mmiimran/**linear-algebra-for-data-science**

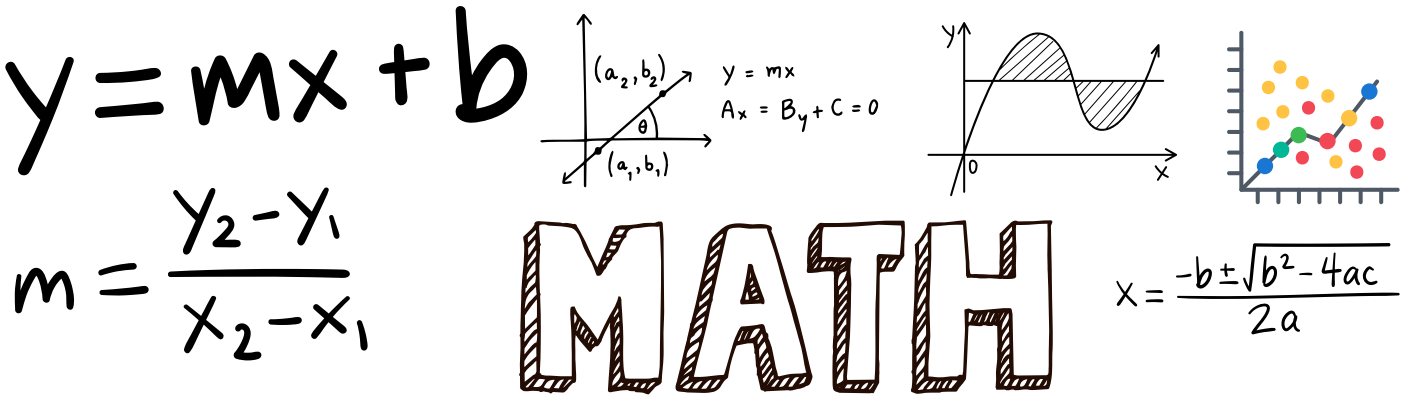
1 Contributor

0 Issues

0 Stars

0 Forks

Vectors (Part 1)



Topic: Vectors

What are vectors?

- Vectors are like arrows in space that show direction and magnitude (or size).

Why are vectors important?

- They form the foundation of linear algebra, which is a big part of math used in many fields.

What will you learn?

- You'll learn everything about vectors: what they are, what they're used for, how to understand them, and how to work with them using Python.

Key operations:

- You'll learn about important things you can do with vectors, like adding them together, multiplying them, and finding the dot product.

Vector decompositions:

- This is about breaking down vectors into simpler parts, which is a big deal in linear algebra.

So, by the end of this chapter, you'll be a vector expert, ready to tackle more advanced stuff in linear algebra!

Creating and Visualizing Vectors in NumPy

---

1. Introduction to Vectors:

- A vector is like a well-organized list of numbers that helps us solve problems in linear algebra.
- While vectors can contain various mathematical objects, we'll focus on numbers for practical applications.

2. Important Characteristics of Vectors:

- **Dimensionality:**
  - **It tells us how many numbers are in the vector.**
- **Orientation:**
  - **It describes whether the vector is arranged as a column (standing tall) or a row (lying flat).**

3. Fancy Symbols and Examples:

- We often use special symbols like $\mathbb{R}^{\mathbf{N}}$ (N is like power) to indicate dimensionality, where $\mathbb{R}$ stands for real numbers.
- For example, $\mathbb{R}^{\mathbf{2}}$ (2 is like base here) represents a vector with two elements.
- Let's look at some examples:

Equation 1-1. Examples of column vectors and row vectors

$$\mathbf{x} = \begin{bmatrix} 1 \\ 4 \\ 5 \\ 6 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} .3 \\ -7 \end{bmatrix}, \quad \mathbf{z} = \begin{bmatrix} 1 & 4 & 5 & 6 \end{bmatrix}$$

- Answers: x is a 4D column vector, y is a 2D column vector, and z is a 4D row vector. We can also write, e.g., $x \in \mathbb{R}4$, where the $\in$ symbol means "is contained in the set of."

4. Understanding Vector Orientation:

- Despite having the same elements (**x** and **z**), vectors in different orientations are technically different.
- We'll delve deeper into this in "Does Vector Orientation Matter?"

Does Vector Orientation Matter?

- **Introduction:**
  - Vector orientation refers to whether vectors are represented as column vectors, row vectors, or orientationless 1D arrays.
  - Understanding vector orientation is crucial in certain situations to avoid errors and unexpected results in Python.
- **Vector Orientation in Data Storage:**
  - In most cases, when using vectors to store data, the orientation doesn't matter.
  - Whether a vector is represented as a column or row doesn't affect the data it holds.
- **Importance in Operations:**
  - However, certain operations in Python may produce errors or unexpected outcomes if the vector orientation is incorrect.
  - It's essential to ensure that the orientation matches the requirements of the operation being performed.
- **Debugging Challenges:**
  - Incorrect vector orientation can lead to frustrating debugging experiences.
  - Spending time debugging code only to realize that the issue stems from mismatched vector orientations can be headache-inducing.

- **Conclusion:**
  - While vector orientation might seem like a minor detail, it can have a significant impact on the outcome of operations in Python.
  - Understanding when vector orientation matters and ensuring alignment with operation requirements can save time and prevent headaches during the coding process.

---

5. Math vs. Coding:

- There are differences between math and coding when dealing with vectors.
- Understanding these differences is crucial for solving problems effectively.

6. Vector Representation in Python:

- We can represent vectors in Python using various data types.
- NumPy arrays are preferred for most linear algebra operations due to their efficiency.
- In Python, we can represent vectors using different data types. One simple option is using lists. Lists are versatile and easy to understand, but they don't directly support many linear algebra operations. That's where NumPy arrays come in. NumPy arrays are like supercharged versions of lists, specifically designed for numerical computations and linear algebra. They offer optimized implementations of mathematical operations and come with built-in support for handling vectors efficiently. So, while lists are great for basic tasks, when it comes to serious number crunching and linear algebra, NumPy arrays are the way to go! Therefore, most of the time it's best to create vectors as NumPy array.
- Let's see how to create vectors in Python using NumPy arrays:

```
asList = [1, 2, 3]

asArray = np.array([1, 2, 3])  # 1D array

rowVec = np.array([[1, 2, 3]]) # row

colVec = np.array([[1], [2], [3]]) # column
```

- In NumPy, the variable **asArray** represents an **orientationless** array, which means it's neither a row nor a column vector but simply a **1D** list of numbers. Orientation in NumPy is denoted by brackets: the outermost brackets encapsulate all the numbers into one object. Additional sets of

brackets indicate rows: a row vector, like **rowVec**, contains all numbers in one row, while a column vector, such as **colVec**, consists of multiple rows, with each row containing one number.

7. Understanding Vector Orientation in Python:

- Python treats vectors differently based on their orientation.
- We can explore this by examining the shapes of the variables.

```
print(f 'asList: { np.shape(asList) }')

print(f 'asArray: { asArray.shape }')

print(f 'rowVec: { rowVec.shape }')

print(f 'colVec: { colVec.shape }')
```

Output:

- **asList: (3,)**
- **asArray: (3,)**
- **rowVec: (1, 3)**
- **colVec: (3, 1)**
-

- The output shows that the 1D array **asArray** is of size (3), whereas the orientation-
- endowed vectors are 2D arrays and are stored as size (1,3) or (3,1) depending on the orientation. Dimensions are always listed as (rows ,columns**)**

Conclusion:

- Understanding vectors and their orientations is crucial for both mathematical and coding applications.
- Practice and experience will help you master these concepts effectively

Geometry of Vectors

Introduction to Ordered Lists and Vectors:

- An ordered list of numbers represents what we call a vector in algebra.
- Think of a vector as a mathematical object that carries both magnitude (length) and direction.

Geometric Interpretation of Vectors:

- When we think about vectors geometrically, we envision them as straight lines with a specific length and direction.
- The **length** of the vector is often referred to as its **magnitude**, while its **direction** is described by an **angle** relative to the positive x-axis.

Understanding Vector Points:

- Every vector has two important points: the **tail** and the **head**.
- The tail is where the vector **starts**, and the head is where it **ends**.
- To **distinguish** between the two, the head often features an **arrow tip**.

Difference Between Vectors and Coordinates:

- While it might seem like vectors and coordinates are similar concepts, they're actually different.
- Vectors represent **direction** and **magnitude**, while coordinates are **fixed point**s in space.
- However, when a vector starts at the origin (0,0), it aligns with a coordinate system.
- This alignment is referred to as the **standard position.**

Illustration of Standard Position:

- The standard position of a vector is depicted when it originates from the origin (0,0) of a coordinate plane.
- This concept helps us understand how vectors relate to coordinate systems and geometric space.

By understanding these key points, you'll have a solid foundation for grasping the relationship between **ordered lists, vectors, coordinates,** and their **geometric interpretations**.

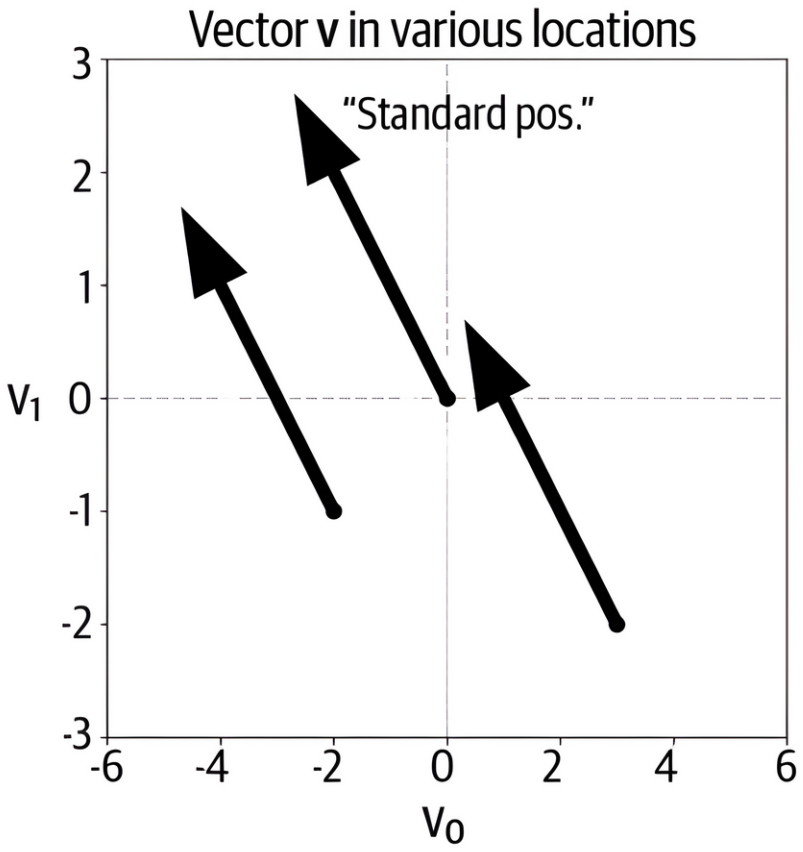


Figure 1-1. All arrows express the same vector. A vector in standard position has its tail at the origin and its head at the concordant geometric coordinate.

Conceptualizing Vectors:

- Vectors can be thought of in two main ways: **geometrically** and **algebraically.**
- These perspectives serve as different lenses through which we understand and apply vector concepts.

Geometric Interpretation:

- In the geometric interpretation, vectors are visualized as **arrows** with a **specific length** and **direction** in space.
- This perspective is particularly useful in fields like physics and engineering, where vectors represent physical quantities such as forces, velocities, or displacements.

- Engineers might use geometric vectors to represent the direction and magnitude of forces acting on structures, for example.

Algebraic Interpretation:

- From an algebraic standpoint, vectors are represented as **ordered lists** of **numbers** or **variables.**
- This interpretation is valuable in fields like **data science** and **computer science**, where vectors are used to **store** and **manipulate numerical data**.
- For **instance**, in **data science**, vectors can represent **features of data points** or **patterns in datasets**, such as sales data over time or customer preferences.

Complementary Applications:

- While geometric interpretation is handy for visualizing real-world phenomena, algebraic interpretation lends itself well to computational analysis and manipulation.
- Both perspectives are essential in various applications, and understanding them allows for a more comprehensive grasp of vector concepts.

Learning Linear Algebra Concepts:

- Often, learners start by understanding vectors geometrically, typically in the context of 2D graphs.
- As they progress, they delve into higher dimensions and more abstract concepts using algebraic representations.
- This gradual transition helps build a solid foundation in linear algebra, paving the way for tackling more complex problems across different disciplines.

By recognizing the interconnectedness of geometric and algebraic interpretations of vectors, students can appreciate the versatility of these concepts and their applications in diverse fields.

Operations on Vectors

Introduction to Vectors:

- Vectors are like the characters in our linear algebra story.
- Just as nouns represent people, places, or things in language, vectors represent **quantities or directions** in mathematics.

The Fun of Linear Algebra:

- The excitement in linear algebra comes from the actions we perform on vectors, similar to verbs in language.
- These actions, called operations, bring vectors to life and allow us to manipulate them to solve problems.

Types of Operations:

- Linear algebra operations can be simple and intuitive, like addition, or more complex, like singular value decomposition.
- Addition is straightforward and behaves just as you'd expect: combining vectors tip-to-tail to find the result.
- Other operations, like singular value decomposition, are more involved and may require dedicated study to understand fully.

Starting with Simple Operations:

- To ease into linear algebra, it's best to start with the simple operations.
- By mastering these basic operations, you'll build a strong foundation for tackling more complex concepts later on.

By understanding that vectors are the **characters** and operations are the **actions** in our linear algebra story, students can approach the subject with a sense of adventure and curiosity, starting with the simple operations and gradually delving into more advanced topics.

Adding Two Vectors

Vector Addition:

- To add two vectors, we simply add each corresponding element together.

Equation 1-2. Adding two vectors

$$\begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 14 \\ 25 \\ 36 \end{bmatrix}$$

- For example, if we have vectors [4, 5, 6] and [10, 20, 30], adding them results in [14, 25, 36].

Dimensionality of Vectors:

- Vector addition is defined only for vectors with the same dimensionality.
- You can't add a vector in $\mathbb{R}^3$ (3-dimensional space) with a vector in $\mathbb{R}^5$ (5-dimensional space).

Vector Subtraction:

- Vector subtraction works similarly to addition; you subtract corresponding elements.

Equation 1-3. Subtracting two vectors

$$\begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} - \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} -6 \\ -15 \\ -24 \end{bmatrix}$$

- For instance, subtracting [10, 20, 30] from [4, 5, 6] yields [-6, -15, -24].

Python Implementation of Vector Addition:

- In Python, adding vectors is straightforward using libraries like NumPy.

```
v = np.array([4,5,6])

w = np.array([10,20,30])

u = np.array([0,3,6,9])

vPlusW = v+w

uPlusW = u+w  # error! dimensions mismatched!
```

- For example, with NumPy arrays v and w, v + w gives the result of adding the two vectors.
- However, it's crucial to ensure that the dimensions of the vectors match; otherwise, you'll encounter an error.

Does vector orientation matter for addition? Consider Equation 1-4:

Equation 1-4. Can you add a row vector to a column vector?

$$\begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} + \begin{bmatrix} 10 & 20 & 30 \end{bmatrix} = ?$$

You might think that there is no difference between this example and the one shown earlier—after all, both vectors have three elements. Let's see what Python does:

```
v = np.array([[4,5,6]]) # row vector

w = np.array([[10,20,30]]).T # column vector

v+w

>> array([[14, 15,16],

[24, 25, 26],

[34, 35, 36]])
```

Orientation and Addition:

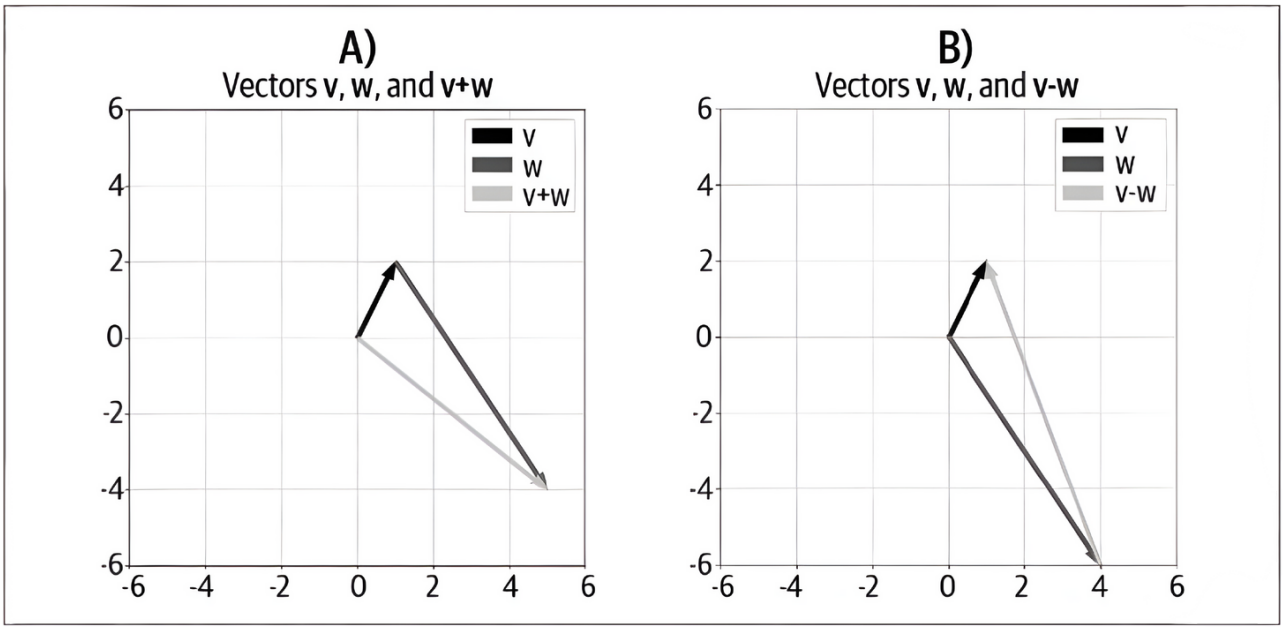
- The orientation of vectors matters when adding them together.
- Consider the case of adding a row vector to a column vector, like [4, 5, 6] and [10, 20, 30].
- Python's broadcasting feature may lead to unexpected results, as it extends dimensions to match.
- This example highlights that vectors can be added only if they have the same dimensionality and orientation.

Broadcasting in Python:

- Broadcasting is a concept in Python that allows operations on arrays with different shapes.
- It extends dimensions to make them compatible for operations like addition.
- While broadcasting can be useful, it's essential to understand its implications and potential for unexpected results.

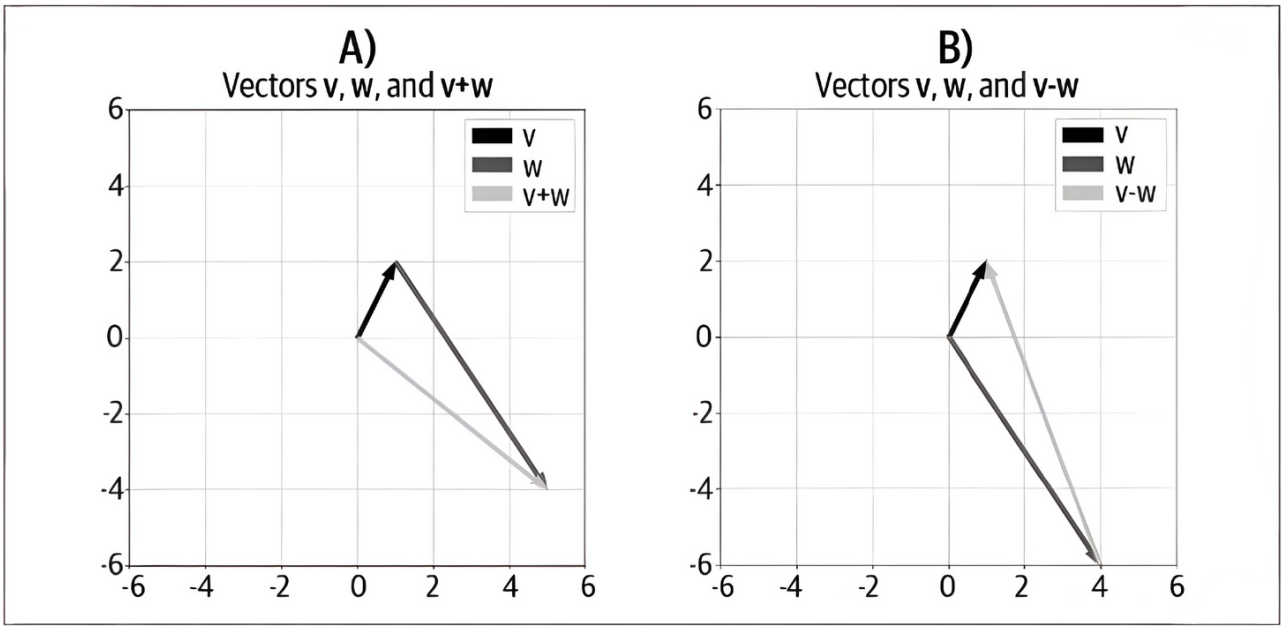
By understanding these concepts, you'll be equipped to perform vector addition and subtraction and comprehend how Python handles these operations, including the nuances of dimensionality and orientation.

Geometry of Vector Addition and Subtraction



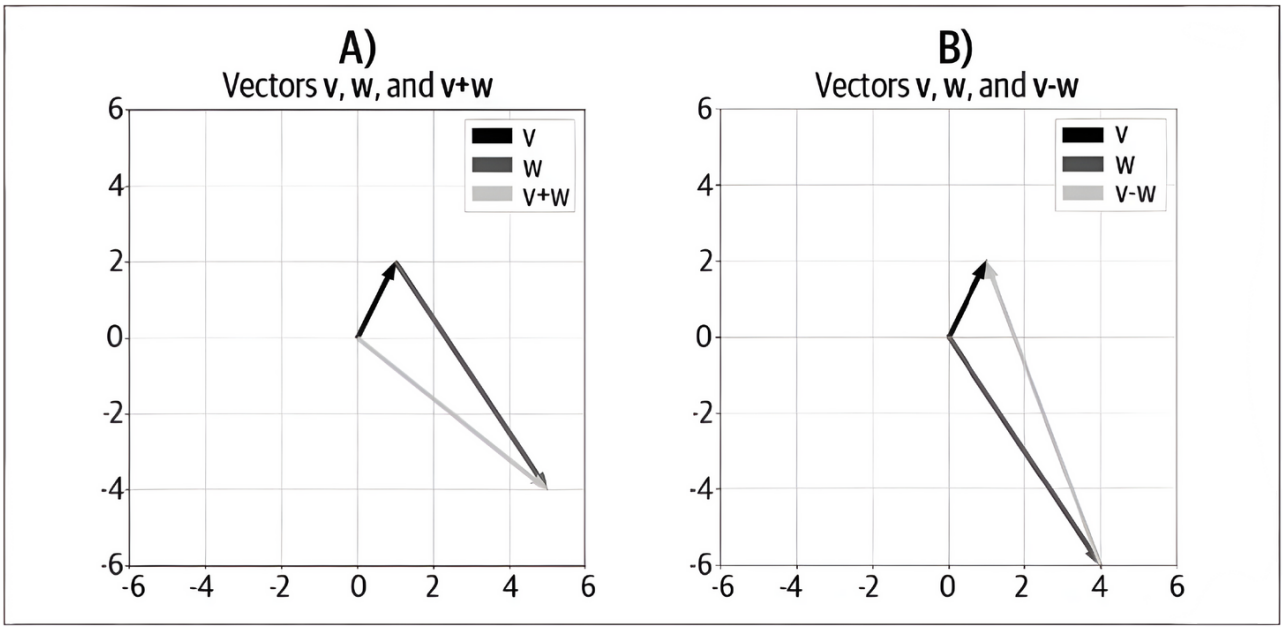
Vector Addition - Geometric Interpretation:

- To add two vectors geometrically, position them so that the tail of one vector is at the head of the other.
- The summed vector then traverses from the tail of the first vector to the head of the second (graph A in Figure 1-2).
- This method extends to summing any number of vectors: stack them tail-to-head, and the sum is the line from the first tail to the final head.



Vector Subtraction - Geometric Interpretation:

- Subtracting vectors geometrically involves aligning them so that their tails coincide (both in standard position).
- The difference vector then extends from the head of the "negative" vector to the head of the "positive" vector (graph B in Figure 1-2).
- This method is equally straightforward and crucial for understanding vector operations geometrically.



Importance of Vector Subtraction Geometry:

- The geometry of vector subtraction is fundamental for various applications in science and engineering.
- It serves as the **foundation for orthogonal vector decomposition**, a technique used extensively in linear algebra.
- Orthogonal vector decomposition, in turn, is crucial for concepts like **linear least squares**, which finds widespread use in scientific and engineering fields.

By understanding the geometric interpretations of vector addition and subtraction, students gain insight into how vectors behave visually and how these operations are applied in real-world scenarios, particularly in scientific and engineering contexts.

Vector-Scalar Multiplication

Understanding Scalars:

- In linear algebra, a scalar is a single number, not embedded within a vector or matrix.
- Scalars are represented using lowercase Greek letters, such as α *(**alpha**)* or λ *(**lambda**)*.

Indicating Scalar-Vector Multiplication:

- When performing multiplication between a scalar and a vector, we use notation like $\beta\mathbf{u}$.
- Here, β is the **scalar**, and **u** is the **vector.**

The Process of Scalar-Vector Multiplication:

- Scalar-vector multiplication is straightforward: multiply each element of the vector by the scalar.
- This operation scales the vector by the value of the scalar, affecting both its magnitude and direction.

Example of Scalar-Vector Multiplication (Equation 1-5):

Equation 1-5. Vector-scalar multiplication (or: scalar-vector multiplication)

$$\lambda = 4,\ \mathbf{w} = \begin{bmatrix} 9 \\ 4 \\ 1 \end{bmatrix}, \quad \lambda\mathbf{w} = \begin{bmatrix} 36 \\ 16 \\ 4 \end{bmatrix}$$

- Let's consider an example: λ **=4** and **w=[[9 4 1]]**
- To find $\lambda\mathbf{w}$, we multiply each element of **w** by λ.
- The result is **[[36 16 4]]**

In summary, scalar-vector multiplication involves multiplying each element of a vector by a scalar. This operation is fundamental in linear algebra and allows us to scale vectors according to specific factors.

Introduction to Zeros Vector:

- In linear algebra, a vector consisting of all zeros is called the zeros vector.
- It's represented using a boldfaced zero, **0**, and serves as a special case in vector algebra.

The Trivial Solution:

- Using the zeros vector to solve a problem is often referred to as the trivial solution.
- This solution is considered trivial because it doesn't offer meaningful information or insights into the problem at hand.

Nontrivial Solutions:

- Linear algebra problems often seek nonzeros vectors or nontrivial solutions.
- These solutions provide valuable information and contribute to the understanding or resolution of the problem.

Examples:

- Statements like "find a nonzeros vector that can solve..." or "find a nontrivial solution to..." are common in linear algebra.
- These statements emphasize the importance of finding meaningful solutions that aren't simply the result of using the zeros vector.

In summary, while the zeros vector is a valid vector in linear algebra, its use in solving problems is often considered trivial. Linear algebra problems typically seek nontrivial solutions that provide valuable insights and understanding.

Data Type Importance in Scalar-Vector Multiplication:

- The code snippet demonstrates how the behavior of the **asterisk operator (*)** in Python depends on the data type of the operands.
- When multiplying a scalar and a list, the asterisk operator repeats the list ***s*** times, where ***s*** is an integer.
- However, when the vector is stored as a NumPy array, the asterisk operator performs **element-wise multiplication**, consistent with scalar-vector multiplication in linear algebra.

```
s = 2

a = [3,4,5] # as list

b = np.array(a) # as np array

print(a*s)

print(b*s)

>> [ 3, 4, 5, 3, 4, 5 ]

>> [ 6 8 10 ]
```

Exercise - Setting *s*=2.0:

- If ***s*** is set to **2.0**, it becomes a floating-point number.
- When used in scalar multiplication:
  - If the vector is a list, the asterisk operator will still repeat the list ***s*** times, as 2.0 is considered a float. However, Python will convert it to an integer, resulting in the same behavior as before.
  - If the vector is a NumPy array, the result will remain the same as before, performing element-wise multiplication.

Real-world Coding Consideration:

- The distinction between list repetition and vector-scalar multiplication is important in real-world coding scenarios.
- Understanding how Python treats different data types and operators is crucial for writing correct and efficient code.

In summary, the behavior of the asterisk operator in scalar-vector multiplication varies depending on the data type of the operands. It's essential to be aware of this distinction when working with vectors in Python.

Scalar-Vector Addition

Adding a Scalar to a Vector in Linear Algebra:

- Formally, in linear algebra, adding a scalar to a vector is not a defined operation.
- Scalars and vectors are distinct mathematical objects with different properties and operations, and they cannot be combined directly through addition in the context of linear algebra.

Implementation in Numerical Processing Programs:

- Despite the lack of formal definition in linear algebra, numerical processing programs like **Python allow adding scalars to vectors**.
- In such programs, adding a scalar to a vector is **interpreted as scalar-vector multiplication**, where the scalar is added to each element of the vector.

Illustration with Python Code:

- The provided Python code demonstrates the concept of adding a scalar to a vector using a simple example.

```
import numpy as np

# Define a scalar

s = 2

# Define a vector

v = np.array([3, 6])
```

```
# Add the scalar to the vector

result = s + v

print(result)

>> [5,8]
```

Explanation of the Code:

- In the code, a NumPy array **v** represents the vector [3, 6].
- The scalar **s** is set to **2**.
- Adding **s** to **v (s + v)** results in adding **2** to each element of the vector **v**.
- The result is **[5, 8]**, where each element of the vector **v** has been incremented by **2**.

Considerations:

- While numerical processing programs like Python allow for such operations, it's important to recognize that they deviate from formal linear algebra definitions.
- When performing such operations, especially in mathematical contexts, it's crucial to understand the underlying interpretation and to use caution to ensure consistency with mathematical principles.

> **In summary, while adding a scalar to a vector is not formally defined in linear algebra, numerical processing programs like Python allow for such operations, treating them as scalar-vector multiplication. However, it's important to recognize the distinction and exercise caution when applying such operations in mathematical contexts.**

---

a*s throws an error, because list repetition can only be done using integers; it's not possible to repeat a list 2.72 times

---

The geometry of vector-scalar multiplication

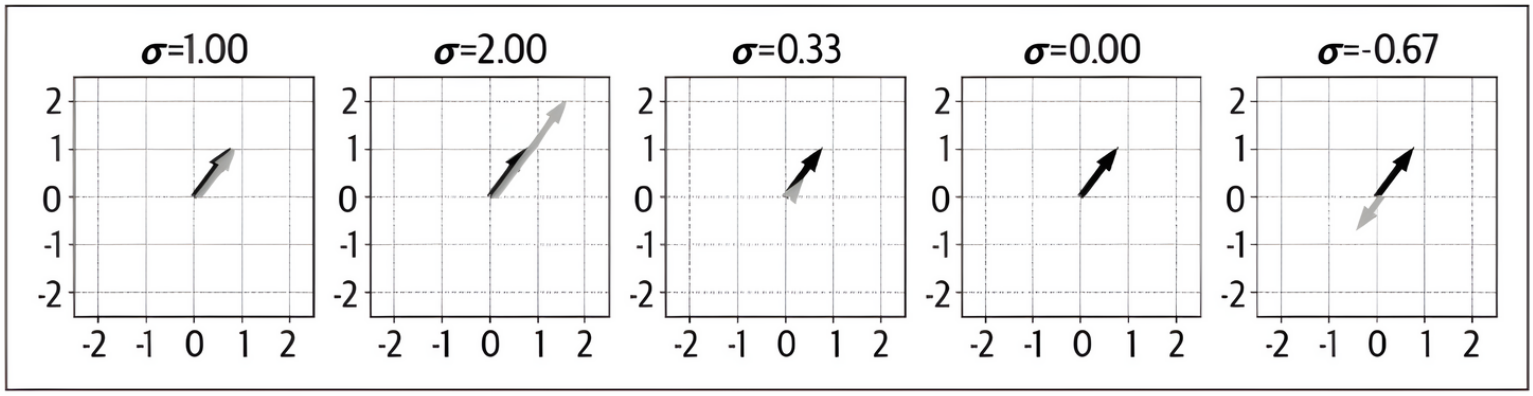


Figure 1-3. The same vector (black arrow) multiplied by different scalars σ (sigma) (gray line; shifted slightly for visibility)

Here's an explanation of why scalars are called "scalars" and their effects on vector-scalar multiplication, along with the concept of vector averaging:

Origin of the Term "Scalars":

- The term "scalars" comes from the geometric interpretation of their effect on vectors.
- Scalars scale vectors without altering their direction, akin to stretching or compressing them along the same line.

Effects of Vector-Scalar Multiplication:

Vector-scalar multiplication has different effects depending on the value of the scalar:

- **Greater than 1**: Increases the magnitude of the vector.
- **Between 0 and 1**: Decreases the magnitude of the vector.
- **Exactly 0**: Results in a zero vector, regardless of the initial vector.
- **Negative**: Flips the direction of the vector by **180°**, but it still points along the same infinite line passing through the origin.

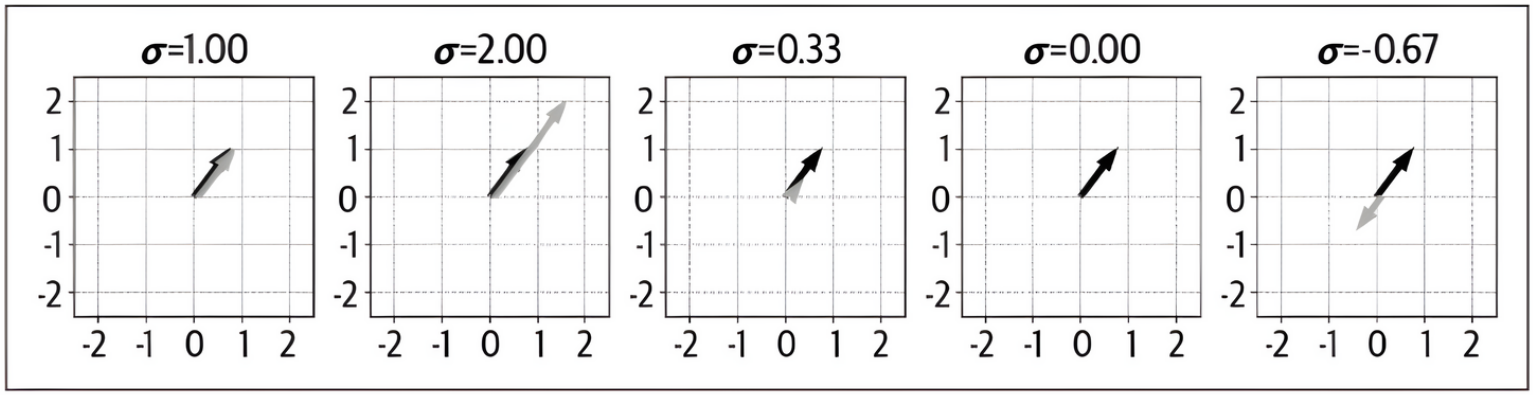


Figure 1-3. The same vector (black arrow) multiplied by different scalars σ (gray line; shifted slightly for visibility)

Interpretation of Negative Scalars:

- While negative scalars might appear to change the direction of vectors, they are interpreted as still pointing along the same infinite line.
- This interpretation is crucial for understanding concepts like **matrix spaces, eigenvectors**, and **singular vectors** introduced in later on.

Vector Averaging:

- Vector-scalar multiplication, combined with vector addition, leads directly to vector averaging.
- Averaging vectors involves summing them and then scalar multiplying the result by the reciprocal of the number of vectors.
- For example, to average two vectors, add them and then scalar multiply by 1/2. Similarly, to average N vectors, sum them and scalar multiply the result by $1/N$.

Understanding the effects of scalars on vectors and their role in vector averaging provides valuable insights into linear algebra concepts and their practical applications.

Transpose

Formal Definition of Transpose:

- It converts column vectors into row vectors, and vice versa
- The transpose operation on a matrix swaps the indices of its elements.
- For a matrix element $m_{i,j}$ its transpose $m_{j,i}$ swaps the row index i with the column index j.

Generalization to Matrices:

- Matrices have both rows and columns, so each element has a (row, column) index pair.
- The transpose operation swaps these indices, effectively swapping rows and columns.

Equation 1-6 - Formal Transpose Operation:

- The formal definition of the transpose operation is given by Equation 1-6:

$$\mathbf{m}^{\mathrm{T}}_{i,j} = \mathbf{m}_{j,i}$$

Orientation of Vectors:

- Vectors can be represented as either row vectors or column vectors, depending on their orientation.
- A row vector with **6 dimensions** has indices i**=1** and j ranging from **1** to **6.**
- Conversely, a column vector with **6 dimensions** has indices i ranging from **1 to 6** and j**=1**.
- Swapping the i **and** j indices effectively swaps rows and columns, transforming between row and column vectors.

Important Rule - Transposing Twice:

- Transposing a vector or matrix twice returns it to its original orientation.

- Symbolically, $\boldsymbol{vTT = v}$.
- This property is crucial in various proofs in data science and machine learning, including the creation of symmetric covariance matrices and principal components analysis.

In summary, the transpose operation swaps the indices of matrix elements, effectively swapping rows and columns. Understanding this operation is essential for manipulating matrices and vectors in various mathematical contexts, including data science and machine learning.

Vector Broadcasting in Python

Introduction to Broadcasting:

- Broadcasting is an operation specific to modern computer-based linear algebra.
- It enables the repeated application of an operation between one vector and each element of another vector, facilitating efficient and compact computations.

Illustration with Equations:

- Consider the series of equations provided:

$$[1\ \ 1] + [10\ \ 20]$$
$$[2\ \ 2] + [10\ \ 20]$$
$$[3\ \ 3] + [10\ \ 20]$$

- These equations exhibit a pattern where each element of the first vector [1 2 3] is added to each element of the second vector [10 20].

Implementation in Python:

- The provided Python code demonstrates how to implement the above equations using broadcasting:

```
import numpy as np

v = np.array([[1,2,3]]).T  # column vector
```

```
w = np.array([[10,20]])   # row vector

result = v + w            # addition with broadcasting

print(result)
```

```
>> array([[11, 21],
          [12, 22],
          [13, 23]])
```

- The output is an array where each element of the first vector has been added to each element of the second vector.

Importance of Orientation:

- The orientation of vectors (row or column) is crucial in linear algebra operations, including broadcasting.
- Changing the orientation of vectors can lead to different results or errors in computations.

Efficiency in Numerical Coding:

- Broadcasting allows for efficient and compact computations, making it a common technique in numerical coding.
- It simplifies the implementation of operations involving arrays and matrices, enhancing computational efficiency.

Further Examples:

- Broadcasting is commonly used in various mathematical and computational tasks, including k-means clustering, as illustrated in section 3 of the book.

In summary, broadcasting in linear algebra enables efficient and compact computations by applying operations between vectors and arrays. It is a powerful technique used extensively in numerical coding to streamline computations and enhance computational efficiency. Understanding broadcasting is essential for effectively leveraging linear algebra operations in modern computer-based applications.

Python still broadcasts, but the result is a 3 × 2 matrix instead of a 2 × 3 matrix.

Vector Magnitude and Unit Vectors

Magnitude of a Vector (Norm):

- The **magnitude**, also known as the **norm** or **geometric length**, represents the distance from the **tail** to the **head** of a vector.
- It is computed using the standard **Euclidean** distance formula, which involves taking the square root of the sum of squared vector elements.
- The notation for vector magnitude is indicated using double-vertical bars around the vector: $||\boldsymbol{v}||$.
- The formula for the norm of a vector is given by Equation 1-7, which sums the squares of the vector elements and takes the square root of the result.

$$\| \mathbf{v} \| = \sqrt{\sum_{i=1}^{n} v_i^2}$$

Applications and Terminology:

- Some applications may use squared magnitudes ($||\boldsymbol{v}||^2$), where the square root term drops out.
- There are discrepancies in terminology between "chalkboard" linear algebra and Python linear algebra.
- In mathematics, the dimensionality of a vector refers to the number of elements, while the length is a geometric distance.
- In Python, the function **len()** returns the dimensionality of an array, while **np.linalg.norm()** returns the geometric length (**magnitude**) of a vector.
- In this context, the term "**magnitude**" (or "**geometric length**") is used instead of "**length**" to avoid confusion.

Unit Vectors:

- A unit vector is a vector with a geometric length of one ($||\boldsymbol{v}||=\mathbf{1}$).
- Examples of applications requiring unit vectors include orthogonal matrices, rotation matrices, eigenvectors, and singular vectors.
- Creating a unit vector from a non-unit vector involves scaling the vector by the reciprocal of its norm, as described in Equation 1-8.
- Unit vectors are conventionally indicated by adding a hat symbol ($\boldsymbol{v}$^) to the vector in the same direction as their parent vector $\boldsymbol{v}$.

Equation 1-8. Creating a unit vector

$$\hat{\mathbf{v}} = \frac{1}{\| \mathbf{v} \|}\mathbf{v}$$

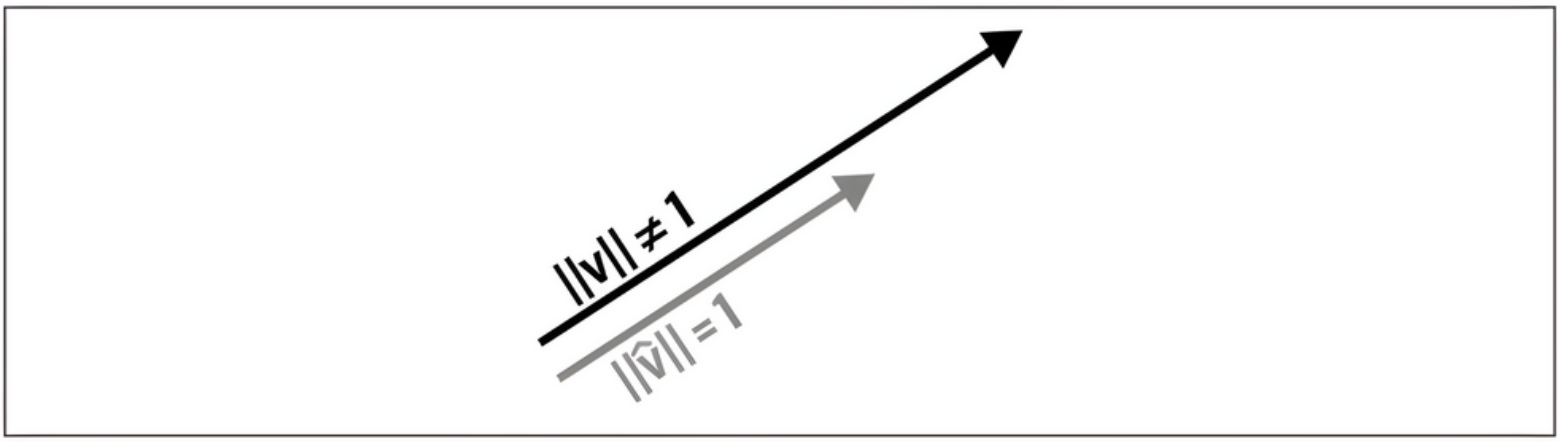


Figure 1-4 A unit vector (gray arrow) can be crafted from a nonunit vector (black arrow); both vectors have the same angle but different magnitudes

Exception to the Claim:

- While it's generally true that any non-unit vector has an associated unit vector, there is one exception.
- The zeros vector (**0**), which has zero length, does not have an associated unit vector because it lacks direction and cannot be scaled to have a nonzero length.

Actually, the claim that "any nonunit vector has an associated unit vector" is not entirely true.There is a vector that has nonunit length and yet has no associated unit vector. Can you guess which vector it is?

Ans: The zeros vector has a length of 0 but no associated unit vector, because it has no direction and because it is impossible to scale the zeros vector to have nonzero length

---

Python Code for Creating Unit Vectors:

- The text mentions that creating unit vectors is one of the exercises at the end of the chapter, and therefore, Python code for this purpose is not provided within the text.

In summary, the text provides comprehensive explanations of vector magnitude, unit vectors, and their applications, along with the related Python terminology. It also highlights a notable exception regarding the association of unit vectors with non-unit vectors and leaves the creation of unit vectors as an exercise for the reader.

The Vector Dot Product

Significance of the Dot Product(Inner product):

- The dot product is a fundamental operation in linear algebra, serving as a basic computational building block for various operations and algorithms.
- It plays a key role in operations like **convolution, correlation, Fourier transform, matrix multiplication, linear feature extraction,** and **signal filtering**.

Notations for Dot Product:

- There are different notations for indicating the dot product between two vectors, such as $a\boldsymbol{T}b$, $\boldsymbol{a \cdot b}$, or $\boldsymbol{(a,b)}$.

- The notation ***aTb*** is commonly used and will become clearer after learning about matrix multiplication.

Algorithm for Computing the Dot Product:

- To compute the dot product, one multiplies the corresponding elements of the two vectors and then sums over all the individual products.
- This involves element-wise multiplication followed by summation, as shown in Equation 1-9.

$$\delta = \sum_{i=1}^{n} a_i b_i$$

Example of Dot Product Calculation:

- Equation 1-10 provides a numerical example of computing the dot product between two vectors.

$$\begin{aligned}[1\ 2\ 3\ 4] \cdot [5\ 6\ 7\ 8] &= 1 \times 5 + 2 \times 6 + 3 \times 7 + 4 \times 8 \\ &= 5 + 12 + 21 + 32 \\ &= 70\end{aligned}$$

- It demonstrates how to multiply corresponding elements of the vectors and then sum the individual products to obtain the dot product.

Irritations of Indexing

- In math and some programming languages like MATLAB and Julia, counting starts at 1 and goes up to N.
- However, in languages like Python and Java, counting starts at 0 and goes up to N - 1.
- While we won't discuss which convention is better, it's crucial to remember this difference when translating math formulas into Python code.
- This inconsistency in indexing can sometimes lead to bugs in code, so it's essential to be aware of it while coding in Python.

Implementation in Python:

- Python provides multiple ways to implement the dot product, with the most straightforward being the **np.dot()** function from the **NumPy library**.
- The **np.dot()** function is commonly used for computing dot products between vectors or performing matrix multiplication.

```
v = np.array([1,2,3,4])

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

np.dot(v,w)
```

Note regarding np.dot():

The np.dot() function in NumPy doesn't perform the vector dot product directly; instead, it handles matrix multiplication, which involves a series of dot products. This distinction becomes clearer once you delve into the rules and mechanics of matrix multiplication in section 4. However, if you're eager to explore this concept now, you can experiment with modifying previous code examples to specify the orientation of both vectors (row versus column). You'll notice that the output corresponds to the dot product only when the first input is a row vector and the second input is a column vector.

Scalar Multiplication and Dot Product: When we scalar multiply one vector, it scales the dot product by the same amount. Let's say we have vectors **v** and **w**, and we scalar multiply **v** by a scalar value, denoted as **s**. Computing the dot product of **sv** and **w** is equivalent to scaling the dot product of **v** and **w** by the same scalar value.

```
s = 10

np.dot(s*v,w)
```

For example, if the dot product of **v** and **w** is **70**, then the dot product using sv (written as **σ vTw** in math notation) would be **700** when **s = 10**. Trying it with a negative scalar, like **-1**, we notice that

while the magnitude of the dot product remains the same, the sign is reversed. And when **s = 0**, the dot product becomes **zero**.

Interpretation of the Dot Product: Now, let's understand the interpretation of the dot product. It serves as a measure of similarity or mapping between two vectors. For instance, consider collecting height and weight data from **20** people and storing them in two vectors. Since taller people tend to weigh more, we'd expect these variables to be related, resulting in a large dot product between the vectors. However, the magnitude of the dot product depends on the scale of the data. Data measured in grams and centimeters would yield a larger dot product than data measured in pounds and feet. To address this, we use a normalization factor. The normalized dot product between two variables is known as the **Pearson correlation coefficient**, which is a crucial analysis in data science, further explored in Chapter 3!

In summary, the dot product is a crucial operation in linear algebra with widespread applications across various domains. Understanding how to compute, interpret, and manipulate dot products is essential for effectively using linear algebra in practical applications, including data science and machine learning.

The Dot Product Is Distributive

Distributive Property:

- The distributive property of mathematics states that $\boldsymbol{a \cdot (b+c) = a \cdot b + a \cdot c}$.
- Translated into vectors and the dot product, this means that the dot product of a vector sum equals the sum of the vector dot products.

$$\mathbf{aT\ b + c = aTb + aTc}$$

Illustration with Python Code:

```
a = np.array([ 0,1,2 ])

b = np.array([ 3,5,8 ])
```

```
c = np.array([ 13,21,34 ])

# the dot product is distributive

res1 = np.dot( a, b+c )

res2 = np.dot( a,b ) + np.dot( a,c )
```

- The provided Python code demonstrates the distributivity property using NumPy arrays.
- Three vectors ***a***, ***b***, and ***c*** are defined.
- The dot product of ***a*** with the sum of ***b*** and ***c*** is computed and stored in res1.
- The sum of the dot products of ***a*** with ***b*** and ***c*** individually is computed and stored in **res2**.
- Both **res1** and **res2** should yield the same result, demonstrating the distributive property.

Understanding the Code:

- The code accurately translates the mathematical formula into Python code, showcasing the application of the distributive property in a programming context.
- Demonstrating how to translate mathematical concepts into code is emphasized as an important skill in math-oriented coding.

In summary, the text effectively explains the distributive property of mathematics in the context of vectors and the dot product, and provides a clear illustration of this property using Python code. Understanding and applying such properties is crucial for mathematical modeling and computational tasks.

Geometry of the Dot Product

Geometric Definition of the Dot Product:

- The dot product can also be defined geometrically as the product of the magnitudes of the two vectors, scaled by the c**osine of the angle between them**.
- This geometric definition is represented by Equation 1-11, where α represents the dot product, $\boldsymbol{\theta}$ is the angle between the vectors, and $||v||$ and $||w||$ are the magnitudes of the vectors v and w, respectively.

$$\alpha = \cos\left(\theta_{\mathbf{v},\mathbf{w}}\right) \| \mathbf{v} \| \| \mathbf{w} \|$$

Equivalence with Mathematical Definition:

- Equation 1-9 and Equation 1-11 represent two different forms of expressing the dot product, but they are mathematically equivalent.
- Proving their equivalence requires principles such as the **Law of Cosines**, but the proof is omitted in the context of this book.

Sign of the Dot Product:

- The sign of the dot product is entirely determined by the geometric relationship between the two vectors.
- Vector magnitudes are always positive (except for the zeros vector), while the cosine of an angle can range from -1 to +1.
- Figure 1-5 illustrates five cases of the dot product sign based on the angle between the vectors, demonstrating how the sign changes with varying angles.

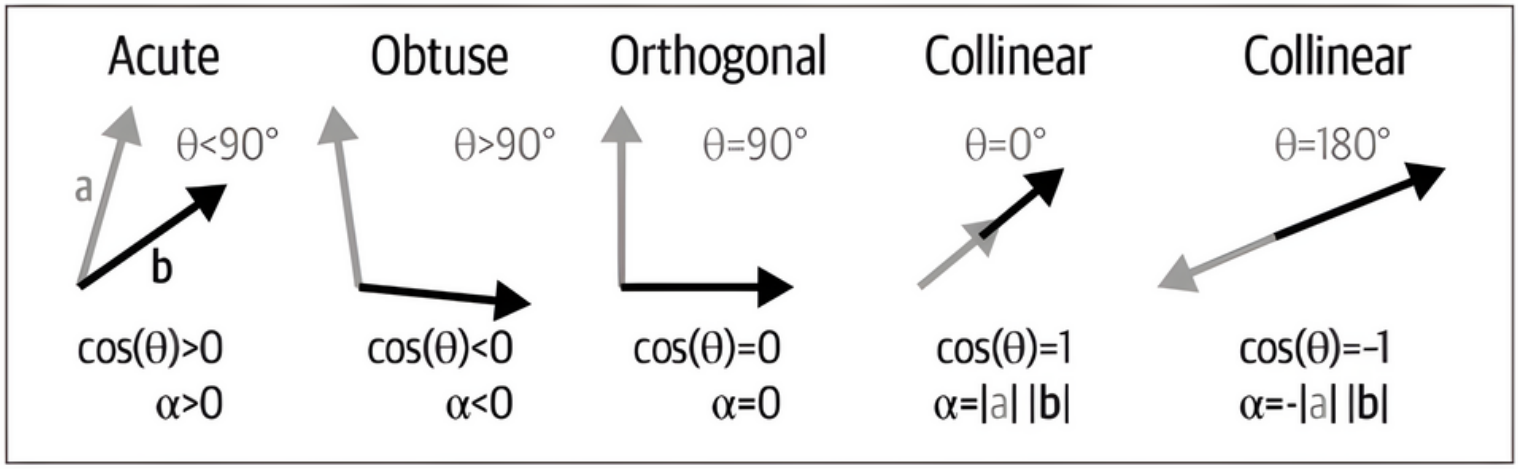


Figure 1-5. The sign of the dot product between two vectors reveals the geometric relationship between those vectors

In summary, the geometric definition of the dot product provides insight into its relationship with vector magnitudes and the angle between vectors. Understanding this definition enhances the geometric interpretation of the dot product and its significance in various mathematical and computational contexts.

NOTE:

Memorization in Mathematics:

- While understanding procedures and proofs is essential in mathematics, memorization also plays a crucial role.
- Linear algebra requires some degree of memorization, although it's not excessively heavy compared to other mathematical subjects.

Key Concept: Orthogonal Vectors:

- One crucial concept to memorize is that orthogonal vectors have a dot product of zero.
- Conversely, if the dot product of two vectors is zero, then they are orthogonal.
- Orthogonal vectors are those that meet at a 90° angle.

Equivalence of Statements:

- The note emphasizes the equivalence of statements regarding orthogonal vectors: being orthogonal, having a dot product of zero, and meeting at a 90° angle are all equivalent statements.
- It encourages repetition of this equivalence until it becomes firmly ingrained in one's memory.

In summary, memorizing the relationship between orthogonal vectors and the dot product is essential in understanding linear algebra. Recognizing the equivalence of statements regarding orthogonality helps solidify understanding and facilitates problem-solving in various mathematical contexts.

Other Vector Multiplications

Introduction to Dot Product:

- The dot product is a fundamental operation used to multiply vectors.
- It's considered the most important and commonly used method for vector multiplication.

Frequent Usage:

- We use the dot product extensively in various fields such as physics, engineering, computer science, and more.
- It helps us understand the relationship between vectors and is crucial in many mathematical and computational applications.

Existence of Other Methods:

- Despite its importance, it's essential to know that there are other ways to multiply vectors.
- These alternative methods serve different purposes and are useful in specific situations.

Exploring Alternatives:

- By learning about these other methods, we can deepen our understanding of vector algebra.
- We can expand our problem-solving skills by knowing when and how to use different multiplication techniques.

Conclusion:

- While the dot product reigns supreme in terms of importance and frequency of use, being aware of other vector multiplication methods enriches our mathematical knowledge and enhances our ability to tackle diverse problems.

In essence, while the dot product takes the spotlight, exploring the variety of ways to multiply vectors broadens our mathematical horizons and empowers us to excel in different areas of study and application.

Hadamard Multiplication

Introduction to Hadamard Multiplication:

- Hadamard multiplication is also known as element-wise multiplication.
- It involves multiplying each corresponding element in two vectors to produce a new vector.

Implementation in Python:

- In Python, the **asterisk (*)** operator is used for element-wise multiplication of vectors or matrices.
- For example, if we have vectors **a** and **b**, **a * b** performs Hadamard multiplication.

$$\begin{bmatrix}5\\4\\8\\2\end{bmatrix} \odot \begin{bmatrix}1\\0\\.5\\-1\end{bmatrix} = \begin{bmatrix}5\\0\\4\\-2\end{bmatrix}$$

```
a = np.array([5,4,8,2])

b = np.array([1,0,.5])

a*b  #python will give an error. see pink color information given below
```

Example:

- Consider vectors **[5, 4, 8, 2]** and **[1, 0, 0.5, -1]**.
- Their Hadamard product results in **[5, 0, 4, -2]**, where each element is the product of corresponding elements from the two vectors.

Handling Errors:

- When attempting to perform Hadamard multiplication in Python, you might encounter errors.
- These errors often arise due to differences in vector dimensions.

Learning from Errors:

- By fixing errors encountered during Hadamard multiplication, we learn about the importance of ensuring vectors have compatible dimensions.
- Hadamard multiplication requires vectors to have the same length.

Practical Application:

- Hadamard multiplication is useful for organizing multiple scalar operations efficiently.
- For instance, in business scenarios, it can be used to calculate revenue per item for different products or stores.

Difference from Dot Product:

- It's important to note that Hadamard multiplication differs from the dot product, which calculates the total revenue across all stores.

In summary, Hadamard multiplication offers a straightforward way to perform element-wise operations on vectors. Understanding its implementation in Python and its practical applications enhances our ability to analyze and manipulate data efficiently.

The error is that the two vectors have different dimensionalities, which shows that Hadamard multiplication is defined only for two vectors of equal dimensionality. You can fix the problem by removing one number from a or adding one number to b.

Outer Product

Introduction to Outer Product:

- The outer product is a method to create a matrix from a column vector and a row vector.
- It results in a matrix where each row is the scalar multiplication of the row vector by the corresponding element in the column vector.

Example Illustration:

- For vectors **[a, b, c] (column vector)** and **[d, e] (row vector)**, their outer product matrix is:

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \begin{bmatrix} d & e \end{bmatrix} = \begin{bmatrix} ad & ae \\ bd & be \\ cd & ce \end{bmatrix}$$

Using Letters in Linear Algebra:

- Just like using letters as variables in middle school algebra deepens understanding, in linear algebra, letters inside matrices serve as placeholders for numbers.

- These letters are treated as variables, allowing for more abstract and flexible manipulation of matrices.

Comparison with Dot Product:

- The outer product differs from the dot product:
  - It produces a matrix instead of a scalar.
  - The two vectors in an outer product can have different dimensionalities, unlike the dot product where they must have the same dimensionality.

Notation and Orientation:

- The outer product is denoted as **vwT**, assuming vectors are in column orientation.
- This notation differs from the dot product notation (**vTw**), and it will become clearer after learning about matrix multiplication.

Difference from Broadcasting:

- The outer product and broadcasting are distinct operations:
  - Broadcasting expands vectors in arithmetic operations like addition, multiplication, and division.
  - The outer product is a specific mathematical procedure for multiplying two vectors.

Computing Outer Product in NumPy:

- NumPy provides functions to compute the outer product:
  - **np.outer()** computes the outer product directly.
  - Alternatively, np.dot() can be used if the vectors are appropriately oriented (column and row, respectively).

Understanding the outer product expands our toolkit for manipulating vectors and matrices, offering versatility in mathematical operations and data transformations.

Cross and Triple Products

Introduction to Other Vector Multiplication Methods:

- Apart from the dot product and outer product, there exist other multiplication methods for vectors.
- Notable examples include the cross product and triple product.

Usage in Geometry and Physics:

- These methods, like the cross product and triple product, find significant applications in geometry and physics.
- They are particularly useful for solving problems related to spatial relationships, forces, and rotations.

Limited Relevance in Tech-Related Applications:

- While essential in certain fields, these methods are not frequently encountered in technology-related applications.
- Thus, they are not extensively covered in this book.

Passing Familiarity:

- Despite their limited discussion, it's beneficial to have a basic understanding of these methods and their names.
- This passing familiarity can help recognize their significance in specific contexts where they might be encountered.

In summary, while the cross product and triple product are important in geometry and physics, they are not extensively covered in this book due to their limited relevance in technology-related applications. However, being aware of their existence and basic functionality can be valuable for broader mathematical knowledge.

Orthogonal Vector Decomposition

Introduction to Prime Factorization:

- Prime factorization involves decomposing a number into its prime factors.
- It's akin to breaking down a complex mathematical object into simpler pieces.

- This process finds applications in numerical processing and cryptography.

Introduction to Vector Decomposition:

- Similar to prime factorization, vector decomposition involves breaking down a vector into simpler components.
- We'll explore a specific type called orthogonal vector decomposition.
- This decomposition splits a vector into two parts: one orthogonal to a reference vector and the other parallel to it.

Simple Examples:

Decomposition is like breaking a big problem into smaller, easier parts to solve. For example, think about the number 42.01. We can break it into two parts: 42 and 0.01. This splitting can help us deal with the number more easily, like if we want to ignore the small part or make it take up less space in our computer's memory. It's similar to how we break down 42 into the numbers 2, 3, and 7 when we're finding its prime factors. This idea of breaking things down into simpler pieces is useful in lots of math and science areas, like when we're working with data or keeping information secure.

Visualizing the Decomposition:

- A visual aid (**Figure 1-6**) illustrates the goal of the decomposition.
- Consider two vectors, **a** and **b**, where the objective is to find the point on vector a closest to the head of vector **b**.
- This can be framed as an optimization problem: minimize the distance by projecting vector **b** onto vector **a**.
- The projected point on vector a will be a scaled version of a, denoted as **βa**, where **β** is a scalar.

By decomposing vectors in this manner, we lay the groundwork for understanding advanced procedures like the Gram-Schmidt procedure and QR decomposition, crucial in statistical analysis and problem-solving.

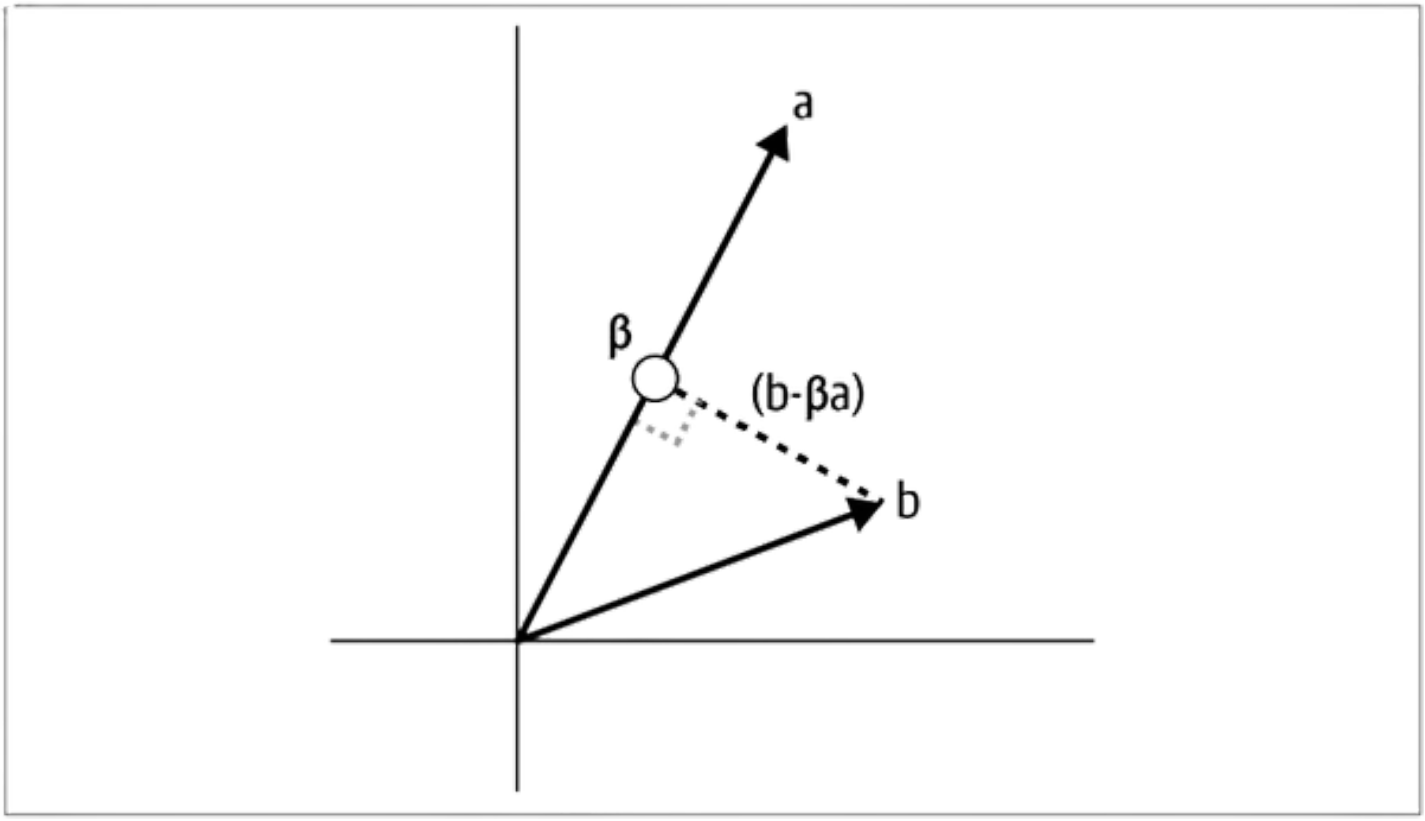


Figure 1-6. To project a point at the head of b onto a vector a with minimum distance, we need a formula to compute β such that the length of the projection vector b – βa is minimized

Using Vector Subtraction:

- Vector subtraction helps define the line from vector b to the point βa.
- Although a new vector (e.g., vector **c**) could represent this line, subtraction aids in finding the solution.

Key Insight and Geometric Interpretation:

- The closest point on vector **a** to the head of vector **b** is where a line from **b** meets a at a right angle.
- Imagining a triangle formed by the origin, the head of **b**, and **βa** helps visualize this concept.
- The length of the line from **b** to **βa** increases as the angle between them deviates from **90 degrees.**

Deriving the Formula:

- Vector **b minus β** times vector a is orthogonal to vector **a**, implying they are perpendicular.
- This orthogonality leads to the realization that the dot product between them must be **zero**.

- This insight is translated into the equation
- **aTb - β(aTa) = 0**
- Solving for **β** yields **β = aTb / aTa**, a formula for projecting a point onto a line with minimum distance, known as **orthogonal projection.**

Equation 1-12. Solving the orthogonal projection problem

$$\mathbf{a}^{\mathrm{T}}\mathbf{b} - \beta\mathbf{a}^{\mathrm{T}}\mathbf{a} = 0$$

$$\beta\mathbf{a}^{\mathrm{T}}\mathbf{a} = \mathbf{a}^{\mathrm{T}}\mathbf{b}$$

$$\beta = \frac{\mathbf{a}^{\mathrm{T}}\mathbf{b}}{\mathbf{a}^{\mathrm{T}}\mathbf{a}}$$

Relation to Orthogonal Vector Decomposition:

- The minimum distance projection serves as the foundation for orthogonal vector decomposition.
- In this context, the target vector **(t)** is decomposed into two components: perpendicular to the reference vector **(t ⊥ r)** and parallel to it **(t ∥ r).**
- This decomposition is illustrated in Figure 1-7.

Terminology Clarification:

- The target vector is denoted as **t**, and the reference vector as **r**.
- The components formed from the target vector are named the perpendicular component **(t ⊥ r)** and the parallel component **(t ∥ r)**.

This explanation sets the stage for understanding how to decompose vectors into orthogonal and parallel components, a fundamental concept with applications in various fields like statistics and machine learning.

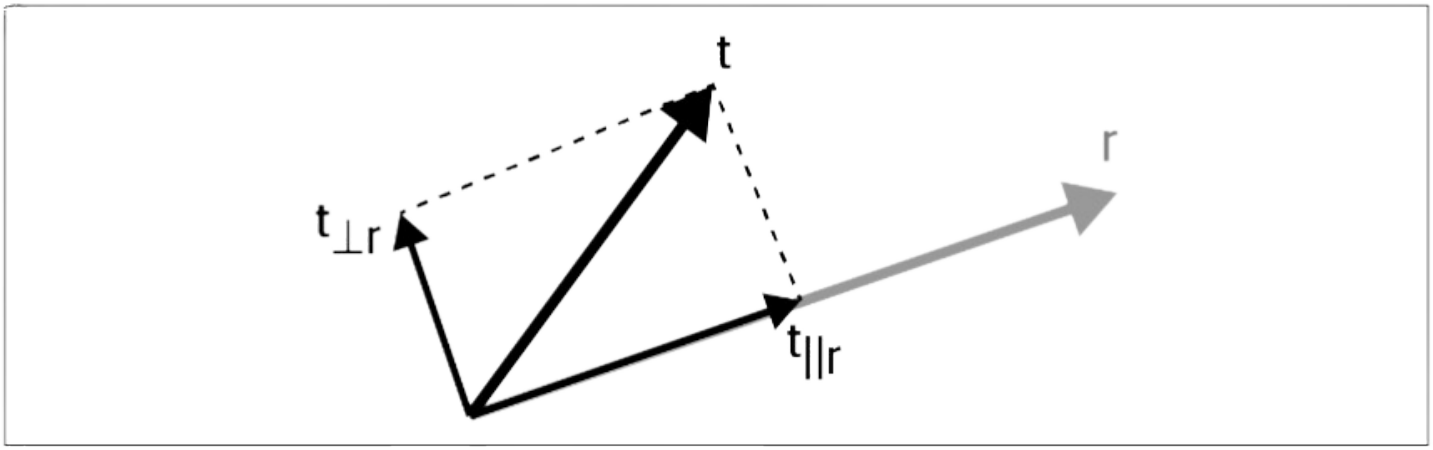


Figure 1-7. Illustration of orthogonal vector decomposition: decompose vector t into the sum of two other vectors that are orthogonal and parallel to vector r

Finding the Parallel Component:

- Any scaled version of vector ***r*** is parallel to ***r***.
- Therefore, we find ***t*||*r*** by applying the orthogonal projection formula (**Equation 1-13**) to ***r***.
- The formula for ***t*||*r*** is:

Equation 1-13. Computing the parallel component of t with respect to r

$$\mathbf{t}_{\parallel \mathbf{r}} = \mathbf{r}\,\frac{\mathbf{t}^{\mathrm{T}}\mathbf{r}}{\mathbf{r}^{\mathrm{T}}\mathbf{r}}$$

- Note the difference from Equation 1-12: here, we compute the scaled vector ***βr***.

Computing the Perpendicular Component:

- We know that the sum of the two vector components must equal the original target vector.
- Therefore, we can find ***t*⊥*r*** by subtracting the parallel component (***t*||*r***) from the original vector ***t***:

$$\mathbf{t}_{\parallel \mathbf{r}} = \mathbf{r}\,\frac{\mathbf{t}^{\mathrm{T}}\mathbf{r}}{\mathbf{r}^{\mathrm{T}}\mathbf{r}}$$

This process allows us to decompose the target vector ***t*** into two components: one parallel to the reference vector ***r*** and one perpendicular to it.

Absolutely! The proof that the perpendicular component is orthogonal to the reference vector involves showing that their **dot product equals zero**.

Here's the mathematical representation of the proof:

$$(\mathbf{t}_{\perp \mathbf{r}})^{\mathrm{T}}\mathbf{r} = 0$$

$$\left(\mathbf{t} - \mathbf{r}\frac{\mathbf{t}^{\mathrm{T}}\mathbf{r}}{\mathbf{r}^{\mathrm{T}}\mathbf{r}}\right)^{\mathrm{T}}\mathbf{r} = 0$$

This proof establishes that the perpendicular component $\boldsymbol{t}_{\perp}$ is indeed orthogonal to the reference vector $\boldsymbol{r}$. While the algebraic steps are straightforward, they can be tedious. However, you can gain intuition and verify this relationship using Python code in the exercises.

Orthogonal vector decomposition is a powerful concept, allowing us to break down vectors into components that satisfy specific constraints. The particular decomposition we discussed here, orthogonal and parallel to a reference vector, provides valuable insights into various mathematical and analytical problems.

Summary:

In summary, linear algebra provides a powerful framework for understanding and manipulating vectors, which are ordered lists of numbers representing points in space. Here are the key takeaways from this chapter:

Understanding Vectors: Vectors are represented either as columns or rows of numbers, and their dimensionality refers to the number of elements they contain. Geometrically, vectors can be visualized as lines in a space with axes corresponding to their dimensionality.

Arithmetic Operations: Basic arithmetic operations like addition, subtraction, and Hadamard multiplication (element-wise multiplication) on vectors are straightforward and operate element-

wise.

Dot Product: The dot product is a fundamental operation that yields a single number representing the relationship between two vectors of the same dimensionality. It's computed by multiplying corresponding elements of the vectors and then summing the products. The dot product is zero for orthogonal vectors, indicating that they meet at a right angle geometrically.

Orthogonal Vector Decomposition: This involves breaking down a vector into two components: one parallel and one orthogonal to a reference vector. The formula for this decomposition can be derived from geometric principles, and it's essential to remember the concept of "mapping over magnitude" that this formula expresses.

By mastering these concepts, you lay the foundation for more advanced topics in linear algebra, which are crucial for various fields such as data science, machine learning, and image processing.

Exercises with code:

I hope you don’t see these exercises as tedious work that you need to do. Instead, these exercises are opportunities to polish your math and coding skills, and to make sure that you really understand the material in this chapter.

I also want you to see these exercises as a springboard to continue exploring linear algebra using Python. Change the code to use different numbers, different dimensionalities, different orientations, etc. Write your own code to test other concepts mentioned in the chapter. Most importantly: have fun and embrace the learning experience.

As a reminder: the solutions to all the exercises can be viewed or downloaded from

mmiimran/**linear-algebra-for-data-science**

1 Contributor 0 Issues 0 Stars 0 Forks

Exercise 1-1:

The online code repository is "missing" code to create Figure 1-2. (It's not really missing—I moved it into the solution to this exercise.) So, your goal here is to write your own code to produce Figure 1-2.

Exercise 1-2:

Write an algorithm that computes the norm of a vector by translating Equation 1-7 into code. Confirm, using random vectors with different dimensionalities and orientations, that you get the same result as np.linalg.norm(). This exercise is designed to give you more experience with indexing NumPy arrays and translating formulas into code; in practice, it's often easier to use np.linalg.norm().

Exercise 1-3:

Create a Python function that will take a vector as input and output a unit vector in the same direction. What happens when you input the zeros vector?

Exercise 1-4:

You know how to create unit vectors; what if you want to create a vector of any arbitrary magnitude? Write a Python function that will take a vector and a desired magnitude as inputs and will return a vector in the same direction but with a magnitude corresponding to the second input.

Exercise 1-5:

Write a for loop to transpose a row vector into a column vector without using a built-in function or method such as np.transpose() or v.T. This exercise will help you create and index orientation-endowed vectors.

Exercise 1-6:

Here is an interesting fact: you can compute the squared norm of a vector as the dot product of that vector with itself. Look back to Equation 1-8 to convince yourself of this equivalence. Then confirm it using Python.

Exercise 1-7:

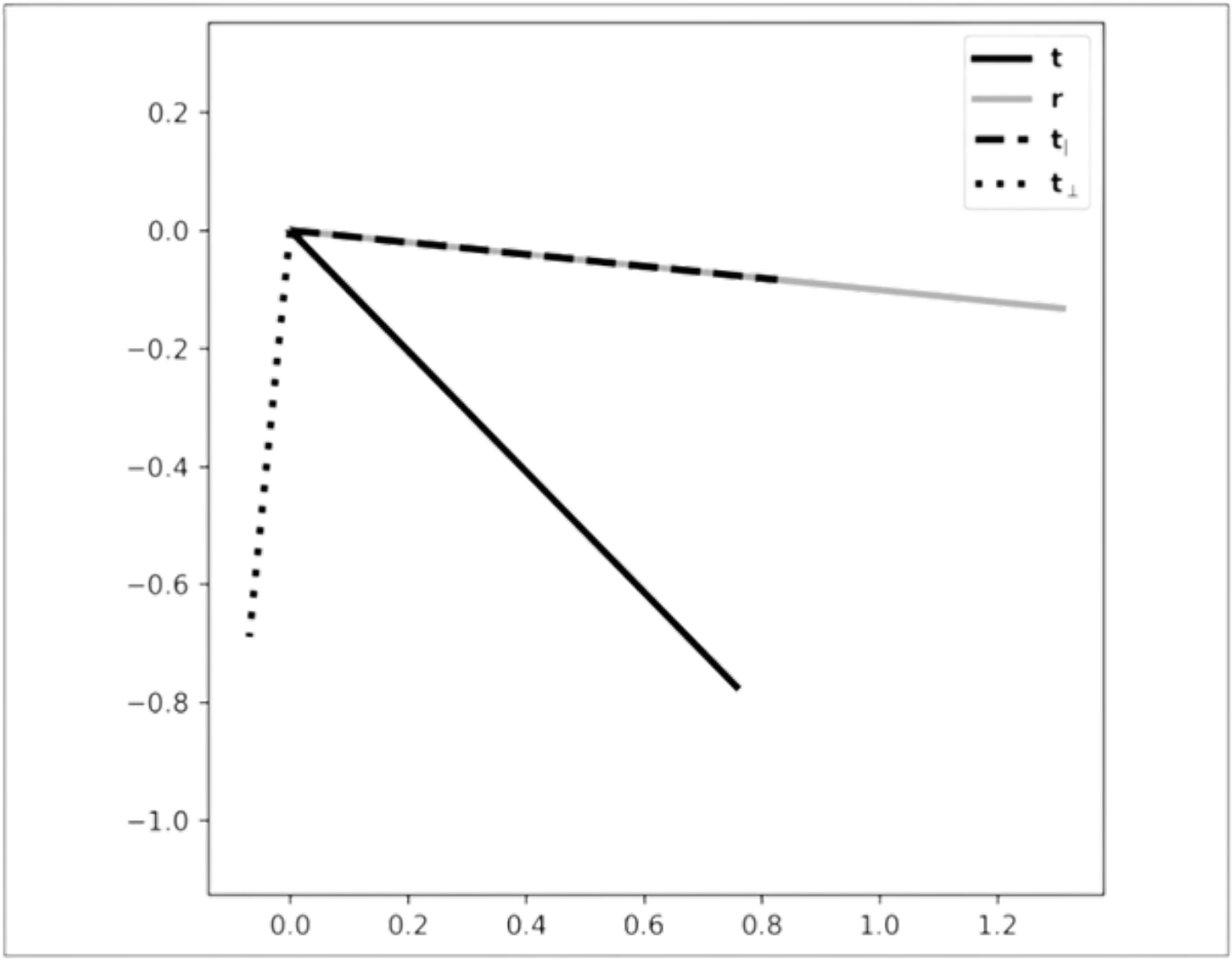
Write code to demonstrate that the dot product is commutative. Commutative means that $a \times b = b \times a$, which, for the vector dot product, means that $a^Tb = b^Ta$. After demonstrating this in code, use equation Equation 1-9 to understand why the dot product is commutative.

Exercise 1-8:

Write code to produce Figure 1-6. (Note that your solution doesn't need to look exactly like the figure, as long as the key elements are present.)

Exercise 1-9:

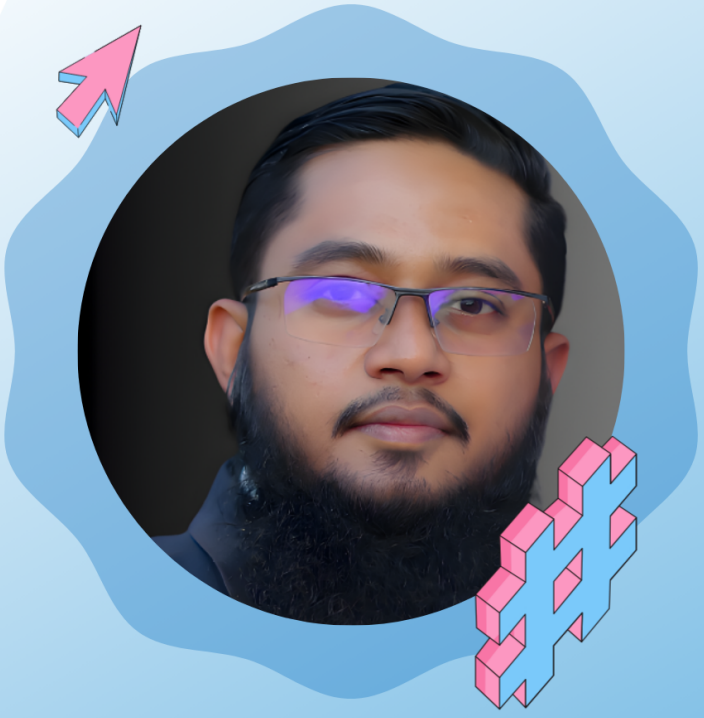
Implement orthogonal vector decomposition. Start with two random-number vectors t and r, and reproduce Figure 1-8 (note that your plot will look somewhat different due to random numbers). Next, confirm that the two components sum to t and that $t \perp r$ and $t \parallel r$ are orthogonal.



Exercise 9 (Figure 1-8)

Exercise 2-10:

An important skill in coding is finding bugs. Let's say there is a bug in your code such that the denominator in the projection scalar of Equation 1-13 is tTt instead of rTr (an easy mistake to make, speaking from personal experience while writing this chapter!). Implement this bug to check whether it really deviates from the accurate code. What can you do to check whether the result is correct or incorrect? (In coding, confirming code with known results is called sanity-checking.)



Thanks from:

mmislam IMRAN

www.go2imran.com

mmiimran
go2imran6@gmail.com
@code_2_learn6666
www.go2imran.com