CHAPTER 4

Linear Algebra

Changing gears a little bit, let's venture away from probability and statistics and into linear algebra. Sometimes people confuse linear algebra with basic algebra, thinking maybe it has to do with plotting lines using the algebraic function $y = mx + b$. This is why linear algebra probably should have been called "vector algebra" or "matrix algebra" because it is much more abstract. Linear systems play a role but in a much more metaphysical way.

يهتم

So, what exactly is linear algebra? Well, *linear algebra* concerns itself with linear systems but represents them through vector spaces and matrices. If you do not know what a vector or a matrix is, do not worry! We will define and explore them in depth. Linear algebra is hugely fundamental to many applied areas of math, statistics, operations research, data science, and machine learning. When you work with data in any of these areas, you are using linear algebra and perhaps you may not even know it.

You can get away with not learning linear algebra for a while, using machine learning and statistics libraries that do it all for you. But if you are going to get intuition behind these black boxes and be more effective at working with data, understanding the fundamentals of linear algebra is inevitable. Linear algebra is an enormous topic that can fill thick textbooks, so of course we cannot gain total mastery in just one chapter of this book. However, we can learn enough to be more comfortable with it and navigate the data science domain effectively. There will also be opportunities to apply it in the remaining chapters in this book, including Chapters 5 and 7.

What Is a Vector?

Simply put, a *vector* is an arrow in space with a specific direction and length, often representing a piece of data. It is the central building block of linear algebra, including matrices and linear transformations. In its fundamental form, it has no concept of location so always imagine its tail starts at the origin of a Cartesian plane (0,0).

Figure 4-1 shows a vector $\vec{v}$ that moves three steps in the horizontal direction and two steps in the vertical direction.

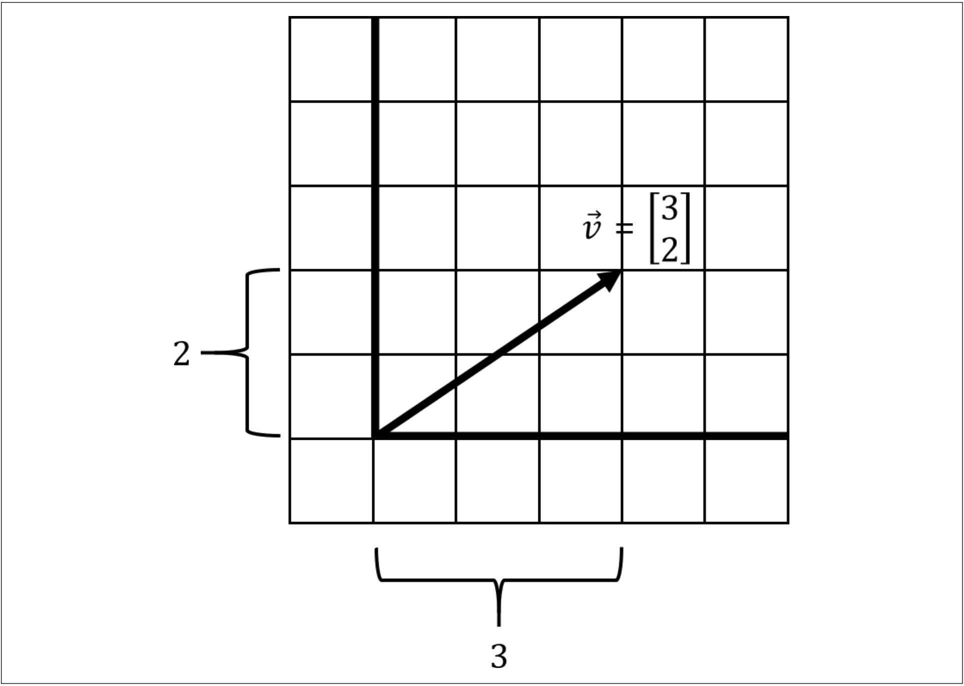


Figure 4-1. A simple vector

أؤكد

To emphasize again, the purpose of the vector is to visually represent a piece of data. If you have a data record for the square footage of a house 18,000 square feet and its valuation $260,000, we could express that as a vector [18000, 2600000], stepping 18,000 steps in the horizontal direction and 260,000 steps in the vertical direction.

We declare a vector mathematically like this:

$$\vec{v} = \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\vec{v} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

We can declare a vector using a simple Python collection, like a Python list as shown in Example 4-1.

Example 4-1. Declaring a vector in Python using a list

```
v = [3, 2]
print(v)
```

حسابات

However, when we start doing mathematical computations with vectors, especially when doing tasks like machine learning, we should probably use the NumPy library as it is more efficient than plain Python. You can also use SymPy to perform linear algebra operations, and we will use it occasionally in this chapter when decimals become inconvenient. However, NumPy is what you will likely use in practice so that is what we will mainly stick to.

To declare a vector, you can use NumPy's `array()` function and then can pass a collection of numbers to it as shown in Example 4-2.

Example 4-2. Declaring a vector in Python using NumPy

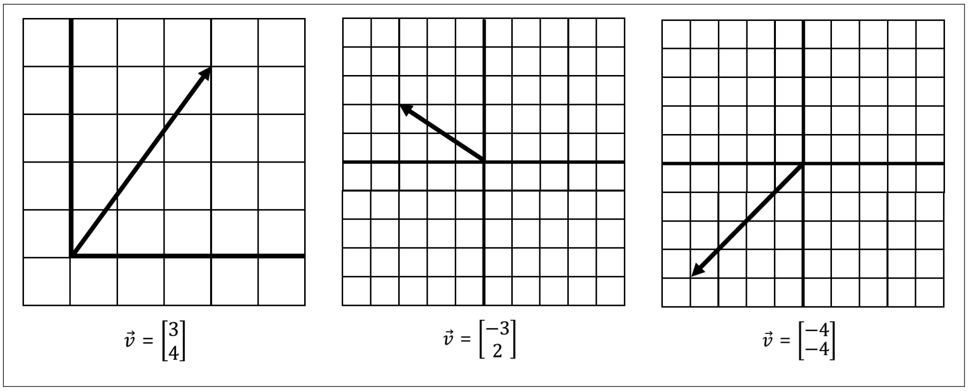
```
import numpy as np
v = np.array([3, 2])
print(v)
```

Python Is Slow, Its Numerical Libraries Are Not

Python is a computationally slow language platform, as it does not compile to lower-level machine code and bytecode like Java, C#, C, etc. It is dynamically interpreted at runtime. However, Python's numeric and scientific libraries are not slow. Libraries like NumPy are typically written in low-level languages like C and C++, hence why they are computationally efficient. Python really acts as "glue code" integrating these libraries for your tasks.

A vector has countless practical applications. In physics, a vector is often thought of as a direction and magnitude. In math, it is a direction and scale on an XY plane, kind of like a movement. In computer science, it is an array of numbers storing data. The computer science context is the one we will become the most familiar with as data science professionals. However, it is important we never forget the visual aspect so we do not think of vectors as esoteric grids of numbers. Without a visual understanding, it is almost impossible to grasp many fundamental linear algebra concepts like linear dependence and determinants.

Here are some more examples of vectors. In Figure 4-2 note that some of these vectors have negative directions on the X and Y scales. Vectors with negative directions will have an impact when we combine them later, essentially subtracting rather than adding them together.



$\vec{v} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$
$\vec{v} = \begin{bmatrix} -3 \\ 2 \end{bmatrix}$
$\vec{v} = \begin{bmatrix} -4 \\ -4 \end{bmatrix}$

Figure 4-2. A sampling of different vectors

Why Are Vectors Useful?

A struggle many people have with vectors (and linear algebra in general) is understanding why they are useful. They are a highly abstract concept but they have many tangible applications. Computer graphics are easier to work with when you have a grasp of vectors and linear transformations (the fantastic Manim visualization library defines animations and transformations with vectors). When doing statistics and machine learning work, data is often imported and turned into numerical vectors so we can work with it. Solvers like the one in Excel or Python PuLP use linear programming, which uses vectors to maximize a solution while meeting those constraints. Even video games and flight simulators use vectors and linear algebra to model not just graphics but also physics. I think what makes vectors so hard to grasp is not that their application is unobvious, but rather these applications are so diverse it is hard to see the generalization.

Note also vectors can exist on more than two dimensions. Next we declare a three-dimensional vector along axes x, y, and z:

$$\vec{v} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \\ 2 \end{bmatrix}$$

To create this vector, we are stepping four steps in the x direction, one in the y direction, and two in the z direction. Here it is visualized in Figure 4-3. Note that we no longer are showing a vector on a two-dimensional grid but rather a three-dimensional space with three axes: x, y, and z.

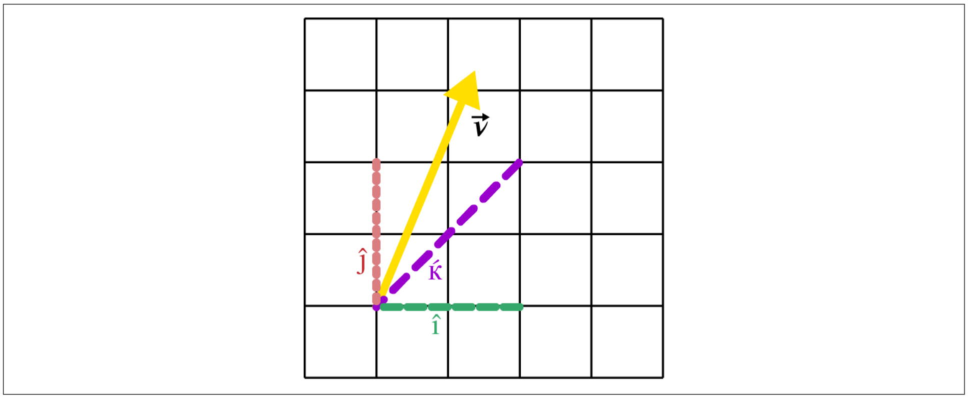


Figure 4-3. A three-dimensional vector

Naturally, we can express this three-dimensional vector in Python using three numeric values, as declared in Example 4-3.

Example 4-3. Declaring a three-dimensional vector in Python using NumPy

```
import numpy as np
v = np.array([4, 1, 2])
print(v)
```

Like many mathematical models, visualizing more than three dimensions is challenging and something we will not expend energy doing in this book. But numerically, it is still straightforward. Example 4-4 shows how we declare a five-dimensional vector mathematically in Python.

$$\vec{v} = \begin{bmatrix} 6 \\ 1 \\ 5 \\ 8 \\ 3 \end{bmatrix}$$

Example 4-4. Declaring a five-dimensional vector in Python using NumPy

```
import numpy as np
v = np.array([6, 1, 5, 8, 3])
print(v)
```

إضافة دمجها

Adding and Combining Vectors

On their own, vectors are not terribly interesting. You express a direction and size, kind of like a movement in space. But when you start combining vectors, known as *vector addition*, it starts to get interesting. We effectively combine the movements of two vectors into a single vector.

Say we have two vectors $\vec{v}$ and $\vec{w}$ as shown in Figure 4-4. How do we add these two vectors together?

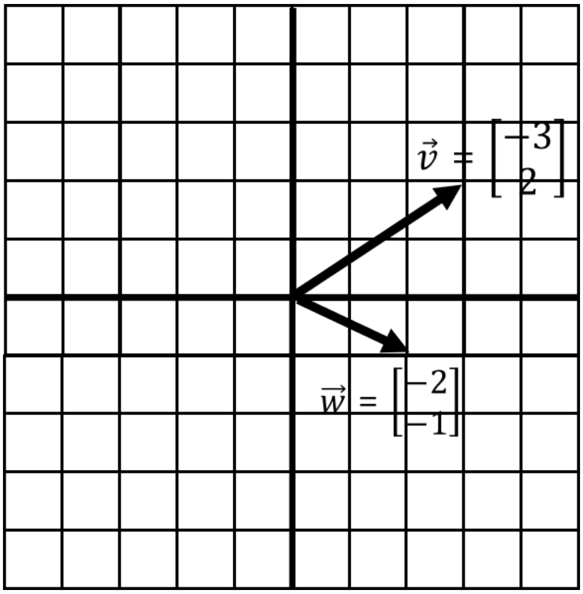

$\vec{v} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$
$\vec{w} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$

Figure 4-4. Adding two vectors together

We will get to why adding vectors is useful in a moment. But if we wanted to combine these two vectors, including their direction and scale, what would that look like? Numerically, this is straightforward. You simply add the respective x-values and then the y-values into a new vector as shown in Example 4-5.

$$\vec{v} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

$$\vec{w} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

$$\vec{v} + \vec{w} = \begin{bmatrix} 3+2 \\ 2+-1 \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \end{bmatrix}$$

Example 4-5. Adding two vectors in Python using NumPy

```
from numpy import array

v = array([3,2])
w = array([2,-1])

# sum the vectors
v_plus_w = v + w

# display summed vector
print(v_plus_w) # [5, 1]
```

But what does this mean visually? To visually add these two vectors together, connect one vector after the other and walk to the tip of the last vector (Figure 4-5). The point you end at is a new vector, the result of summing the two vectors.

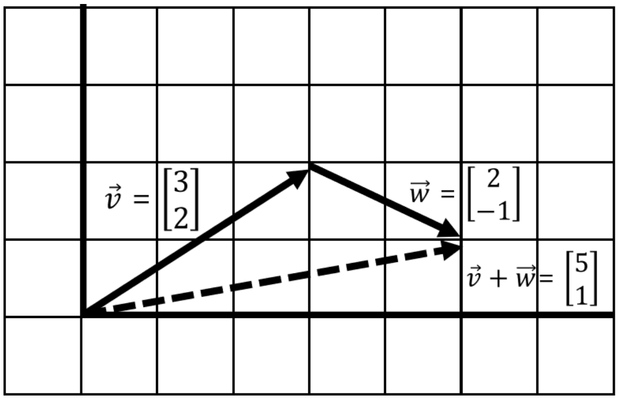
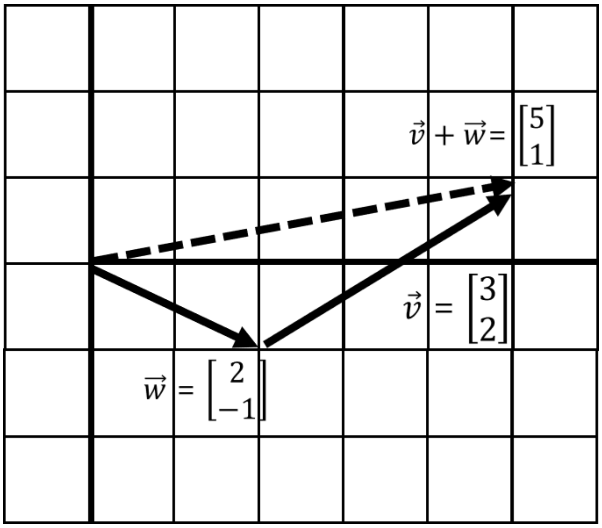

$\vec{v} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$
$\vec{w} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$
$\vec{v} + \vec{w} = \begin{bmatrix} 5 \\ 1 \end{bmatrix}$

Figure 4-5. Adding two vectors into a new vector

As seen in Figure 4-5, when we walk to the end of the last vector $\vec{w}$ we end up with a new vector [5, 1]. This new vector is the result of summing $\vec{v}$ and $\vec{w}$. In practice, this can be simply adding data together. If we were totaling housing values and their square footage in an area, we would be adding several vectors into a single vector in this manner.

* Note that it does not matter whether we add $\vec{v}$ before $\vec{w}$ or vice versa, which means it is *commutative* and order of operation does not matter. If we walk $\vec{w}$ before $\vec{v}$ we end up with the same resulting vector [5, 1] as visualized in Figure 4-6.

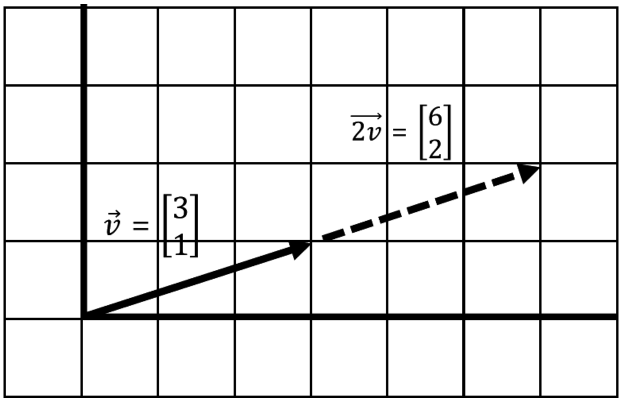


$\vec{v} + \vec{w} = \begin{bmatrix} 5 \\ 1 \end{bmatrix}$
$\vec{v} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$
$\vec{w} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$

Figure 4-6. Adding vectors is commutative

Scaling Vectors

Scaling is growing or shrinking a vector's length. You can grow/shrink a vector by multiplying or scaling it with a single value, known as a *scalar*. Figure 4-7 is vector $\vec{v}$ being scaled by a factor of 2, which doubles it.



$\overrightarrow{2v} = \begin{bmatrix} 6 \\ 2 \end{bmatrix}$
$\vec{v} = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$

Figure 4-7. Scaling a vector

Mathematically, you multiply each element of the vector by the scalar value:

$$\vec{v} = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

$$2\vec{v} = 2\begin{bmatrix} 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \times 2 \\ 1 \times 2 \end{bmatrix} = \begin{bmatrix} 6 \\ 2 \end{bmatrix}$$

Performing this scaling operation in Python is as easy as multiplying a vector by the scalar, as coded in Example 4-6.

Example 4-6. Scaling a number in Python using NumPy

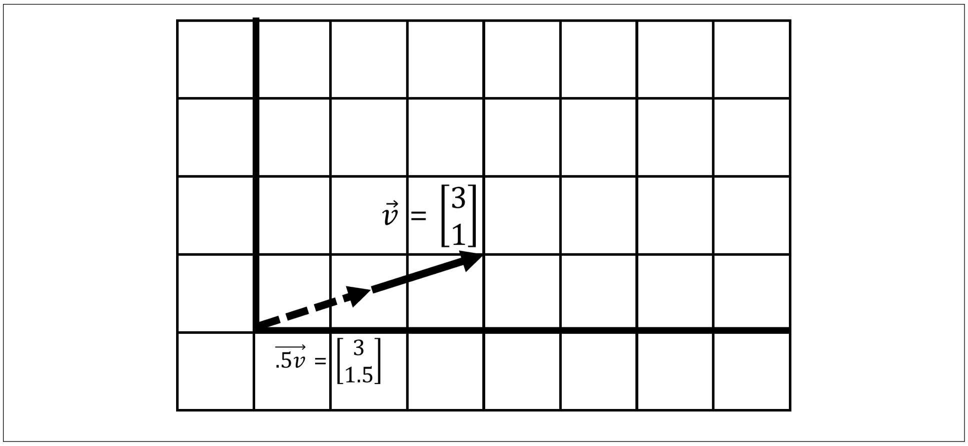
```
from numpy import array

v = array([3,1])

# scale the vector
scaled_v = 2.0 * v

# display scaled vector
print(scaled_v) # [6 2]
```

Here in Figure 4-8 $\vec{v}$ is being scaled down by factor of .5, which halves it.



$\vec{v} = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$
$\overrightarrow{.5v} = \begin{bmatrix} 3 \\ 1.5 \end{bmatrix}$

Figure 4-8. Scaling down a vector by half

Manipulating Data Is Manipulating Vectors

Every data operation can be thought of in terms of vectors, even simple averages.

Take scaling, for example. Let's say we were trying to get the average house value and average square footage for an entire neighborhood. We would add the vectors together to combine their value and square footage respectively, giving us one giant vector containing both total value and total square footage. We then scale down the vector by dividing by the number of houses N, which really is multiplying by $1/N$. We now have a vector containing the average house value and average square footage.

An important detail to note here is that scaling a vector does not change its direction, only its magnitude. But there is one slight exception to this rule as visualized in Figure 4-9. When you multiply a vector by a negative number, it flips the direction of the vector as shown in the image.

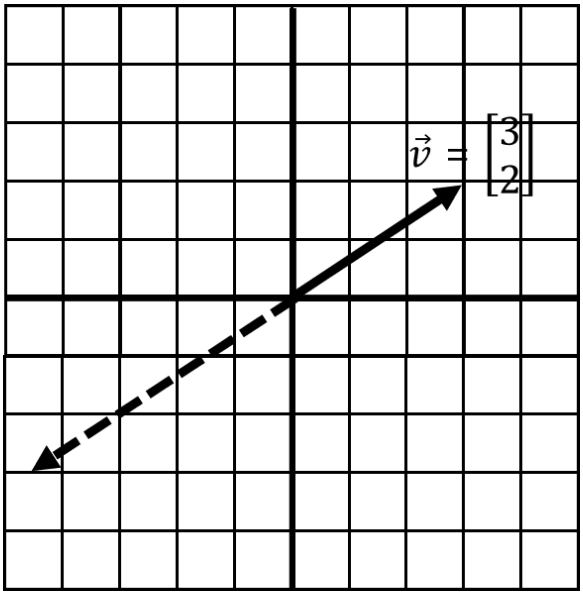


Figure 4-9. A negative scalar flips the vector direction

When you think about it, though, scaling by a negative number has not really changed direction in that it still exists on the same line. This segues to a key concept called linear dependence.

Span and Linear Dependence

These two operations, adding two vectors and scaling them, brings about a simple but powerful idea. With these two operations, we can combine two vectors and scale them to create any resulting vector we want. Figure 4-10 shows six examples of taking two vectors $\vec{v}$ and $\vec{w}$, and scaling and combining. These vectors $\vec{v}$ and $\vec{w}$, fixed in two different directions, can be scaled and added to create *any* new vector $\overrightarrow{v + w}$.

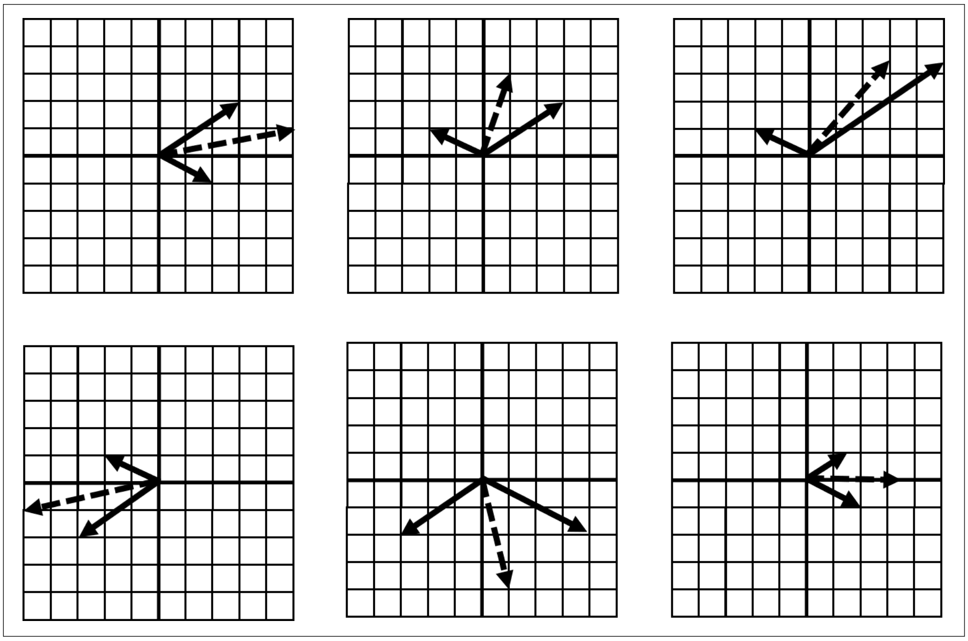


Figure 4-10. Scaling two added vectors allows us to create any new vector

Again, $\vec{v}$ and $\vec{w}$ are fixed in direction, except for flipping with negative scalars, but we can use scaling to freely create any vector composed of $\overrightarrow{v + w}$.

This whole space of possible vectors is called *span*, and in most cases our span can create unlimited vectors off those two vectors, simply by scaling and summing them. When we have two vectors in two different directions, they are *linearly independent* and have this unlimited span.

But in what case are we limited in the vectors we can create? Think about it and read on.

What happens when two vectors exist in the same direction, or exist on the same line? The combination of those vectors is also stuck on the same line, limiting our span to just that line. No matter how you scale it, the resulting sum vector is also stuck on that same line. This makes them *linearly dependent*, as shown in Figure 4-11.

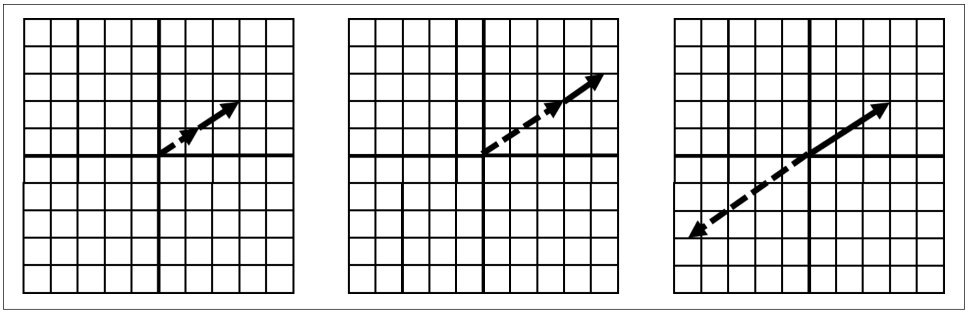


Figure 4-11. Linearly dependent vectors

The span here is stuck on the same line as the two vectors it is made out of. Because the two vectors exist on the same underlying line, we cannot flexibly create any new vector through scaling.

In three or more dimensions, when we have a linearly dependent set of vectors, we often get stuck on a plane in a smaller number of dimensions. Here is an example of being stuck on a two-dimensional plane even though we have three-dimensional vectors as declared in Figure 4-12.

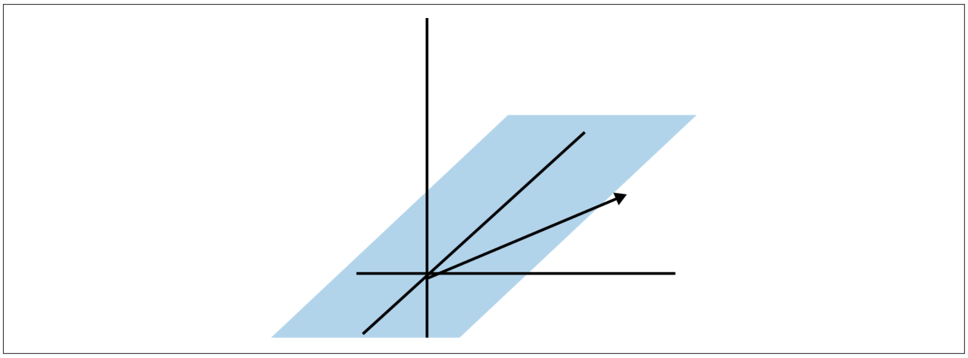


Figure 4-12. Linear dependence in three dimensions; note our span is limited to a flat plane

Later we will learn a simple tool called the determinant to check for linear dependence, but why do we care whether two vectors are linearly dependent or independent? A lot of problems become difficult or unsolvable when they are linearly dependent. For example, when we learn about systems of equations later in this chapter, a linearly dependent set of equations can cause variables to disappear and make the problem unsolvable. But if you have linear independence, that flexibility to create any vector you need from two or more vectors becomes invaluable to solve for a solution!