Lecture 4

Linear Algebra and Gradient Descent

Dr. David Zmiaikou

Linear Algebra

Next Exercise

$$\begin{cases} 4x - 2y = 20 \\ -5x - 5y = -10 \end{cases}$$

Solution

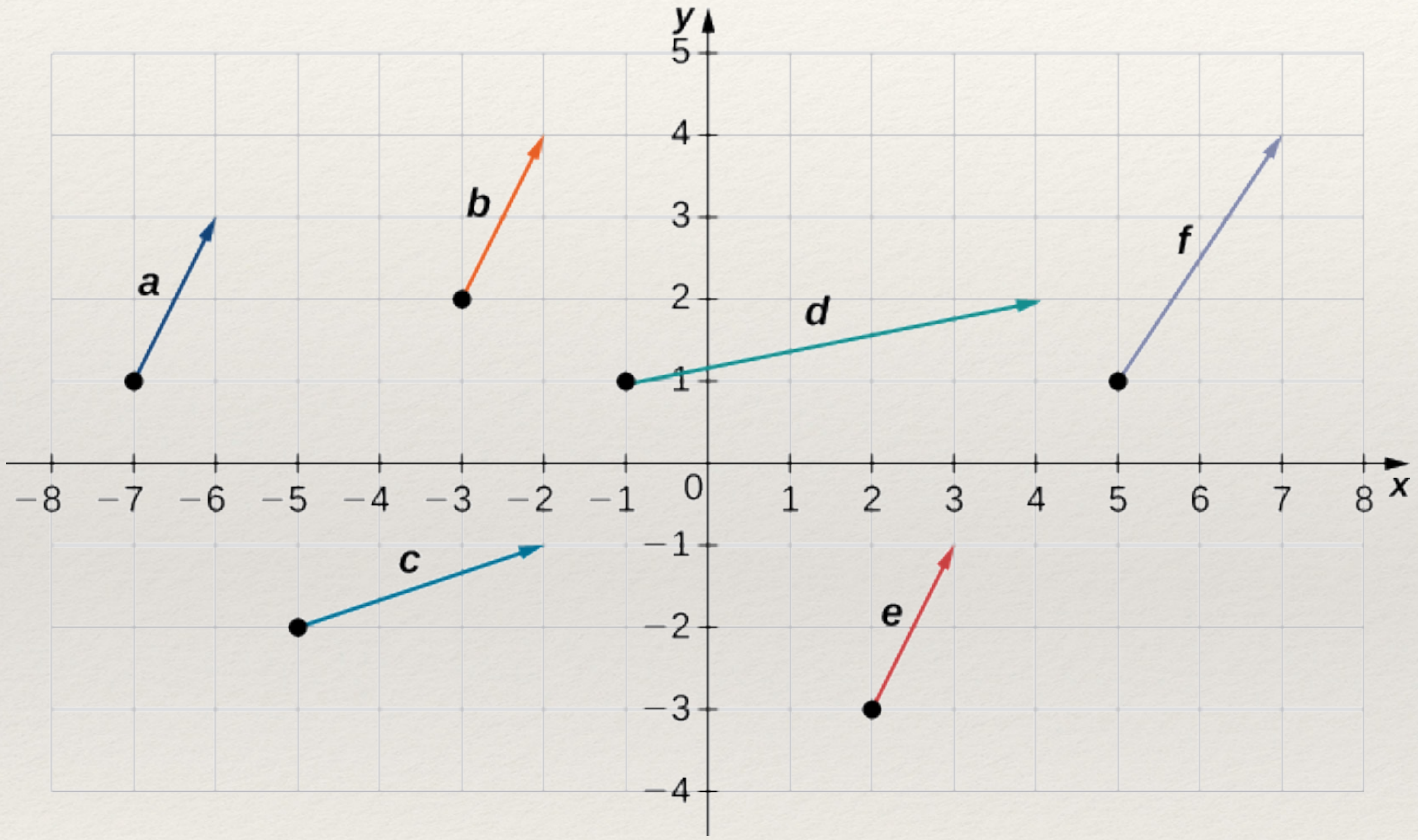
$$\therefore \quad \begin{cases} x = 4 \\ y = -2 \end{cases}$$

Linear Algebra is a domain of mathematics whose primary goal is to be able to solve systems of linear equations.

Main notions of linear algebra include:

- vector
- vector space
- matrix

Vectors

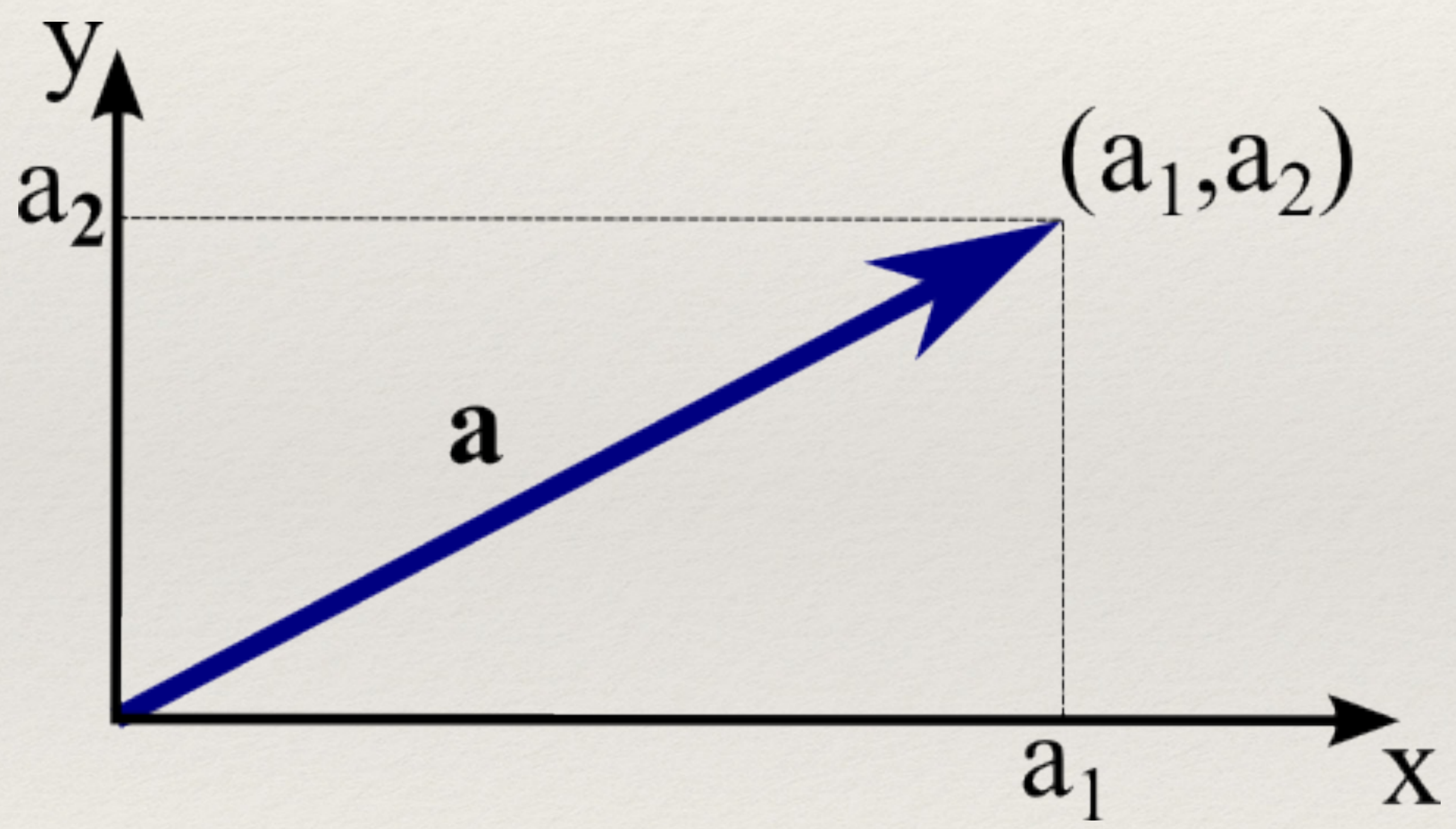


Vectors are elements of a **vector space**.
A **vector space** is a set of objects that can be added together and that can be multiplied by scalars.

What you need to know:

- two vectors can be **added** to form a new vector
- two vectors can be **subtracted** to form a new vector
- a vector can be **multiplied** by a number to form a new vector
- one can assign **coordinates** to every vector

Vector: coordinates

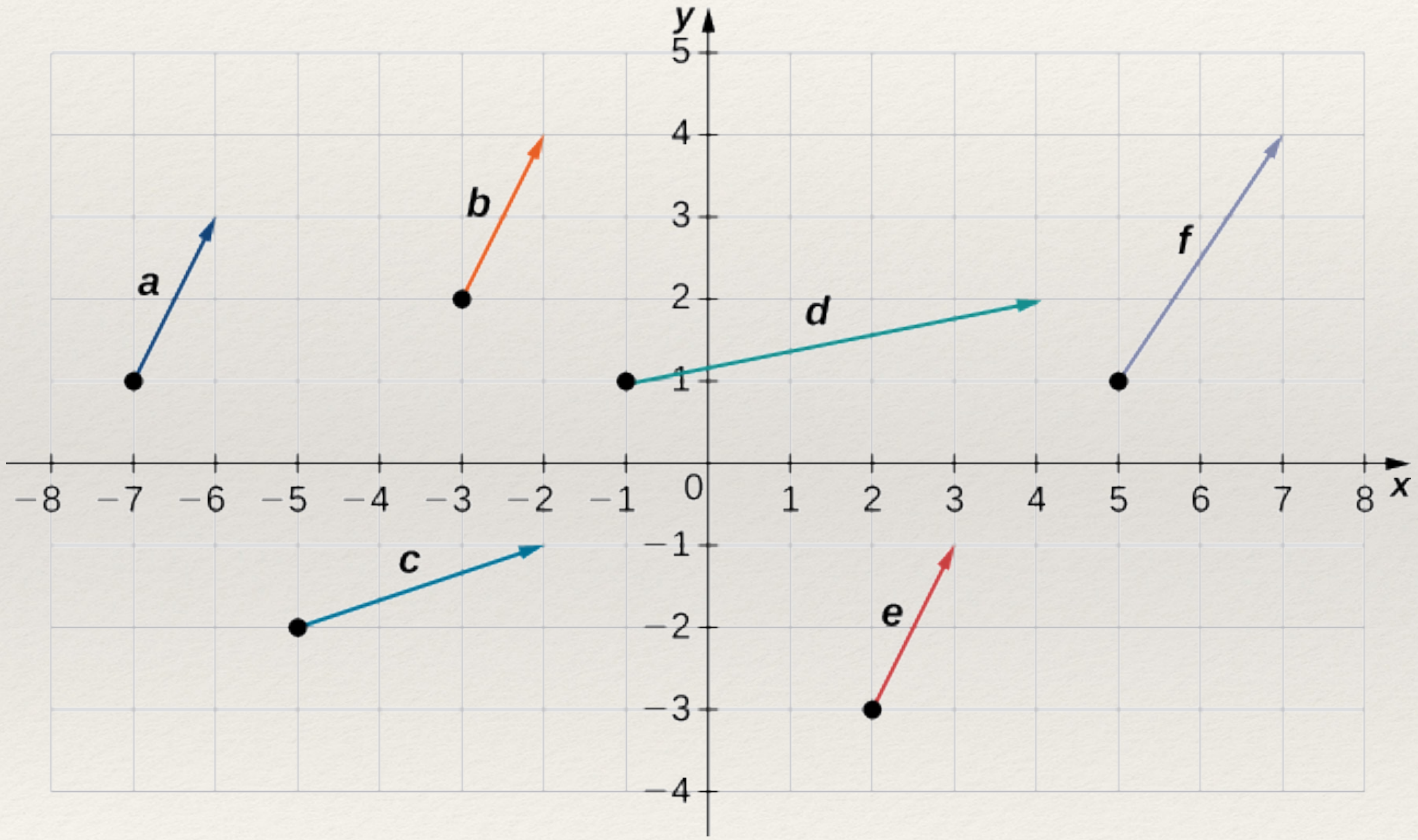


Vectors are elements of a **vector space**.

What you need to know:

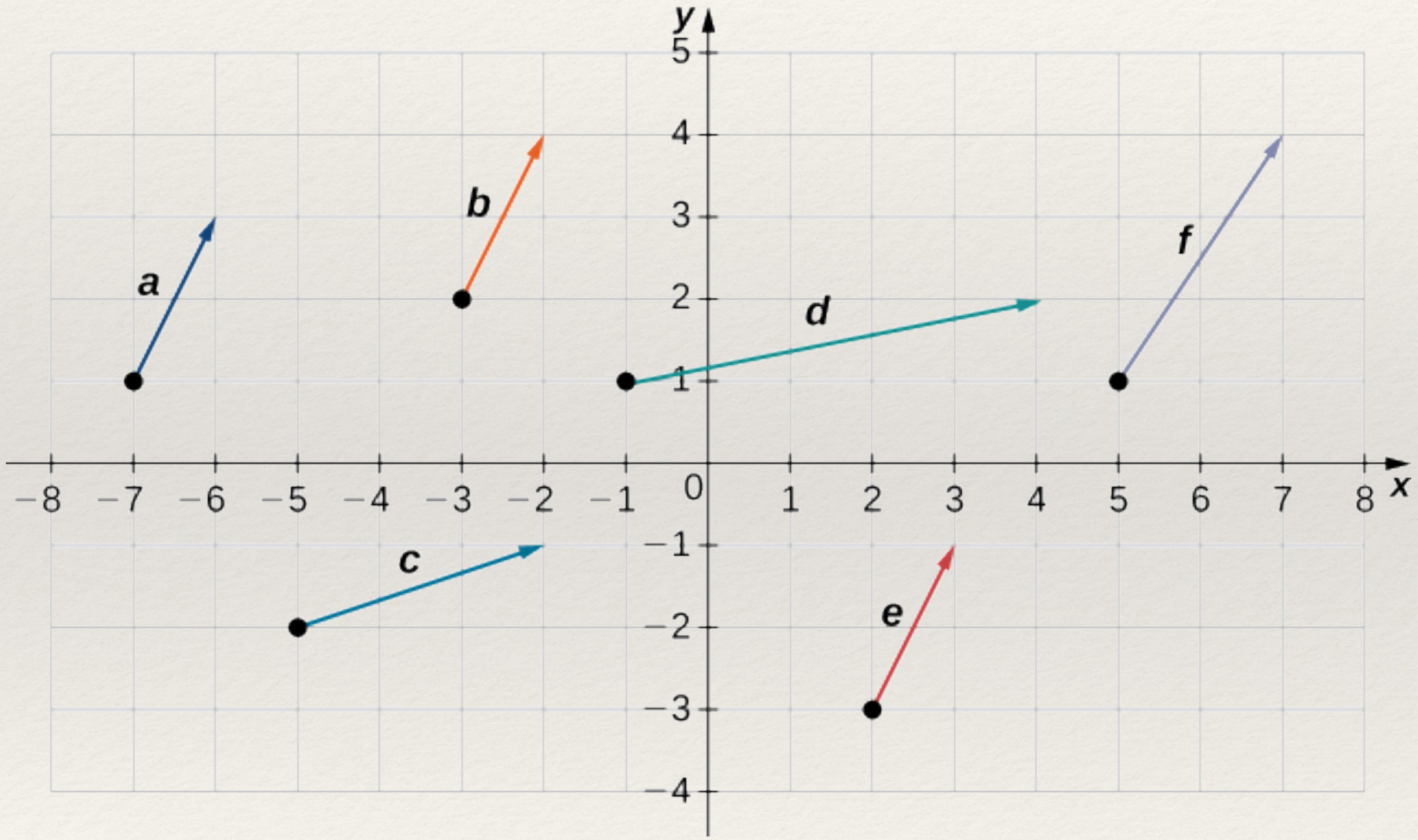
- one can assign **coordinates** to every vector
- if the vector space is n-**dimensional**, then every vector has n coordinates

Example 1



Question. What are the coordinates of the vectors **a**, **b**, **c**, **d**, **e**, **f**?

Example 1



Question. What are the coordinates of the vectors **a**, **b**, **c**, **d**, **e**, **f**?

Answer.

a = [1, 2] = [-6, 3] — [-7, 1] = [-6+7, 3-1]

b = [1, 2] = **a**

c = [3, 1]

d = [5, 1]

e = [2, 1]

f = [2, 3]

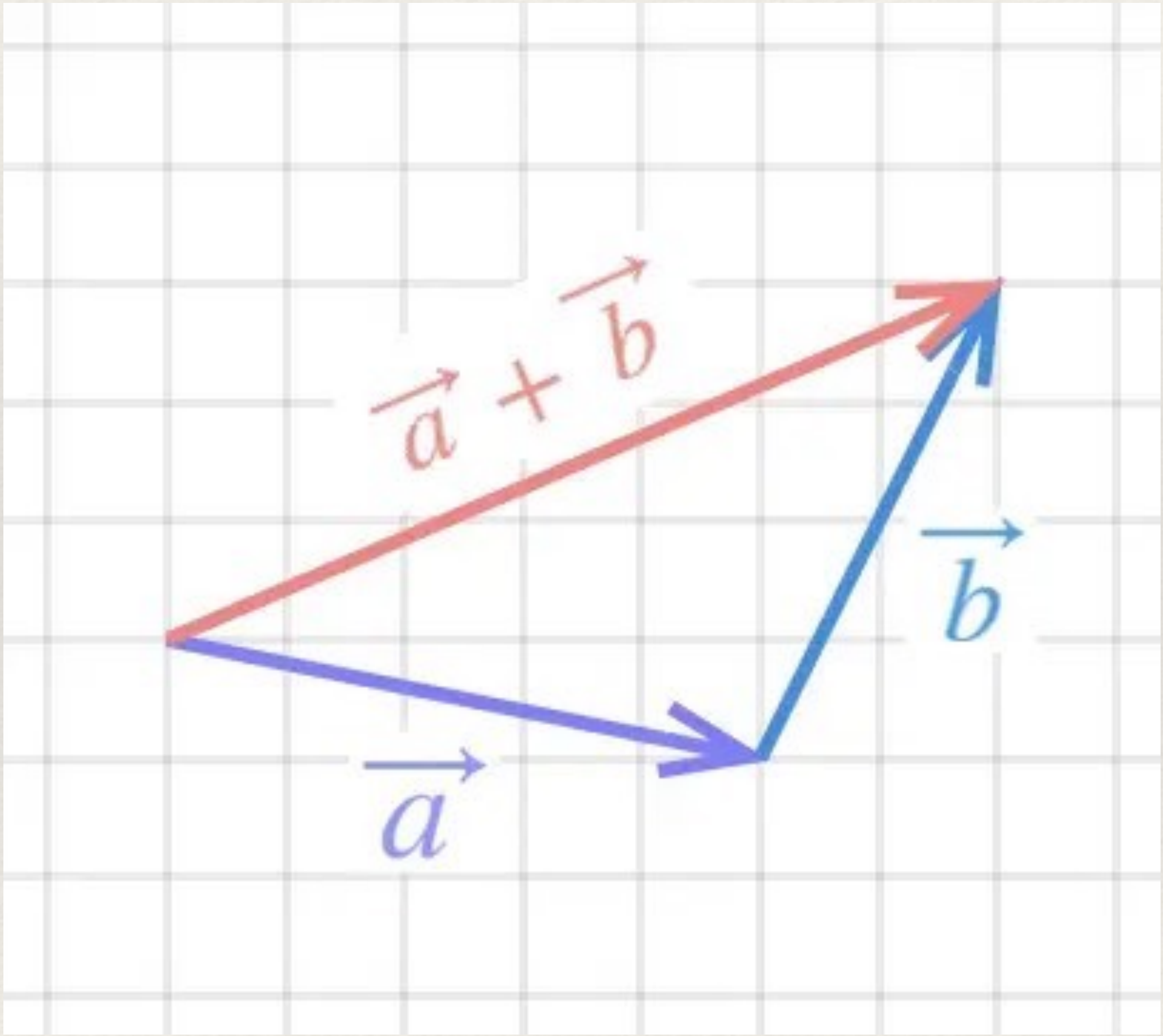
Example 2

```
from typing import List
Vector = List[float]

height_weight_age = [70,  # inches,
                     170, # pounds,
                     40 ]  # years
grades = [95, # exam1
          80, # exam2
          75, # exam3
          62 ] # exam4
```

If you have the heights, weights, and ages of a large number of people, you can treat your data as 3-dimensional vectors
[height, weight, age].

If you're teaching a class with four exams, you can treat student grades as 4-dimensional vectors
[exam1, exam2, exam3, exam4].

Vectors: addition

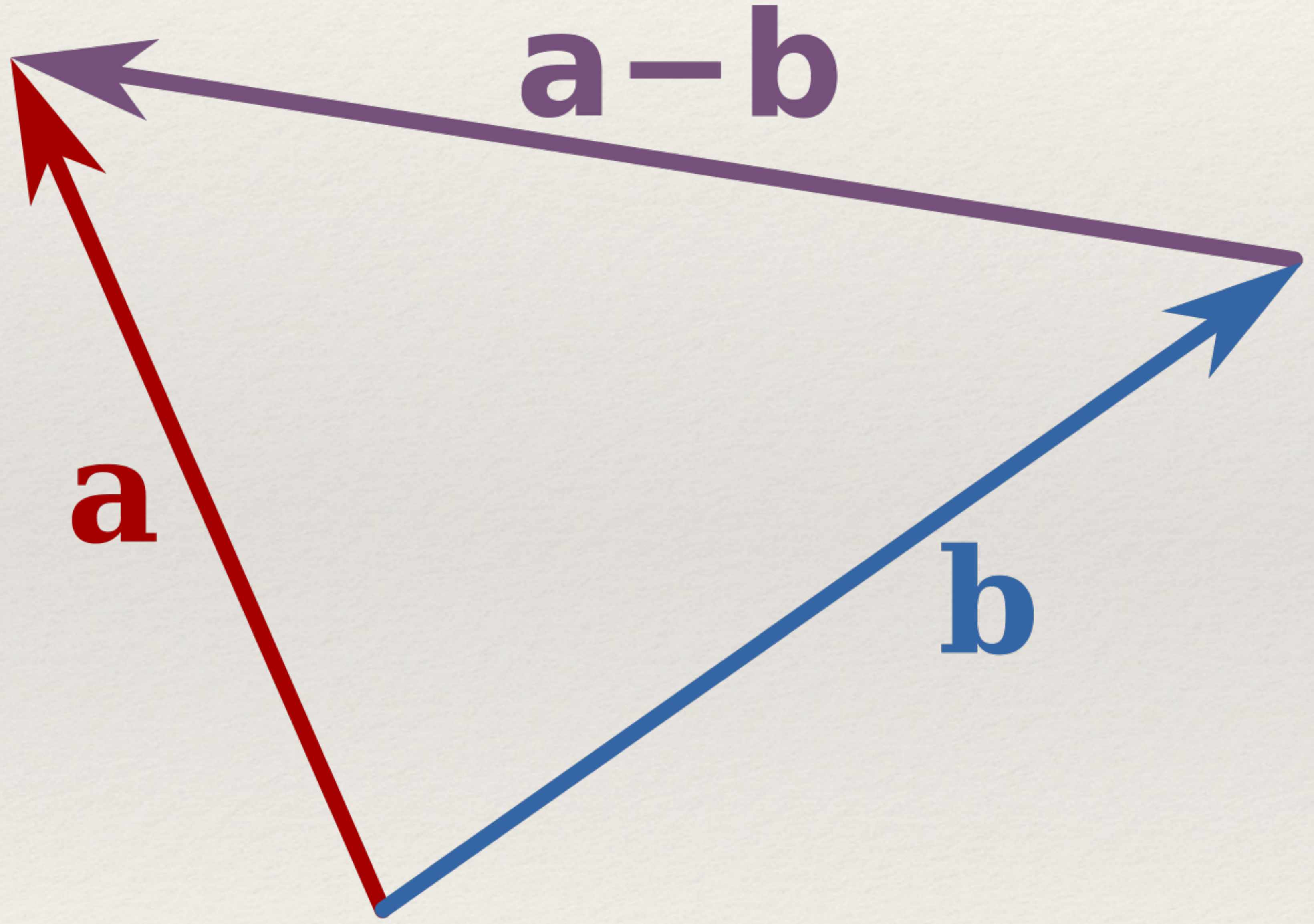


Vectors are elements of a **vector space**.

We will frequently need to add two vectors. Vectors add *component-wise*.

To add two vectors, they **must have the same number of coordinates**.

Vectors: subtraction



Vectors also subtract *component-wise*.

To subtract two vectors, they **must have the same number of coordinates**.

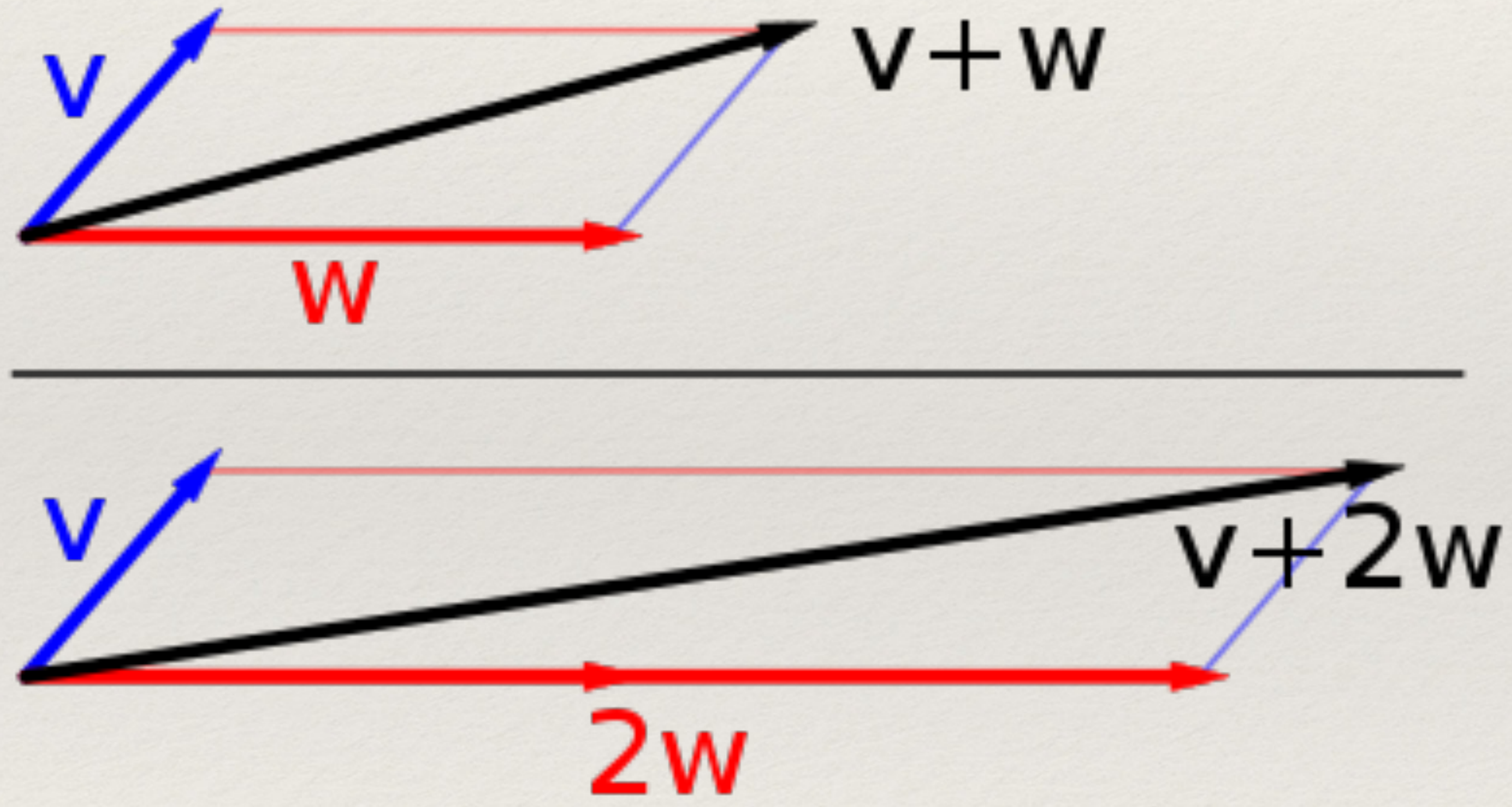
Example 3

```
def add(v: Vector, w: Vector) -> Vector:
    """Adds corresponding elements"""
    assert len(v) == len(w), "vectors must be the same length”
    return [v_i + w_i for v_i, w_i in zip(v, w)]

add([1, 2, 3], [4, 5, 6]) # = [5, 7, 9]


def subtract(v: Vector, w: Vector) -> Vector:
    """Subtracts corresponding elements"""
    assert len(v) == len(w), "vectors must be the same length"
    return [v_i - w_i for v_i, w_i in zip(v, w)]

subtract([5, 7, 9], [4, 5, 6]) # [1, 2, 3]
```

Example 4

```
def vector_sum(vectors: List[Vector]) -> Vector:
    """Sums all corresponding elements"""
    # Check that vectors is not empty
    assert vectors, "no vectors provided!"

    # Check the vectors are all the same size
    num_elements = len(vectors[0])
    assert all(len(v) == num_elements for v in vectors), "different sizes!"

    # the i-th element of the result is the sum of every vector[i]
    return [sum(vector[i] for vector in vectors)
            for i in range(num_elements)]

vector_sum([[1, 2], [3, 4], [5, 6], [7, 8]]) # [16, 20]
```

Vector: multiplication by a scalar



We will also need to be able to **multiply a vector by a scalar**, which is simply done by multiplying each coordinate of the vector by that number.

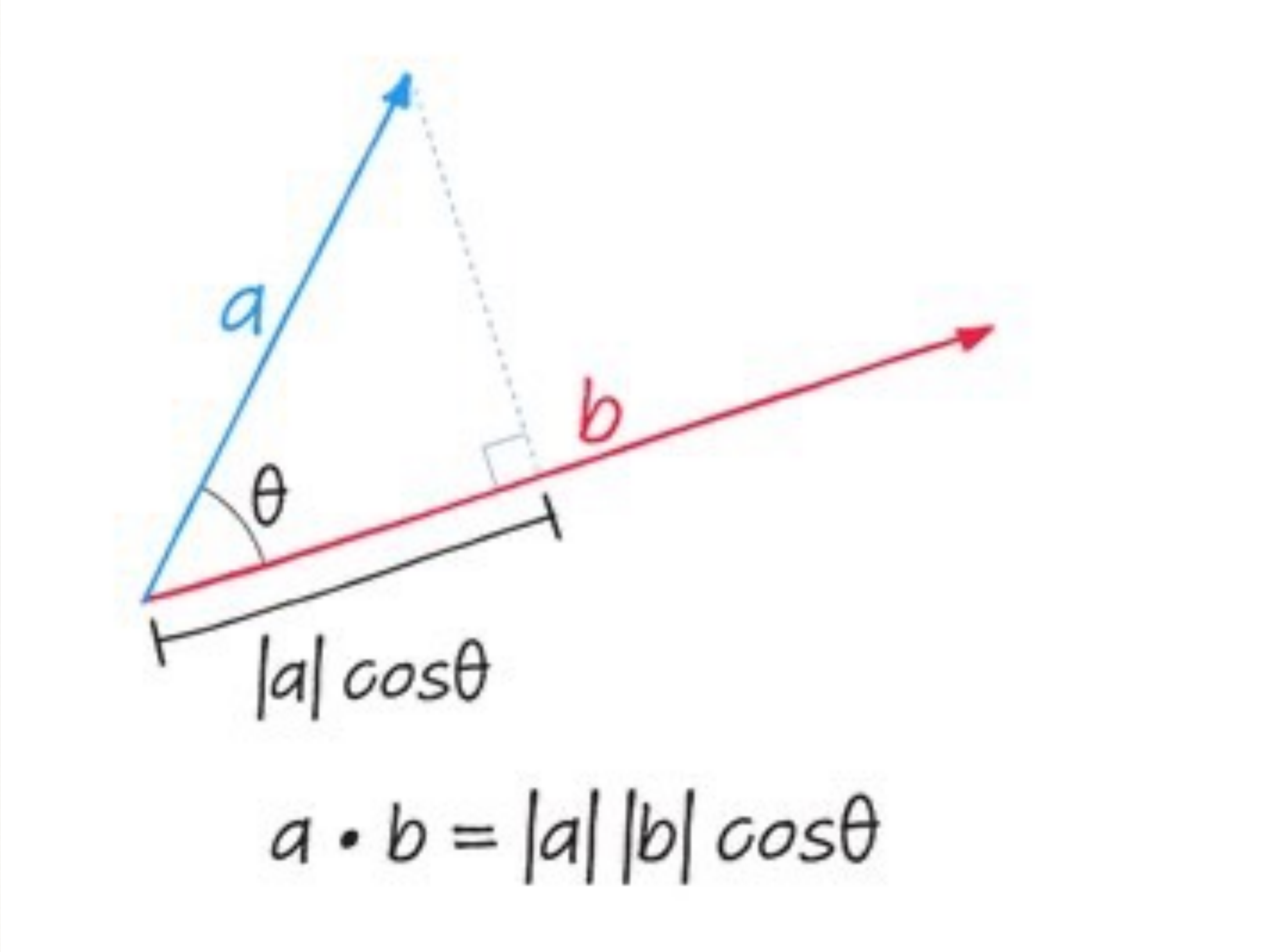
Example 5

```
def scalar_multiply(c: float, v: Vector) -> Vector:
   """Multiplies every element by c"""
   return [c * v_i for v_i in v]

scalar_multiply(5, [1, 2, 3]) # [5, 10, 15]
```

Example 6

```
def vector_mean(vectors: List[Vector]) -> Vector:
    """Computes the element-wise average"""
    n = len(vectors)
    return scalar_multiply(1/n, vector_sum(vectors))

vector_mean([[1, 2], [3, 5], [4, 7]])  # [8/3, 14/3]
```

Vectors: dot product



The **dot product** of two vectors $\mathbf{a} = [x_1, y_1]$ and $\mathbf{b} = [x_2, y_2]$ is the sum of their component-wise products:

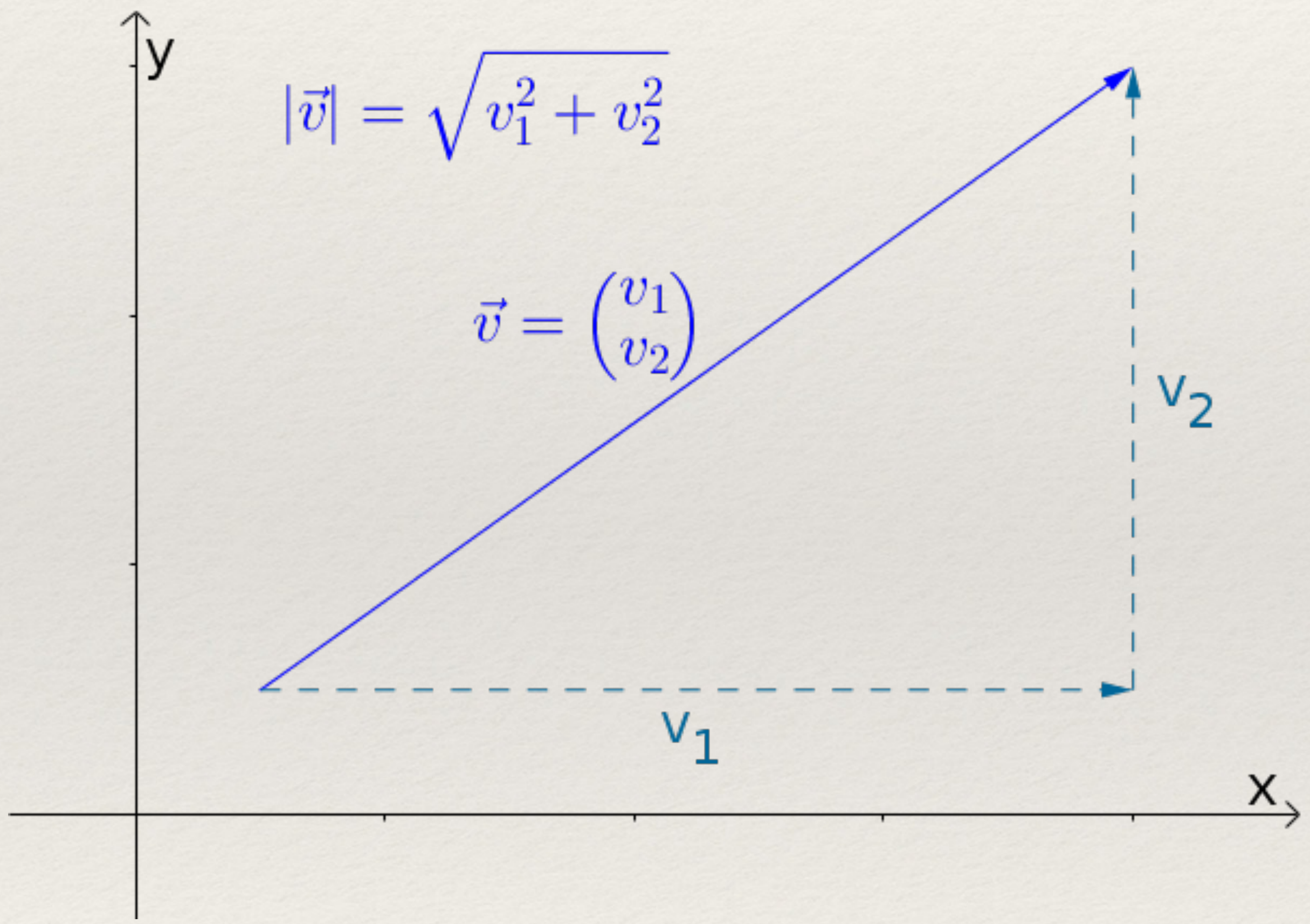
$$\mathbf{a} \cdot \mathbf{b} = x_1 x_2 + y_1 y_2$$

Another way of defining this is that it is the length of the vector you would get if you projected **a** onto **b**.

Example 7

```
def dot(v: Vector, w: Vector) -> float:
    """Computes v_1 * w_1 + ... + v_n * w_n"""
    assert len(v) == len(w), "vectors must be same length"
    return sum(v_i * w_i for v_i, w_i in zip(v, w))

dot([1, 2, 3], [4, 5, 6])  # 32
```

Vector: length

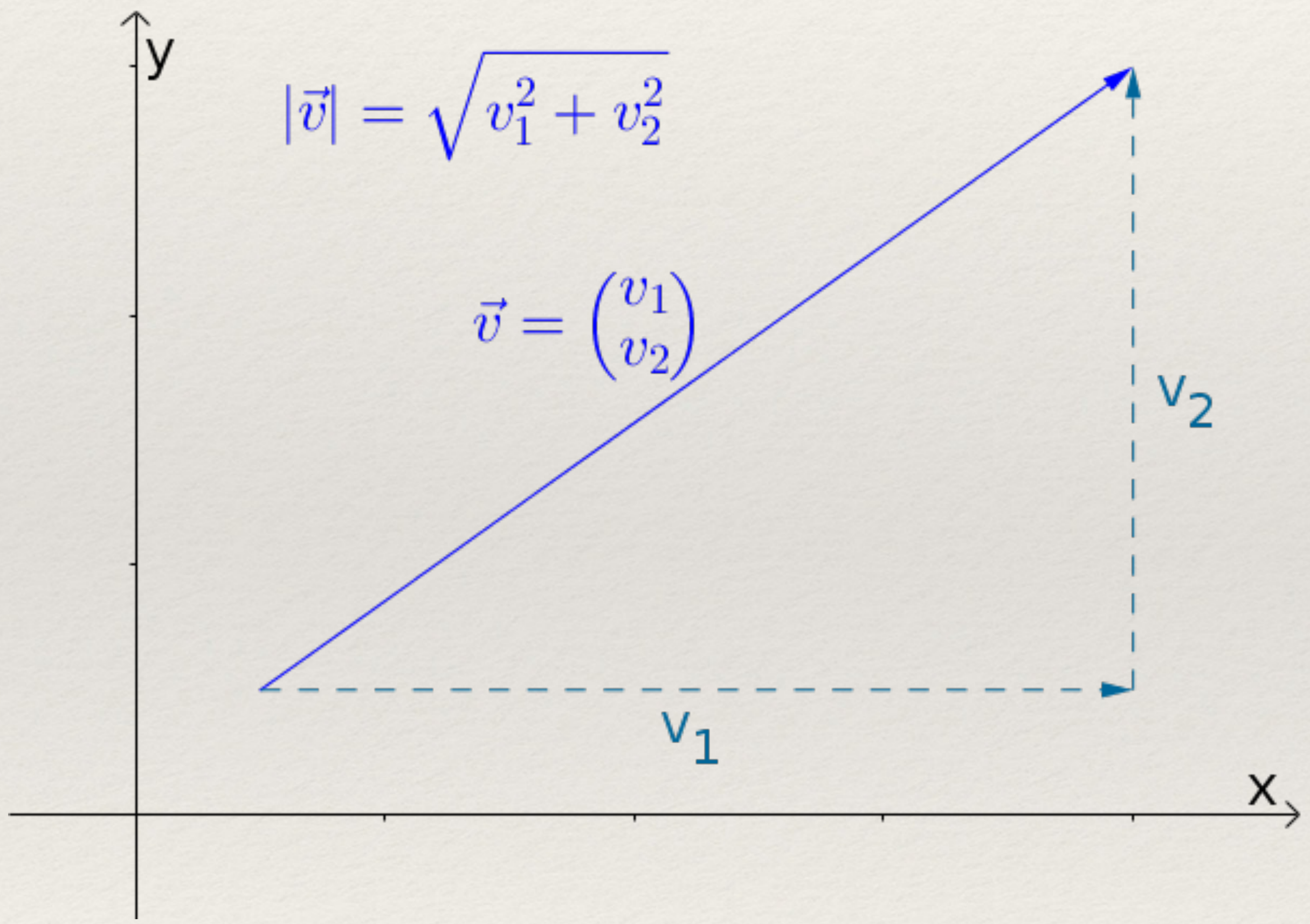


A vector **v** has a **length** |**v**|.

Example 8

```
def sum_of_squares(v: Vector) -> float:
   """Returns v_1 * v_1 + ... + v_n * v_n"""
   return dot(v, v)

import math
def magnitude(v: Vector) -> float:
   """Returns the length of v"""
  return math.sqrt(sum_of_squares(v)) # math.sqrt is square root function

magnitude([1, 2, 2]) # 3
```

Vectors: distance



The **distance** between two vectors **u** and **v** is the length of their difference $|\mathbf{u} - \mathbf{v}|$.

Example 9

```
def distance(v: Vector, w: Vector) -> float:
    return magnitude(subtract(v, w))
```

Matrices

Matrices

$$\begin{array}{cc}
 & \begin{array}{ccccc} 1 & 2 & 3 & \cdots & n \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ \vdots \\ m \end{array} &
\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \\ a_{m1} & a_{m2} & a_{m3} & & a_{mn} \end{bmatrix}
\end{array}$$

A **matrix** is a two-dimensional collection of numbers.

What you need to know:

- two matrices of same size can be **added** to form a new matrix
- two matrices of same size can be **subtracted** to form a new matrix
- a matrix can be **multiplied** by a number to form a new matrix
- two matrices of appropriate sizes can be **multiplied** to form a new matrix

Matrices

$$
\begin{array}{c c}
 & \begin{array}{cccccc} 1 & 2 & 3 & \cdots & n \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ \vdots \\ m \end{array} &
\begin{bmatrix}
a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\
a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\
a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\
\vdots & \vdots & \vdots & \ddots & \\
a_{m1} & a_{m2} & a_{m3} & & a_{mn}
\end{bmatrix}
\end{array}
$$

A **matrix** is a two-dimensional collection of numbers.

We will represent matrices as **lists of lists**, with each inner list having the same size and representing a row of the matrix. If A is a matrix, then A[i][j] is the element in the i-th row and the j-th column.

Example 10

```
Matrix = List[List[float]]

A = [[1, 2, 3], # A has 2 rows and 3 columns
     [4, 5, 6]]

B = [[1, 2], # B has 3 rows and 2 columns
     [3, 4],
     [5, 6]]
```

Example 11

```
from typing import Tuple
def shape(A: Matrix) -> Tuple[int, int]:
    """Returns (# of rows of A, # of columns of A)"""
    num_rows = len(A)

    # number of elements in first row
    num_cols = len(A[0]) if A else 0

    return num_rows, num_cols

shape([[1, 2, 3], [4, 5, 6]]) # (2, 3)
```

A matrix A has len(A) **rows** and len(A[0]) **columns**, which we consider its **shape**.

Example 12

```
def get_row(A: Matrix, i: int) -> Vector:
    """Returns the i-th row of A (as a Vector)"""
    return A[i] # A[i] is already the ith row

def get_column(A: Matrix, j: int) -> Vector:
    """Returns the j-th column of A (as a Vector)"""
    return [A_i[j] # jth element of row A_i
            for A_i in A] # for each row A_i
```

If a matrix has n rows and k columns, we will refer to it as an $n \times k$ matrix. We can think of each row of an $n \times k$ matrix as a vector of length k, and each column as a vector of length n.

Example 13

```
from typing import Callable
def make_matrix(num_rows: int,
                num_cols: int,
                entry_fn: Callable[[int, int], float]) -> Matrix:
    """
    Returns a num_rows x num_cols matrix
    whose (i,j)-th entry is entry_fn(i, j)
    """
    return [[entry_fn(i, j)              # given i, create a list
             for j in range(num_cols)]   # [entry_fn(i, 0), ... ]
            for i in range(num_rows)]    # create one list for each i
```

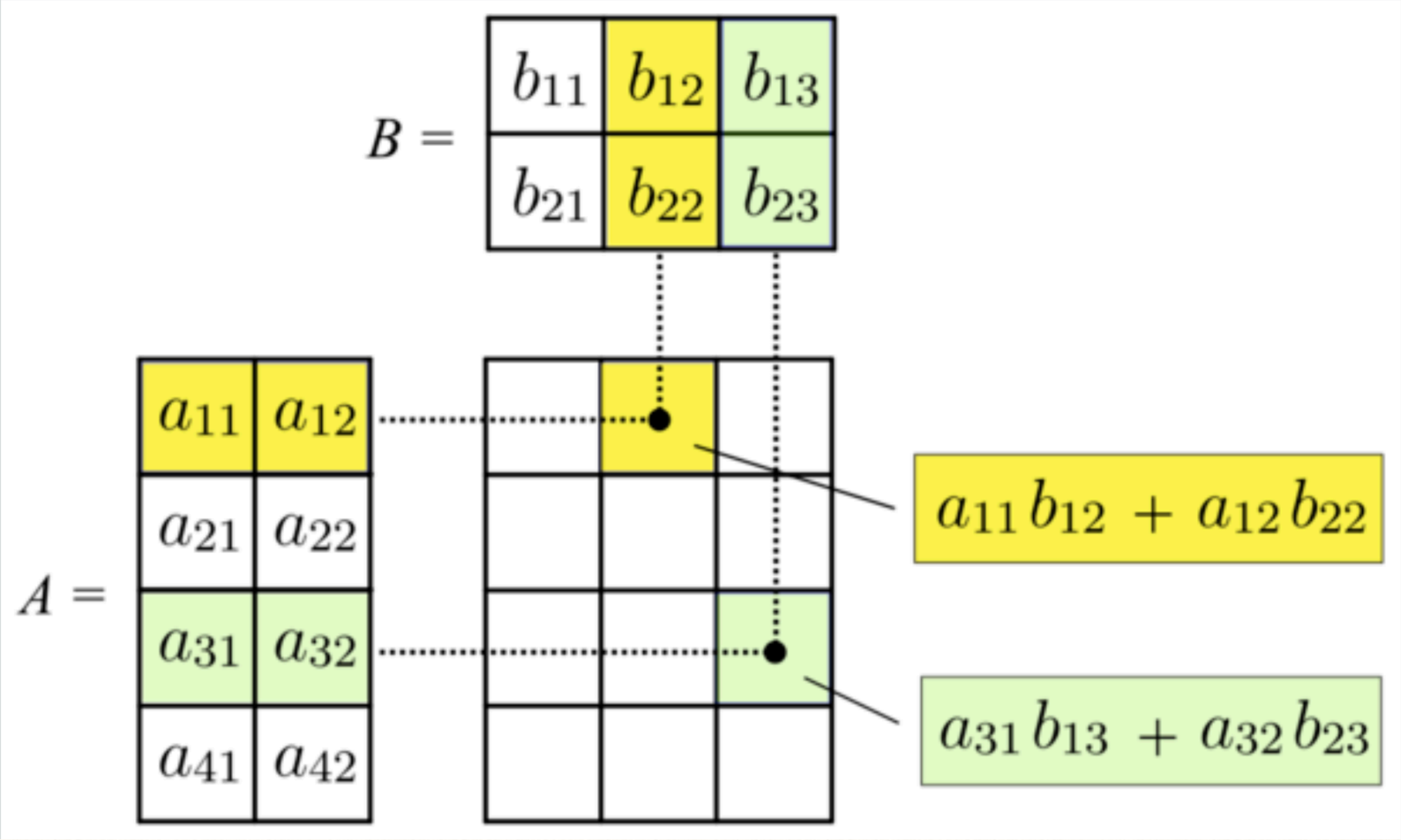
We will be able to create a matrix given its shape and a function for generating its elements.

Example 14: identity matrix

```
def identity_matrix(n: int) -> Matrix:
  """Returns the n x n identity matrix"""
  return make_matrix(n, n, lambda i, j: 1 if i == j else 0)

identity_matrix(5)
"""
[[1, 0, 0, 0, 0],
[0, 1, 0, 0, 0],
[0, 0, 1, 0, 0],
[0, 0, 0, 1, 0],
[0, 0, 0, 0, 1]]
"""
```

Matrices: product

The matrix A has 2 **columns** and B has 2 **raws**, that is why we can multiply these matrices $A \cdot B$.



Why matrices are important to us?

1. We can use a matrix to represent a **dataset** consisting of multiple vectors, simply by considering each vector as a row of the matrix.

2. We can use an $n \times k$ matrix to represent a **linear function** that maps k-dimensional vectors to n-dimensional vectors.

3. Matrices can be used to represent **binary relationships**.

Example 15

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
               (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
friend_matrix = [[0, 1, 1, 0, 0, 0, 0, 0, 0, 0], # user 0
                 [1, 0, 1, 1, 0, 0, 0, 0, 0, 0], # user 1
                 [1, 1, 0, 1, 0, 0, 0, 0, 0, 0], # user 2
                 [0, 1, 1, 0, 1, 0, 0, 0, 0, 0], # user 3
                 [0, 0, 0, 1, 0, 1, 0, 0, 0, 0], # user 4
                 [0, 0, 0, 0, 1, 0, 1, 1, 0, 0], # user 5
                 [0, 0, 0, 0, 0, 1, 0, 0, 1, 0], # user 6
                 [0, 0, 0, 0, 0, 1, 0, 0, 1, 0], # user 7
                 [0, 0, 0, 0, 0, 0, 1, 1, 0, 1], # user 8
                 [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]] # user 9
friend_matrix[0][2] # 1
friends_of_five = [i for i, is_friend in
                   enumerate(friend_matrix[5])
                   if is_friend]  # [4, 6, 7]
```

If there are very few connections, this is a much more inefficient representation, since you end up having to store a lot of zeros.

However, with the matrix representation it is much quicker to check whether two nodes are connected.

Gradient Descent

Optimisation

Often we need to solve a number of optimisation problems:

- find the best model for a certain situation
- minimise the error of its predictions
- maximise the likelihood of the data

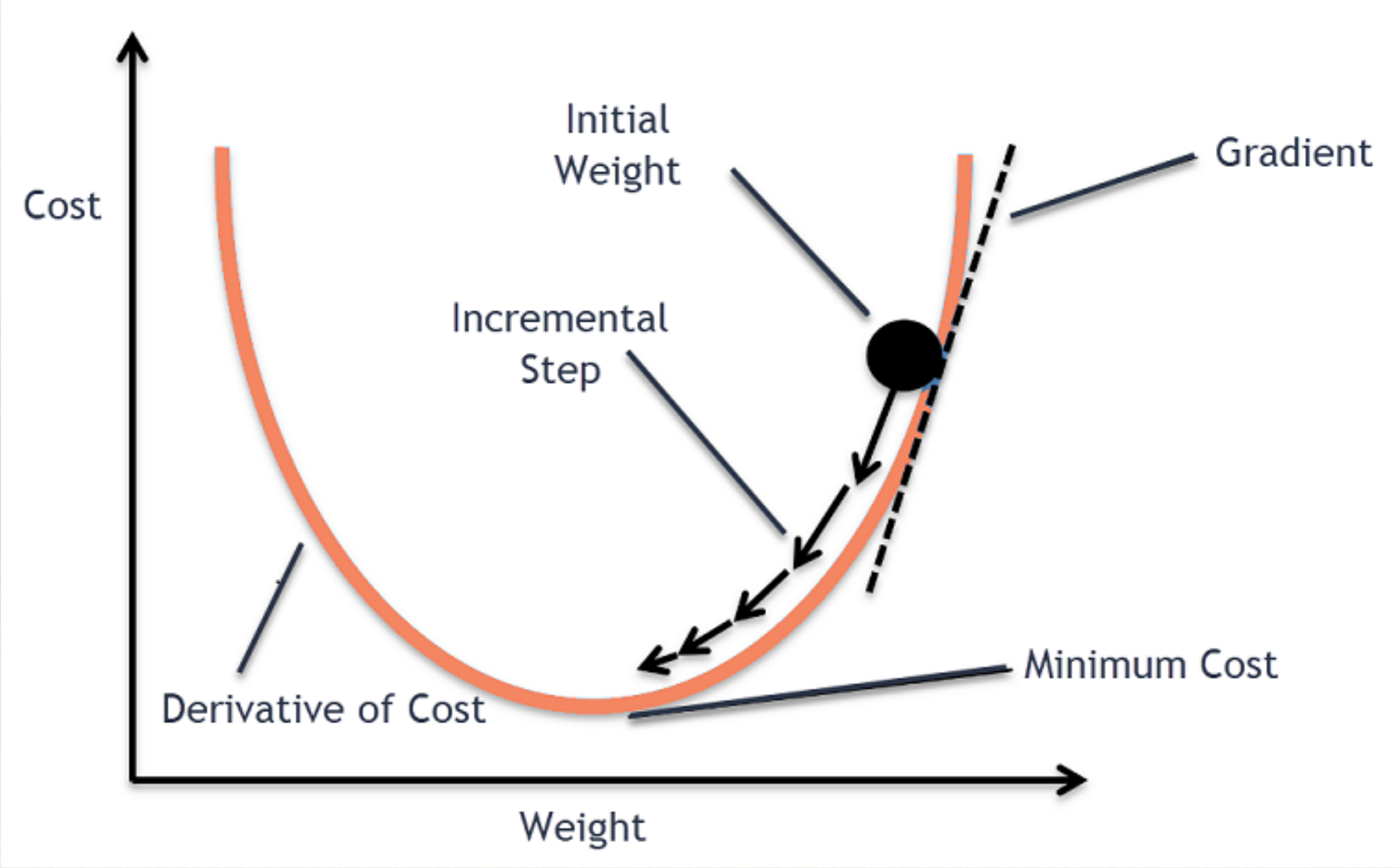
Example 16

```
from scratch.linear_algebra import Vector, dot

def sum_of_squares(v: Vector) -> float:
    """Computes the sum of squared elements in v"""
    return dot(v, v)
```

Suppose we have some function f that takes as input a vector of real numbers and outputs a single real number.

How to maximise or minimise such a function?

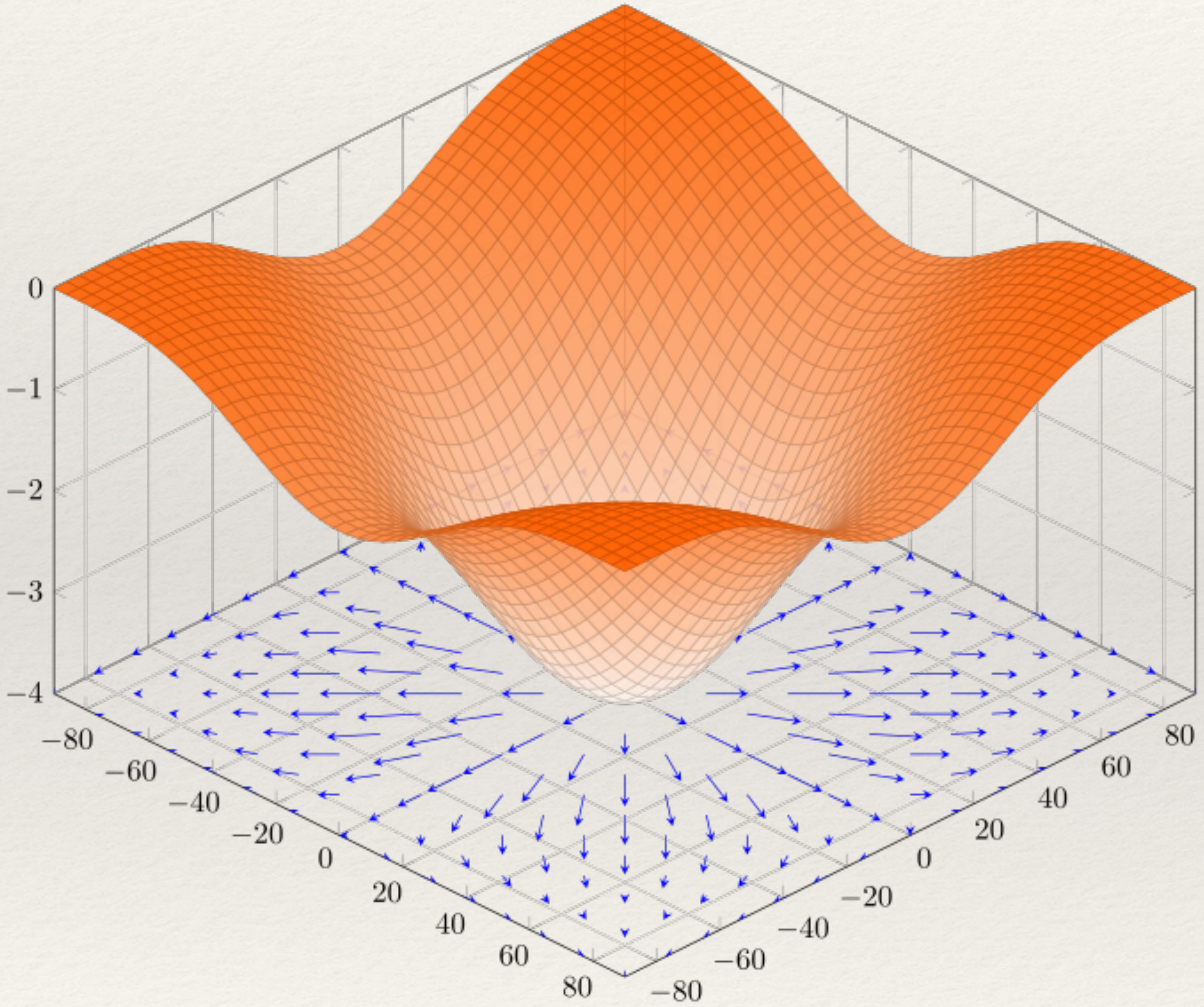
Gradient Descent



One approach to minimising a function is:

- to pick a random starting point,
- compute the gradient,
- take a small step in the direction of the gradient (i.e., the direction that causes the function to decrease the most),
- repeat with the new starting point.

Gradient of a function



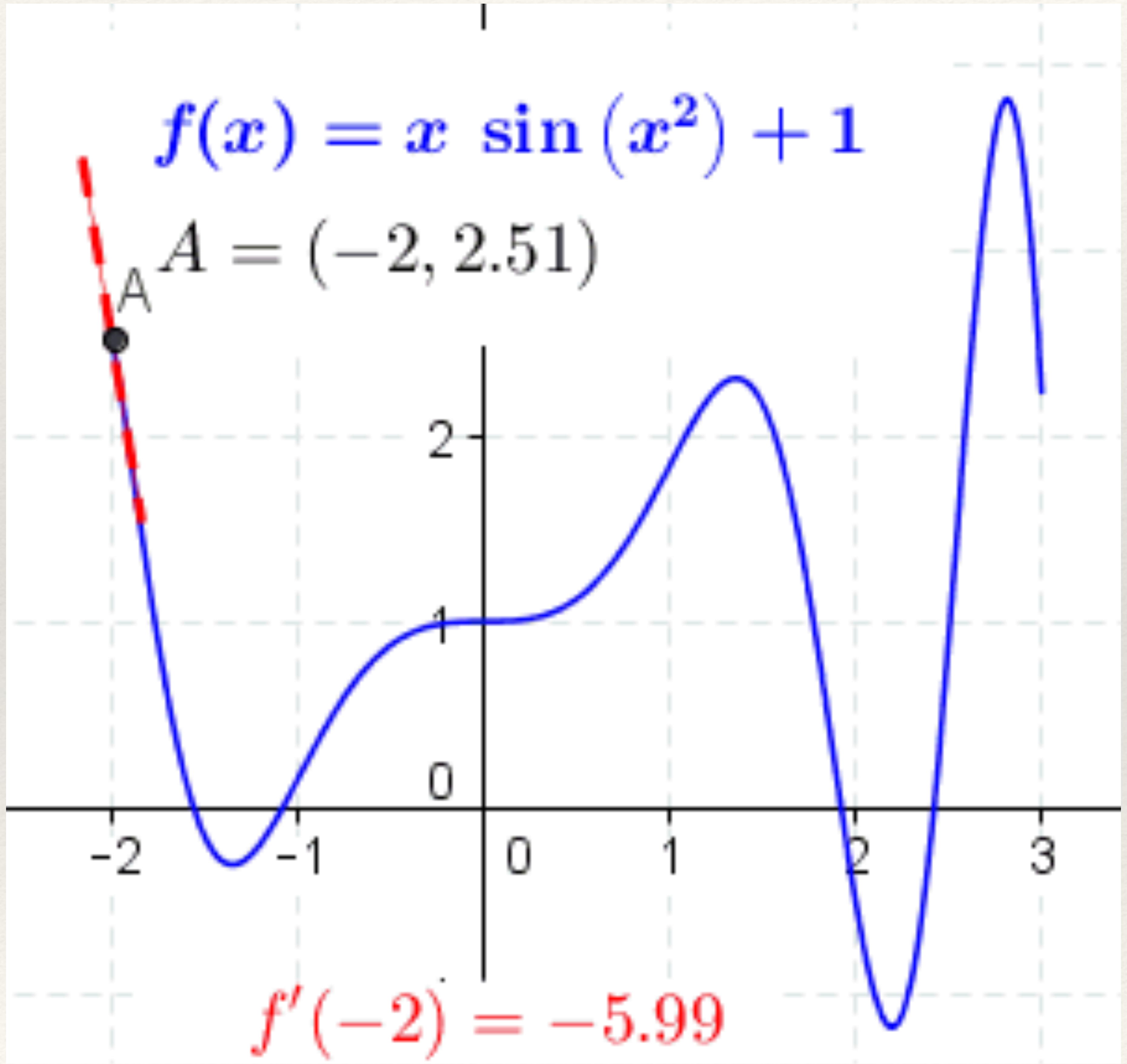
The **gradient** of a one-variable function $f : \mathbb{R} \to \mathbb{R}$ is simply its derivative:

$$\nabla f(x) = f'(x)$$

The **gradient** of an n-variable function $f : \mathbb{R}^n \to \mathbb{R}$ is the vector of its **partial derivatives**:

$$\nabla f(x) = \left[\frac{\partial f}{\partial x_1}(x) , \frac{\partial f}{\partial x_2}(x), \ \ldots, \frac{\partial f}{\partial x_n}(x) \right]$$

Derivative of a function



If f is a function of one variable, its **derivative at a point** x measures how $f(x)$ changes when we make a very small change to x.

Formally, the **derivative** is defined as the limit of the difference quotients:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Thank you!