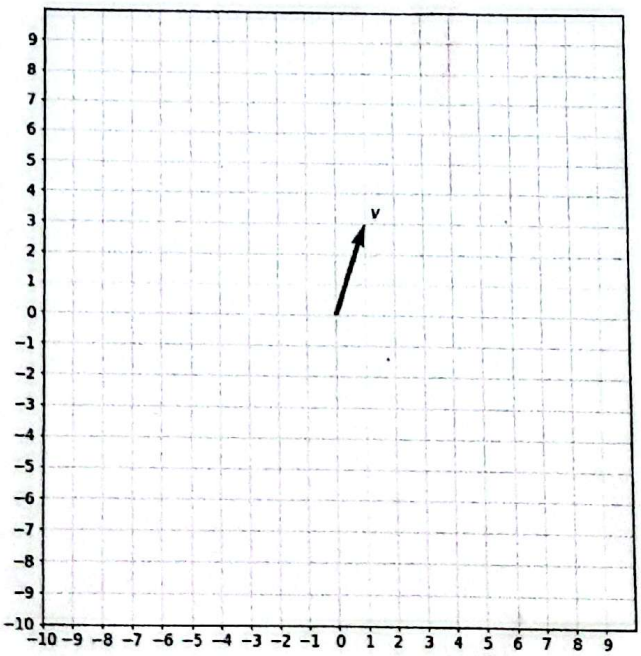
In [2]:
```
import matplotlib.pyplot as plt

def plot_vectors(list_v, list_label, list_color):
    _, ax = plt.subplots(figsize=(10, 10))
    ax.tick_params(axis='x', labelsize=14)
    ax.tick_params(axis='y', labelsize=14)
    ax.set_xticks(np.arange(-10, 10))
    ax.set_yticks(np.arange(-10, 10))


    plt.axis([-10, 10, -10, 10])
    for i, v in enumerate(list_v):
        sgn = 0.4 * np.array([[1] if i==0 else [1] for i in np.sign(v)])
        plt.quiver(v[0], v[1], color=list_color[i], angles='xy', scale_units='xy', scale=1)
        ax.text(v[0]-0.2+sgn[0], v[1]-0.2+sgn[1], list_label[i], fontsize=14, color=list_color[i])

    plt.grid()
    plt.gca().set_aspect("equal")
    plt.show()

v = np.array([[1],[3]])
# Arguments: list of vectors as NumPy arrays, labels, colors.
plot_vectors([v], [f"$v$"], ["black"])
```

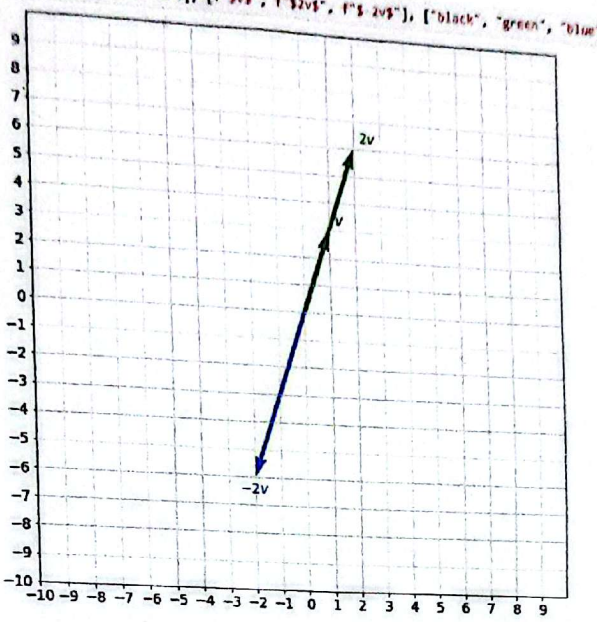


The vector is defined by its **norm (length, magnitude)** and **direction**, not its actual position. But for clarity and convenience vectors are often plotted starting in the origin (in $\mathbb{R}^2$ it is a point $(0, 0)$).

Date No

2] Scalar multiplication

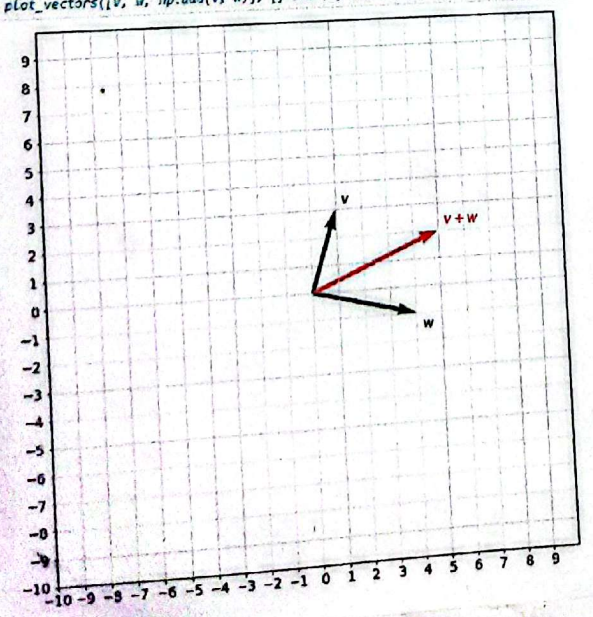
```
In [3]: plot_vectors([v, 2*v, -2*v], [f"$v$", f"$2v$", f"$-2v$"], ["black", "green", "blue"])
```



3] Sum of vectors

```
In [3]: v = np.array([[1],[3]])
        w = np.array([[4],[-1]])

        plot_vectors([v, w, v + w], [f"$v$", f"$w$", f"$v + w$"], ["black", "black", "red"])
        # plot_vectors([v, w, np.add(v, w)], [f"$v$", f"$w$", f"$v + w$"], ["black", "black", "red"])
```



Date

4] Norm of a vector

```
print ("Norm of a vector v is",
       np.linalg.norm(v))
```

Norm of a vector v is 3.1622776601683795

· Dot product

1] Algebric Definition of the Dot product

$$x = [x_1 \quad x_2 \quad \dots \quad x_n]^T \in \mathbb{R}^n$$
$$y = [y_1 \quad y_2 \quad \dots \quad y_n]^T \in \mathbb{R}^n$$

$$x \cdot y = \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \dots x_n y_n$$

$x = [1, -2, -5]$
$y = [4, 3, -1]$

Date No

2] Dot product using python

```
def dot (x,y):
    S = 0
    for xi, yi in zip (x,y):
        S += xi * yi
    returns

print ("The dot product of x and y is",
                    dot (x,y))
```

=> The dot product of x and y is 3

```
print ("np.dot(x,y) function returns dot product
        of x and y:", np.dot(x,y))
```

=> np.dot (x,y) function returns dot product of x and y : 3

```
# @ operator
print ("This line output is a dot product of
        x and y", np.array(x) @ np.array(y))
```

=> This line output is a dot product of x and y : 3

*] @ operator canbe only applied to arrays, not work with lists (x and y)

Date

3] Speed of calculations in vectorized Form

function dot() → loop version of the dot product
*) one by one

@ / np.dot() → vectorized form of the dot product
*) parallel

```
import time
a = np.random.rand(1000000)
b = np.random.rand(1000000)

tic = time.time()
c = dot(a, b)
toc = time.time()
print("Dot product:" c)
print("Time for the loop version:" +
                    str(1000*(toc-tic)) + "ms")
```

=) Dot product: 250028.18622
Time for the loop version: 364.674 ms

```
tic = time.time()
c = np.dot(a, b)
toc = time.time()
print("Dot product : ", c)
print("Time for the vectorized version,
      np.dot() function : " + str(1000*(toc-tic))
                                + "ms")
```

Dot product : 250028.18422
Time for the vectorized version, np.dot() function : 1.884222C3 ms

*) Similarly for @ method,

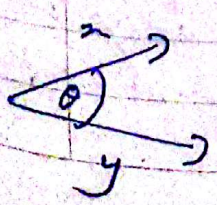
Time for the vectorized version, @ function : 1.8122 19619 ms

*) Vectorization is extremely beneficial in terms of speed of calculation

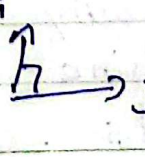
4) Geometric definition of the dot product

```
i = np.array([1, 0, 0])
j = np.array([0, 1, 0])
print("The dot product of i and j is", dot(i,j))
```

=) The dot product of i and j is 0



$x.y = |x||y| \cos\theta$



(orthogonal)

$x.y = 0$ //

Date No

5] Application of the dot product: Vector Similarity

1] Dot product measures similarity

*] In geometry, the dot product relates to the angle between two vectors

*] In NLP words are turned into vectors, and the dot product helps check how similar they are

2] Cosine Similarity formula

$$\cos\theta = \frac{x \cdot y}{\|x\| \|y\|}$$

1 → Vectors point in same direction
Words are very similar

0 → Vectors are orthogonal
no similarity

-1 → Vectors point in opposite directions
Completely dissimilar

*] That's why cosine similarity is widely used in Machine Learning NLP to compare words, documents or embeddings.

14] Matrix Multiplication (programming)

1] Definition

$[A]_{m \times n}$ $\quad C = AB$

$[B]_{n \times p}$ $\quad [C]_{m \times p}$

$$C_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

a_{ik} = Elements of $[A]$

b_{kj} = Elements of $[B]$

2] Matrix Multiplication using python

```
A = np.array([[4,9,9], [9,1,6], [9,2,3]])
print("Matrix A (3by3):\n", A)
B = np.array([[2,2], [5,7], [4,4]])
print("Matrix B (3by2):\n", B)
```

Matrix A (3 by 3):

```
[[4 9 9]
 [9 1 6]
 [9 2 3]]
```

Matrix B (3 by 2):

```
[[2 2]
 [5 7]
 [4 4]]
```

Date

np.matmul (A, B)

=) array ([[89, 107],
[47, 49],
[40, 44]])

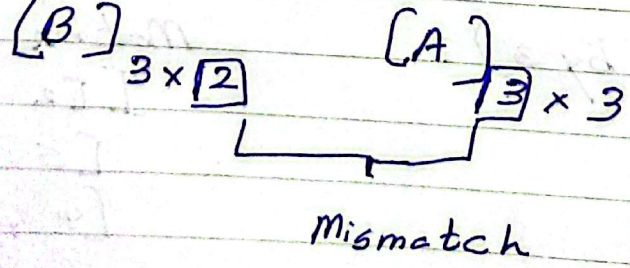
A @ B

=) array ([[89, 107],
[47, 49],
[40, 44]])

3) Matrix Convention and Broadcasting

try:
np.matmul(B, A) # B@A
except ValueError as err:
print (err)

matmul: Input operand 1 has a mismatch in its core Dimension 0, with gufunc signature (n?, k), (k, m?) -> (n? m?) (size 3 is different from 2)



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

print("Shape of vector x:", x.shape)
print("Number of dimension of vector x:", x.ndim)
print("Shape of vector x, reshaped to a matrix:", x.reshape((3,1)).shape)
print("Number of dimensions of vector x reshaped to a matrix:", x.reshape((3,1)).ndim)
```

Shape of vector x: (3,)
Number of dimensions of vector x: 1
Shape of vector x, reshaped to a matrix: (3,1)
Number of dimensions of vector x, reshaped to a matrix: 2

Now that we change x to (3,1) we can multiply

```
np.matmul(x,y)
```
3

In here automatically python will take the transpose of x Instead of $x \cdot y$ $(3\times1 \cdot 3\times1)$ $x^T y$ $(1\times3 \cdot 3\times1)$

Date

np.dot(A, B)

array ([[89, 107],
[47, 49],
[40, 44]])

np.dot() Also works, because here what happening is called "Broadcasting

A - 2

array ([[2, 7, 7],
[7, -1, 4],
[7, 0, 1]])

*) Here although 2 scalar is not defined it will broadcast to $\begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}$.

15] Linear Transformations

1] Transformations

$$T: R^2 \rightarrow R^3$$
$$v \in R^2 \qquad u \in R^3$$

$$T\left(\begin{bmatrix} v_1 \\ v_2 \end{bmatrix}\right) = \begin{bmatrix} 3v_1 \\ 0 \\ -2v_2 \end{bmatrix}$$

```
def T(v):
    w = np.zeros((3,1))
    w[0,0] = 3 * v[0,0]
    w[2,0] = -2 * v[1,0]

    return w

v = np.array([[3],[5]])
w = T(v)

print("Original vector:\n", v, "\n\n Result of transformation:\n", w)
```

Original vector:

```
[[3]
 [5]]
```

Result of the transformation

```
[[ 9.]
 [ 0.]
 [-10.]]
```

Date

2] Linear Transformations

*] A transformation T is said to be linear if the following two properties are true for any scalar k, and any input vectors u and v

$$T(ku) = T\left(\begin{bmatrix} ku_1 \\ ku_2 \end{bmatrix}\right) = \begin{bmatrix} 3ku_1 \\ 0 \\ -2ku_1 \end{bmatrix} = k\begin{bmatrix} 3u_1 \\ 0 \\ -2u_1 \end{bmatrix}$$

$$= kT(u)$$

$$T(u+v) = T\left[\begin{pmatrix} u_1 + v_1 \\ u_2 + v_2 \end{pmatrix}\right] = \begin{matrix} 3(u_1+v_1) \\ 0 \\ -2(u_2+v_2) \end{matrix} = \begin{bmatrix} 3u_1 \\ 0 \\ -2u_2 \end{bmatrix}$$

$$+ \begin{bmatrix} 3v_1 \\ 0 \\ -2v_2 \end{bmatrix} = T(u) + T(v)$$

```
u = np.array([[1], [-2]])
v = np.array([[2], [4]])
k = 7
print("T(k*v):\n", T(k*v), "\n k*T(v):\n",
      k*T(v), "\n\n")
```

print ("T(u+v): \n", T(u+v), "\n T(u) + T(v):", T(u) + T(v))

T(k*v):
[[42.]
[0.]
[-56.]]

k*T(v):
[[42.]
[0.]
[-56.]]

T(u+v):
[[9.]
[0.]
[-4.]]

T(u) + T(v):
[[9.]
[0.]
[-4.]]

3] Transformations Defined as a Matrix Multiplication

$L : R^m \rightarrow R^n$

$L(v) = A(v)$ $\quad A_{(n \times m)} \cdot V_{(m \times 1)}$

$w_{(n \times 1)}$

$$L\left(\begin{bmatrix} v_1 \\ v_2 \end{bmatrix}\right) = \begin{bmatrix} 3v_1 \\ 0 \\ -2v_2 \end{bmatrix} = \begin{bmatrix} ? & ? \\ ? & ? \\ ? & ? \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

$$L\left(\begin{bmatrix} v_1 \\ v_2 \end{bmatrix}\right) = A\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} a_{1,1} + a_{1,2} v_2 \\ a_{2,1} v_1 + a_{2,2} v_2 \\ a_{3,1} v_1 + a_{3,2} v_2 \end{bmatrix} = \begin{bmatrix} 3v_1 \\ 0 \\ -2v_2 \end{bmatrix}$$

```
def L(v):
    A = np.array([[3, 0], [0,0], [0,-2]])
    print("Transformation matrix:\n", A, "\n")
    w = A @ v
    return w

v = np.array([[3], [5]])
w = L(v)

print("original vector:\n", v, "\n\n Result
      of the tranformation:\n", w)
```

```
[[ 3  0]
 [ 0  0]
 [ 0 -2]]

original vector:
[[3]
 [5]]

Result of the transformation:
[[ 9]
 [ 0]
 [-10]]
```

4] Standard Transformations in a Plane

$e_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $e_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, $L(e_1) = Ae_1$ $L(e_2) = Ae_2$

$\therefore$ $A[e_1\ e_2] = [Ae_1\ Ae_2] = [L(e_1)\ L(e_2)]$

$[e_1\ e_2] = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \longrightarrow A[e_1\ e_2] = AI = A$

$A = [L(e_1)\ L(e_2)]$

4.1 Horizontional Scaling (Dilation)

```
def T_hscaling(v):
    A = np.array([[2,0],[0,1]])
    w = A@v
    return w

def transform_vectors(T, v1, v2):
    V = np.hstack((v1,v2))
    W = T(V)
    return W

e1 = np.array([[1],[0]])
e2 = np.array([[0],[1]])

transformation_result_hscaling = transform_vectors
                                 (T_hscaling, e1, e2)

print("Original vectors:\n e1 =\n", e1,
      "\n e2 =\n", e2,
      "\n\n Result of the transformation
       (matrix form):\n", transformation_result_
       hscaling)
```

Original vectors:

e1 = [[1]
      [0]]

e2 = [[0]
      [1]]

Result of the transformation (matrix form):

[[2 0]
 [0 1]]

Date

~~def T_reflection_yaxis(v):~~

4.2 Reflection about y axis (vertical axis)

```
def T_reflection_yaxis():
    A = np.array([[-1,0],[0,1]])
    w = A @v

    return w

e1 = np.array([[1],[0]])
e2 = np.array([[0],[1]])

R = transform_vectors(T_reflection_axis, e1,e

print("original vectors:\n e1=\n", e1,
      "\n e2=\n", e2,
      "\n\n Result of transformation (matrix for
      :\n", R)
```

Original vectors:

$e_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

$e_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

Result of transformation (matrix form)

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

*) In Linear Transformation most of the cases ($AB \neq BA$) Therefore we cannot change the order of the matrix multiplication