Linear Algebra with Python

O'Reilly Media

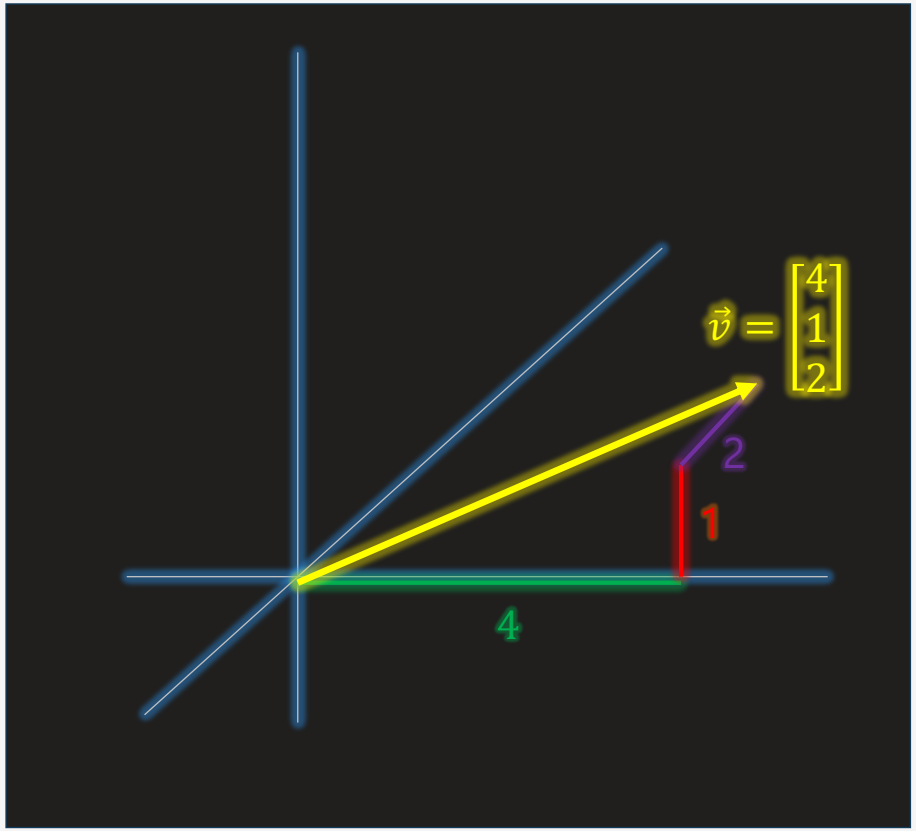
Thomas Nield

Overview

Agenda

Here is what we will do for the next 2 hours:

1. **Introduction** to Linear Algebra
2. Where linear algebra **is used**
3. **Vector** operations using **NumPy Arrays**
4. **Matrix** operations using **NumPy Arrays**
5. Solving **systems of equations**
6. Eigenvalues and Eigenvectors



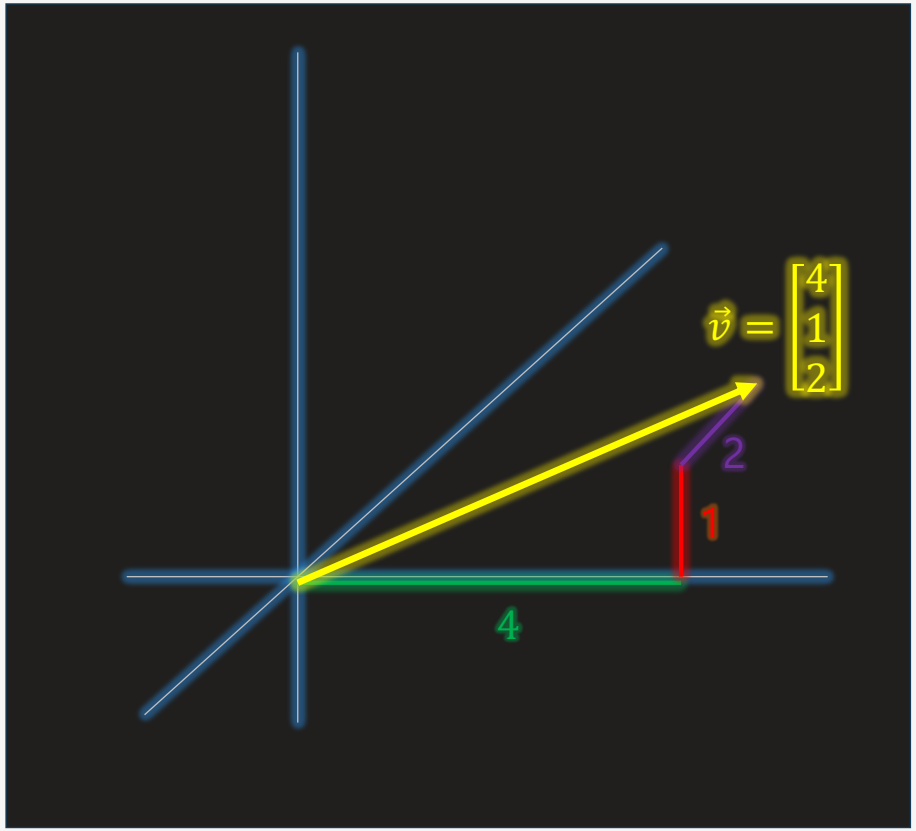
Why Learn Linear Algebra?

Linear algebra can be found in many areas of science, technology, engineering, and data science.

Computers can more efficiently model data as vectors and matrices and perform operations more effectively.

It is also the backbone of machine learning, data management, graphical modeling, and other computer science areas.

If you want to advance your knowledge in machine learning, statistical modeling, or other areas in computer science then linear algebra is a must!



I. Vectors, Combination, and Scaling

What Are Vectors?

Vectors are an arrow in space, with a specific direction and length.

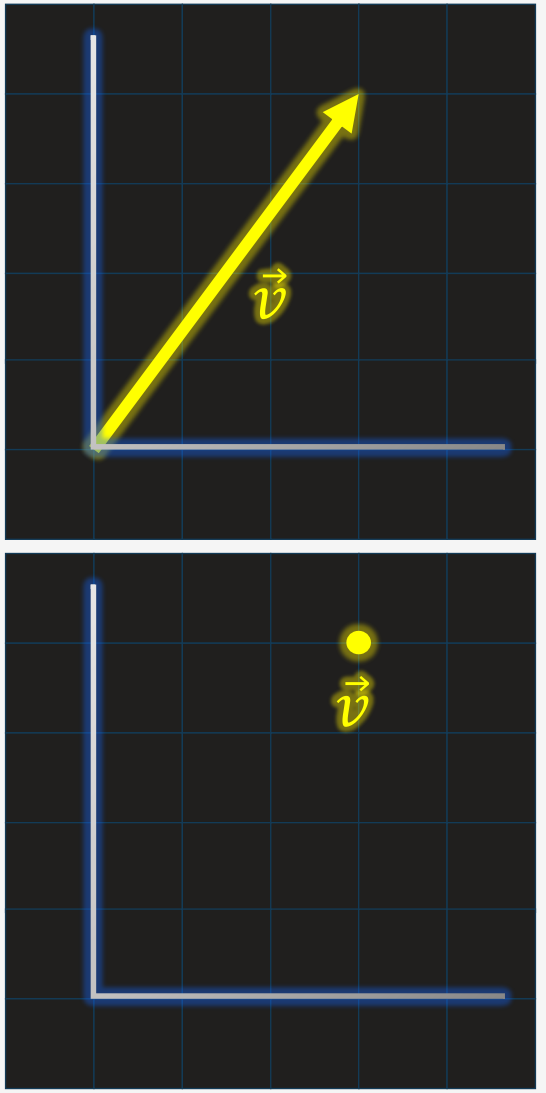
This can mean different things to different disciplines.

Physics – Direction and magnitude.

Computer science – An array of numbers storing data

Math – A direction and scale on an XY plane.

To truly understand linear algebra, you must first think of a vector as an arrow (top right) or a point in space (bottom right).



Vectors as Numbers

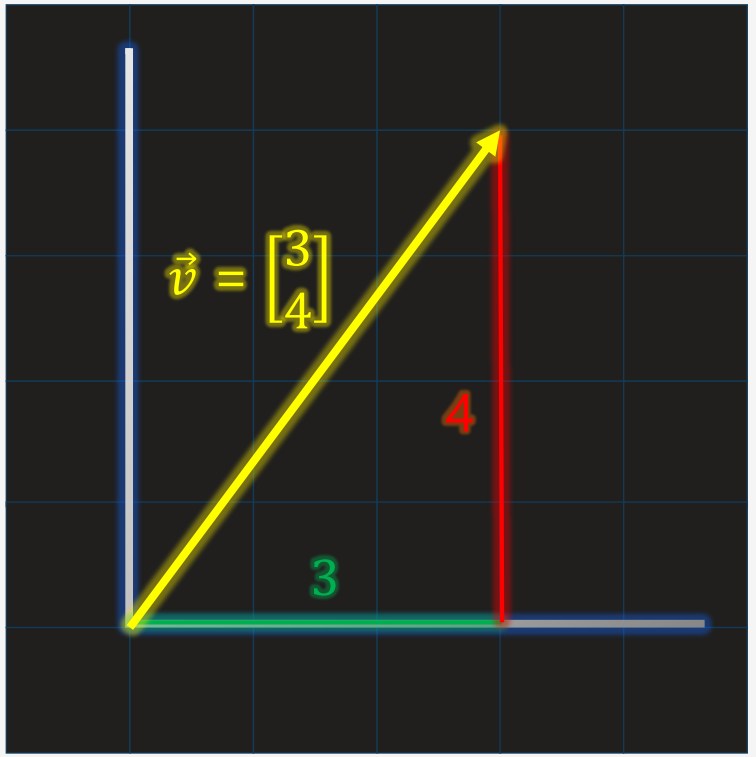
We can think of a two-dimensional vector as a pair of numbers.

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

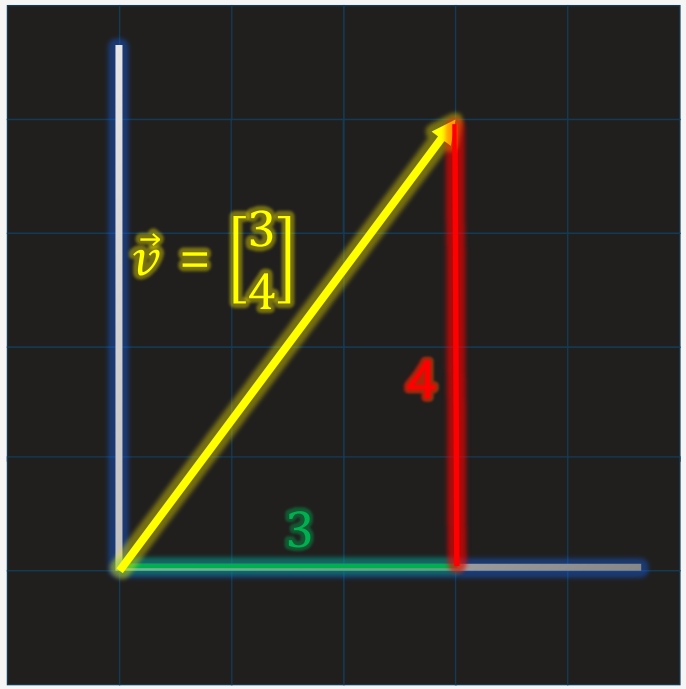
We specify a vector with the name $\vec{v}$, and from the origin (0,0) we make 3 steps along the x-axis, and 4 steps up the y-axis.

Think of vectors as more than a data point, it's a *movement* hence its representation as an arrow.

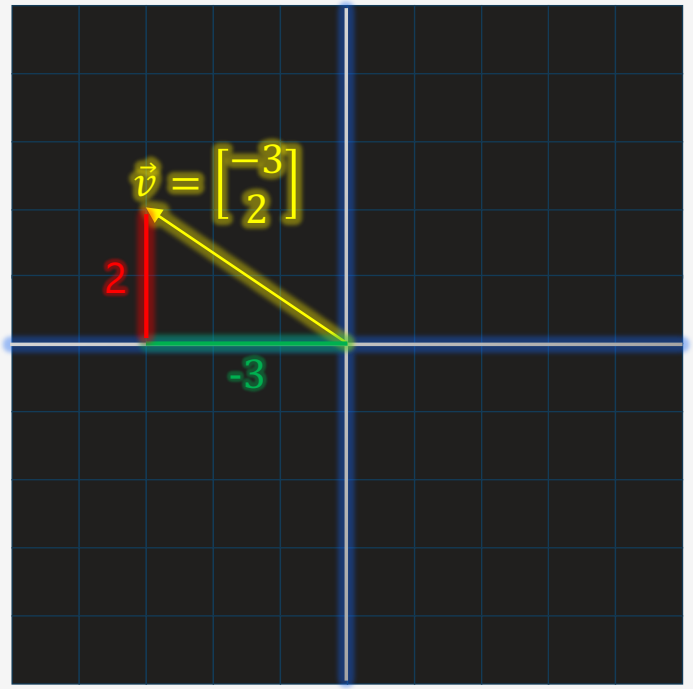
Vectors must start at the origin (0,0) and cannot arbitrarily start at any point in space.



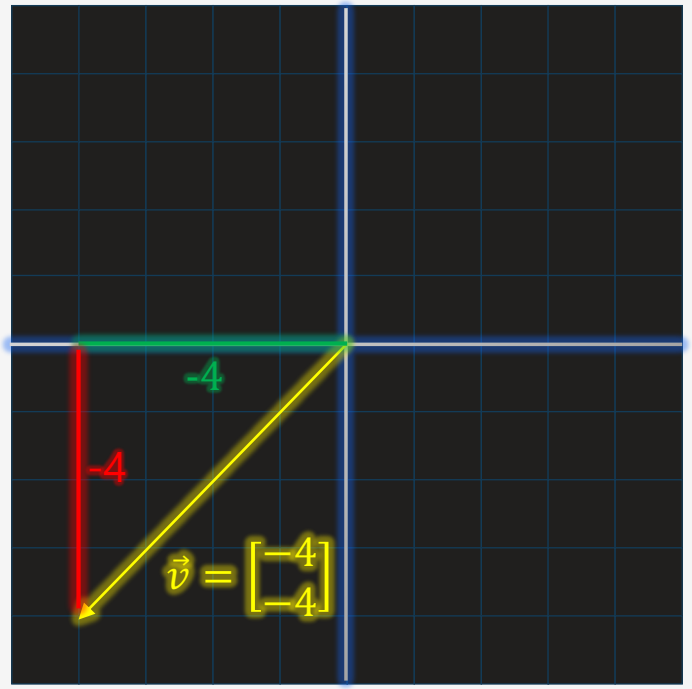
Some Vector Examples



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



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



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

Quick Quiz

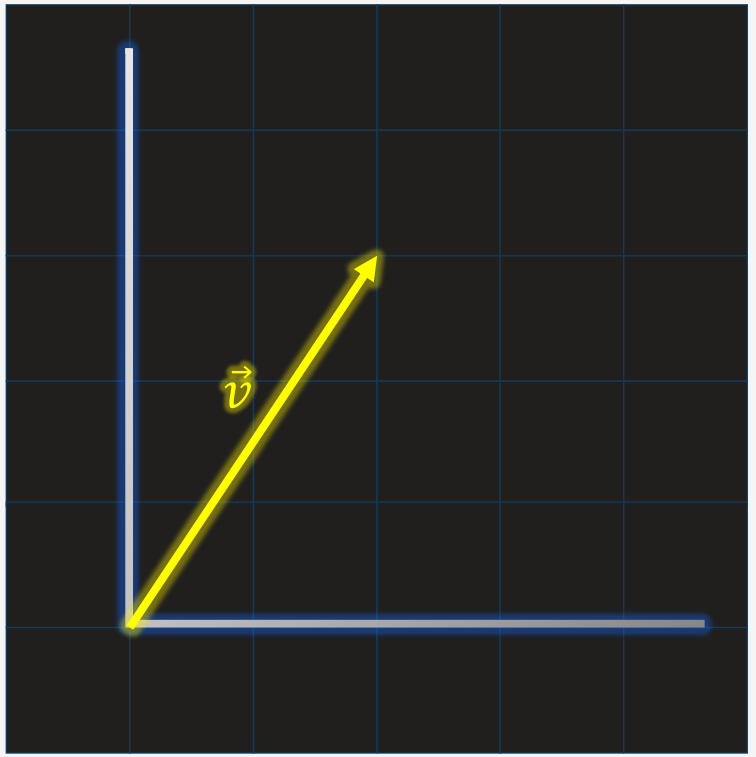
What is the numerical value for the vector $\vec{v}$?

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

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

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

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



Quick Quiz

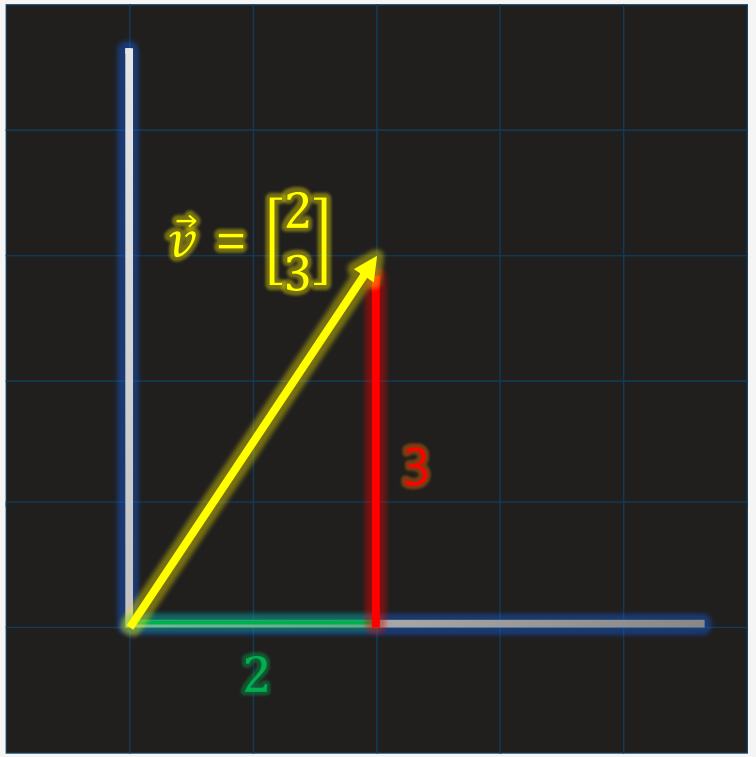
What is the numerical value for the vector $\vec{v}$?

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

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

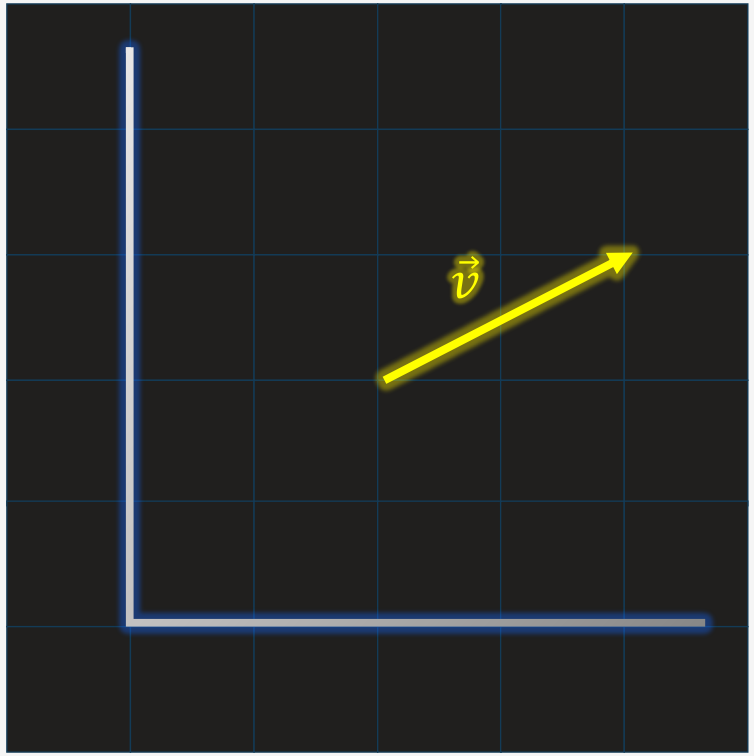
C) $\vec{v} = \begin{bmatrix} -2 \\ 3 \end{bmatrix}$

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



Quick Quiz

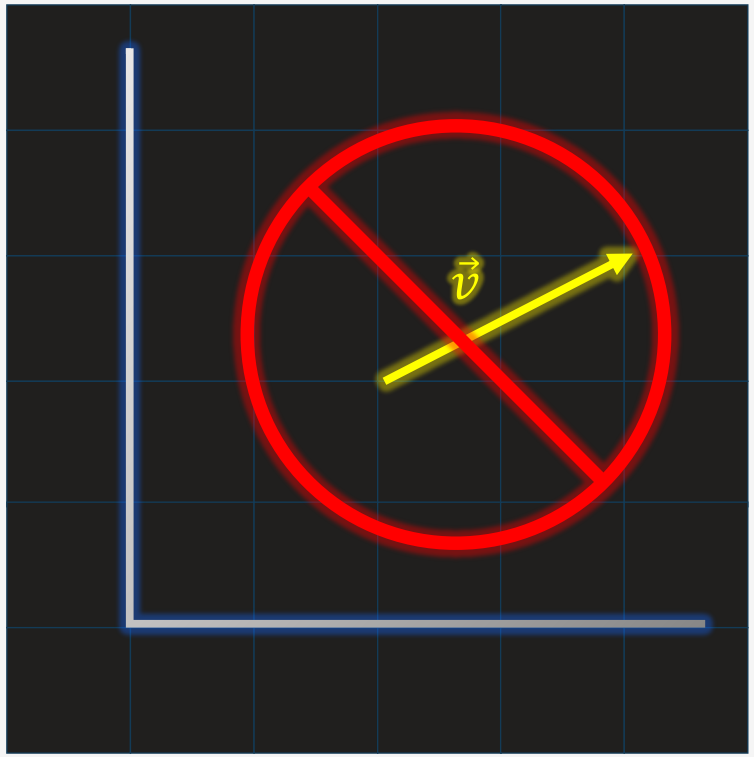
Is this a valid vector on the right?



Quick Quiz

Is this a valid vector on the right?

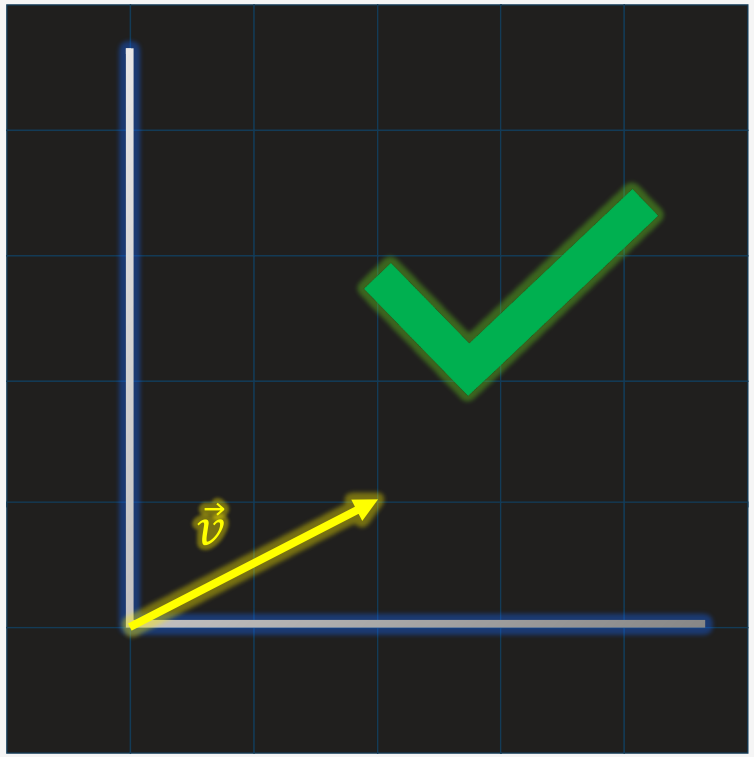
No! A vector must start at the origin (0,0) and cannot start anywhere else.



Quick Quiz

Is this a valid vector on the right?

No! A vector must start at the origin (0,0) and cannot start anywhere else.

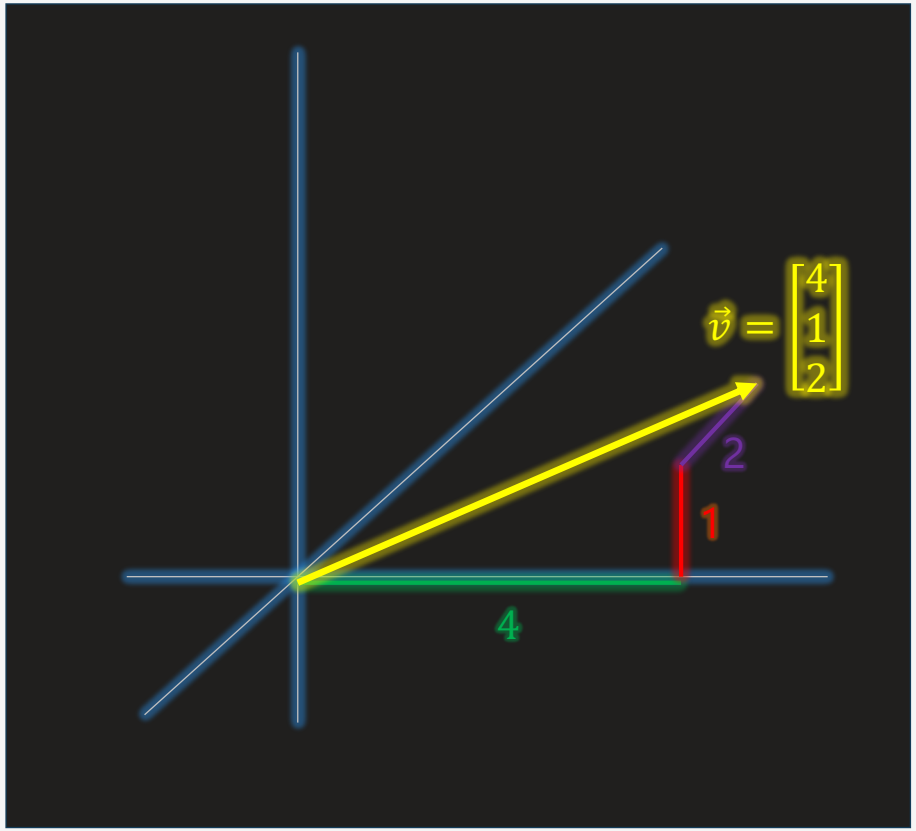


3D Vectors and Beyond

Vectors can apply beyond just a 2-dimensional space.

Vectors can exist in any number of dimensions, although it gets hard to visualize outside of 3 dimensions.

To the right we see a 3D vector represented on 3 axes X, Y, and Z.



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

Declaring Vectors in Python

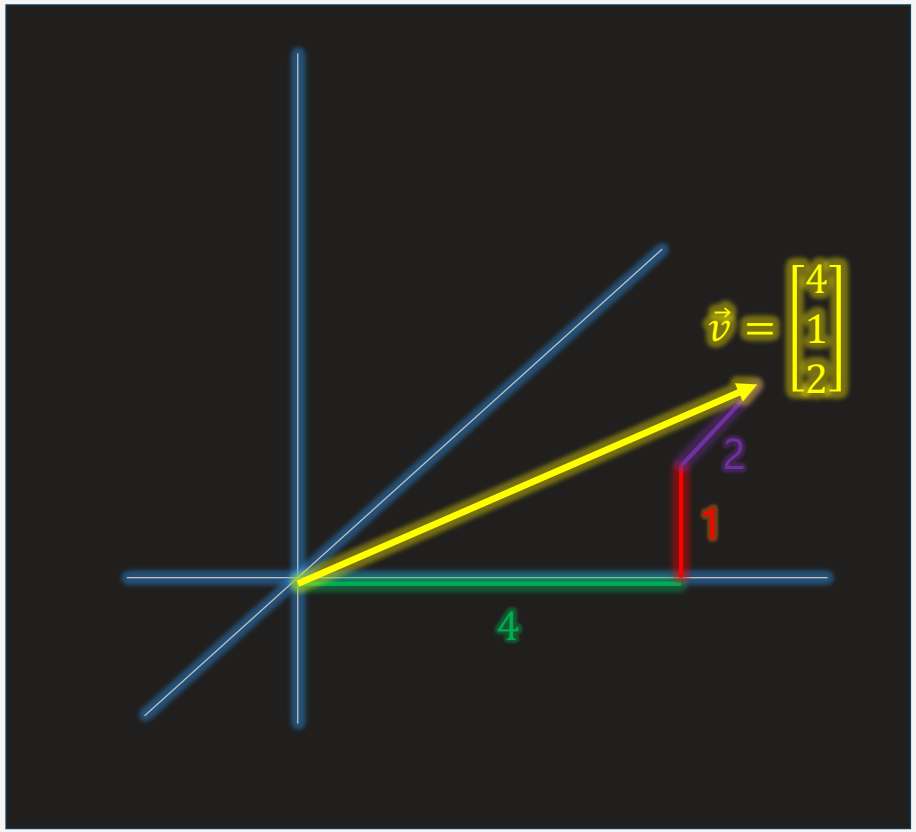
In Python, you will typically represent a vector using NumPy's `ndarray`.

To the right we declare a 3-dimensional vector (holding 3 elements) in Python using a 1-dimensional array in NumPy.

```
from numpy import array

data = [4.0, 1.0, 2.0]
v = array(data)

# display vector
print(v)
```

OUTPUT:

```
[4. 1. 2.]
```

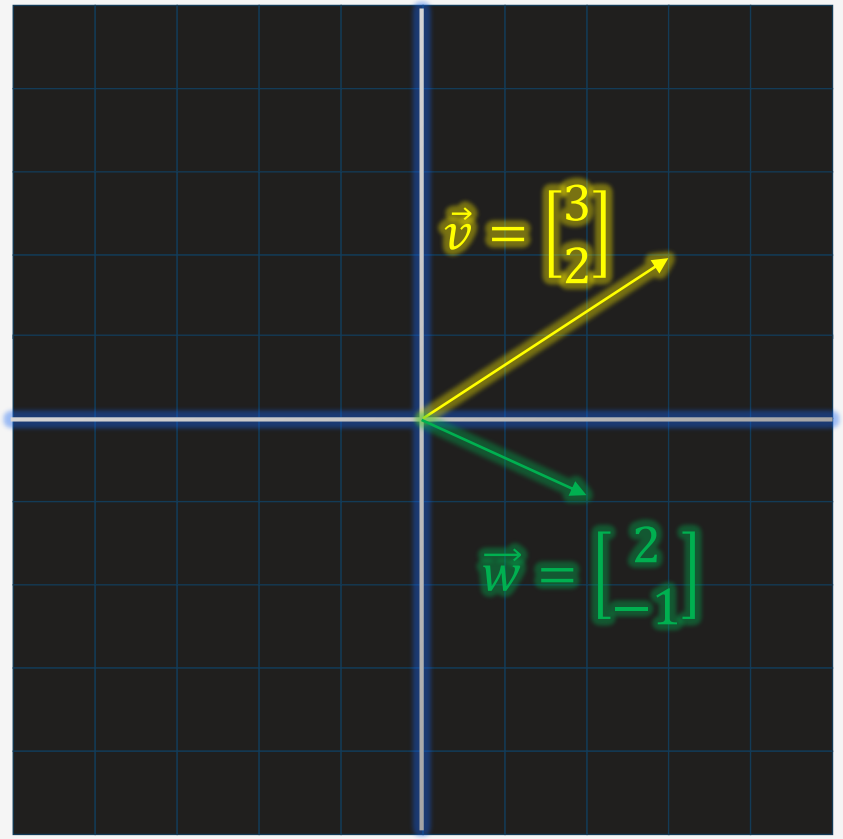


Adding/Combining Vectors

We think of vectors as movements, so what if we wanted to combine multiple movements?

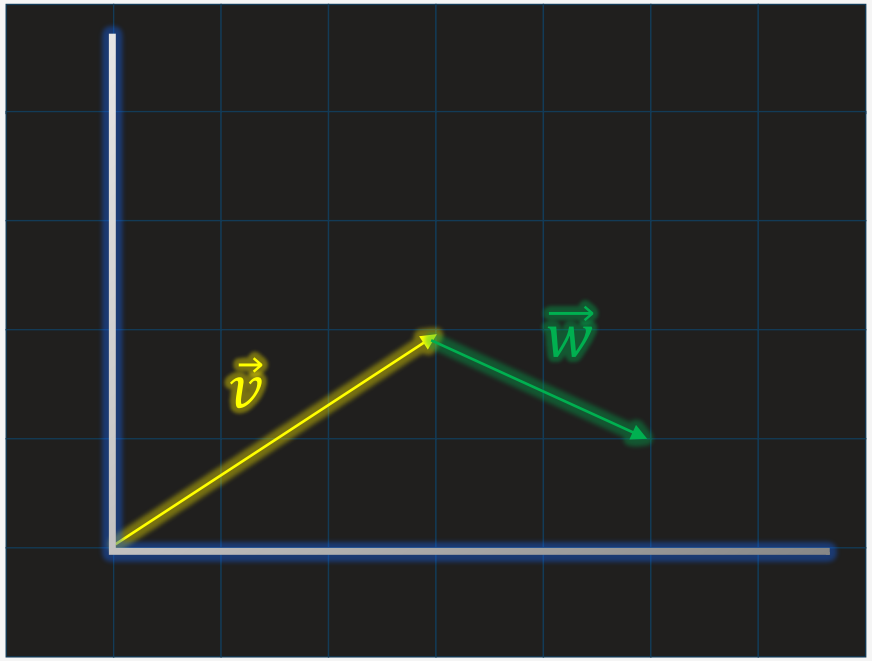
This is where we would add together two vectors, such $\vec{v}$ and $\vec{w}$, and combine them.

How would we add these two vectors to create a single vector?



Adding/Combining Vectors

To visually add these two vectors together, connect one vector after the other and walk to the tip of the last vector.



Adding/Combining Vectors

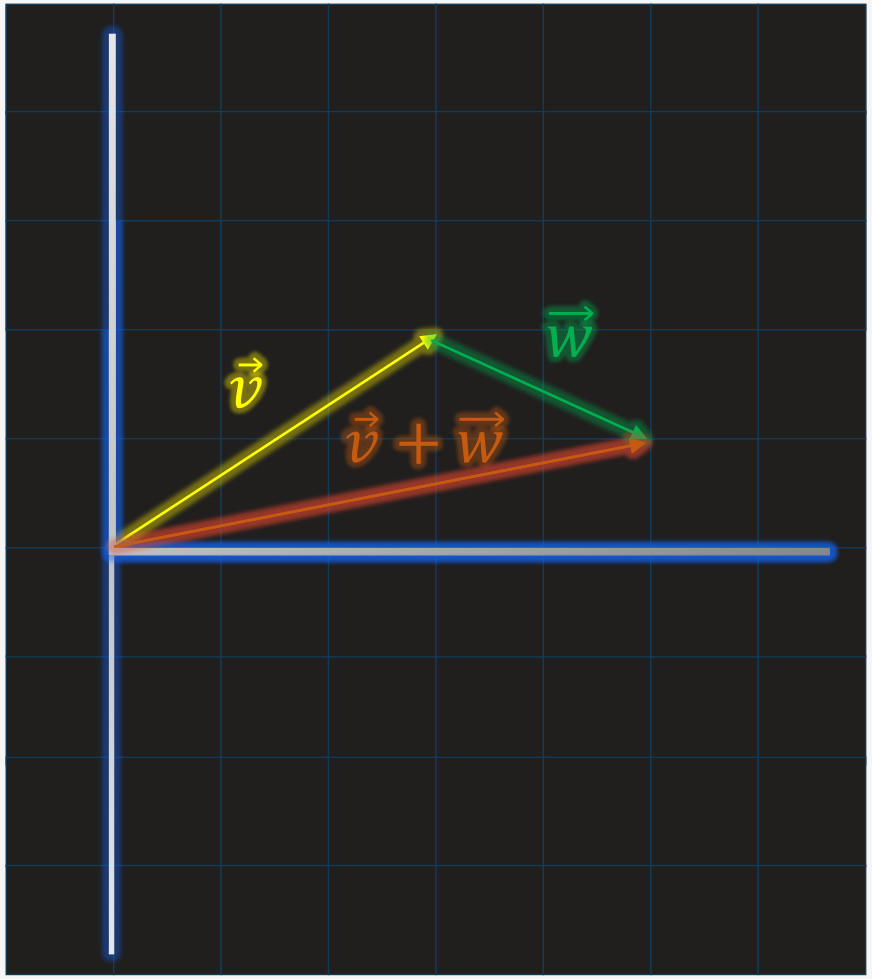
To visually add these two vectors together, connect one vector after the other and walk to the tip of the last vector.

The point you end at is a new vector, the result of summing the two vectors.

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

This new vector $\vec{v} + \vec{w}$ reflects the same movement as walking $\vec{v}$ and $\vec{w}$ consecutively.

We can do this same movement with a single vector $\begin{bmatrix} 5 \\ 1 \end{bmatrix}$, rather than walking $\begin{bmatrix} 3 \\ 2 \end{bmatrix}$ and $\begin{bmatrix} 2 \\ -1 \end{bmatrix}$ separately.

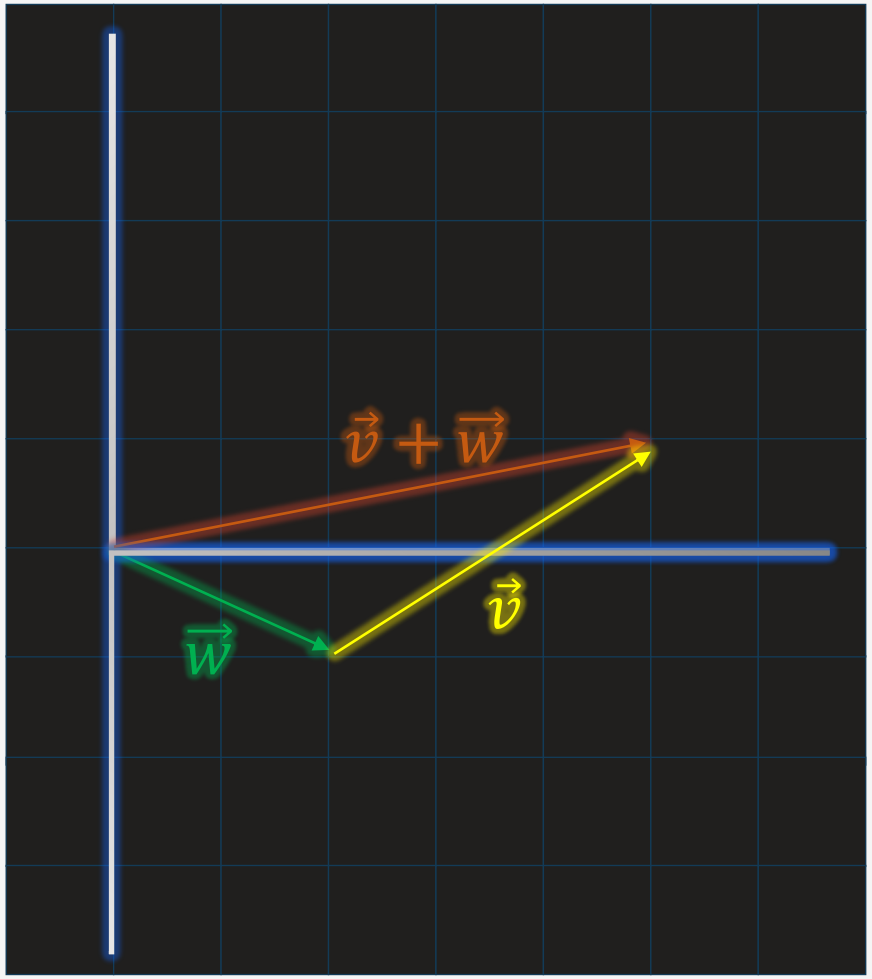


Adding/Combining Vectors

Note the commutative nature of adding vectors, as it does not matter which order you add them.

If we walked $\vec{w}$ before $\vec{v}$, or added them in that order, we will still end up with the same combined vector.

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



Adding/Combining Vectors in Python

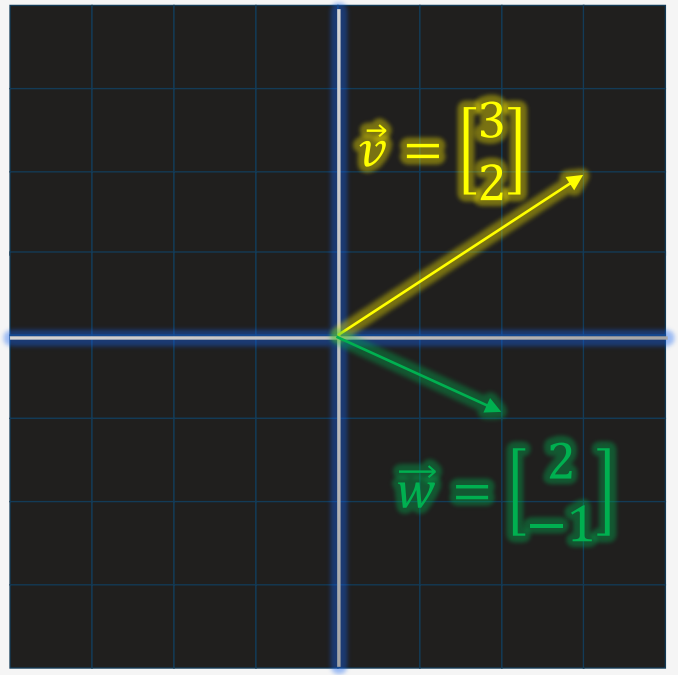
Summing vectors in Python with NumPy should be straightforward.

Declare the vectors as NumPy arrays and use the + operator to sum the vectors.

```
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)
```

OUTPUT:

```
[5 1]
```



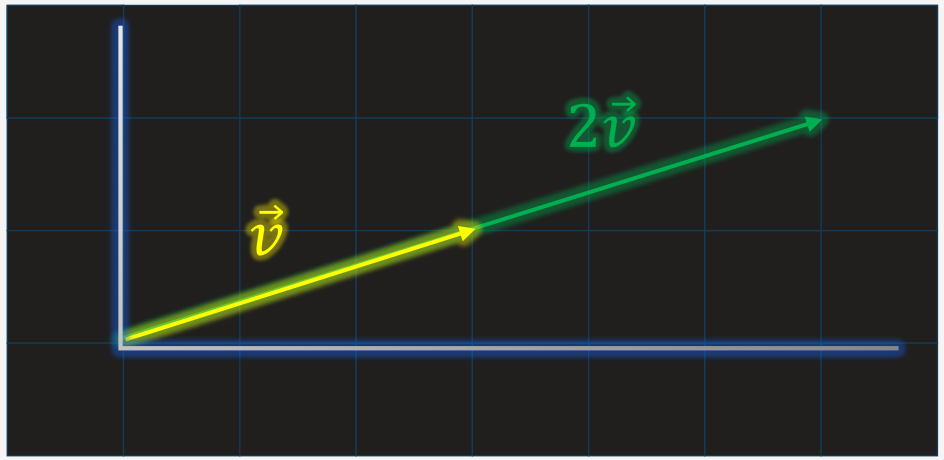
Multiplying/Scaling Vectors

You can also grow/shrink a vector by multiplying or scaling it with a scalar value.

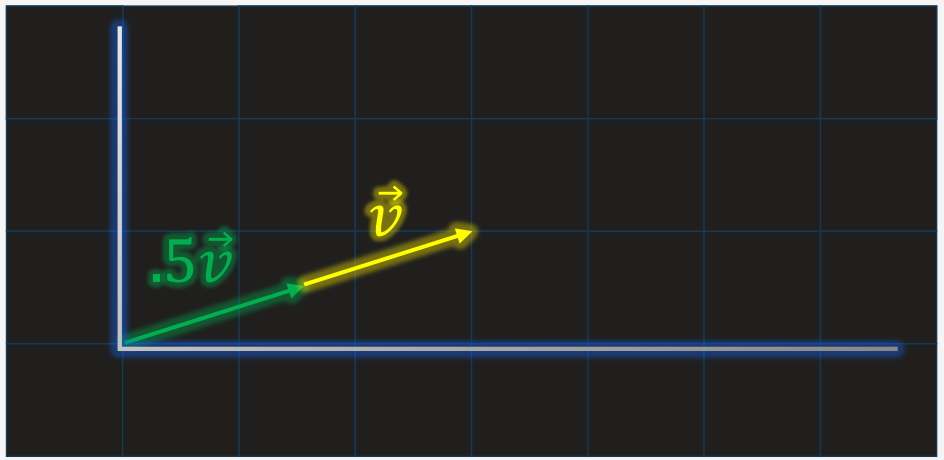
To the right are two examples, the top doubles the length of the vector while and the bottom halves the vector.

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

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



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

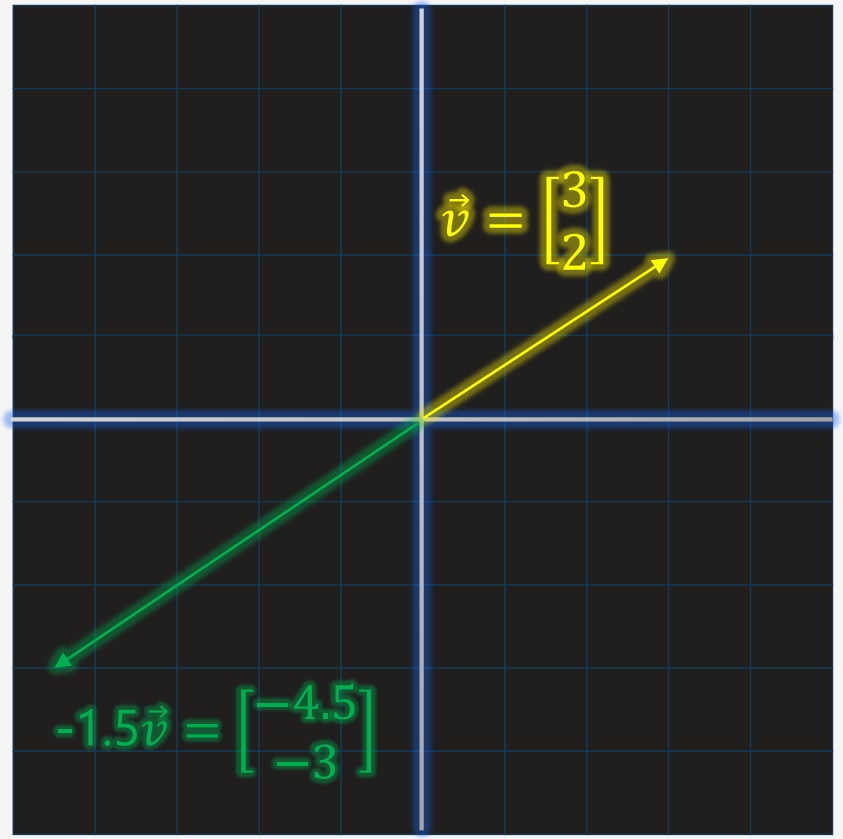


Multiplying/Scaling Vectors

When you multiply/scale a vector with a negative number, it will flip the vector in the opposite direction.

Note that multiplying/scaling a vector only grows or shrinks a vector.

Scaling does not change a vector's direction except for a negative scalar which flips its direction.



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

$$-1.5\vec{v} = -1.5 \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} -4.5 \\ -3 \end{bmatrix}$$

Multiplying/Scaling Vectors in Python

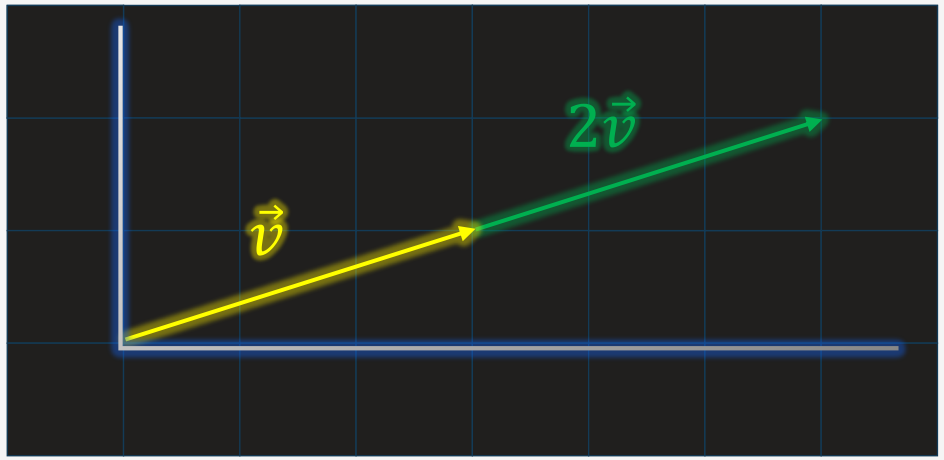
Scaling a vector in NumPy is as simple as using the multiplication * operator against an array.

To the right we scale the vector $\vec{v}$ by a factor of 2.

```
from numpy import array

v = array([3,1])

# scale the vector
scaled_v = 2.0 * v

# display scaled vector
print(scaled_v)
```

OUTPUT:

```
[6. 2.]
```



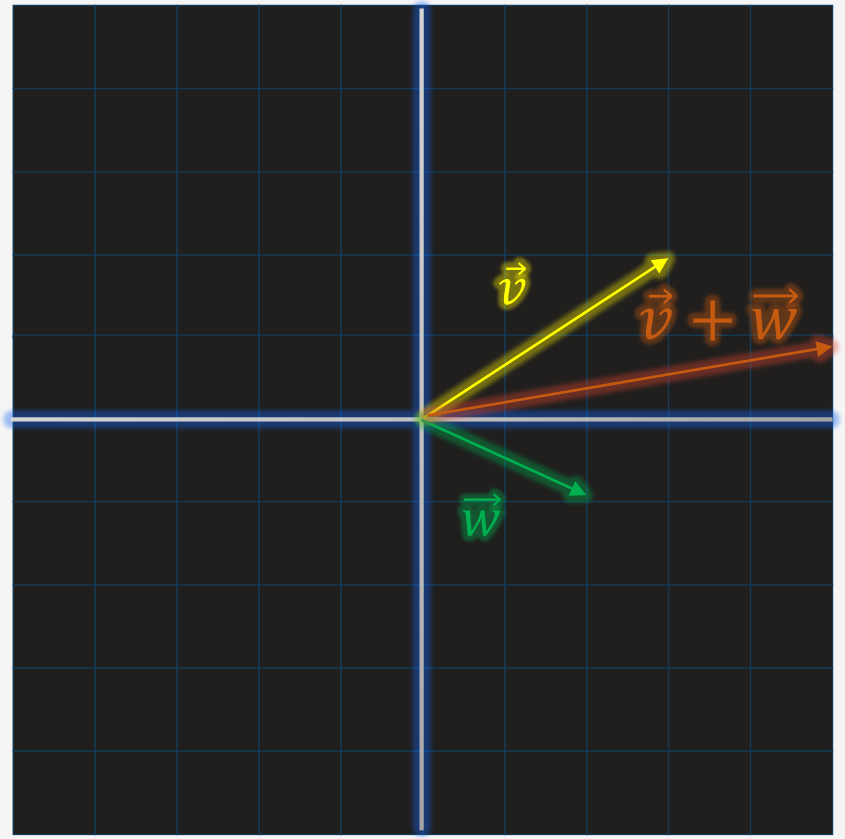
Scaling and Adding Vectors

We learned how to scale and add two vectors together.

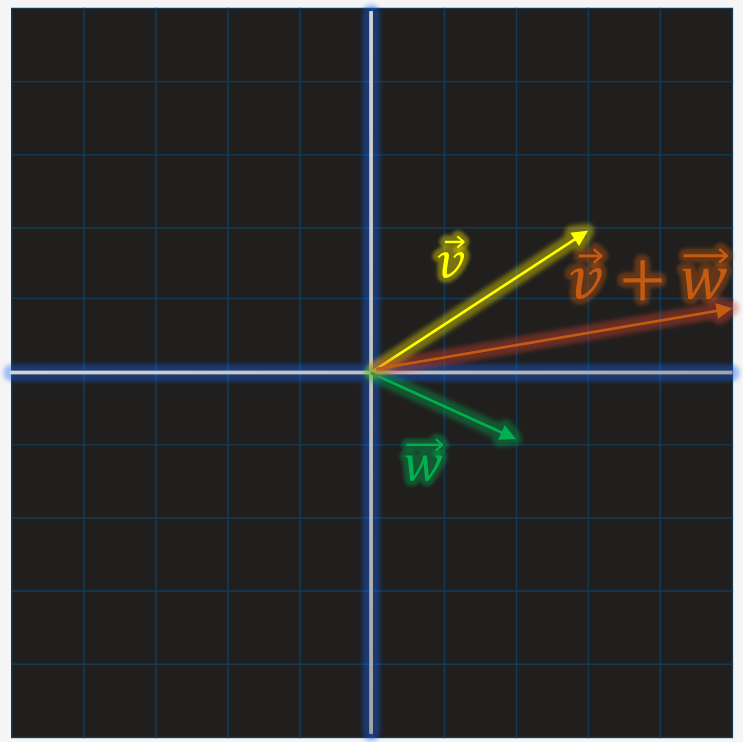
These vectors $\vec{v}$ and $\vec{w}$, fixed in two different directions, can be scaled and added to create ***ANY*** new vector $\vec{v} + \vec{w}$.

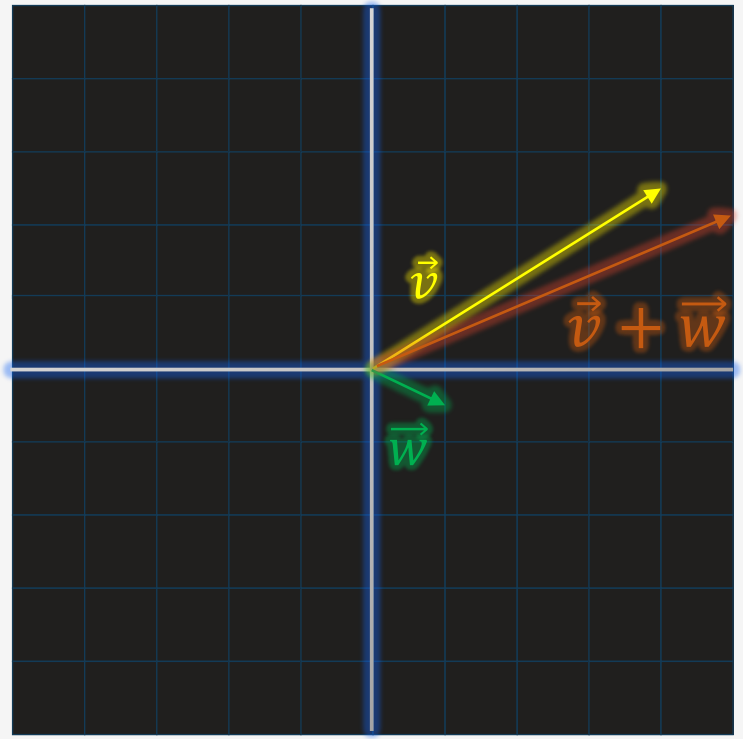
AGAIN: $\vec{v}$ and $\vec{w}$ are fixed in direction, except for flipping with negative scalars, but we can use scaling to freely create any vector composed of $\vec{v} + \vec{w}$.

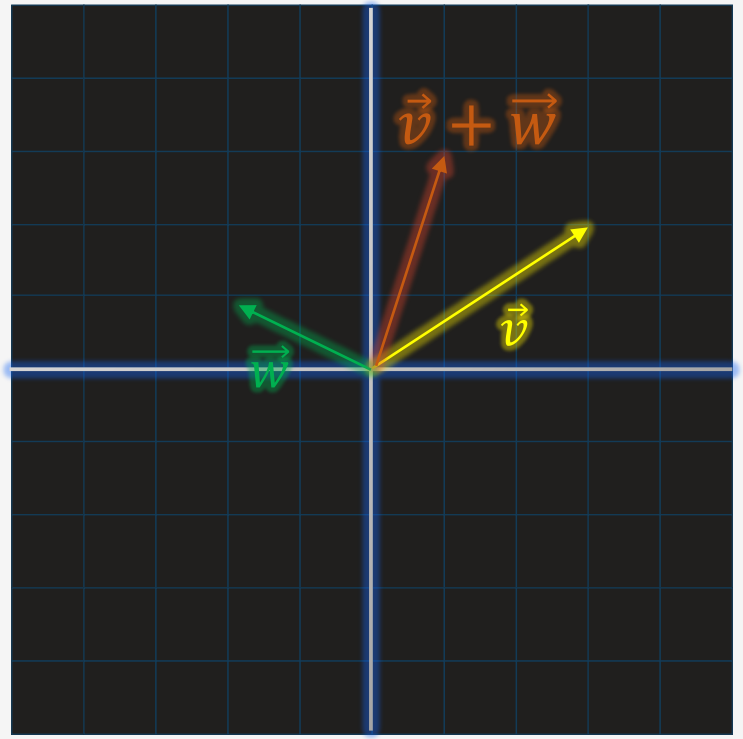
This entire space of vectors I can create off two vectors, fixed in direction but allowed to scale, is known as span.



Span – Linearly Independent



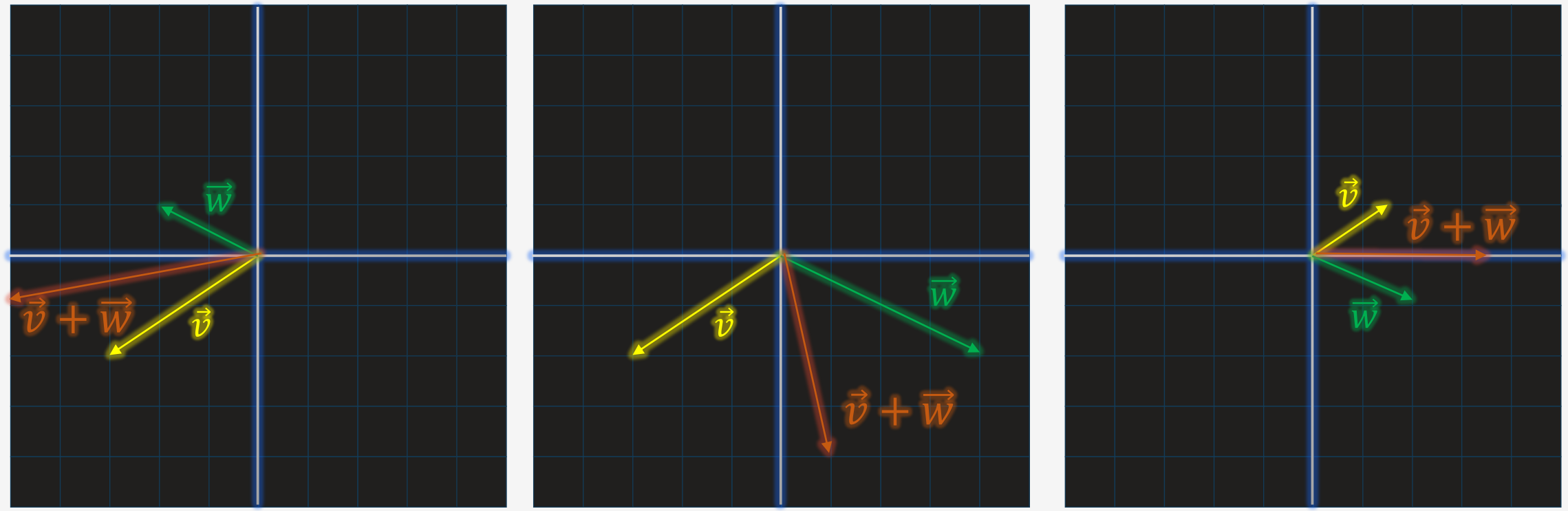




Notice above that I can scale $\vec{v}$ and $\vec{w}$, while keeping its direction fixed (or reversed), and have $\vec{v}$ + $\vec{w}$ be an infinite number of possible vectors.

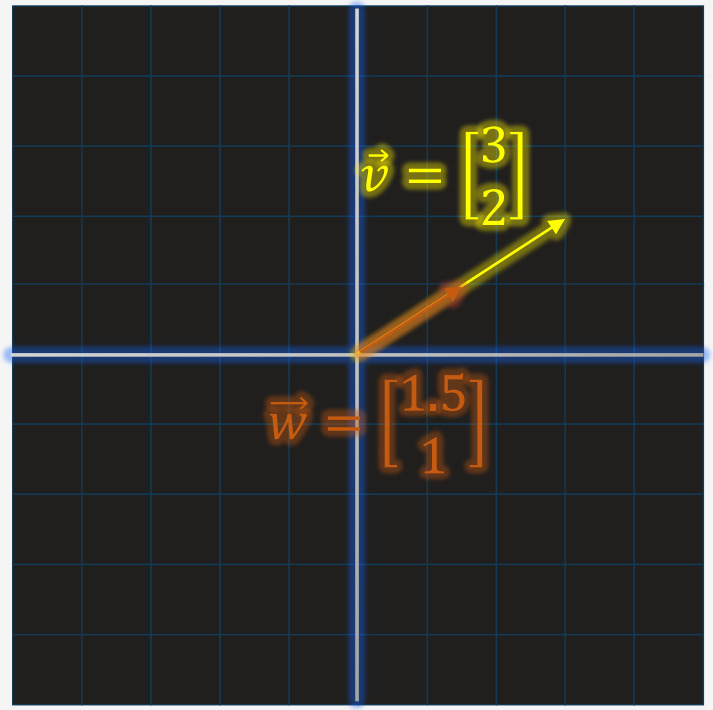
Because the sum of these two vectors give me access to all vectors in space, they are **linearly independent**.

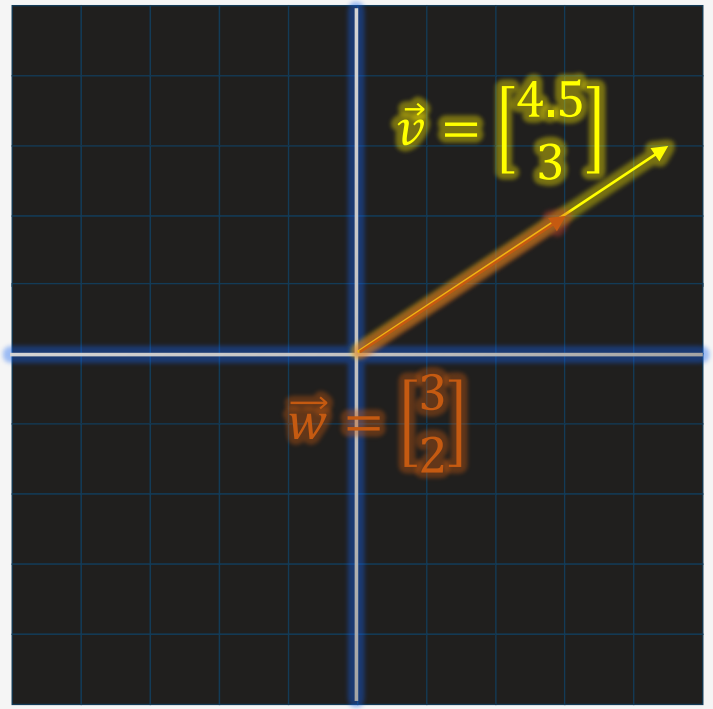
Span – Linearly Independent

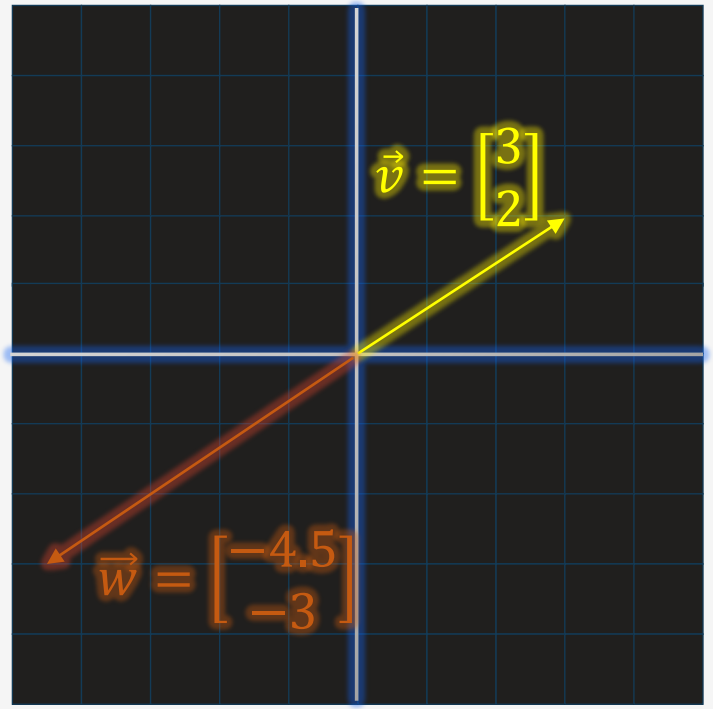


$\vec{v}$ + $\vec{w}$ can reach any point in space just by scaling these two vectors and summing them; there is no changing direction for $\vec{v}$ or $\vec{w}$ except for reversing through negative scalars.

Span – Linearly Dependent







Is there an unlimited span for $\vec{v}$ + $\vec{w}$ when $\vec{v}$ and $\vec{w}$ share a straight line?

Nope! In this unlucky case, we can only create vectors on their shared straight line.

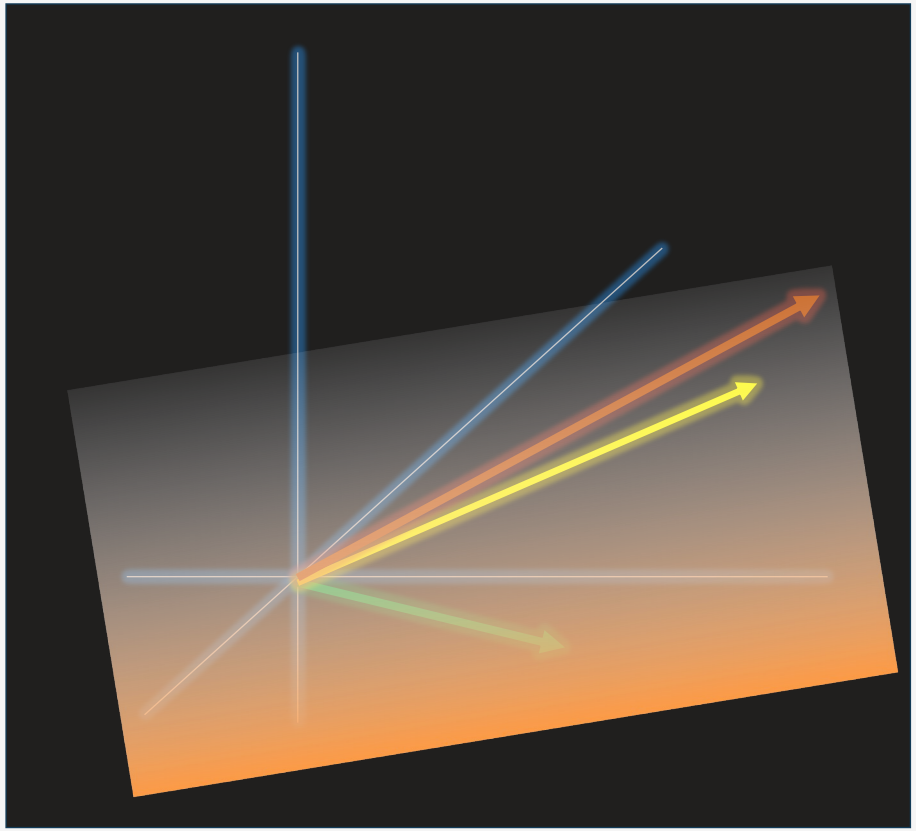
Because these vectors share a common direction, they are **linearly dependent**.

Span – Linear Dependence in 3D or More

With a 3D vector space, if two vectors are linearly dependent (share a direction) but the third vector is linearly independent from the other two, the span will be a flat sheet in space.

If all three vectors were linearly dependent, their span would only be a line in space.

This concept applies to any number of dimensions, not just 2 or 3.

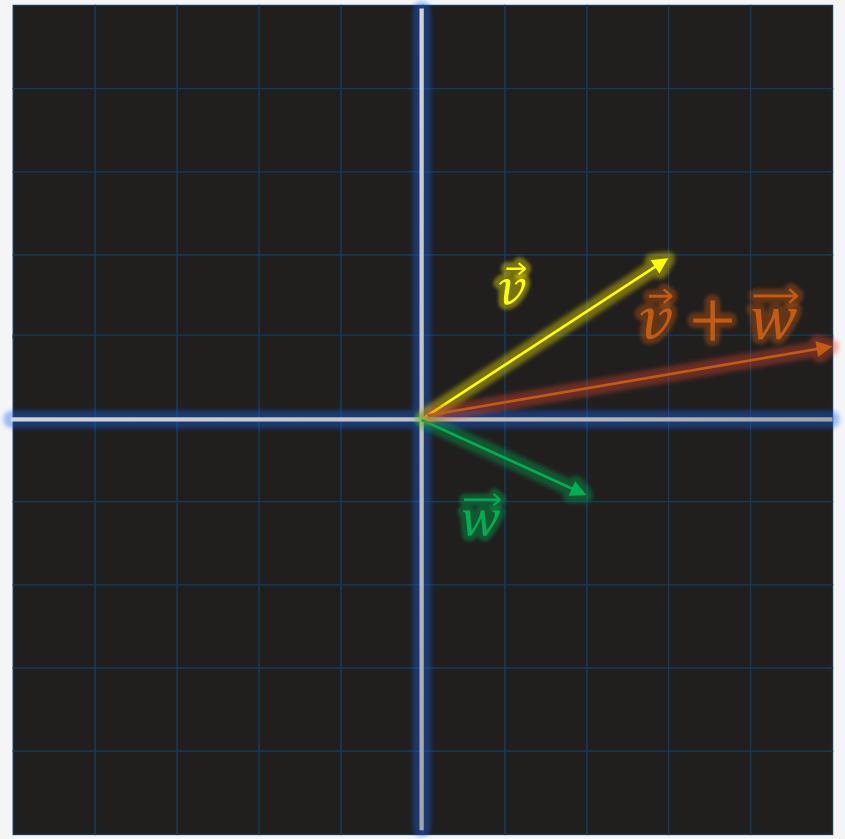


Discovering Linear Transformations

This concept of adding two vectors with fixed direction, but scaling them to get different combined vectors, is hugely important.

This combined vector, except in cases of linear dependence, can point in any direction and have any length we choose.

This sets up an intuition for linear transformations, where we use a vector to transform another vector in a functional-like manner.



II. Transformations and Matrices

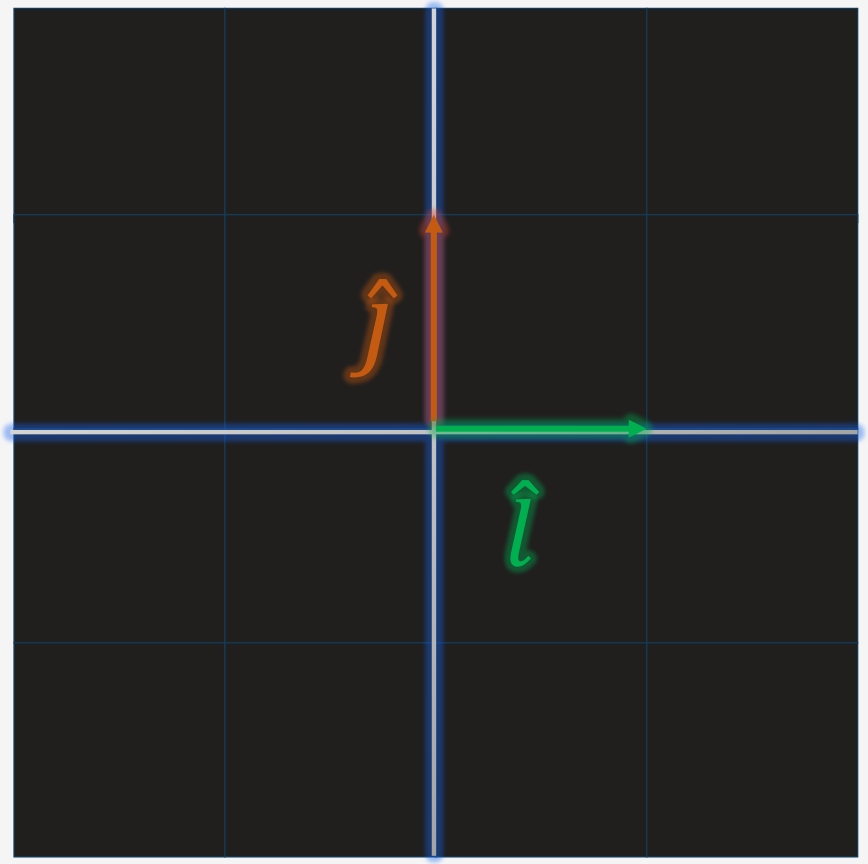
$\hat{\imath}$ and $\hat{\jmath}$ (i-hat and j-hat)

Imagine we have two simple vectors, $\hat{\imath}$ and $\hat{\jmath}$ (i-hat and j-hat).

$$\hat{\imath} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\hat{\jmath} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

These are known as basis vectors, and typically have a length of 1 and point in perpendicular positive directions.



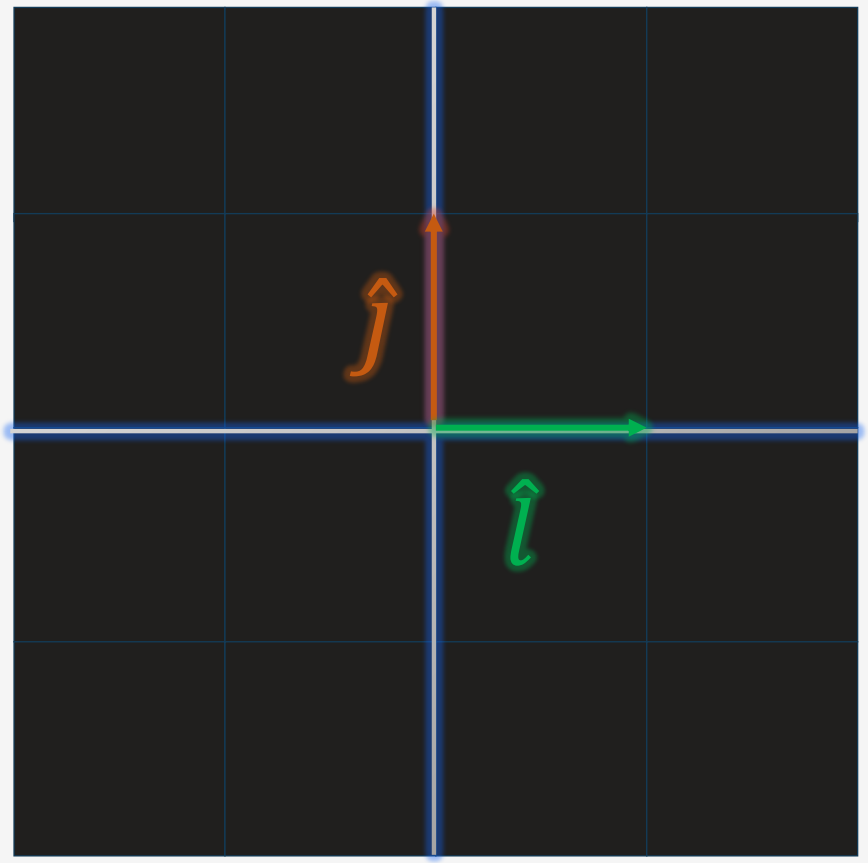
$\hat{\imath}$ and $\hat{\jmath}$ (i-hat and j-hat)

Think of the basis vectors as building blocks to build or transform any vector space.

Our basis vector is expressed in a 2 x 2 matrix, where the first column is $\hat{\imath}$ and the second column is $\hat{\jmath}$.

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

A matrix is like a vector (or collection of vectors), and it can have multiple rows and columns, and is a convenient way to package data.



$\hat{\imath}$ and $\hat{\jmath}$ (i-hat and j-hat)

Take this existing vector $\vec{v} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$.

We can think of vector $\vec{v}$ as scalars of $\hat{\imath}$ and $\hat{\jmath}$.

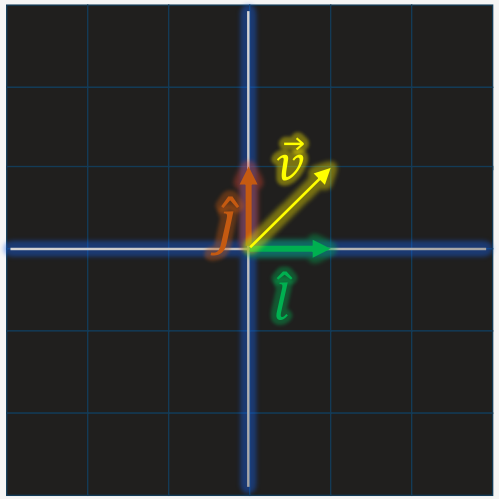
To create $\vec{v}$, we scale $\hat{\imath}$ and $\hat{\jmath}$ to $3\hat{\imath}$ and $2\hat{\jmath}$:

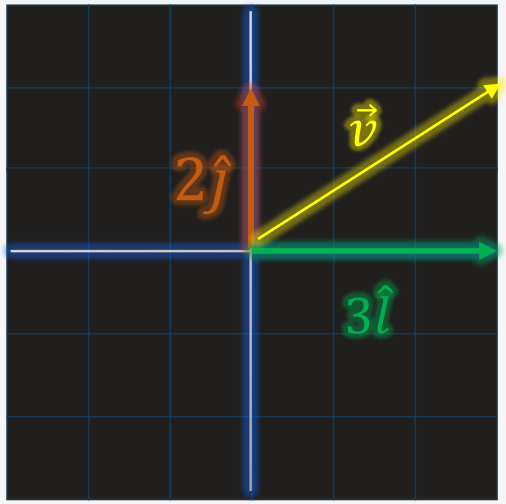
$$3\hat{\imath} = 3\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \end{bmatrix}$$

$$2\hat{\jmath} = 2\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$$

The sum of these two scaled $\hat{\imath}$ and $\hat{\jmath}$ vectors make up $\vec{v}$:

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





Matrix Vector Multiplication

$\hat{\imath}$ and $\hat{\jmath}$ do not have to be $\begin{bmatrix}1\\0\end{bmatrix}$ and $\begin{bmatrix}0\\1\end{bmatrix}$, but can be relative to describe a transformation.

This means we can use them to transform an existing vector.

The formula to transform a given vector $\begin{bmatrix}x\\y\end{bmatrix}$ using $\hat{\imath}$ and $\hat{\jmath}$ expressed as matrix $\begin{bmatrix}a & b\\c & d\end{bmatrix}$, where $\hat{\imath}$ is $\begin{bmatrix}a\\c\end{bmatrix}$ and $\hat{\jmath}$ is $\begin{bmatrix}b\\d\end{bmatrix}$:

$$\begin{bmatrix}x_{new}\\y_{new}\end{bmatrix} = \begin{bmatrix}a & b\\c & d\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$

$$\begin{bmatrix}x_{new}\\y_{new}\end{bmatrix} = \begin{bmatrix}ax + by\\cx + dy\end{bmatrix}$$

This transformation of a vector using basis vectors is known as matrix vector multiplication.

Matrix Vector Multiplication

Think of linear transformation as tracking where $\hat{\imath}$ and $\hat{\jmath}$ land, and then we can use that to transform a vector also.

Say we have a vector $\vec{v} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$, the space got stretched, and $\hat{\imath}$ and $\hat{\jmath}$ moved from $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ to $\begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}$.

What does that do to $\vec{v} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$?

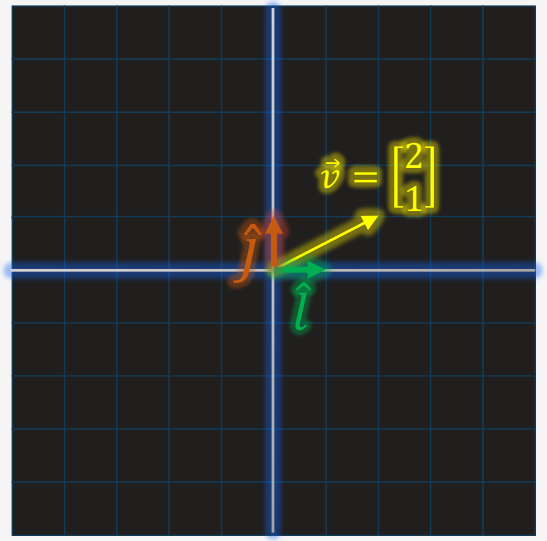
Using our linear transformation formula from the previous page, we can work out what our new vector is:

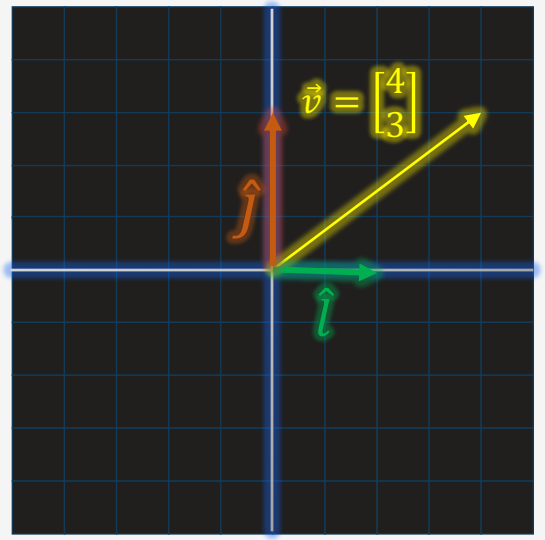
$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} (2)(2) + (0)(1) \\ (2)(0) + (3)(1) \end{bmatrix}$$

$$\begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{bmatrix} = \begin{bmatrix} \boldsymbol{4} \\ \boldsymbol{3} \end{bmatrix}$$

```
from numpy import array

i_hat = array([2,0])
j_hat = array([0,3])

basis = array([i_hat, j_hat]).transpose()

v = array([2,1])

w = basis.dot(v)

print(w)
```





Matrix Vector Multiplication in Python

You may have noticed the Python code from the previous example, and you probably have two questions:

What is the transpose function?

This will swap rows with columns.

This is necessary because NumPy will populate each vector as a row not a column, and $\hat{\imath}$ and $\hat{\jmath}$ need their own columns (not rows).

What is the dot() function?

The dot() function executes the **dot product**, the operation we did earlier with multiplying and adding together a vector and matrix.

We will talk about this function more later, but it is handy to prevent manual loops to add and multiply numbers between vectors/matrices.

```
from numpy import array

# Declare i-hat and j-hat
i_hat = array([2, 0])
j_hat = array([0, 3])

# compose basis matrix using i-hat and j-hat
# also need to transpose rows into columns
basis = array([i_hat, j_hat]).transpose()

# declare vector v 0
v = array([2,1])

# create new vector w
# by transforming v with dot product
w = basis.dot(v)

print(w)
```

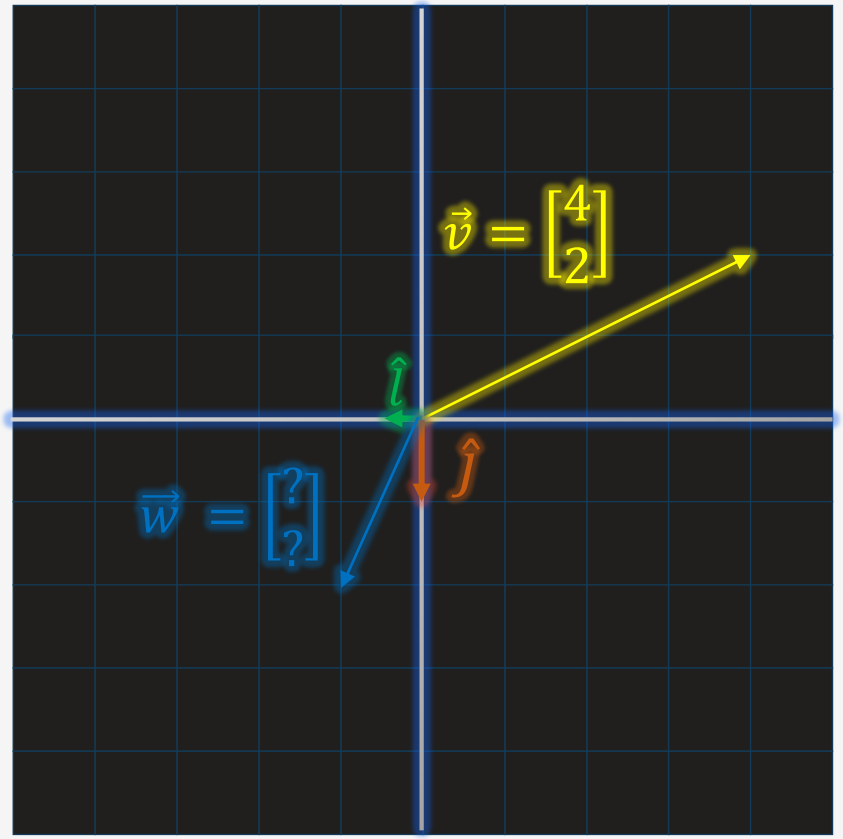
Transforming Vectors with $\hat{\imath}$ and $\hat{\jmath}$

This concept of seeing where $\hat{\imath}$ and $\hat{\jmath}$ land is important, because it allows us not just to create vectors but also transform existing vectors.

Another example: suppose we have this vector $\vec{v} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$, and we transformed it somehow and it became $\vec{w}$.

$\hat{\imath}$ lands at $\begin{bmatrix} -.25 \\ 0 \end{bmatrix}$ and $\hat{\jmath}$ lands at $\begin{bmatrix} 0 \\ -1 \end{bmatrix}$.

How do we describe this transformation? Well we can describe it in terms of $\hat{\imath}$ and $\hat{\jmath}$ and use them to execute the transformation!



Transforming Vectors with $\hat{\imath}$ and $\hat{\jmath}$

Matrix vector multiplication formula:

$$\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

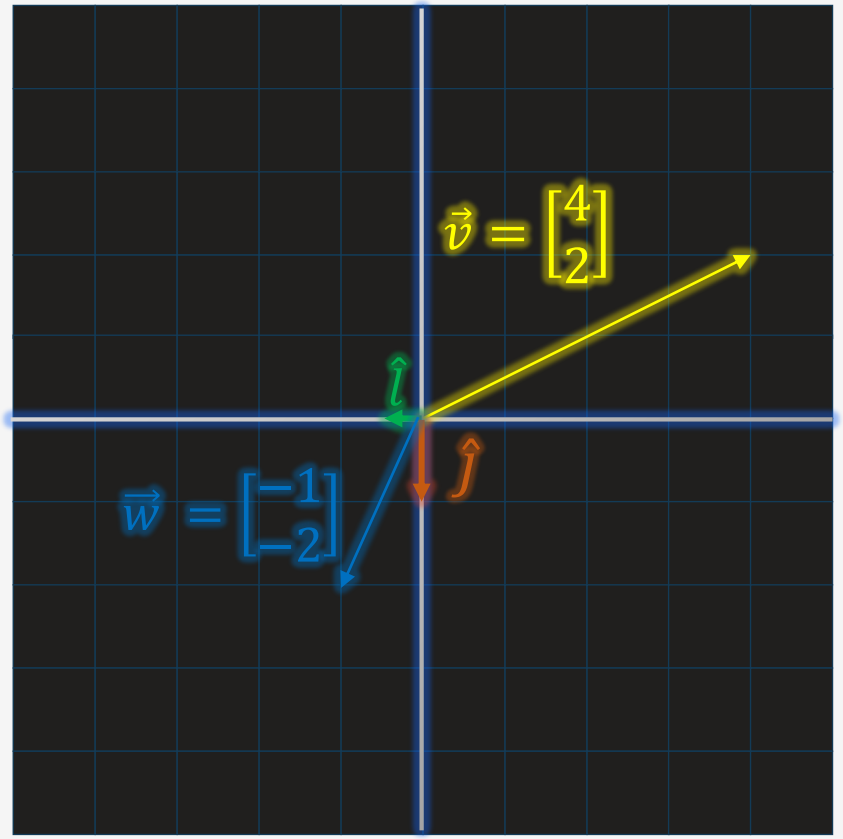
Apply the formula:

$$\vec{w} = \begin{bmatrix} -.25 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ 2 \end{bmatrix}$$

$$\vec{w} = \begin{bmatrix} (-.25)(4) + (0)(2) \\ (4)(0) + (-1)(2) \end{bmatrix}$$

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

```
from numpy import array

i_hat = array([-.25, 0])
j_hat = array([0, -1])

basis = array([i_hat, j_hat]).transpose()

v = array([4,2])

w = basis.dot(v)

print(w)
```



Sheers, Rotations, and Inversions

We can do more than simply scale with linear transformations.

We can also do sheers, rotations and inversions which stretch, turn, and flip our vector space respectively.

To the right we sheer and rotate our space simply using a change to $\hat{\imath}$ and $\hat{\jmath}$, and turn $\vec{v}$ into $\vec{w}$:

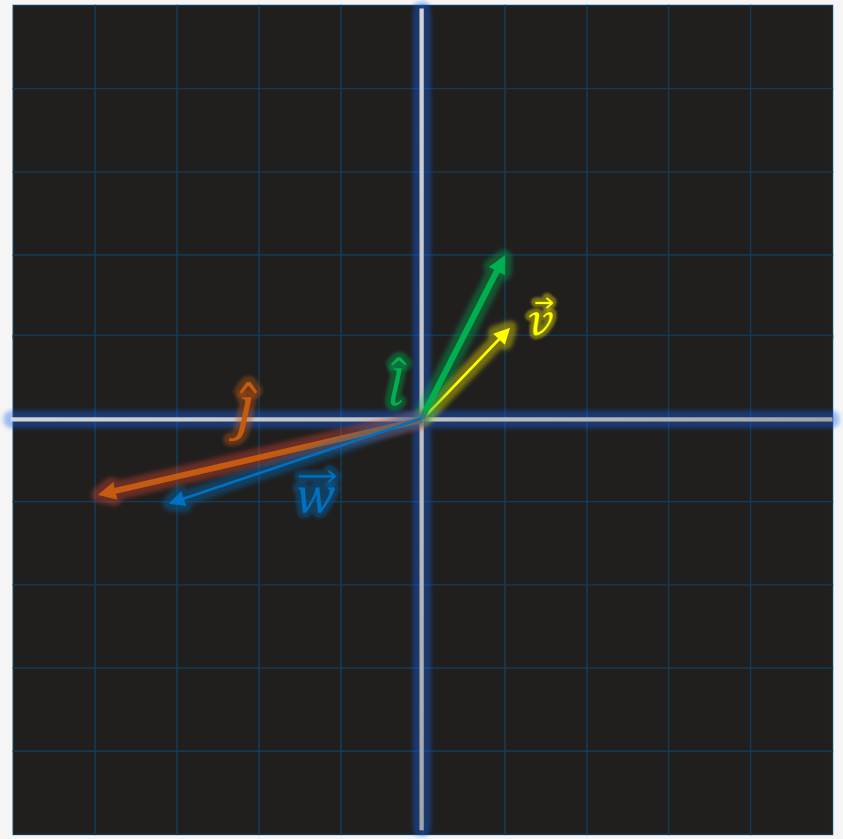
$$\vec{w} = \begin{bmatrix} 1 & -4 \\ 2 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\vec{w} = \begin{bmatrix} (1)(1) + (-4)(1) \\ (2)(1) + (-1)(1) \end{bmatrix}$$

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

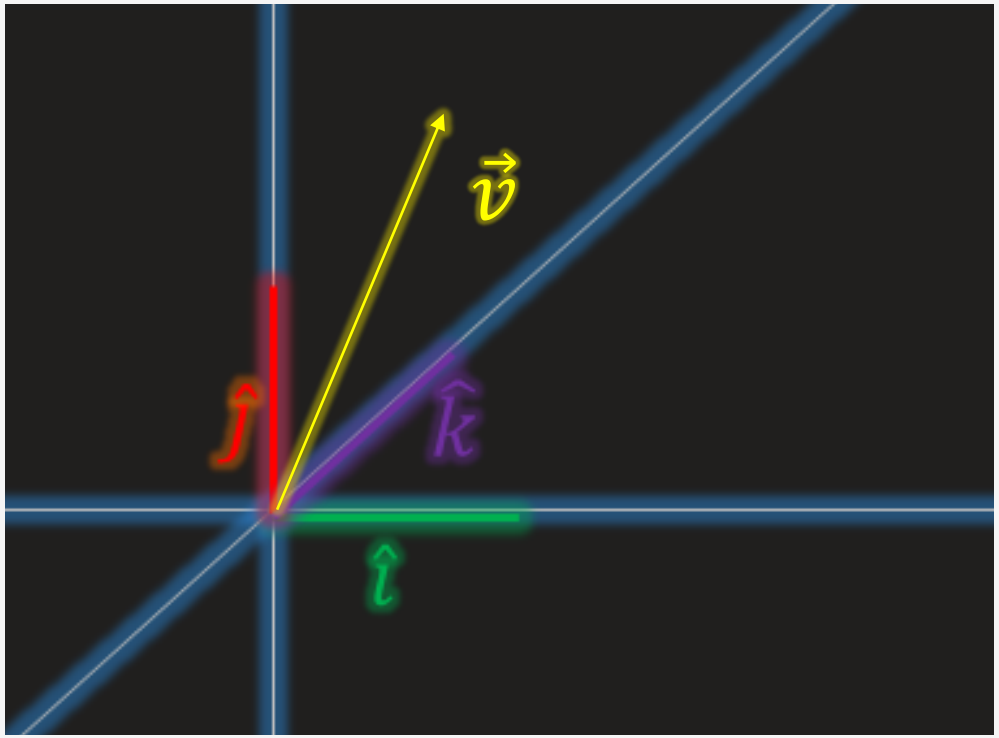
Linear transformations must be linear, so you cannot do nonlinear, curvy, or variable scaling and transformations.

```
from numpy import array

i_hat = array([1, 2])
j_hat = array([-4, -1])

basis = array([i_hat, j_hat]).transpose()

v = array([1,1])

w = basis.dot(v)

print(w)
```



Basis Vectors in 3D and More

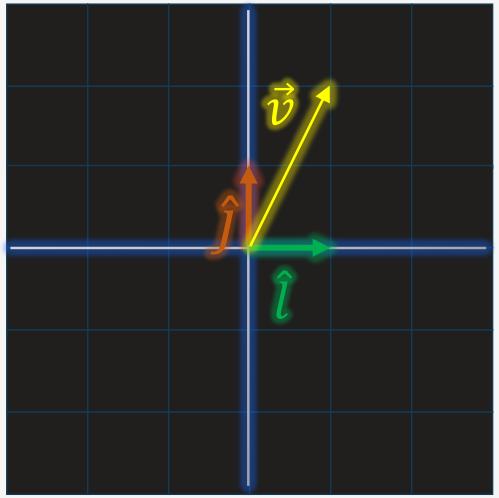
Peeking into beyond more than 2 dimensions, the basis vector concept can be extended beyond $\hat{\imath}$ and $\hat{\jmath}$ with $\hat{k}$, and as many additional dimensions you need.

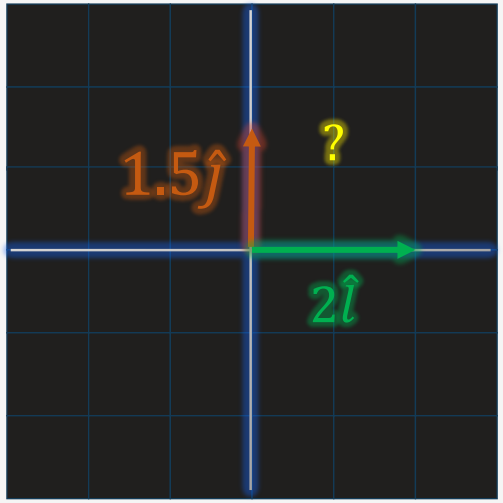


Quick Quiz #1

To the right we have vector $\vec{v} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$, but then we perform a transformation and $\hat{\imath}$ lands at $\begin{bmatrix} 2 \\ 0 \end{bmatrix}$ and $\hat{\jmath}$ lands at $\begin{bmatrix} 0 \\ 1.5 \end{bmatrix}$.

Where does $\vec{v}$ land now after this transformation?





Quick Quiz #1

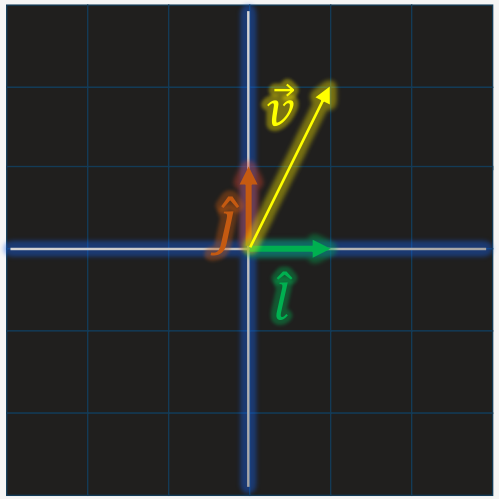
To the right we have vector $\vec{v} = \begin{bmatrix}1\\2\end{bmatrix}$, but then we perform a transformation and $\hat{\imath}$ lands at $\begin{bmatrix}2\\0\end{bmatrix}$ and $\hat{\jmath}$ lands at $\begin{bmatrix}0\\1.5\end{bmatrix}$.

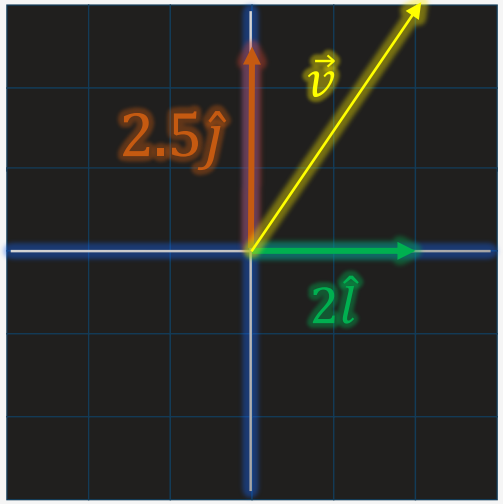
Where does $\vec{v}$ land now after this transformation?

$$\begin{bmatrix}x_{new}\\y_{new}\end{bmatrix} = \begin{bmatrix}a & b\\c & d\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$
$$\begin{bmatrix}x_{new}\\y_{new}\end{bmatrix} = \begin{bmatrix}ax+by\\cx+dy\end{bmatrix}$$

$$\begin{bmatrix}x\\y\end{bmatrix} = \begin{bmatrix}2 & 0\\0 & 1.5\end{bmatrix}\begin{bmatrix}1\\2\end{bmatrix}$$
$$\begin{bmatrix}x\\y\end{bmatrix} = \begin{bmatrix}(2)(1)+(0)(2)\\(1)(0)+(1.5)(2)\end{bmatrix}$$
$$\begin{bmatrix}\boldsymbol{x}\\\boldsymbol{y}\end{bmatrix} = \begin{bmatrix}\boldsymbol{2}\\\boldsymbol{3}\end{bmatrix}$$

```
from numpy import array

i_hat = array([2, 0])
j_hat = array([0, 1.5])

basis = array([i_hat, j_hat]).transpose()

v = array([1,2])

w = basis.dot(v)

print(w)
```

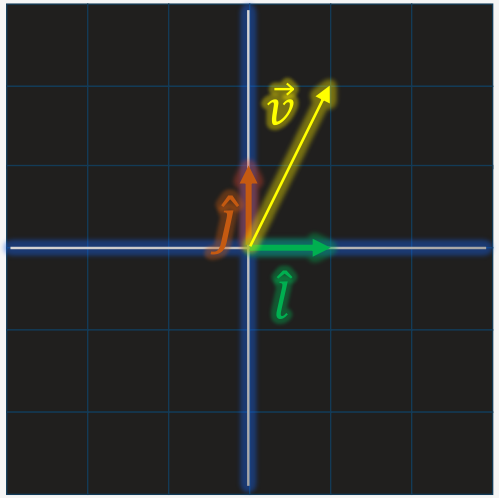


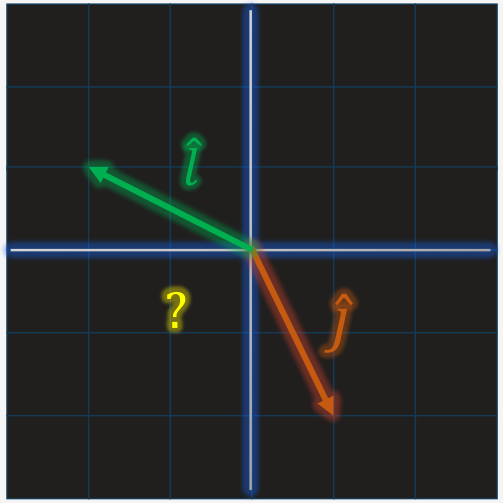


Quick Quiz #2

To the right we have vector $\vec{v} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$, but then we perform a transformation and $\hat{\imath}$ lands at $\begin{bmatrix} -2 \\ 1 \end{bmatrix}$ and $\hat{\jmath}$ lands at $\begin{bmatrix} 1 \\ -2 \end{bmatrix}$.

Where does $\vec{v}$ land now after this transformation?





Quick Quiz #2

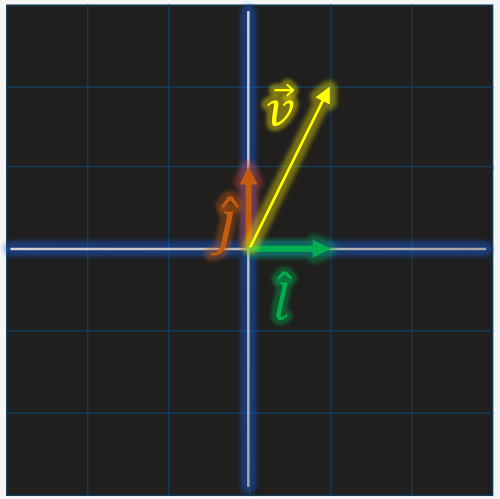
To the right we have vector $\vec{v} = \begin{bmatrix}1\\2\end{bmatrix}$, but then we perform a transformation and $\hat{\imath}$ lands at $\begin{bmatrix}-2\\1\end{bmatrix}$and $\hat{\jmath}$ lands at $\begin{bmatrix}1\\-2\end{bmatrix}$.

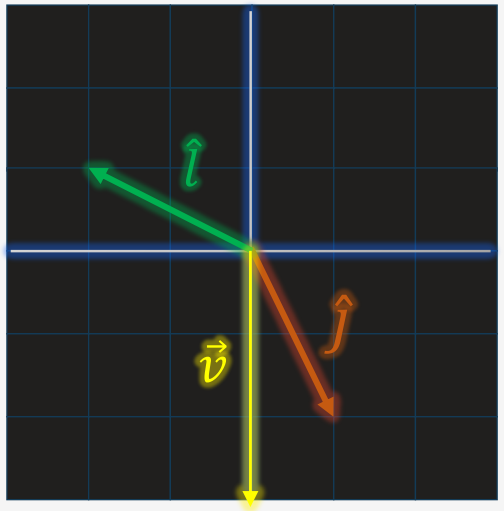
Where does $\vec{v}$ land now after this transformation?

$$\begin{bmatrix}x_{new}\\y_{new}\end{bmatrix} = \begin{bmatrix}a & b\\c & d\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$
$$\begin{bmatrix}x_{new}\\y_{new}\end{bmatrix} = \begin{bmatrix}ax+by\\cx+dy\end{bmatrix}$$

$$\begin{bmatrix}x\\y\end{bmatrix} = \begin{bmatrix}-2 & 1\\1 & -2\end{bmatrix}\begin{bmatrix}1\\2\end{bmatrix}$$
$$\begin{bmatrix}x\\y\end{bmatrix} = \begin{bmatrix}(-2)(1)+(1)(2)\\(1)(1)+(-2)(2)\end{bmatrix}$$
$$\begin{bmatrix}\boldsymbol{x}\\\boldsymbol{y}\end{bmatrix} = \begin{bmatrix}\boldsymbol{0}\\\boldsymbol{-3}\end{bmatrix}$$

```
from numpy import array

i_hat = array([-2, 1])
j_hat = array([1, -2])

basis = array([i_hat, j_hat]).transpose()

v = array([1,2])

w = basis.dot(v)

print(w)
```

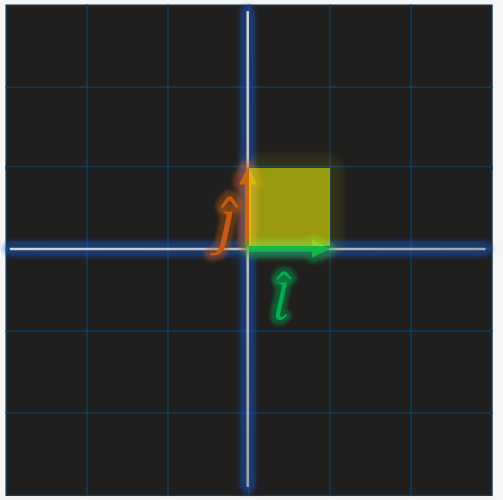


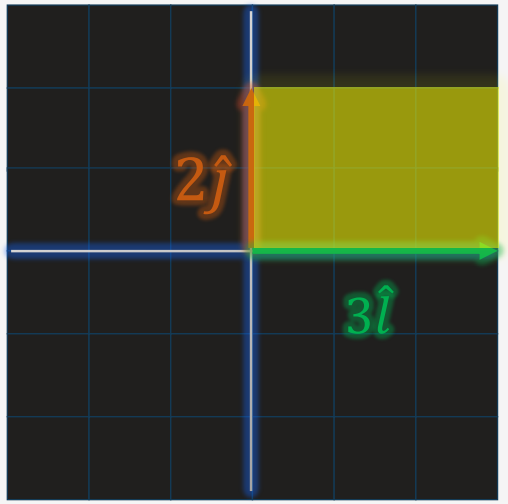


The Determinant

When we perform linear transformations, we sometimes “expand” or “squish” space and the degree this happens can be helpful.

Take a sampled area from the vector space to the right: what happens to it after we scaled $\hat{\imath}$ and $\hat{\jmath}$?





The Determinant

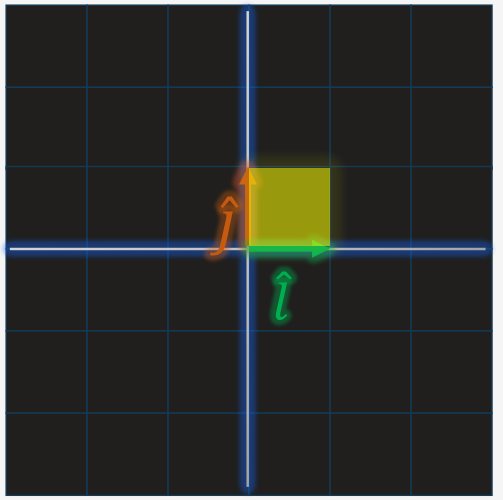
When we perform linear transformations, we sometimes "expand" or "squish" space and the degree this happens can be helpful.

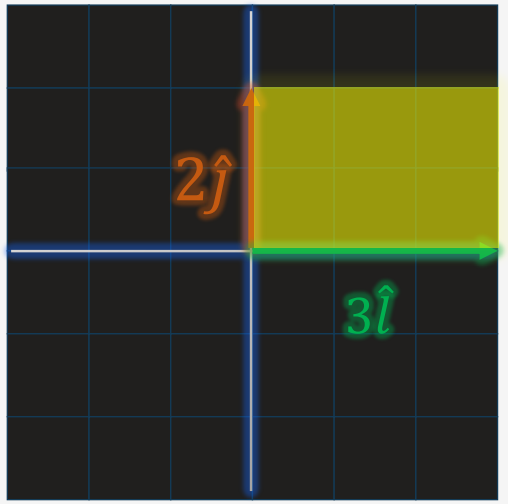
Take a sampled area from the vector space to the right: what happens to it after we scaled $\hat{\imath}$ and $\hat{\jmath}$?

It increases in area by a factor of 6.0, and this factor is known as a determinant.

Determinants describe how much a sampled area in a vector space changes in scale with linear transformations, and this can provide helpful information about the transformation.

```
from numpy.linalg import det
from numpy import array

i_hat = array([3, 0])
j_hat = array([0, 2])

basis = array([i_hat, j_hat]).transpose()

determinant = det(basis)

print(determinant) # prints 6.0
```





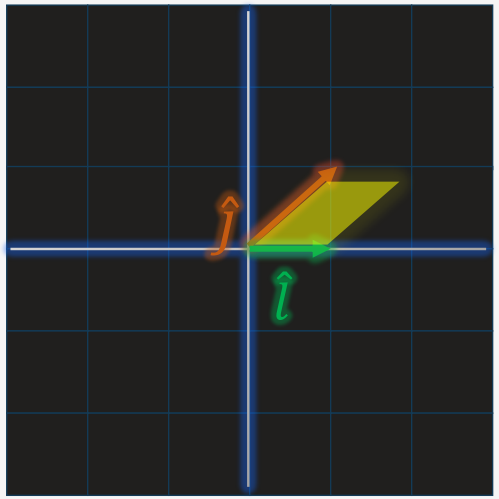
Determinant Behaviors

Simple sheers should not change the determinant.

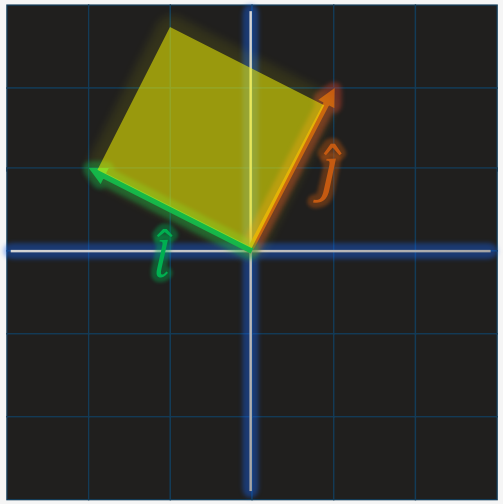
But scaling will increase or decrease the determinant, as that will increase/decrease the sampled area.

When the orientation flips ($\hat{\imath}$ and $\hat{\jmath}$ swap clockwise positions) then the determinant will be negative.

```
from numpy.linalg import det
from numpy import array

i_hat = array([1, 0])
j_hat = array([1, 1])

basis = array([i_hat, j_hat]).transpose()

determinant = det(basis)

print(determinant) # prints 1.0
```




```
from numpy.linalg import det
from numpy import array

i_hat = array([-2, 1])
j_hat = array([1, 2])

basis = array([i_hat, j_hat]).transpose()

determinant = det(basis)

print(determinant) # prints -5.0
```

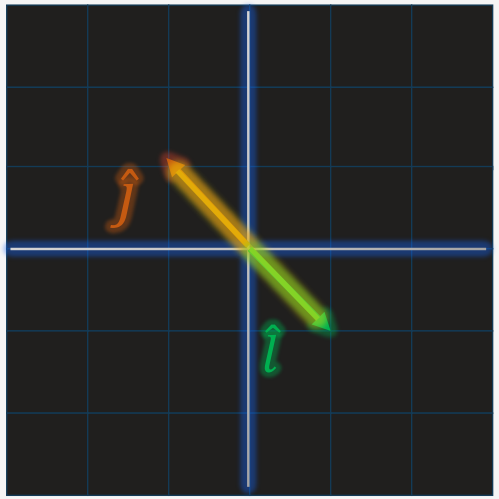


Zero Determinants

When a determinant is 0, that means the transformation is linearly dependent, and has squished all of space into a line.

Testing for a 0 determinant is highly helpful to determine if a transformation has linear dependence.

```
from numpy.linalg import det
from numpy import array

i_hat = array([-1, 1])
j_hat = array([1, -1])

basis = array([i_hat, j_hat]).transpose()

determinant = det(basis)

print(determinant) # prints 0.0
```



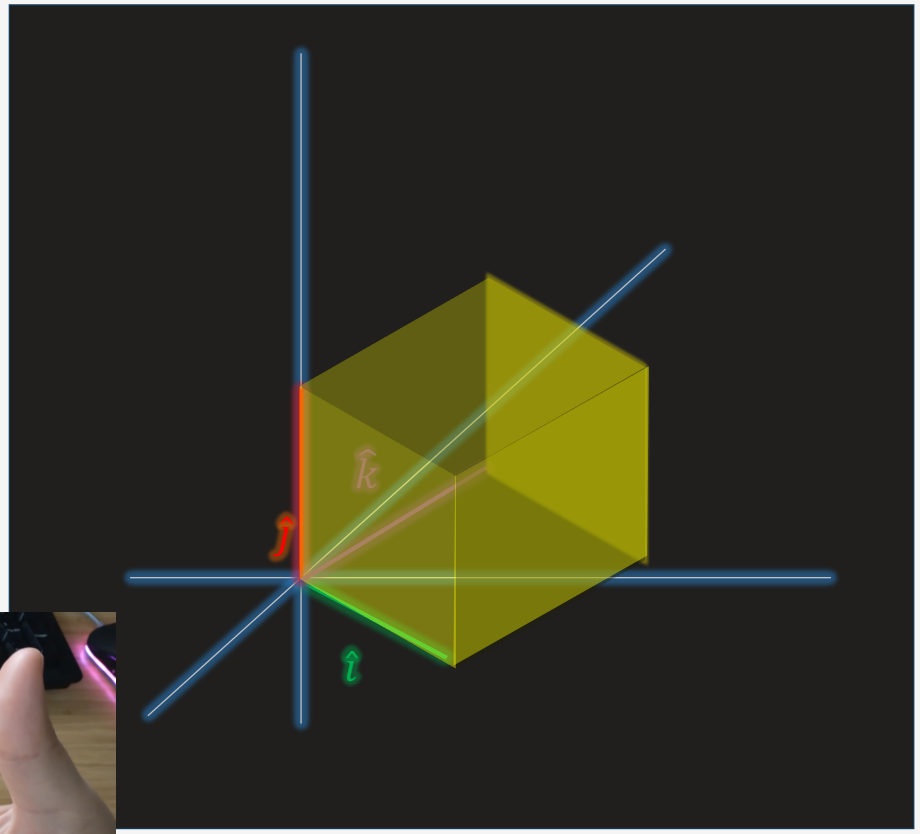
Determinants in 3D

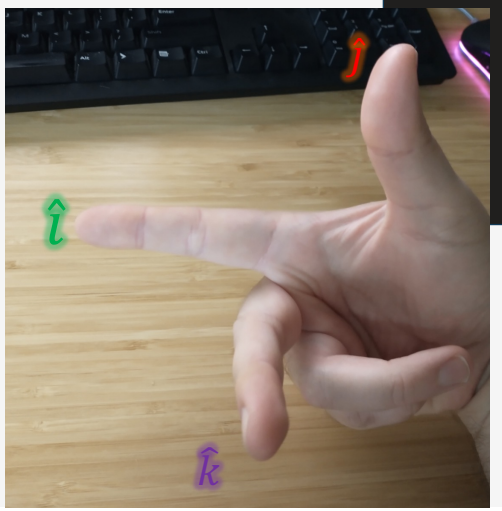
Of course, the determinant extends to 3 or more dimensions.

It becomes a matter of visualizing a sampled volume scaling, rotating, sheering, and flipping.

To figure out if a 3D space has flipped, use the hand rule.

If you cannot relatively orient $\hat{\imath}$, $\hat{\jmath}$, and $\hat{k}$ like my hand to the right, the orientation has flipped, and you should have a negative determinant.

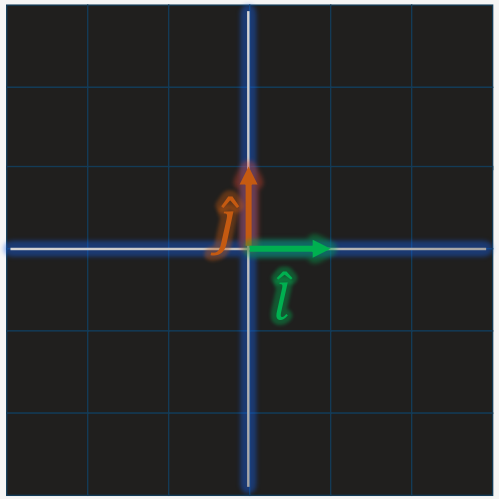


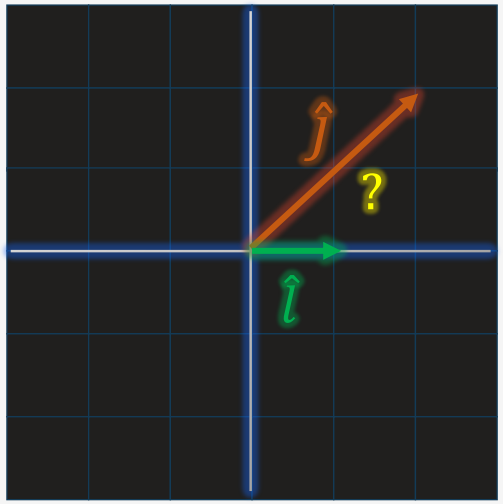


Quick Quiz #3

To the right we have a transformation where $\hat{\imath}$ lands at $\begin{bmatrix}1\\0\end{bmatrix}$ and $\hat{\jmath}$ lands at $\begin{bmatrix}2\\2\end{bmatrix}$.

What is the determinant of this transformation?





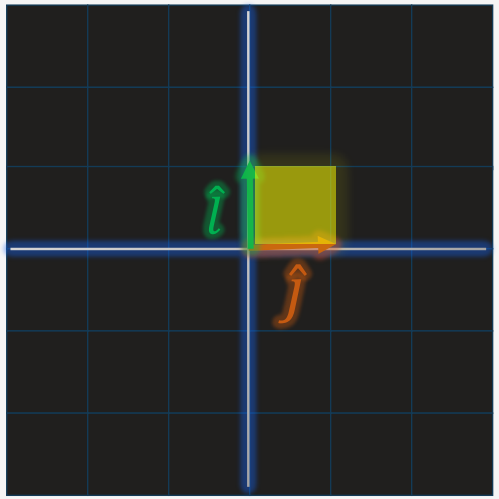
Quick Quiz #3

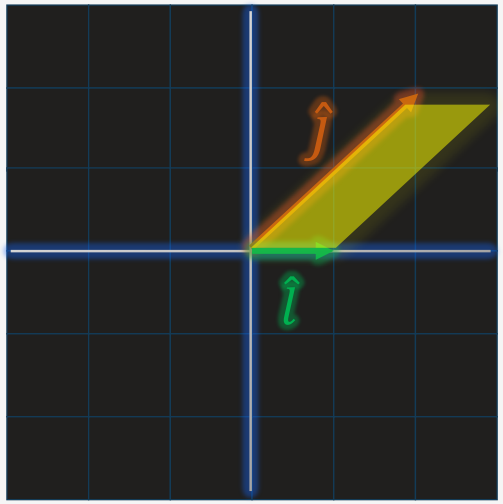
To the right we have a transformation where $\hat{\imath}$ lands at $\begin{bmatrix}1\\0\end{bmatrix}$ and $\hat{\jmath}$ lands at $\begin{bmatrix}2\\2\end{bmatrix}$.

What is the determinant of this transformation?

The determinant is 2.0

```
from numpy.linalg import det
from numpy import array

i_hat = array([1, 0])
j_hat = array([2, 2])

basis = array([i_hat, j_hat]).transpose()

determinant = det(basis)

print(determinant) # prints 2.0
```





Matrix Multiplication

We learned how to multiply a vector and matrix, but what exactly does multiplying two matrices accomplish?

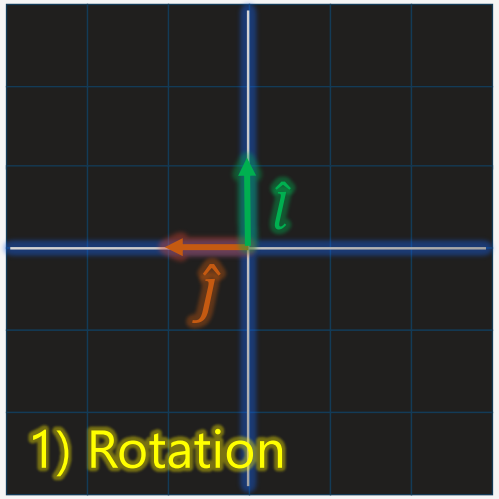
Think of matrix multiplication as applying multiple transformations to a vector space.

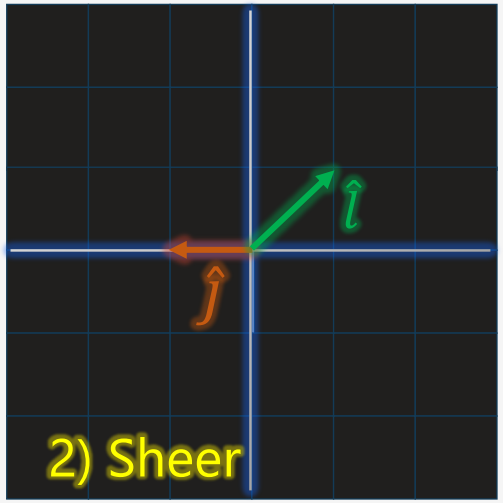
Think of each transformation matrix as a function, where we apply from the inner-most and then outwards.

Here is how we apply a rotation and then a sheer to any vector $\begin{bmatrix} x \\ y \end{bmatrix}$.

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

```
from numpy import array

# Transformation 1
i_hat1 = array([0, 1])
j_hat1 = array([-1, 0])
transform1 = array([i_hat1, j_hat1]).transpose()

# Transformation 2
i_hat2 = array([1, 0])
j_hat2 = array([1, 1])
transform2 = array([i_hat2, j_hat2]).transpose()

# Combine Transformations
combined = transform2.dot(transform1)

print(combined)
```



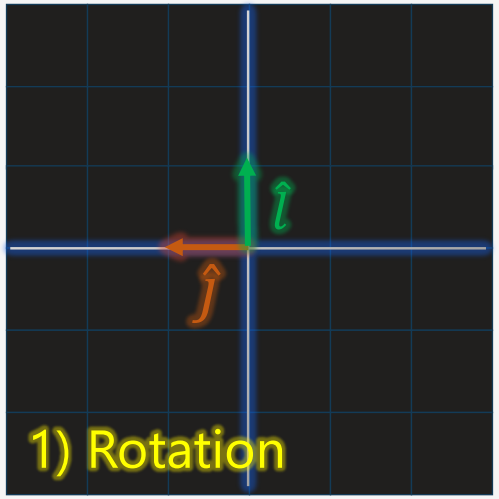


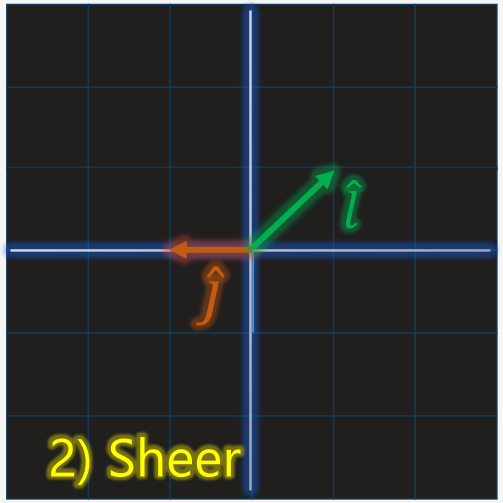
Matrix Multiplication

We typically apply transformations right to left, and the order does matter!

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

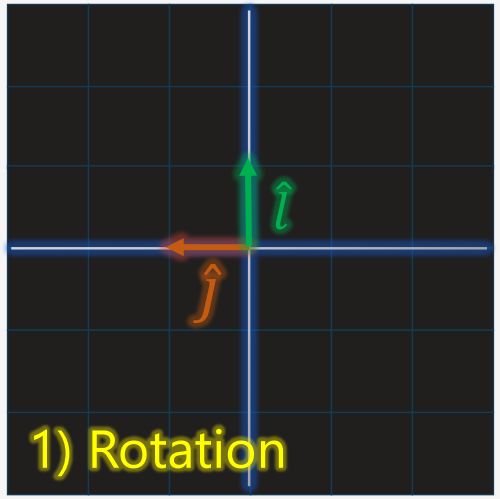
We can chain together as many transformations as we want, but we always apply them from right to left.

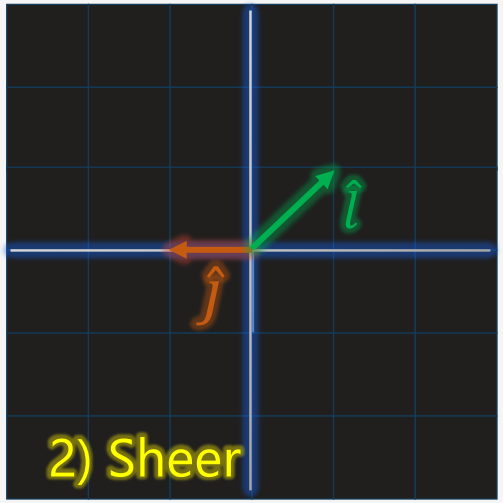




Quick Quiz #4

Can applying two or more transformations be done in a single transformation? Why or why not?





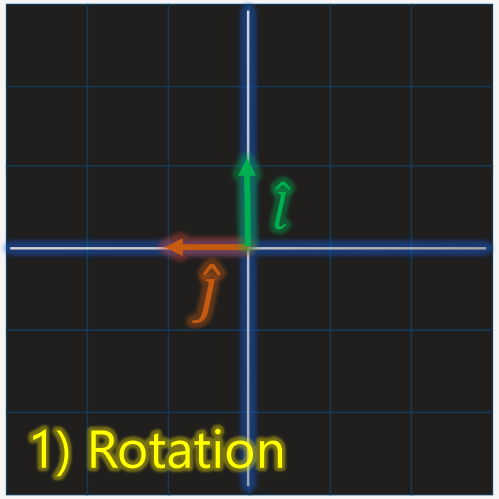
Quick Quiz #4

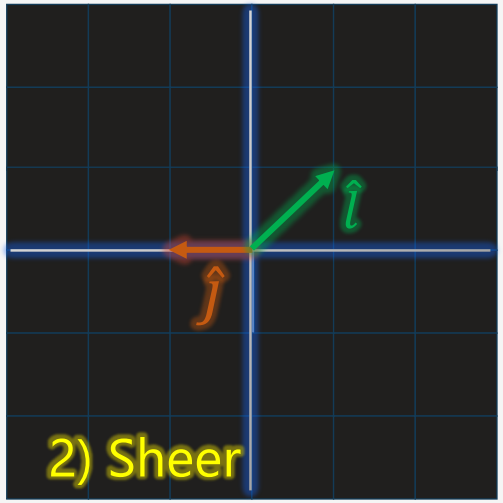
Can applying two or more transformations be done in a single transformation? Why or why not?

We can do any number of transformations as a single transformation.

$\begin{bmatrix}1 & 1\\0 & 1\end{bmatrix}\begin{bmatrix}0 & -1\\1 & 0\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$ is no different than applying $\begin{bmatrix}1 & -1\\1 & 0\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$, as both achieve the same result.

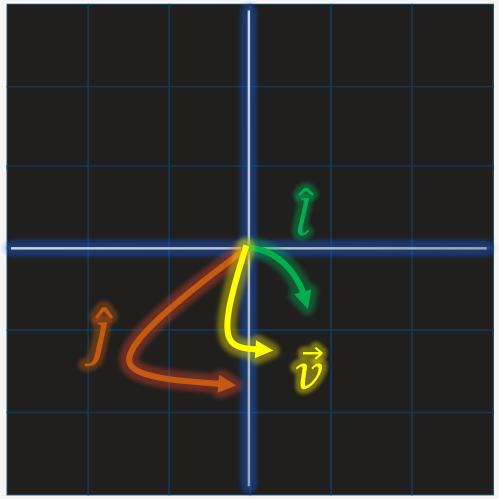
$$\begin{bmatrix}1 & 1\\0 & 1\end{bmatrix}\begin{bmatrix}0 & -1\\1 & 0\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix} = \begin{bmatrix}1 & -1\\1 & 0\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$





Quick Quiz #5

Is this linear transformation to the right valid? Why or why not?

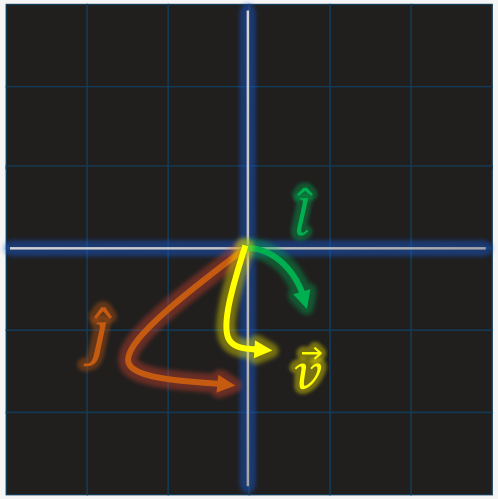


Quick Quiz #5

Is this linear transformation to the right valid? Why or why not?

Linear transformations must be linear and only scale, sheer, rotate, or flip space.

Curvy and variable transformations are not linear and outside the scope of linear algebra.



III. Dot Products

What is the Dot Product?

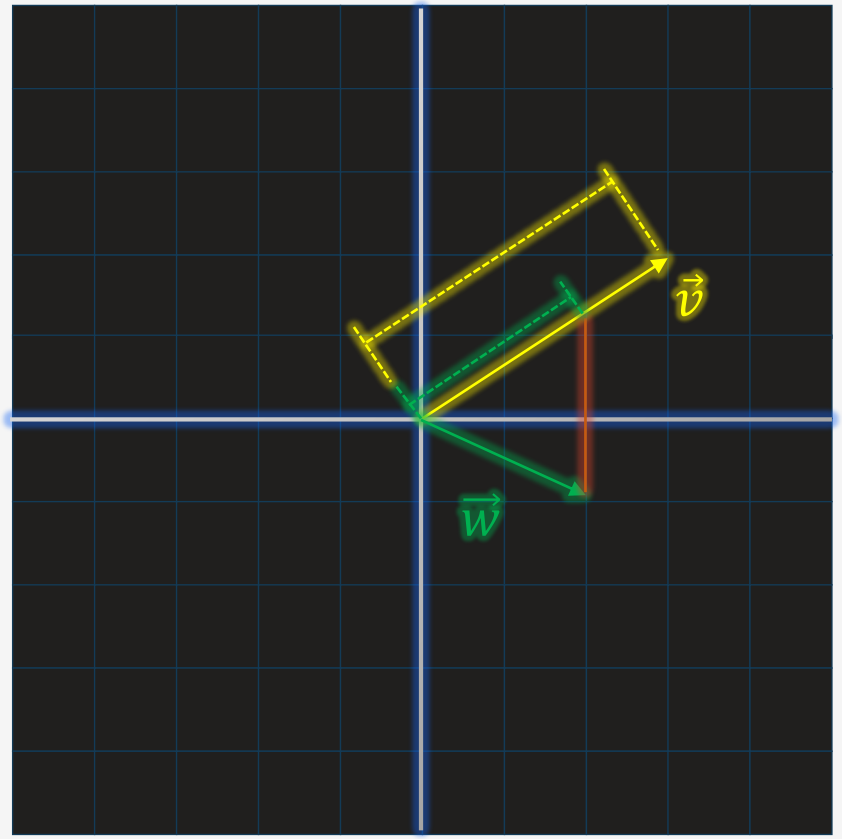
You may have seen the dot product, an operation that does a multiply/add operation between elements in vectors and matrices.

$$\begin{bmatrix}3\\2\end{bmatrix} \cdot \begin{bmatrix}2\\-1\end{bmatrix}$$

$$(3 * 2) + (2 * -1) = 4.0$$

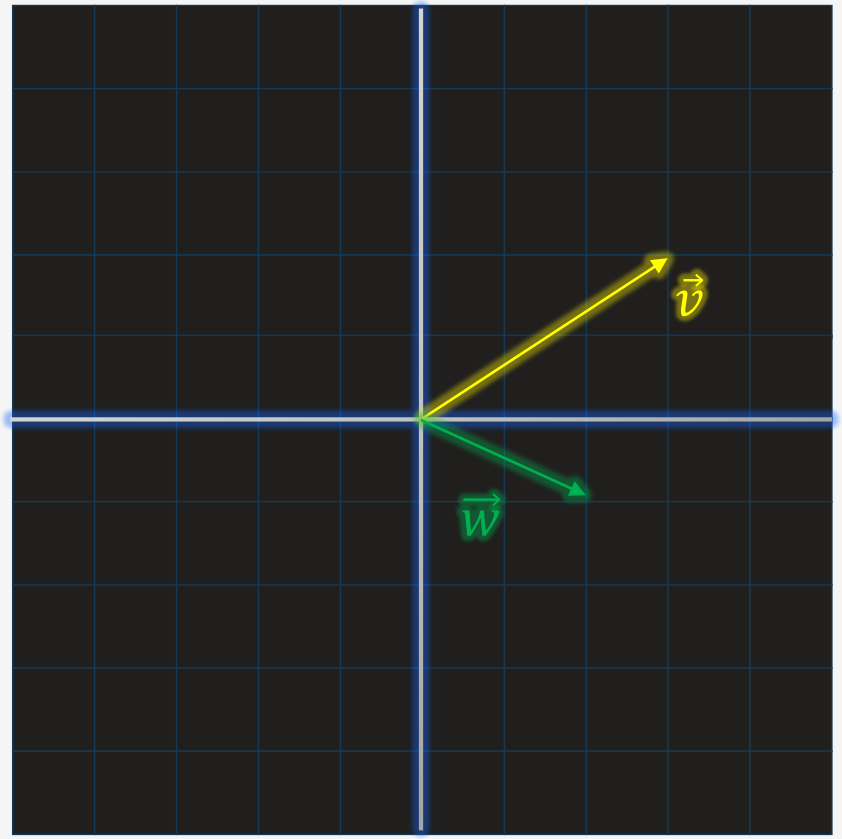
We have already seen the dot product when we combine multiple transformations, but what exactly does it mean?

What is this weird numerical operation accomplishing?



Discovering the Dot Product

Think of the dot product as taking one vector's length, and projecting another vector onto it and multiplying that resulting vector's length.



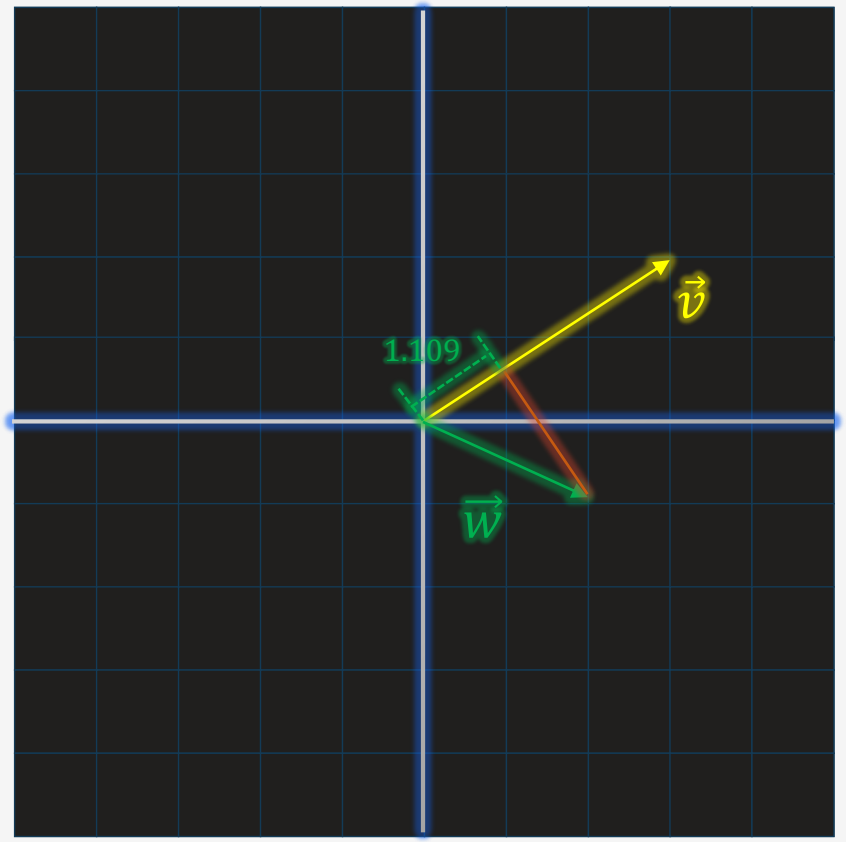
Discovering the Dot Product

Think of the dot product as taking one vector's length, and projecting another vector onto it and multiplying that resulting vector's length.

First, let's project $\vec{w}$ directly onto $\vec{v}$ like so.

This new vector that is linearly dependent to $\vec{v}$ (and portrayed as a dashed green line) has a length of 1.1094003924504583.

You can calculate this using basic algebra to find intersections and trigonometry to compute lengths.



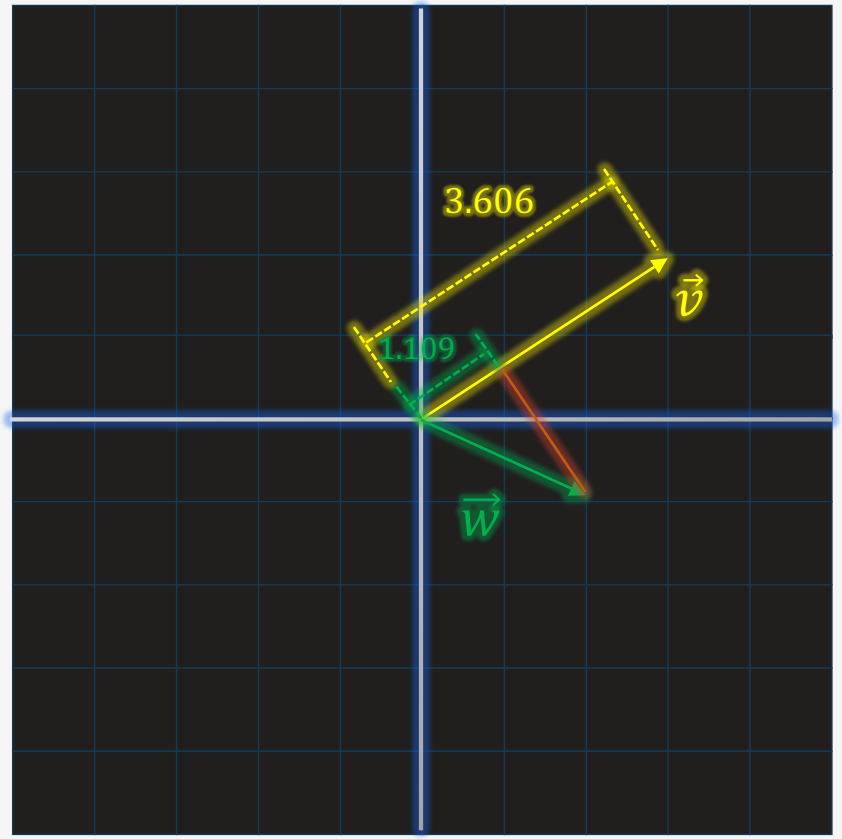
Discovering the Dot Product

This new vector that is linearly dependent to $\vec{v}$ (and portrayed as a dashed green line) has a length of 1.1094003924504583.

Now let's calculate the length of $\vec{v}$ which is 3.606.

The dot product is the multiplication of these two length values:

$3.606 \times 1.1094003924504583 \approx 4.0$



Discovering the Dot Product

The dot product is the multiplication of these two length values:

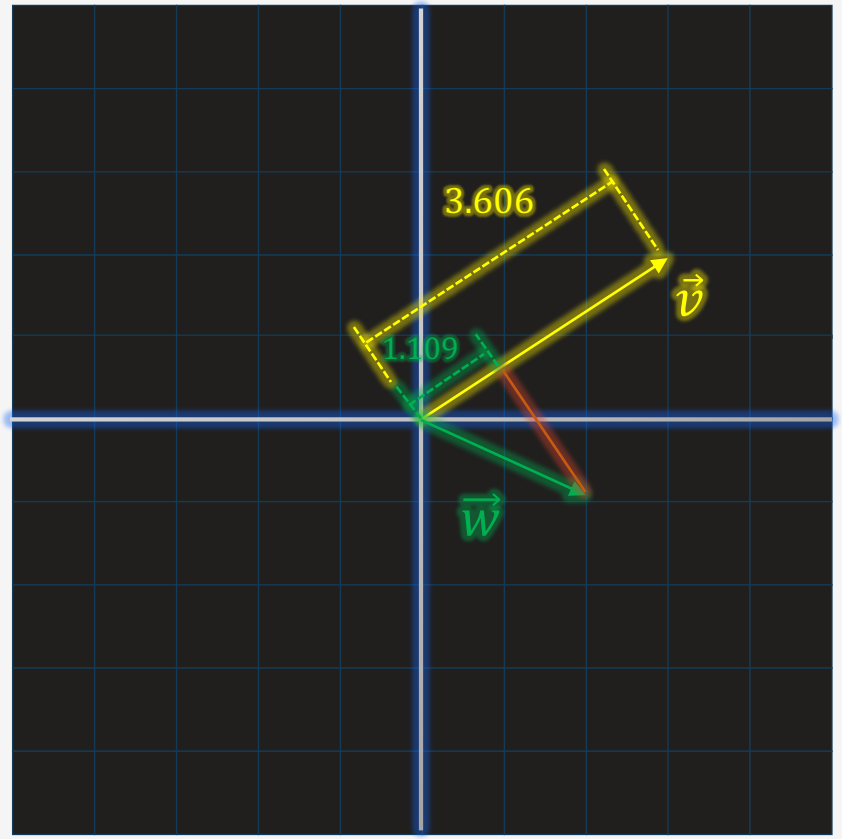
$3.606 \times 1.1094003924504583 \approx 4.0$

This dot product formula is the shorthand we use to compute it:

$$\begin{bmatrix} a \\ b \end{bmatrix} \cdot \begin{bmatrix} c \\ d \end{bmatrix} = ac + bd$$

$$\begin{bmatrix} 3 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ -1 \end{bmatrix} = (3 * 2) + (2 * -1) = 4.0$$

```
from numpy import array

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

dot_product = v.dot(w)

print(dot_product) # prints 4.0
```



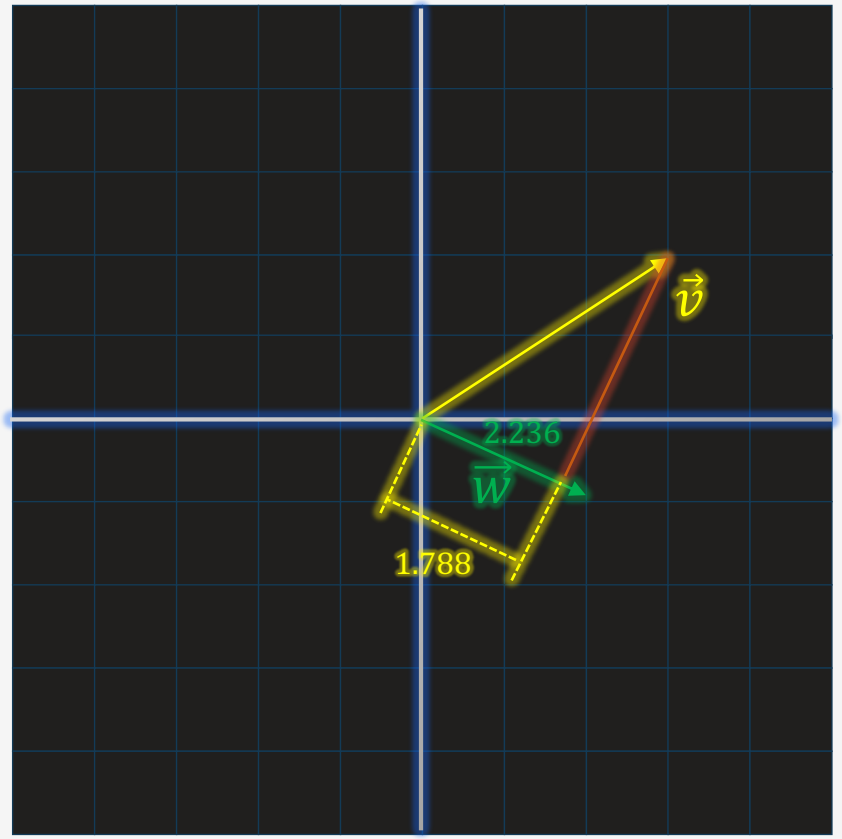
Discovering the Dot Product

Note that geometrically this also works in the reverse direction.

If we projected $\vec{v}$ onto $\vec{w}$ instead of $\vec{w}$ onto $\vec{v}$, we would still end up with the same dot product.

$$1.7888543819998317 * 2.236 = 4.0$$

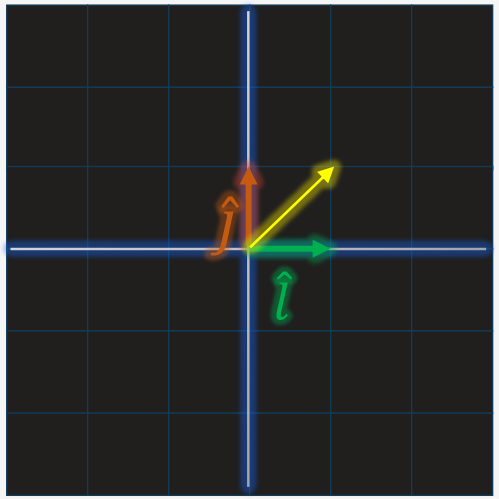
```
from numpy import array

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

dot_product = w.dot(v)

print(dot_product) # prints 4.0
```



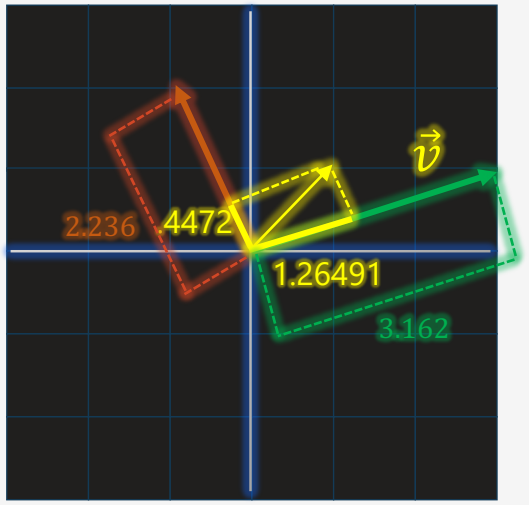
What About Matrix Dot Products?

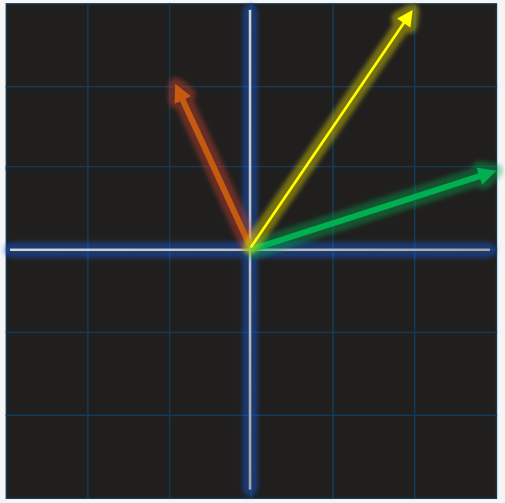
The dot product is the means of executing linear transformations, by means of several projections.

We use dot products to transform with matrices, as we have seen many examples already.




```
from numpy import array, dot

i_hat = array([-1, 2])
j_hat = array([3, 1])

basis = array([i_hat, j_hat]).transpose()

v = array([1, 1])

dot_product = basis.dot(v)

print(dot_product) # prints [2, 3]
```





Over and Down! Over and Down!

To avoid doing a whole bunch of projection work, it is much easier to calculate matrix dot products by hand using the "over and down!" pattern.

$$\begin{bmatrix} 3 & 1 \\ 2 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 & 0 \\ 3 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 3*1+1*3 & 3*2+1*1 & 3*0+1*1 \\ 2*1+0*3 & 2*2+0*1 & 2*0+0*1 \end{bmatrix}$$

$$= \begin{bmatrix} 6 & 7 & 1 \\ 2 & 4 & 0 \end{bmatrix}$$

```
from numpy import array

v = array([[3, 1],
           [2, 0]])

w = array([[1, 2, 0],
           [3, 1, 1]])

dot_product = v.dot(w)

print(dot_product)
```

Over and Down! Over and Down!

To avoid doing a whole bunch of projection work, it is much easier to calculate matrix dot products by hand using the "over and down!" pattern.

$$\begin{bmatrix} \mathbf{3} & \mathbf{1} \\ 2 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{1} & 2 & 0 \\ \mathbf{3} & 1 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{3*1+1*3} & 3*2+1*1 & 3*0+1*1 \\ 2*1+0*3 & 2*2+0*1 & 2*0+0*1 \end{bmatrix}$$

$$= \begin{bmatrix} \mathbf{6} & 7 & 1 \\ 2 & 4 & 0 \end{bmatrix}$$

```
from numpy import array

v = array([[3, 1],
           [2, 0]])

w = array([[1, 2, 0],
           [3, 1, 1]])

dot_product = v.dot(w)

print(dot_product)
```

Over and Down! Over and Down!

To avoid doing a whole bunch of projection work, it is much easier to calculate matrix dot products by hand using the "over and down!" pattern.

$$\begin{bmatrix} \mathbf{3} & \mathbf{1} \\ 2 & 0 \end{bmatrix} \begin{bmatrix} 1 & \mathbf{2} & 0 \\ 3 & \mathbf{1} & 1 \end{bmatrix} = \begin{bmatrix} 3*1+1*3 & \mathbf{3*2+1*1} & 3*0+1*1 \\ 2*1+0*3 & 2*2+0*1 & 2*0+0*1 \end{bmatrix}$$

$$= \begin{bmatrix} 6 & \mathbf{7} & 1 \\ 2 & 4 & 0 \end{bmatrix}$$

```
from numpy import array

v = array([[3, 1],
           [2, 0]])

w = array([[1, 2, 0],
           [3, 1, 1]])

dot_product = v.dot(w)

print(dot_product)
```

Over and Down! Over and Down!

To avoid doing a whole bunch of projection work, it is much easier to calculate matrix dot products by hand using the "over and down!" pattern.

$$\begin{bmatrix} \mathbf{3} & \mathbf{1} \\ 2 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 & \mathbf{0} \\ 3 & 1 & \mathbf{1} \end{bmatrix} = \begin{bmatrix} 3*1+1*3 & 3*2+1*1 & \mathbf{3*0+1*1} \\ 2*1+0*3 & 2*2+0*1 & 2*0+0*1 \end{bmatrix}$$

$$= \begin{bmatrix} 6 & 7 & \mathbf{1} \\ 2 & 4 & 0 \end{bmatrix}$$

```
from numpy import array

v = array([[3, 1],
           [2, 0]])

w = array([[1, 2, 0],
           [3, 1, 1]])

dot_product = v.dot(w)

print(dot_product)
```

Over and Down! Over and Down!

To avoid doing a whole bunch of projection work, it is much easier to calculate matrix dot products by hand using the "over and down!" pattern.

$$\begin{bmatrix} 3 & 1 \\ \mathbf{2} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{1} & 2 & 0 \\ \mathbf{3} & 1 & 1 \end{bmatrix} = \begin{bmatrix} 3*1+1*3 & 3*2+1*1 & 3*0+1*1 \\ \mathbf{2*1+0*3} & 2*2+0*1 & 2*0+0*1 \end{bmatrix}$$

$$= \begin{bmatrix} 6 & 7 & 1 \\ \mathbf{2} & 4 & 0 \end{bmatrix}$$

```
from numpy import array

v = array([[3, 1],
           [2, 0]])

w = array([[1, 2, 0],
           [3, 1, 1]])

dot_product = v.dot(w)

print(dot_product)
```

Over and Down! Over and Down!

To avoid doing a whole bunch of projection work, it is much easier to calculate matrix dot products by hand using the "over and down!" pattern.

$$\begin{bmatrix} 3 & 1 \\ \mathbf{2} & \mathbf{0} \end{bmatrix} \begin{bmatrix} 1 & \mathbf{2} & 0 \\ 3 & \mathbf{1} & 1 \end{bmatrix} = \begin{bmatrix} 3*1+1*3 & 3*2+1*1 & 3*0+1*1 \\ 2*1+0*3 & \mathbf{2*2+0*1} & 2*0+0*1 \end{bmatrix}$$

$$= \begin{bmatrix} 6 & 7 & 1 \\ 2 & \mathbf{4} & 0 \end{bmatrix}$$

```
from numpy import array

v = array([[3, 1],
           [2, 0]])

w = array([[1, 2, 0],
           [3, 1, 1]])

dot_product = v.dot(w)

print(dot_product)
```

Over and Down! Over and Down!

To avoid doing a whole bunch of projection work, it is much easier to calculate matrix dot products by hand using the "over and down!" pattern.

$$\begin{bmatrix} 3 & 1 \\ \mathbf{2} & \mathbf{0} \end{bmatrix} \begin{bmatrix} 1 & 2 & \mathbf{0} \\ 3 & 1 & \mathbf{1} \end{bmatrix} = \begin{bmatrix} 3*1+1*3 & 3*2+1*1 & 3*0+1*1 \\ 2*1+0*3 & 2*2+0*1 & \mathbf{2*0+0*1} \end{bmatrix}$$

$$= \begin{bmatrix} 6 & 7 & 1 \\ 2 & 4 & \mathbf{0} \end{bmatrix}$$

Multiply and sum each row with each respective column, and this will quickly and efficiently apply a series of transformations,

```
from numpy import array

v = array([[3, 1],
           [2, 0]])

w = array([[1, 2, 0],
           [3, 1, 1]])

dot_product = v.dot(w)

print(dot_product)
```

IV. Systems of Linear Equations and Inverse Matrices

Solving Systems of Equations

Let's say you have a system of equations like this:

$$4x + 2y + 4z = 44$$

$$5x + 3y + 7z = 56$$

$$9x + 3y + 6z = 72$$

We need to solve for x, y, and z. First let's declare the equations above as three matrices:

$$A = \begin{bmatrix} 4 & 2 & 4 \\ 5 & 3 & 7 \\ 9 & 3 & 6 \end{bmatrix} \qquad B = \begin{bmatrix} 44 \\ 56 \\ 72 \end{bmatrix} \qquad X = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Solving Systems of Equations.

We need to transform matrix A with some other matrix X that will result in matrix B.

$$AX = B$$

$$\begin{bmatrix} 4 & 2 & 4 \\ 5 & 3 & 7 \\ 9 & 3 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 44 \\ 56 \\ 72 \end{bmatrix}$$

We can isolate X by multiplying each side with the inverse of matrix A, which we will call A^{-1}.

$$A^{-1}AX = A^{-1}B$$

$$X = A^{-1}B$$

Inverse Matrixes

To calculate the **inverse** of matrix A, we should probably use a computer rather than searching for solutions by hand (this is tedious!)

To the right we use NumPy to calculate the inverse of matrix A, and we get this result:

$$A^{-1} = \begin{bmatrix} -\frac{1}{2} & 0 & \frac{1}{3} \\ 5.5 & -2 & \frac{4}{3} \\ -2 & 1 & \frac{1}{3} \end{bmatrix}$$

```
from numpy import array
from numpy.linalg import inv

A = array([
   [4, 2, 4],
   [5, 3, 7],
   [9, 3, 6]
])

print(inv(A))
```

Inverse Matrixes

When we multiply a matrix by its inverse, we get an identity matrix which cancels out and effectively identifies x, y, an z.

$$A = \begin{bmatrix} 4 & 2 & 4 \\ 5 & 3 & 7 \\ 9 & 3 & 6 \end{bmatrix} \qquad A^{-1} = \begin{bmatrix} -\frac{1}{2} & 0 & \frac{1}{3} \\ 5.5 & -2 & \frac{4}{3} \\ -2 & 1 & \frac{1}{3} \end{bmatrix}$$

$$A * A^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
from numpy import array
from numpy.linalg import inv


A = array([
   [4, 2, 4],
   [5, 3, 7],
   [9, 3, 6]
])

identity = inv(A).dot(A)

print(identity)
```

Solving Systems of Equations

$$A = \begin{bmatrix} 4 & 2 & 4 \\ 5 & 3 & 7 \\ 9 & 3 & 6 \end{bmatrix} \qquad B = \begin{bmatrix} 44 \\ 56 \\ 72 \end{bmatrix} \qquad A^{-1} = \begin{bmatrix} -\frac{1}{2} & 0 & \frac{1}{3} \\ 5.5 & -2 & \frac{4}{3} \\ -2 & 1 & \frac{1}{3} \end{bmatrix} \qquad X = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Now we have all the pieces we need to solve for X!

$$A^{-1}B = X$$

$$\begin{bmatrix} -\frac{1}{2} & 0 & \frac{1}{3} \\ 5.5 & -2 & \frac{4}{3} \\ -2 & 1 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} 44 \\ 56 \\ 72 \end{bmatrix} = \begin{bmatrix} 2 \\ 34 \\ -8 \end{bmatrix}$$

So $x = 2$, $y = 34$, and $z = -8$.

Inverse matrices and this method of solving systems of equations is used for linear programming and other areas.

Solving Systems of Equations

```
from numpy import array
from numpy.linalg import inv

# 4x + 2y + 4z = 44
# 5x + 3y + 7z = 56
# 9x + 3y + 6z = 72

A = array([
   [4, 2, 4],
   [5, 3, 7],
   [9, 3, 6]
])

B = array([
   44,
   56,
   72
])

X = inv(A).dot(B)

print(X)
```

Quiz #6: Solve the System of Equations

What is x, y, and z?

$$3x + 1y + 0z = 54$$
$$2x + 4y + 1z = 12$$
$$3x + 1y + 8z = 6$$

Quiz #6: Solve the System of Equations

```
from numpy import array
from numpy.linalg import inv

# 3x + 1y + 0z  = 54
# 2x + 4y + 1z = 12
# 3x + 1y + 8z = 6

A = array([
   [3, 1, 0],
   [2, 4, 1],
   [3, 1, 8]
])

B = array([
   54,
   12,
   6
])

X = inv(A).dot(B)

print(X)
```

$$3x + 1y + 0z = 54$$
$$2x + 4y + 1z = 12$$
$$3x + 1y + 8z = 6$$

$$A = \begin{bmatrix} 3 & 1 & 0 \\ 2 & 4 & 1 \\ 3 & 1 & 8 \end{bmatrix} \qquad B = \begin{bmatrix} 54 \\ 12 \\ 6 \end{bmatrix} \qquad X = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$AX = B$$
$$A^{-1}AX = A^{-1}B$$
$$X = A^{-1}B$$

$$X = \begin{bmatrix} 19.8 \\ -5.4 \\ -6 \end{bmatrix}$$

V. Eigenvectors and Eigenvalues

Matrix Decomposition

Matrix decomposition is breaking up a matrix into its basic components, much like factoring numbers (e.g. 10 can be factored to 2 x 5).

Matrix decomposition is helpful for tasks like finding inverse matrices, calculating determinants, as well as linear regression.

There are many ways to decompose a matrix, but the most common method is **eigendecomposition**, which is often used for machine learning and Principal Component Analysis (PCA).

At this level, just know eigendecomposition is helpful for breaking up a matrix into components that are easier to work with in different machine learning tasks.

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$$

$$\begin{bmatrix} -0.464 \\ 6.464 \end{bmatrix} \qquad \begin{bmatrix} 0.806 & 0.343 \\ 0.59 & -0.939 \end{bmatrix}$$

λ v

Eigendecomposition

If we have a square matrix A, it has the following eigenvalue equation:

$$Av = \lambda v$$

If A is the parent matrix, it has eigenvector v and eigenvalue λ (lambda).

There is one eigenvector and eigenvalue for each dimension of the parent matrix, and not all matrices can be decomposed into an eigenvector and eigenvalue.

Sometimes complex (imaginary) numbers will result!

To the right is how we calculate eigenvectors and eigenvalues in NumPy for a given matrix A.

```
from numpy import array, diag
from numpy.linalg import eig, inv

A = array([
   [1, 2],
   [4, 5]
])

eigenvals, eigenvecs = eig(A)

print("EIGENVALUES")
print(eigenvals)
print("\nEIGENVECTORS")
print(eigenvecs)
```

```
EIGENVALUES
[-0.46410162  6.46410162]

EIGENVECTORS
[[-0.80689822 -0.34372377]
 [ 0.59069049 -0.9390708 ]]
```

Eigendecomposition

So how do we rebuild matrix A from the eigenvectors and eigenvalues?

Recall this equation:

$$Av = \lambda v$$

We need to make a few tweaks to the formula to reconstruct A:

$$A = Q\Lambda Q^{-1}$$

Where Q is the eigenvectors, Λ is the eigenvalues in diagonal form, and Q^{-1} is the inverse matrix of Q.

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$$

$$\begin{bmatrix} -0.464 \\ 6.464 \end{bmatrix} \qquad \begin{bmatrix} 0.806 & 0.343 \\ 0.59 & -0.939 \end{bmatrix}$$

λ v

$$Q = \begin{bmatrix} 0.806 & 0.343 \\ 0.59 & -0.939 \end{bmatrix}$$

$$\Lambda = \begin{bmatrix} -0.464 & 0 \\ 0 & 6.464 \end{bmatrix}$$

$$Q^{-1} = \begin{bmatrix} -0.977 & 0.357 \\ -0.614 & -0.839 \end{bmatrix}$$

Rebuilding from Eigenvectors/Eigenvalues

Rebuilding a matrix from the eigenvectors and eigenvalues requires a few transformational steps as shown to the right.

You can dot product all the components to rebuild matrix A again.

$$Q = \begin{bmatrix} 0.806 & 0.343 \\ 0.59 & -0.939 \end{bmatrix}$$

$$\Lambda = \begin{bmatrix} -0.464 & 0 \\ 0 & 6.464 \end{bmatrix}$$

$$Q^{-1} = \begin{bmatrix} -0.977 & 0.357 \\ -0.614 & -0.839 \end{bmatrix}$$

$$A = Q\Lambda Q^{-1} = \begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$$

```
from numpy import array, diag
from numpy.linalg import eig, inv

A = array([
   [1, 2],
   [4, 5]
])

eigenvals, eigenvecs = eig(A)

print("EIGENVALUES")
print(eigenvals)
print("\nEIGENVECTORS")
print(eigenvecs)

print("\nREBUILD MATRIX")
Q = eigenvecs
R = inv(Q)

L = diag(eigenvals)
B = Q.dot(L).dot(R)

print(B)
```

Appendix

Resources

3Blue1Brown – Linear Algebra

https://www.youtube.com/watch?v=fNk_zzaMoSs&list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab

Python for Data Analysis (O'Reilly)

https://learning.oreilly.com/library/view/python-for-data/9781491957653