LINEAR ALGEBRA

ARIHARASUDHAN

LINEAR ALGEBRA

It is the study of vector spaces, lines and planes, and some mappings that are required to perform the linear transformations.

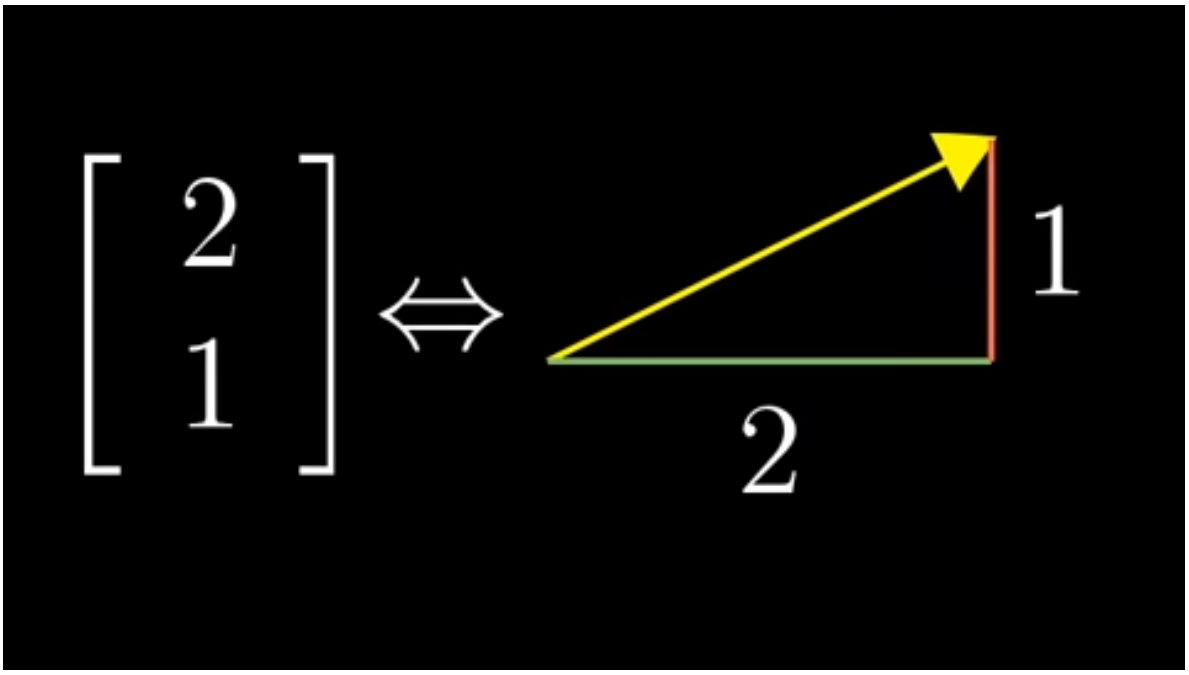
VECTOR

There are distinct but related ideas about vectors. It can be seen from different perspectives such as that of a Physics Student, Computer Student and a Mathematician.

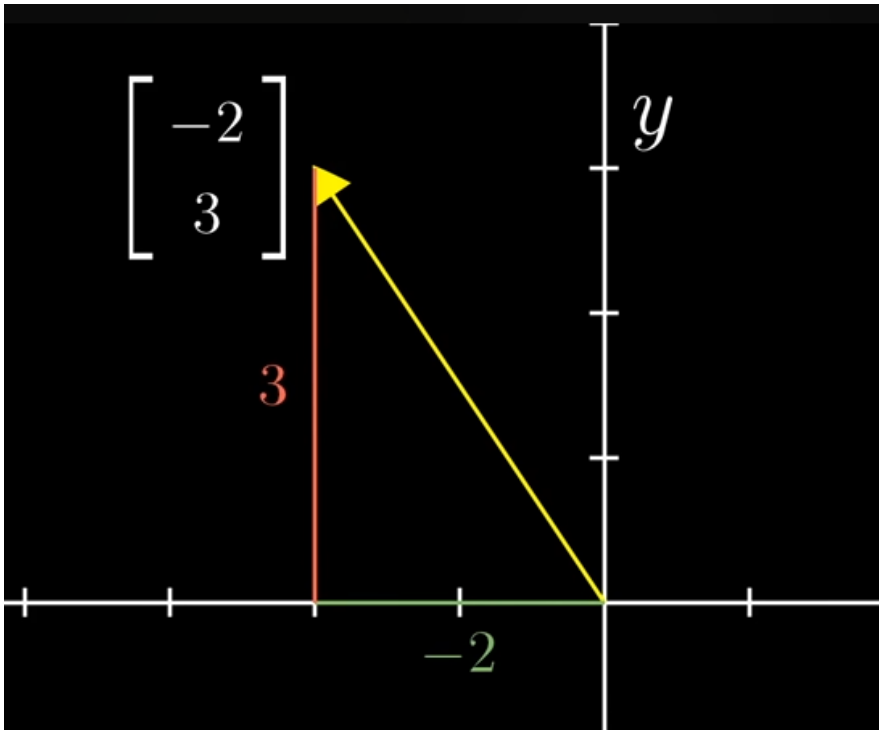
★ For a Physics Student, Vectors are arrows pointing in the space. A vector is defined by it’s length and the direction it is pointing.

★ For a Computer Student, Vectors are the ordered lists of numbers. A House can be modeled with the area and the price.

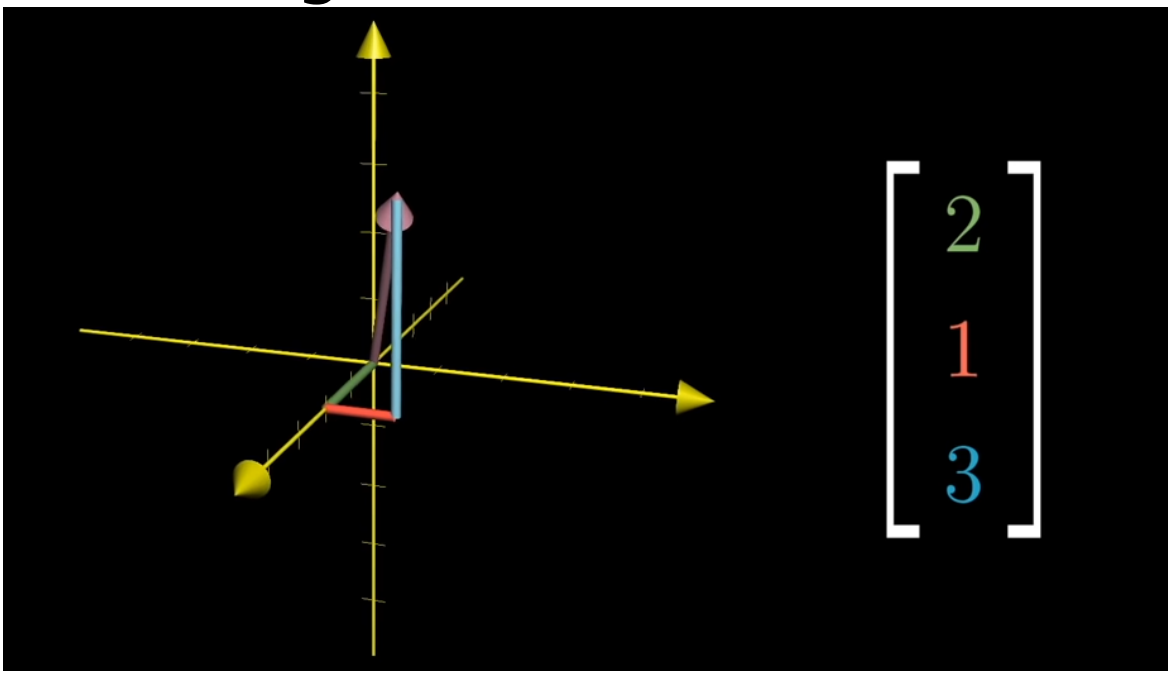
★A Mathematician generalizes these ideas. A 2D Vector can be represented like the following.



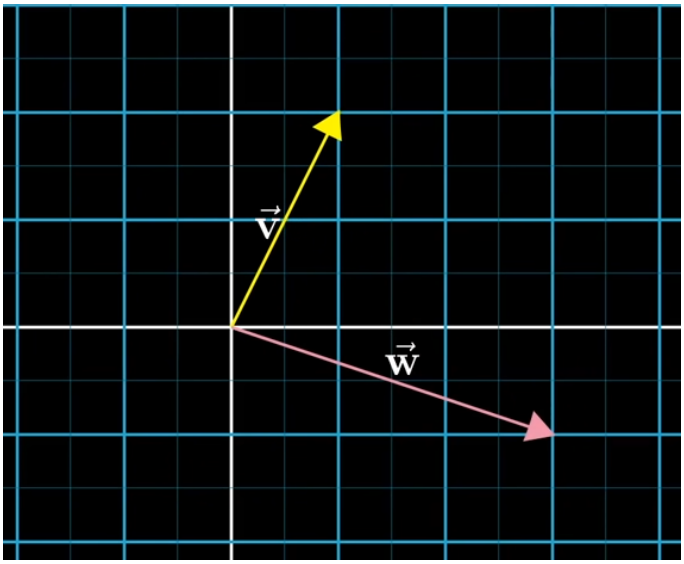
Negative values can also be represented this way.



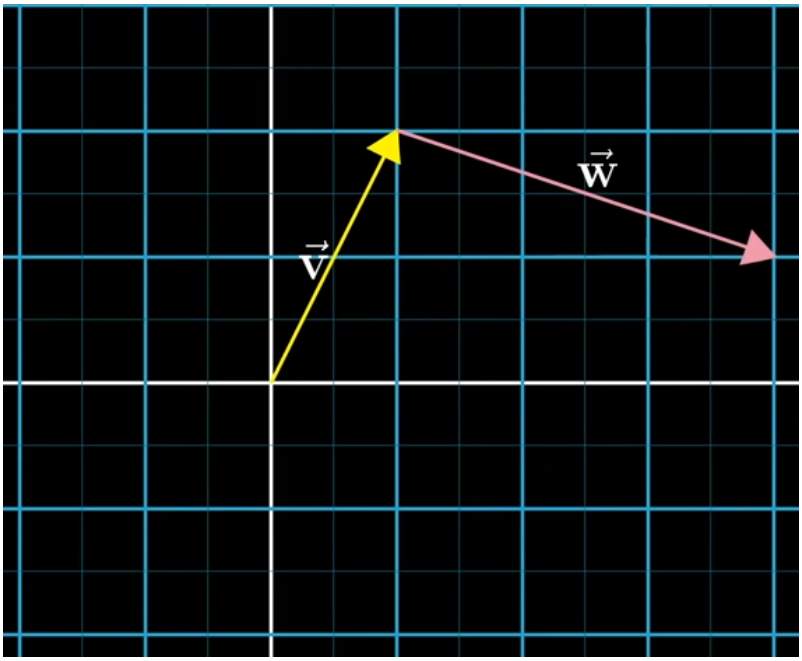
A 3D Vector can be represented using the following notation.



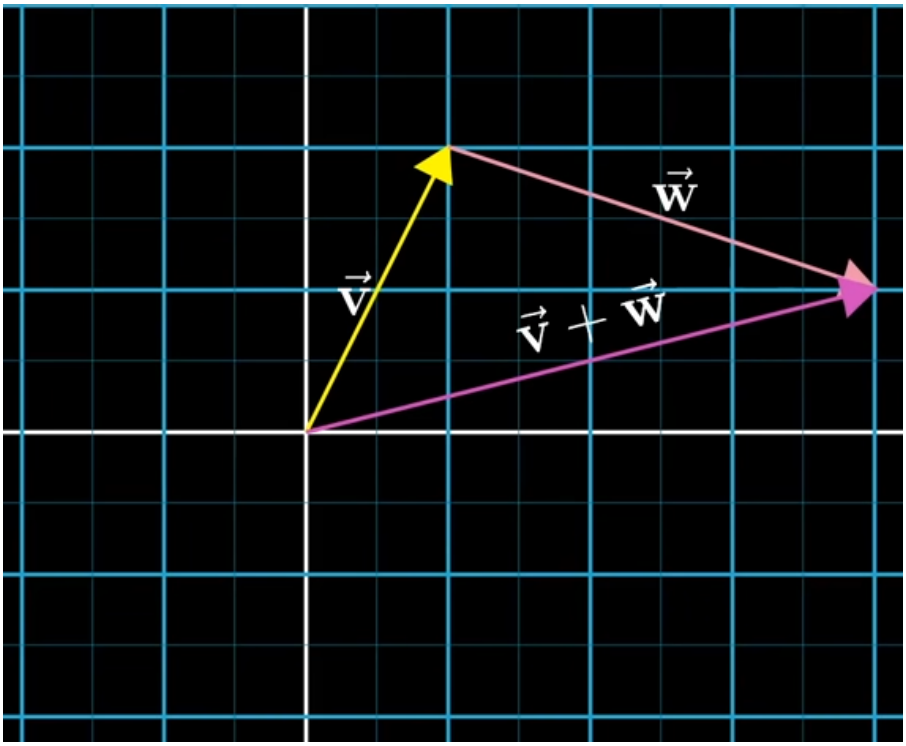
Vector addition is a quite exciting one.



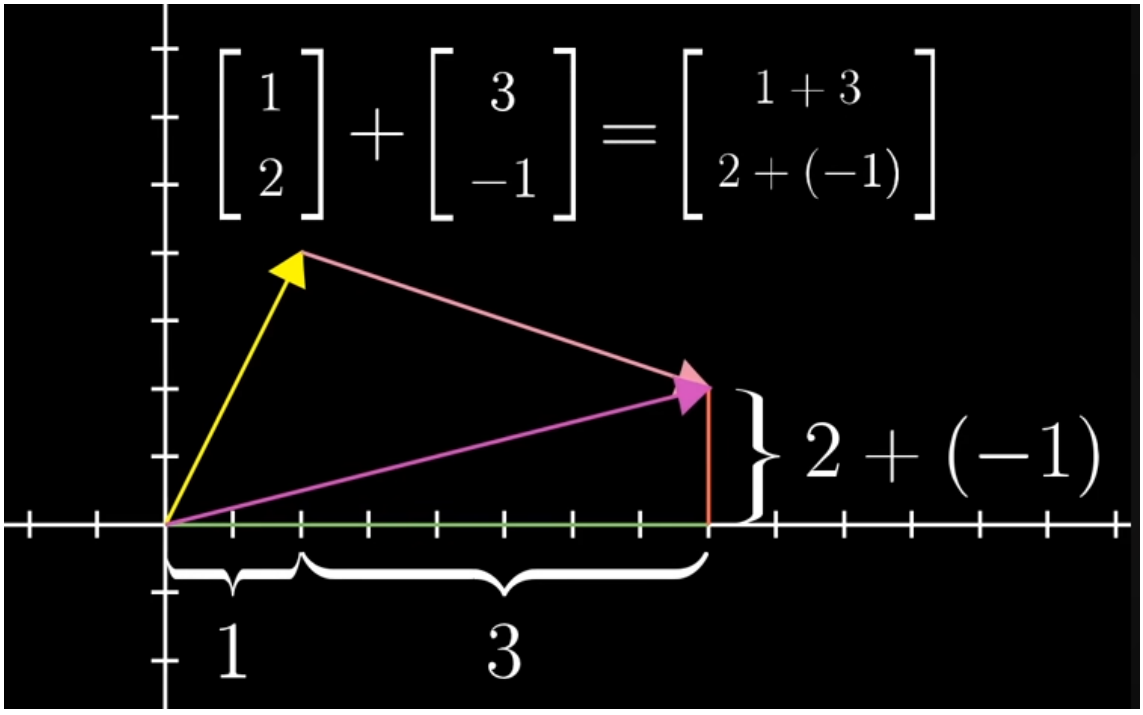
We are provided with two vectors v and w. Move vector w towards the direction of v.



We can find the sum in the following way.



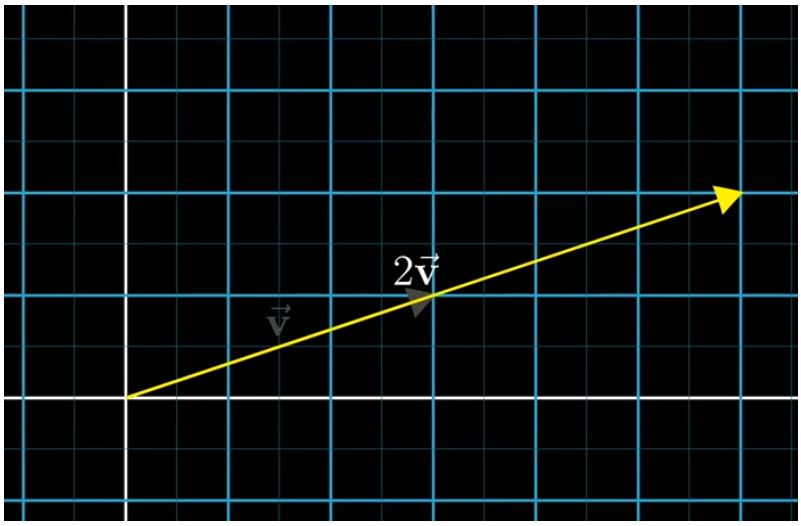
Usually, it’s so explicit while using the matrix addition.



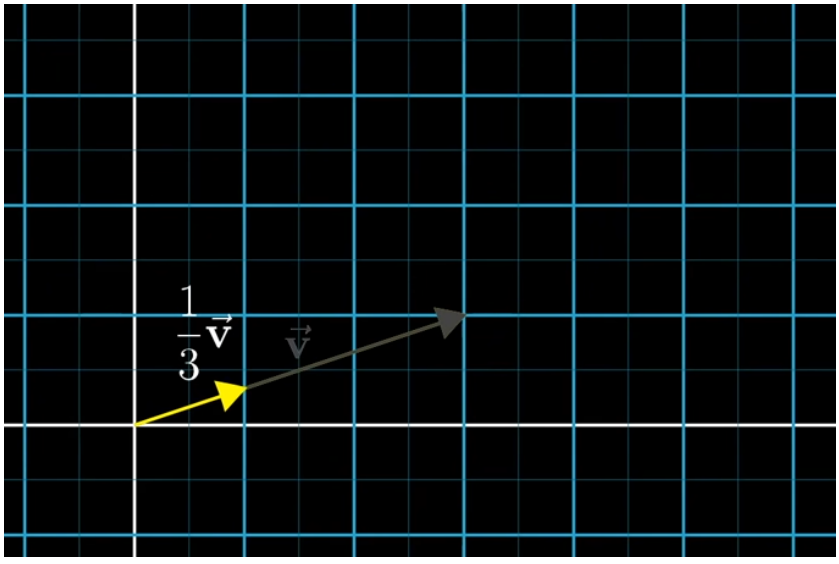
SCALING

We can perform stretching, squishing and reversing of direction over the vector. It is called scaling.

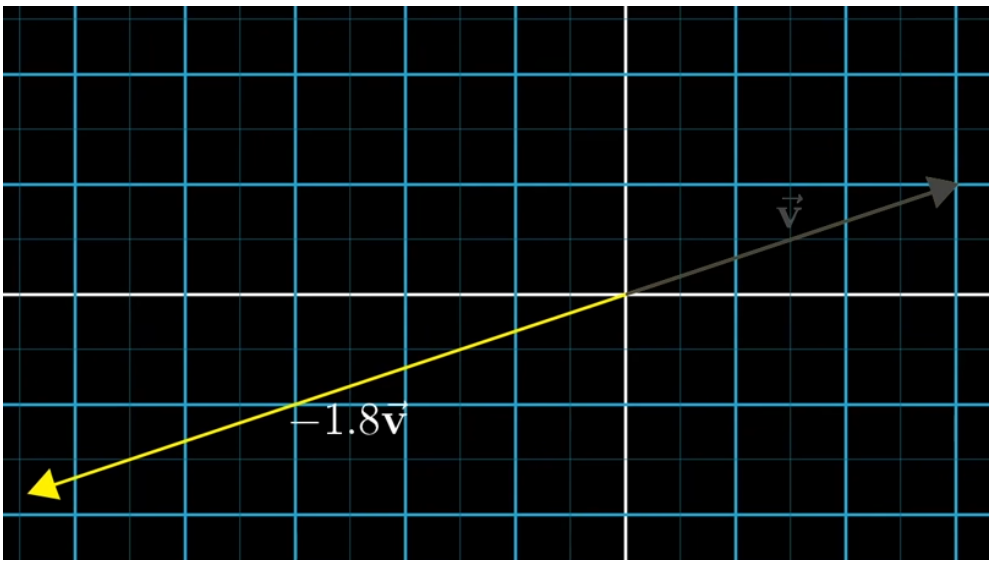
@STRETCHING



@SQUISHING



@REVERSING



As we can observe, the each and every values in the matrix representation of the vector is multiplied by the value provided. This value is called SCALAR.

$$2 \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2x \\ 2y \end{bmatrix}$$

Moreover, it is a concept of broadcasting in NumPy in which the entity of different dimension is added with the matrix.

```
1  from numpy import array
2  # define array
3  a = array([1, 2, 3])
4  print(a)
5  # define scalar
6  b = 2
7  print(b)
8  # broadcast
9  c = a + b
10 print(c)
```

Each values are added with the provided scalar value.

```
[1 2 3]
2
[3 4 5]
```

We can do the same for an n dimensional array also.

```
# broadcast scalar to two-dimensional array
from numpy import array
# define array
A = array([
[1, 2, 3],
[1, 2, 3]])
print(A)
# define scalar
b = 2
print(b)
# broadcast
C = A + b
print(C)
```

Moreover, broadcasting one dimensional array to a two-dimensional array is also possible.

```
# broadcast one-dimensional array to 2D array
from numpy import array
# define two-dimensional array
A = array([
[1, 2, 3],
[1, 2, 3]])
print(A)
# define one-dimensional array
b = array([1, 2, 3])
print(b)
# broadcast
C = A + b
print(C)
```

VECTOR IN NUMPY

A vector is a tuple of one or more values called scalars. Vectors are built from components, which are ordinary numbers. You can think of a vector as a list of numbers, and vector algebra as operations performed on the numbers in the list.

@CREATING A VECTOR

```
# create a vector
from numpy import array
# define vector
v = array([1, 2, 3])
print(v)
```

@VECTOR MULTIPLICATION

```
from numpy import array
a = array([1, 2, 3])
b = array([1, 2, 3])
# multiply vectors
c = a * b   #ElementWise
print(c) # [1 4 9]
```

@VECTOR DIVISION

```
from numpy import array
a = array([1, 2, 3])
b = array([1, 2, 3])
# divide vectors
c = a / b   #ElementWise
print(c) # [1 1 1]
```

@DOT PRODUCT

```
from numpy import array
a = array([1, 2, 3])
b = array([1, 2, 3])
# DOT Product
c = a.dot(b)
print(c) #14
```

We can calculate the sum of the multiplied elements of two vectors of the same length to give a scalar.

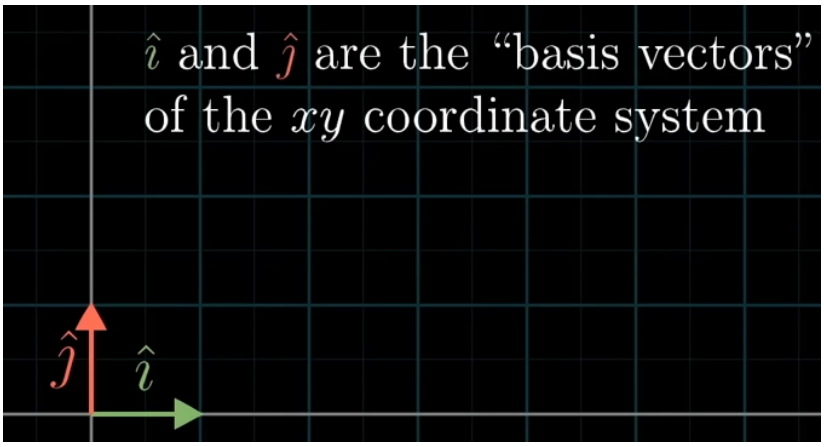
$$c = (a_1 \times b_1 + a_2 \times b_2 + a_3 \times b_3)$$

SCALAR MULTIPLICATION

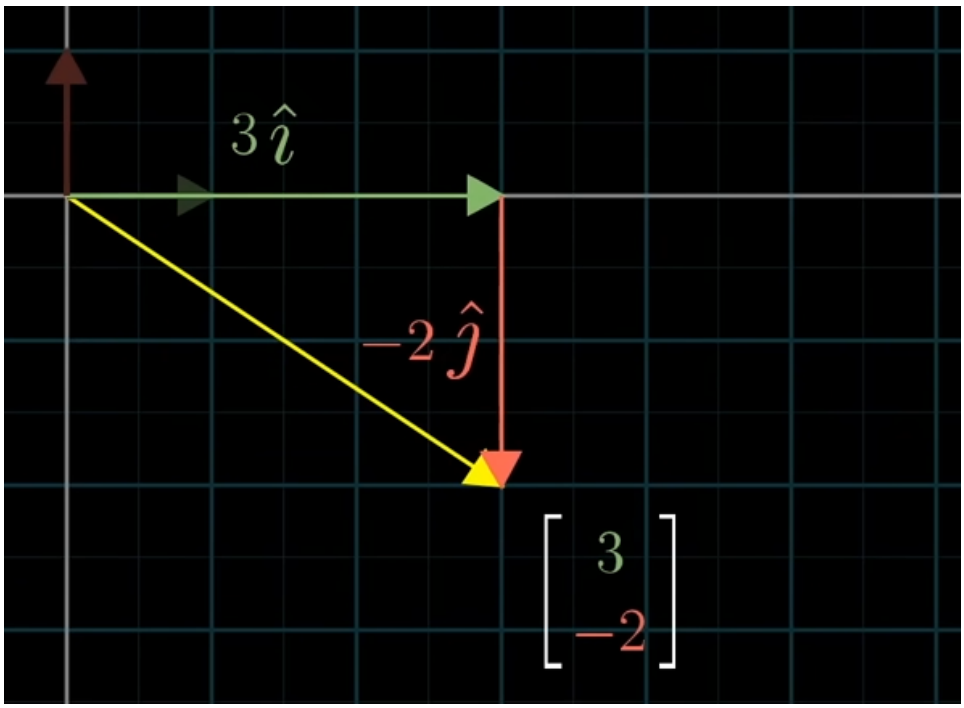
```
from numpy import array
a = array([1, 2, 3])
b = 0.5
c = a*b
print(c) # [0.5 1 1.5]

```

A Vector can be multiplied by a scalar value.

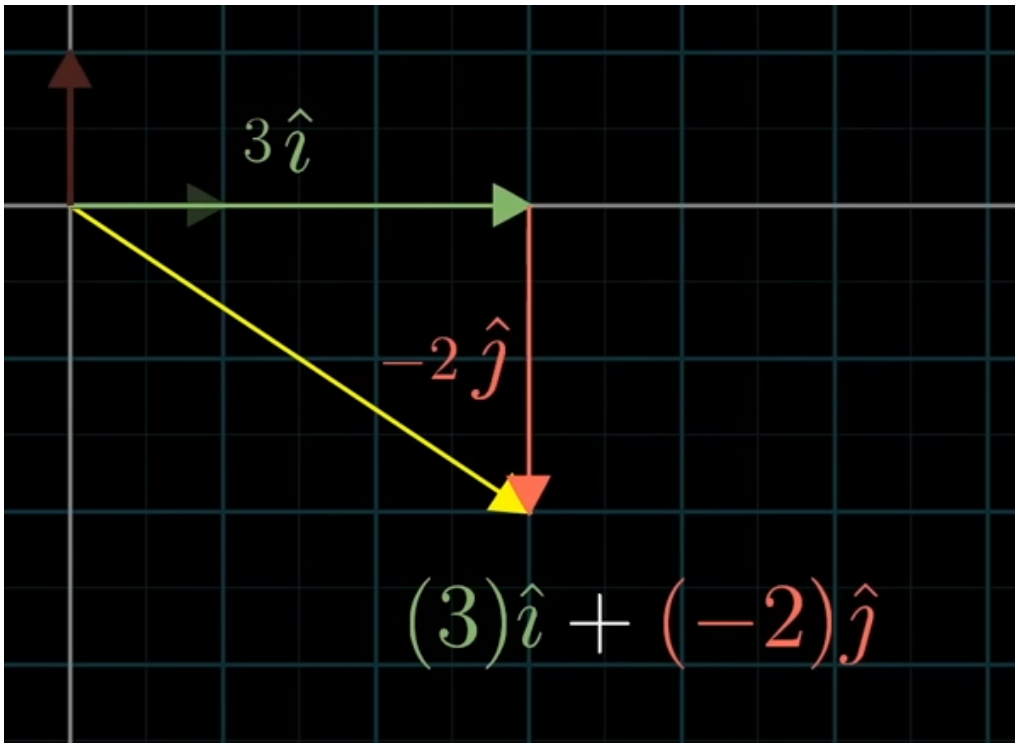
BASIS VECTORS



In the x and y coordinates, let's assume there are two vectors namely î and ĵ.
These are of length 1. Any vectors can be represented in terms of these two.

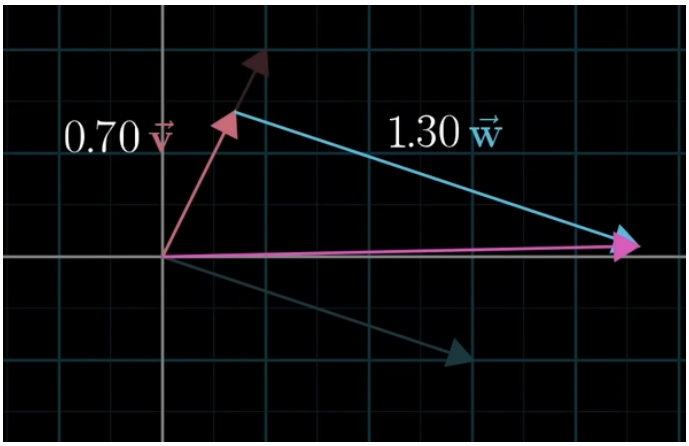


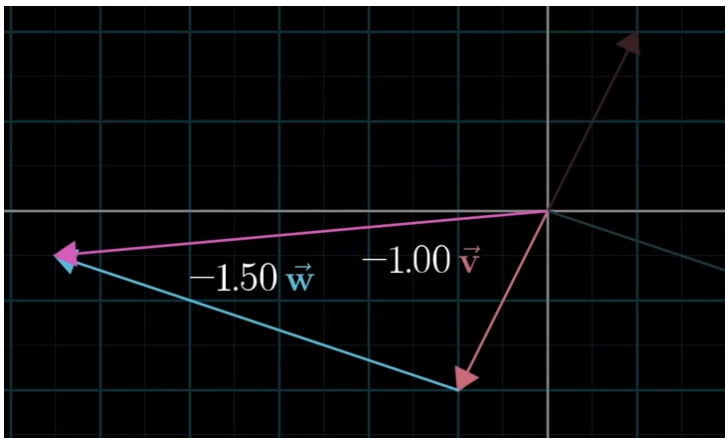
The addition of these vectors can be found as,

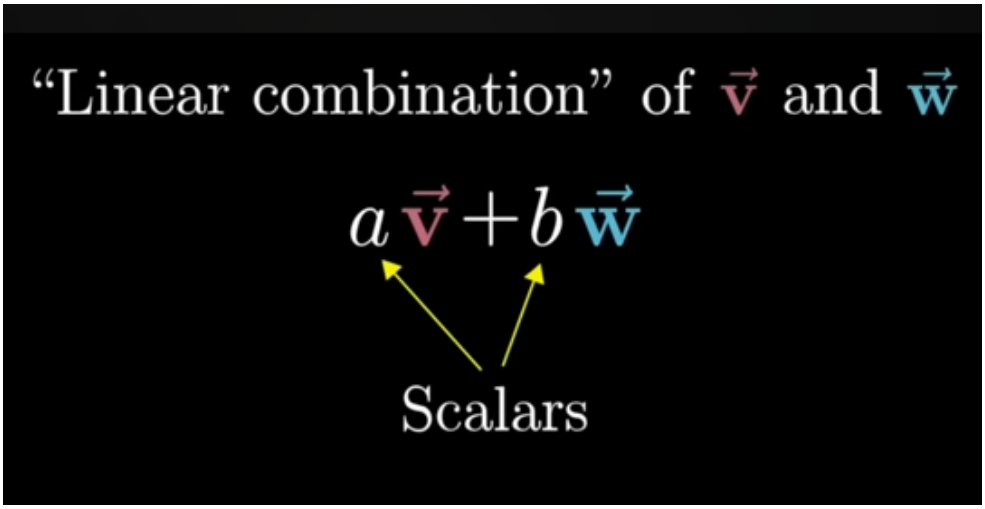


LINEAR COMBINATION

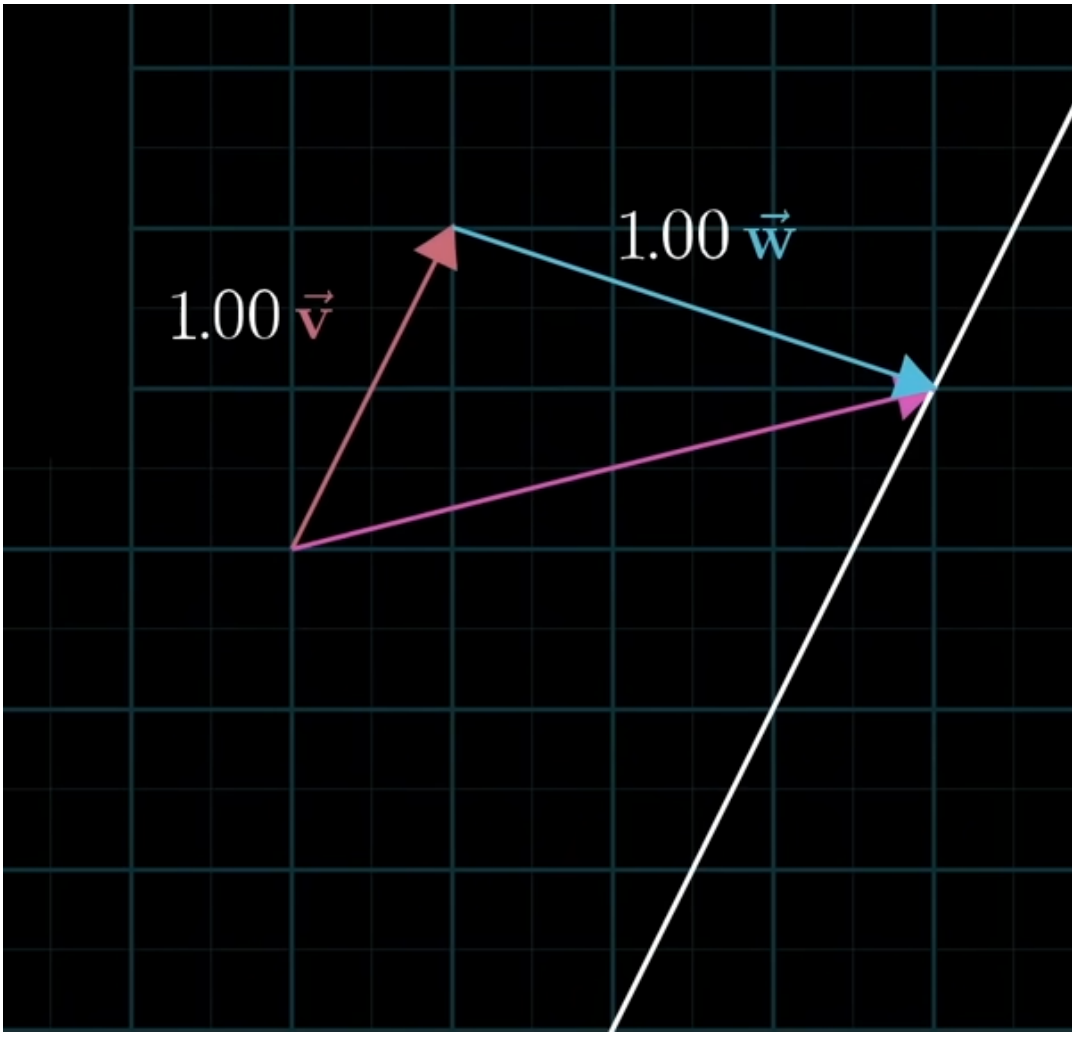
If we use different scalar values for two vectors and compute their scalar product and the vector sum too, we could observe that we form a lot and lot of combinations. We can possibly reach every possible vectors by choosing different scalar values.





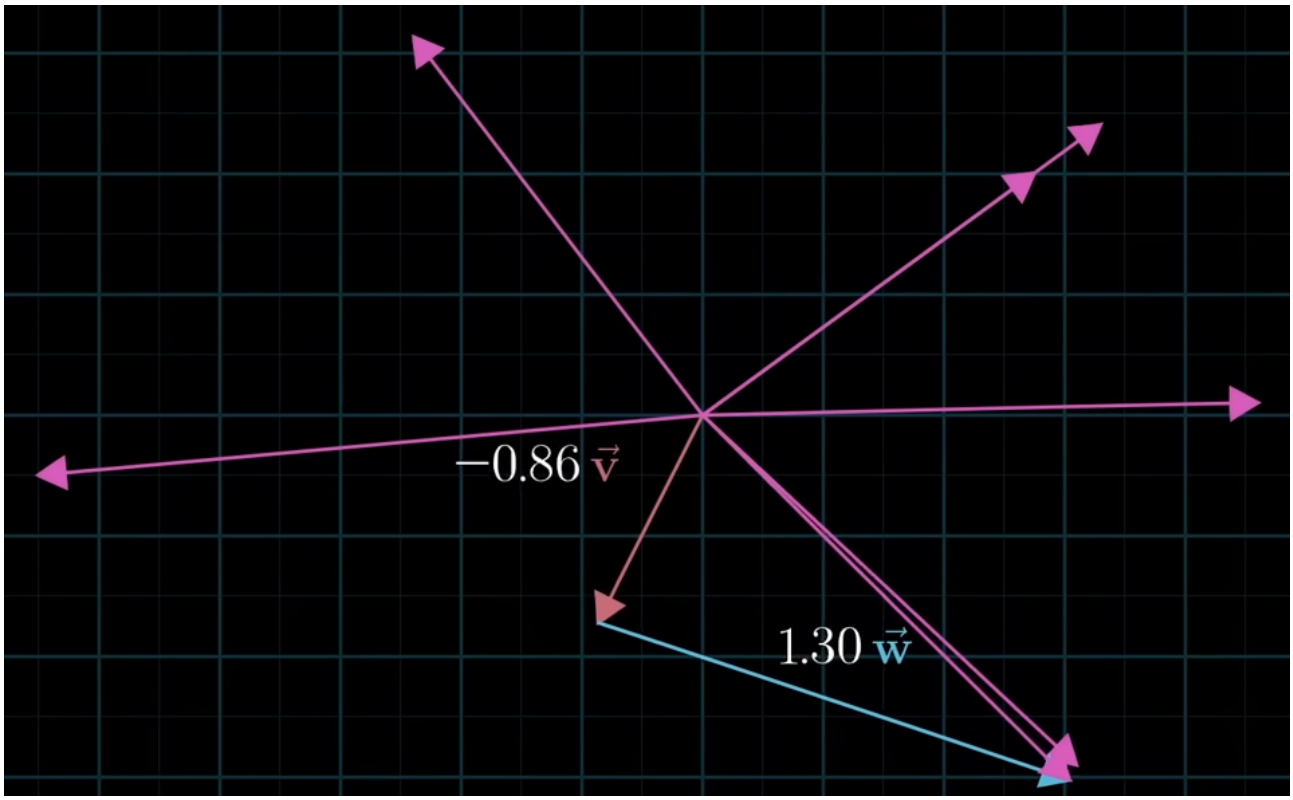


A linear combination is an expression constructed from a set of terms by multiplying each term by a constant and adding the results (A linear combination of *x* and *y* would be any expression of the form *ax* + *by*, where *a* and *b* are constants)



SPAN

A Span of two vectors is simply a set of all the linear combinations of them. The span of a set of vectors, also called linear span, is the linear space formed by all the vectors that can be written as linear combinations of the vectors belonging to the given set.



VECTOR NORMS

Calculating the length or magnitude of vectors is often required.

@ VECTOR NORM

The length of the vector is referred to as the vector norm or the vector's magnitude. The length of a vector is a non-negative number that describes the extent of the vector in space, and is sometimes referred to as the vector's magnitude or the norm.

@ VECTOR L^1 NORM

The L1 norm is calculated as the sum of the absolute vector values, where the absolute value of a scalar uses the notation |a1 |.

$$||v||^1 = |a1 | + |a2 | + |a3 |$$

```
1  from numpy import array
2  from numpy.linalg import norm
3  a = array([1, 2, 3])
4  print(a)
5  l1 = norm(a, 1)
6  print(l1)
```

```
Console 1/A
In [1]: runfile('/home/ari-pt7127/num.py', wdir='/home/ari-pt7127')
[1 2 3]
6.0
```

@ VECTOR L^2 NORM

The L2 norm calculates the distance of the vector coordinate from the origin of the vector space. As such, it is also known as the Euclidean norm as it is calculated as the Euclidean distance from the origin. The result is a positive distance value. The L2 norm is calculated as the square root of the sum of the squared vector values.

```
1  from numpy import array
2  from numpy.linalg import norm
3  # define vector
4  a = array([1, 2, 3])
5  print(a)
6  # calculate norm
7  l2 = norm(a)
8  print(l2)
```

```
Console 1/A

In [2]: runfile('/home/ari-pt7127/num.py', wdir='/home/ari-pt7127')
[1 2 3]
3.7416573867739413
```

@ VECTOR MAX NORM

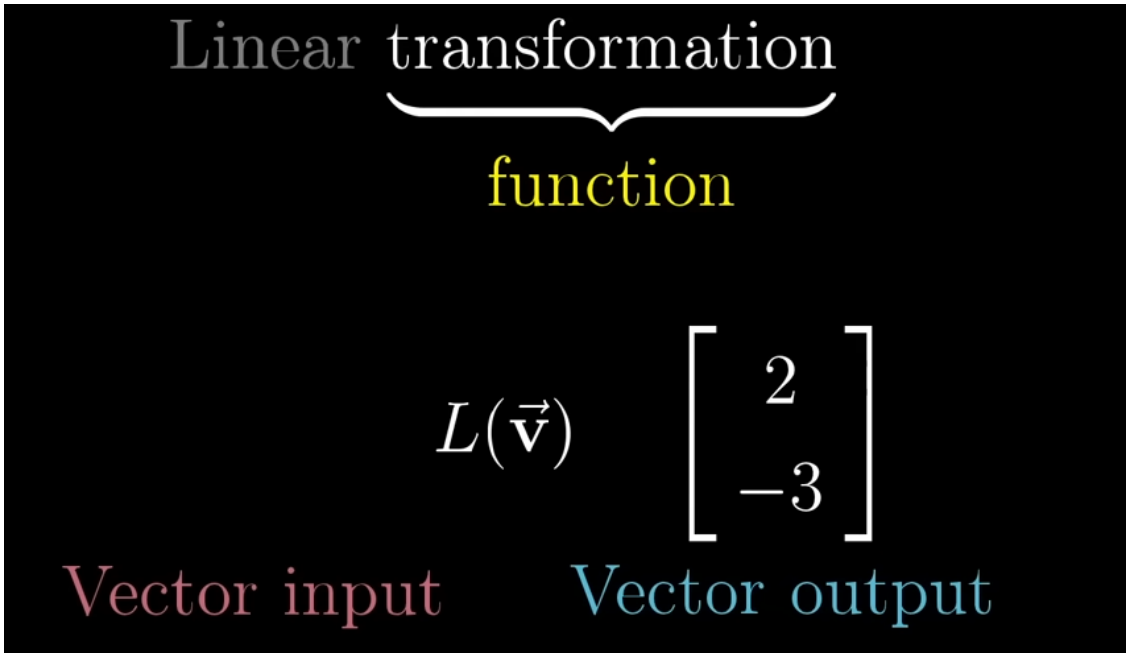
The max norm is calculated as returning the maximum value of the vector, hence the name. The order has to be given as inf.

```
3  from numpy.linalg import norm
4  # define vector
5  a = array([1, 2, 3])
6  print(a)
7  # calculate norm
8  maxnorm = norm(a, inf)
9  print(maxnorm)
```

```
Console 1/A

In [3]: runfile('/home/ari-pt7127/num.py', wdir='/home/ari-pt7127')
[1 2 3]
3.0
```

LINEAR TRANSFORMATIONS

The word transformation is simply defines something like a function



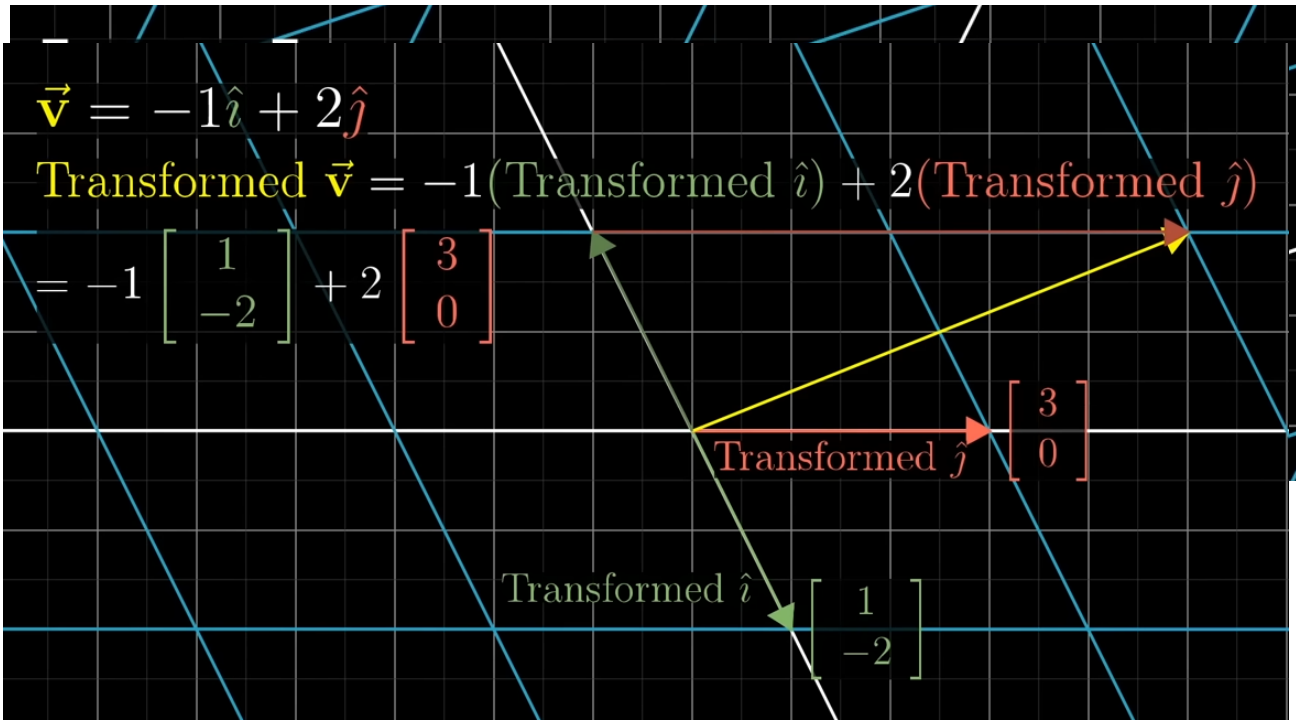
which accepts an input and transforms it into something else. A Vector Input is transformed into A Vector Output using a Transformation. If it is Linear Transformation, all the lines should remain lines. There should be no curves. The Origin must remain fixed. If we have a matrix representation of two vectors say v and w.

It is to be multiplied by the vector [x y] to form a transformation. The applied transformation makes a change in the space.

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

Where all the intuition is

The transformation makes a change in the space. When the space moves, let the grid lines be parallel and evenly spaced.

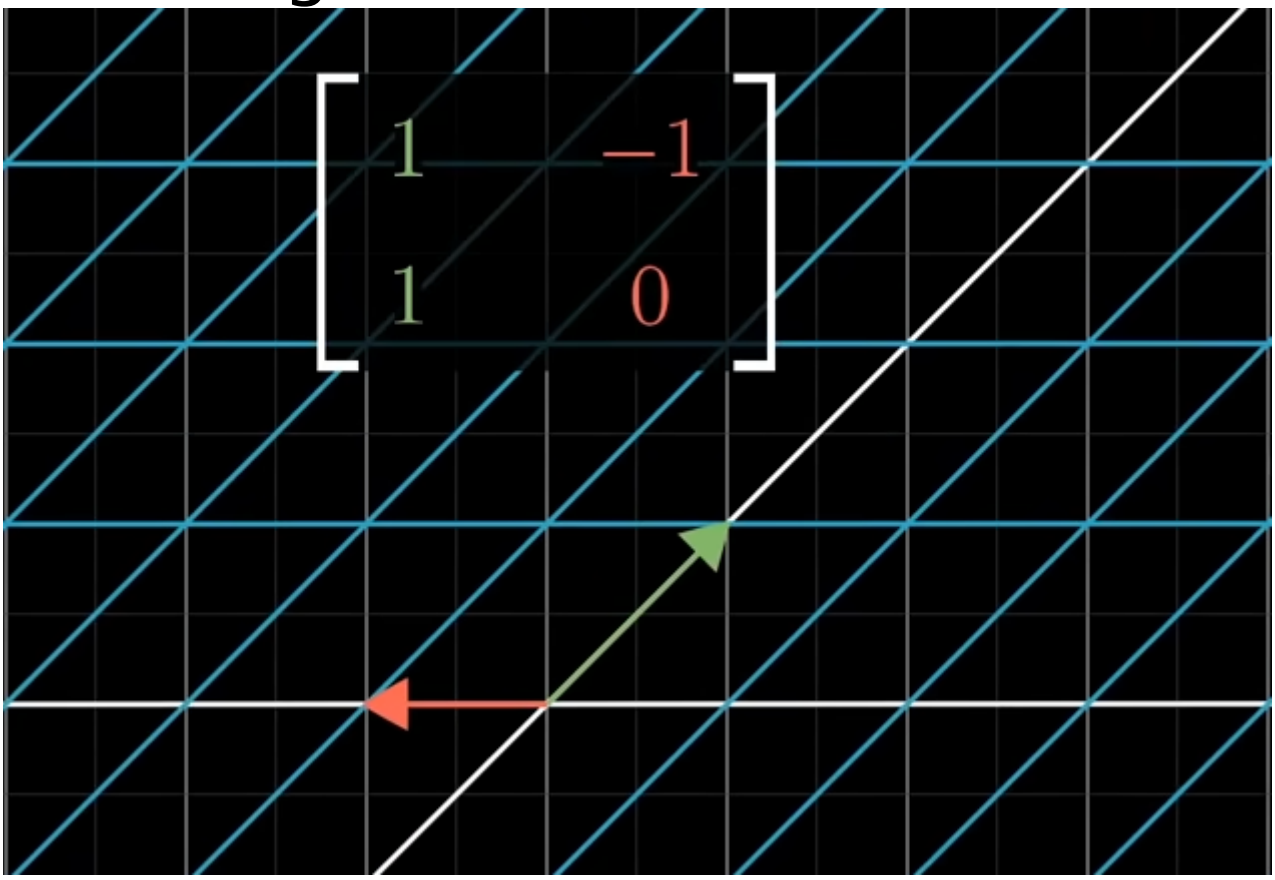


After a transformation the î and ĵ lie somewhere rather than where they were.

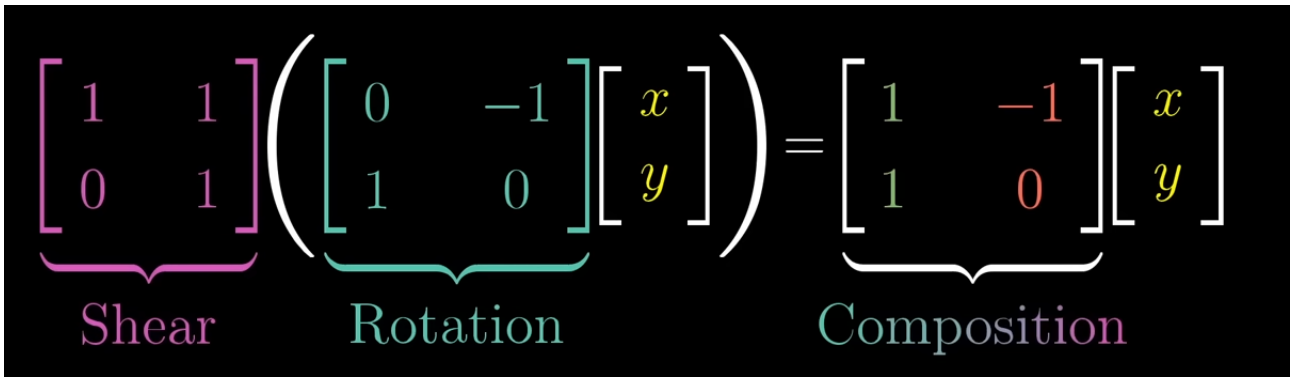
COMPOSITIONS

If we apply two transformations say a rotation of 90º and a shear, the $\hat{i}$ and $\hat{j}$ will lie somewhere else. These basis vectors can be packed into a matrix.

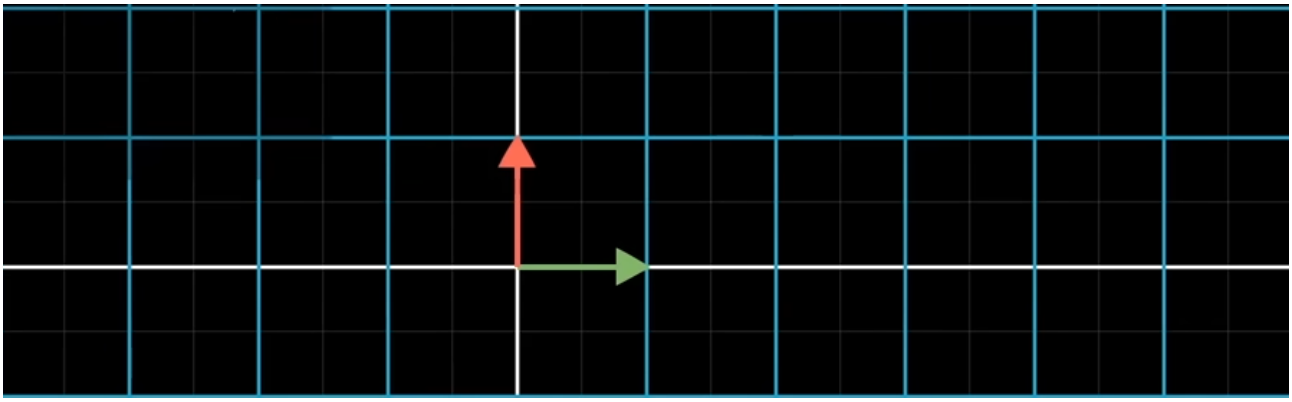
If those basis vectors lie on the following after the transformation,



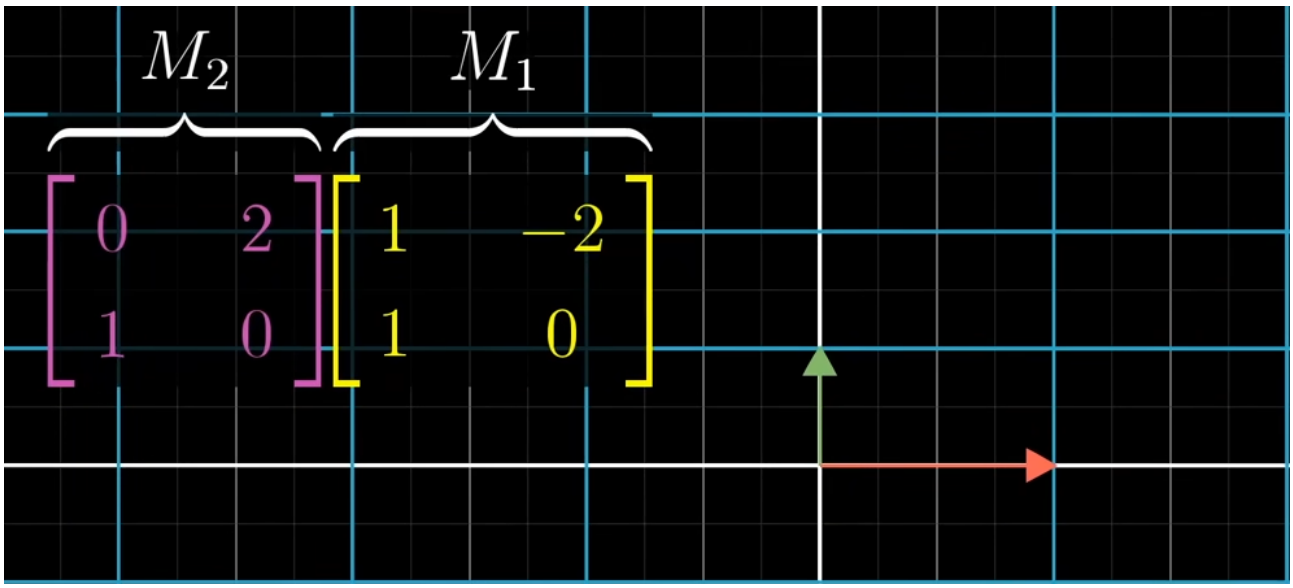
It is performed through matrix multiplication. First of all here, it has to be rotated and then sheared by multiplying with appropriate matrices.



Let’s take an example. We have the basis vectors as,



Let’s apply a couple of transformations over these vectors.

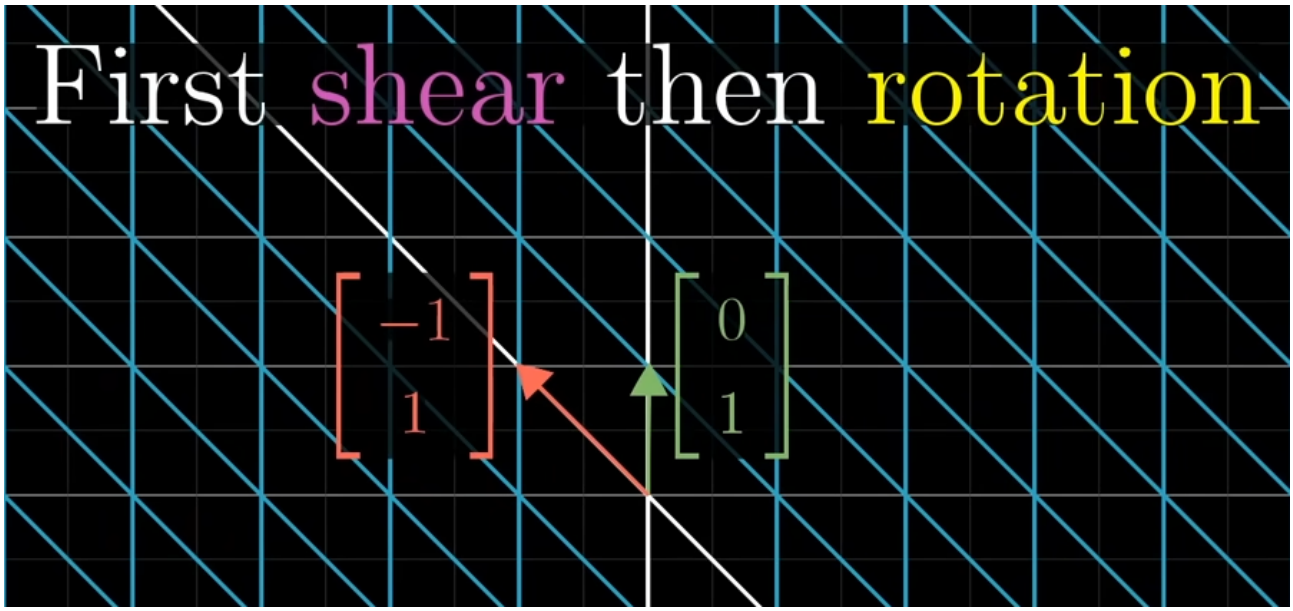


Now, what if we apply these both transformations together ? The transformation will be a composition of those both operations. Multiplying two matrices is like applying the transformations one after other (RIGHT TO LEFT).

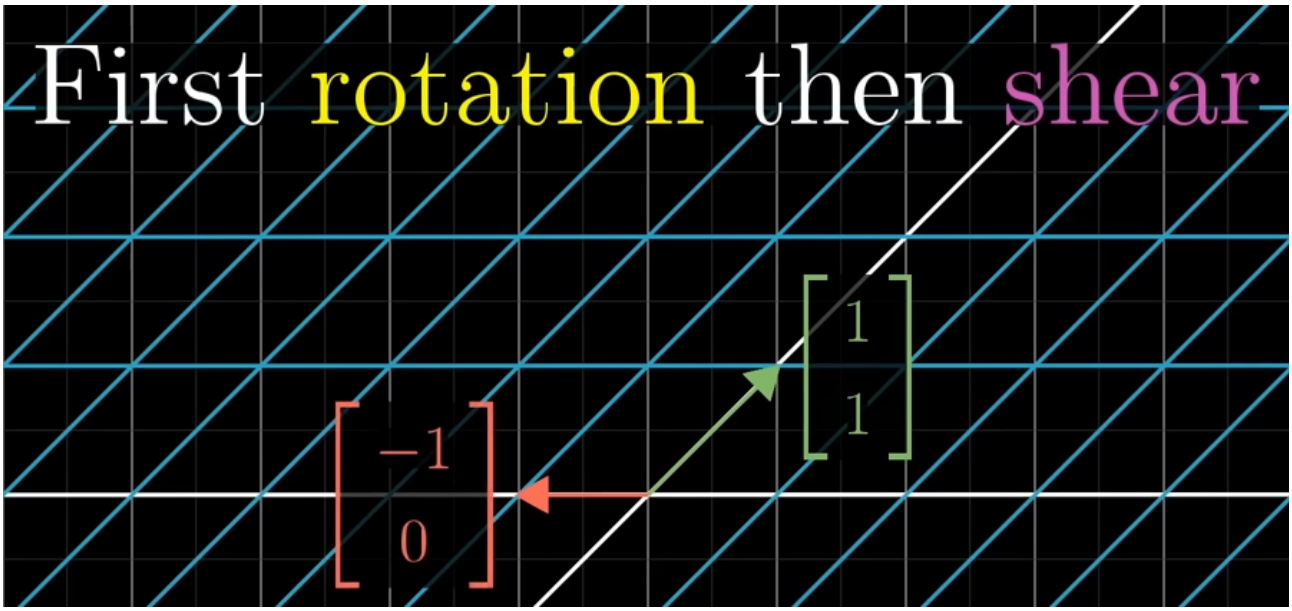
$$\overbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}^{M_2}\overbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}^{M_1} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}$$

Is multiplying M1 with M2 is just the same as multiplying M2 with M1 ? Let’s go graphically.

Let’s do shear first.



If we do rotation first,



Yep! Both are not EQUAL bro…

MATRIX IN NUMPY

@ ADDITION

```
1  # matrix addition
2  from numpy import array
3  A = array([ [1, 2, 3],
4              [4, 5, 6]])
5  B = array([ [1, 2, 3],
6              [4, 5, 6]])
7  C = A + B
8  print(C)
9
```

Console 1/A

```
[[ 2  4  6]
 [ 8 10 12]]
```

@ SUBTRACTION

```
1  from numpy import array
2  A = array([ [1, 2, 3],
3              [4, 5, 6]])
4  B = array([ [1, 2, 3],
5              [4, 5, 6]])
6  C = A - B
7  print(C)
8
9
```

Console 1/A

```
[[0 0 0]
 [0 0 0]]
```

@ HADAMARD MULTIPLICATION

This is often called the element-wise matrix multiplication.

```
1  from numpy import array
2  A = array([ [1, 2, 3],
3              [4, 5, 6]])
4  B = array([ [1, 2, 3],
5              [4, 5, 6]])
6  C = A * B
7  print(C)
8
9
```

Console 1/A

```
[[ 1  4  9]
 [16 25 36]]
```

@ DIVISION

```
1  from numpy import array
2  A = array([ [1, 2, 3],
3              [4, 5, 6]])
4  B = array([ [1, 2, 3],
5              [4, 5, 6]])
6  C = A / B
7  print(C)
8
9
```

Console 1/A

```
[[1. 1. 1.]
 [1. 1. 1.]]
```

@ MAT-MAT MULTIPLICATION

```python
from numpy import array
# define first matrix
A = array([[1, 2],[3, 4],[5, 6]])
B = array([[1, 2],[3, 4]])
C = A.dot(B)
D = A @ B
print(C)
print(D)
```

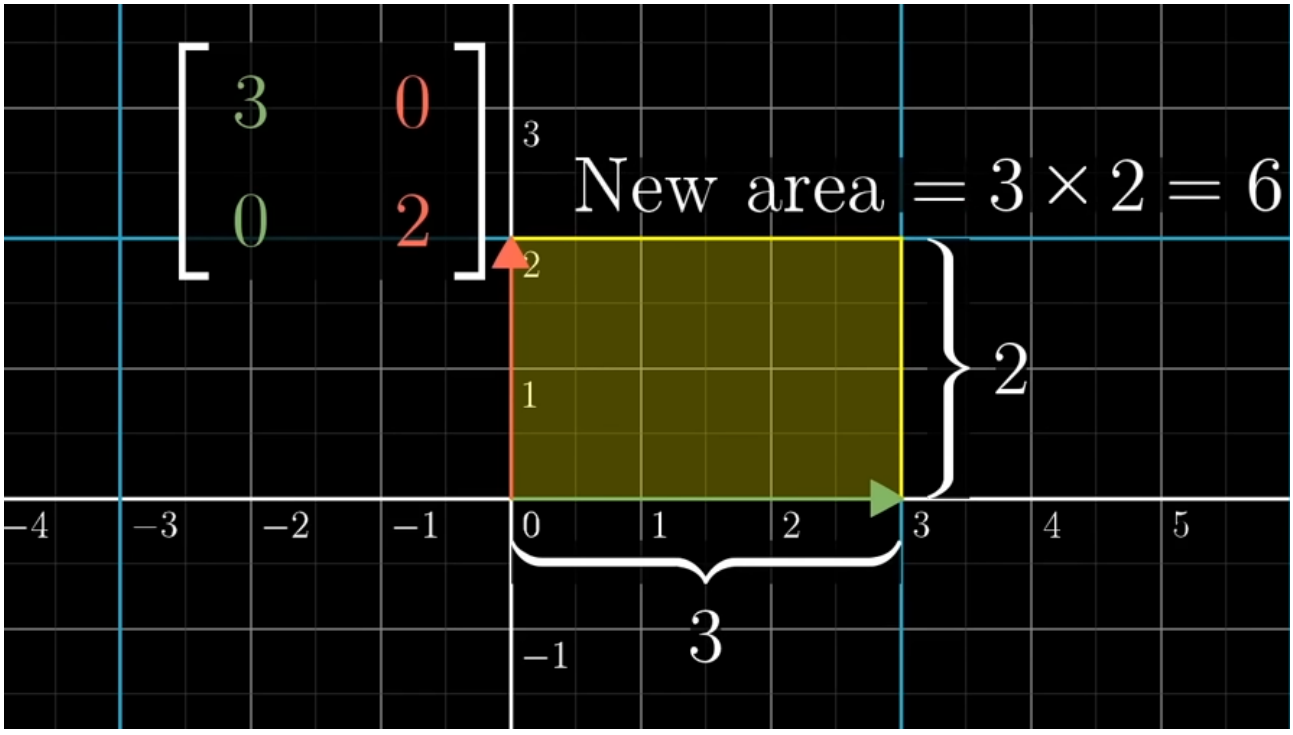
Console 1/A

```
[[ 7 10]
 [15 22]
 [23 34]]
[[ 7 10]
 [15 22]
 [23 34]]
```

The matrix multiplication operation can be implemented in NumPy using the dot() function. It can also be calculated using the newer @ operator, since Python version 3.5.

DETERMINANT

The Determinant of a transformation can be defined as the factor by which the transformation occurs. If an area is increased about the factor of 3, then the determinant would be 3 for that transformation.



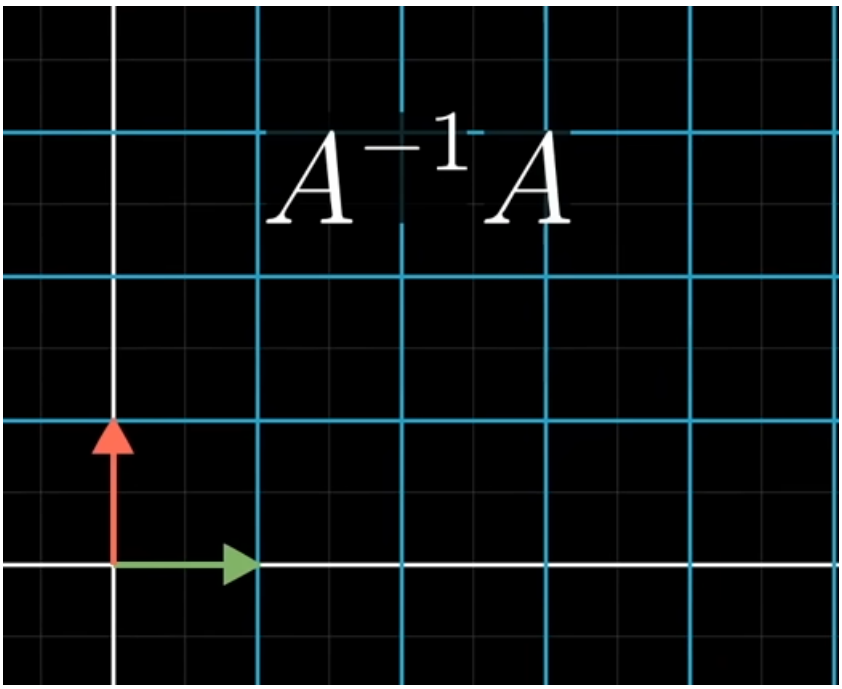
In the above example, there is a new area after the transformation. It is increased about the factor of 6.

In NumPy, there is a det() function in linalg to compute the determinant.

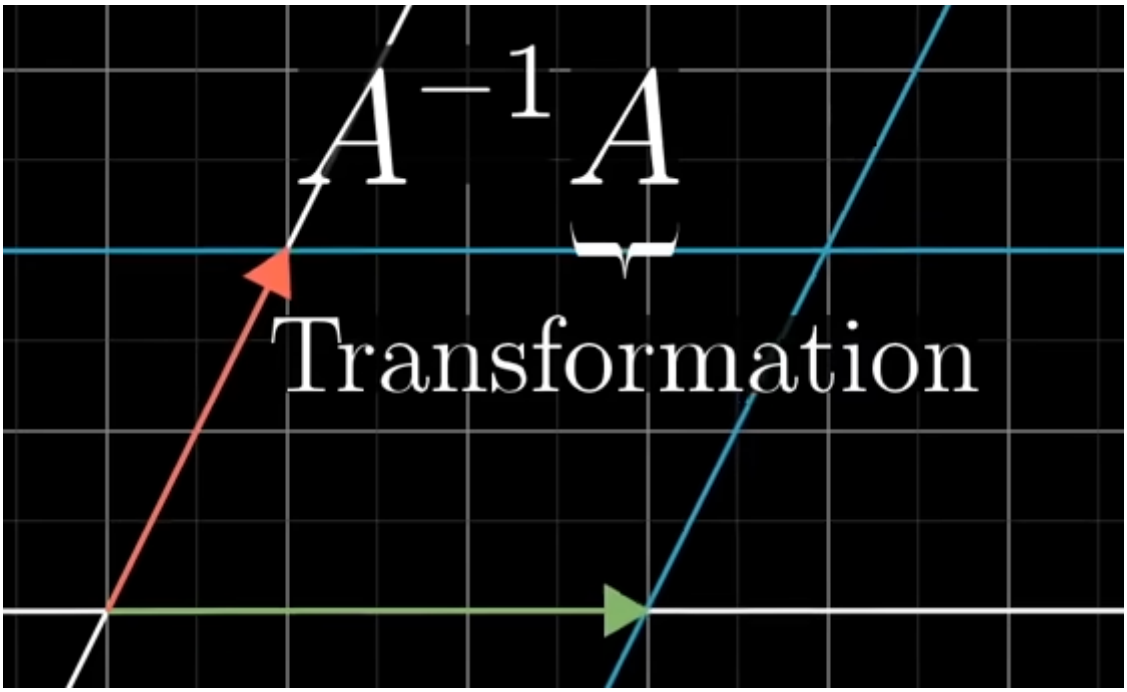
```
1  from numpy import array
2  from numpy.linalg import det
3  # define matrix
4  A = array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
5  # calculate determinant
6  B = det(A)
7  print(B)
```

```
Console 1/A
-9.516197353929 94e-16
```

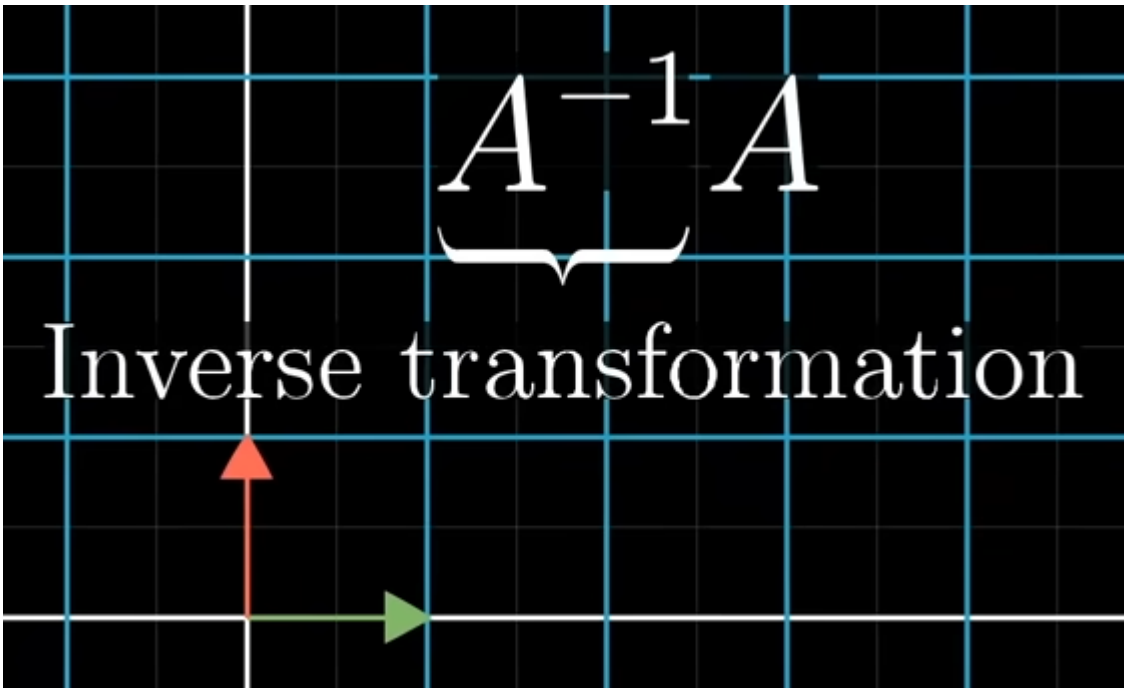
INVERSE MATRIX



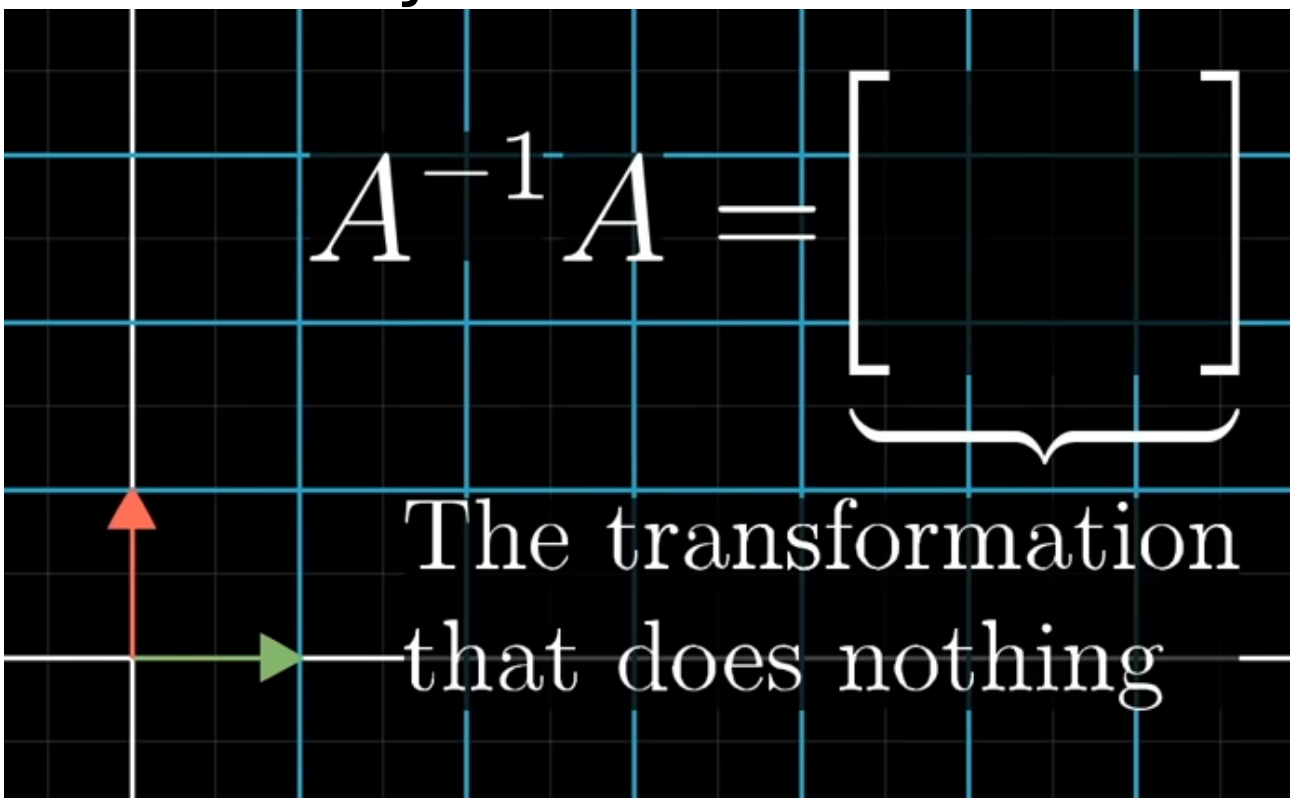
Assume we have the basis vectors as shown. On applying the first transformation A, it will become,



And then, applying the second transformation will become,



Here, as we can observe, there is no change. This sort of transformation is denoted as identity transformation.



In NumPy, there is a function for inverting a matrix.

```
1  from numpy import array
2  from numpy.linalg import inv
3  A = array([ [1.0, 2.0],
4              [3.0, 4.0]])
5  B = inv(A)
6  print(B)
7
```

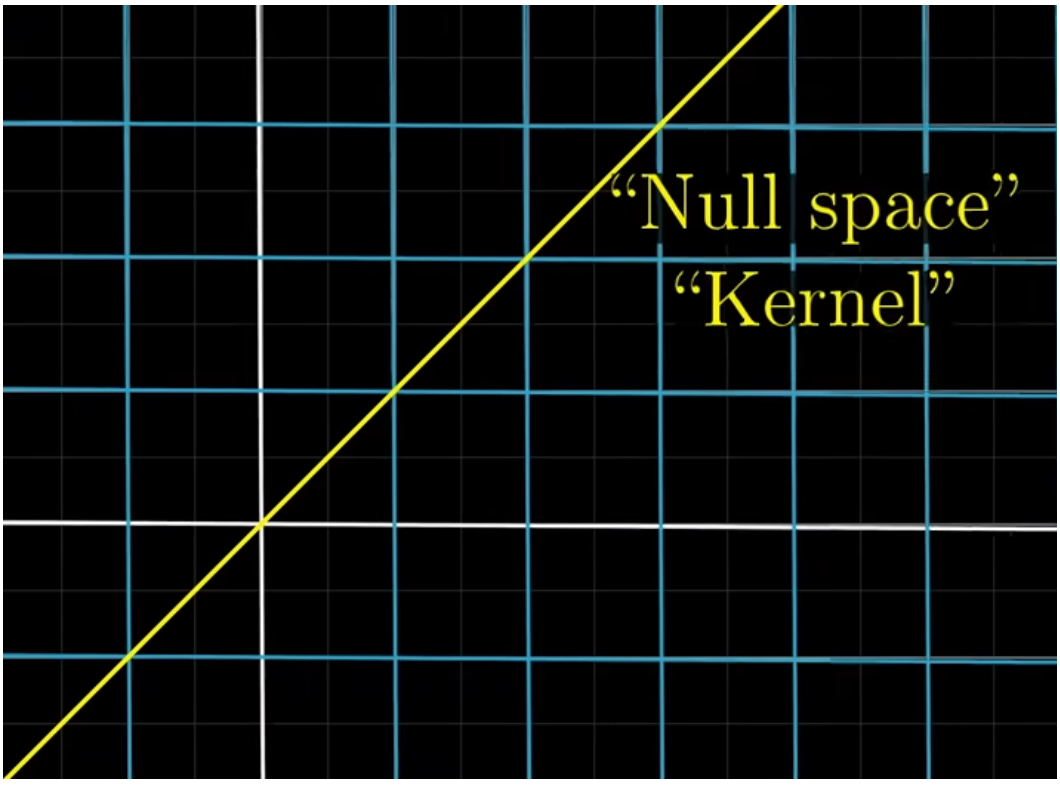
Console 1/A

```
[[-2.   1. ]
 [ 1.5 -0.5]]
```

RANK

When the output dimension of a transformation is 1, the rank of the transformation is 1. When the output dimension of a transformation is 2, the rank of the transformation is 2.

Rank can be defined as the dimension of the output. During the transformations, the vectors may lie on the origin (They become NULL). It is called null space or kernel.



In linear algebra, the column space (also called the range or image) of a matrix A is the span (set of all possible linear combinations) of its column vectors. Rank Computation :

Find the rank of matrix A by using the row echelon form.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 4 \\ 3 & 0 & 5 \end{bmatrix}$$

Solution:

Given,

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 4 \\ 3 & 0 & 5 \end{bmatrix}$$

Now we apply elementary transformations.

$R_2 \rightarrow R_2 - 2R_1$

$R_3 \rightarrow R_3 - 3R_1$

We get

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -2 \\ 0 & -6 & -4 \end{bmatrix}$$

$R_3 \rightarrow R_3 - 2R_2$

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -2 \\ 0 & 0 & 0 \end{bmatrix}$$

The above matrix is in row echelon form.

Number of non-zero rows = 2

Hence the rank of matrix A = 2

In NumPy, we have a method for computing the rank of a matrix.

```
1  from numpy import array
2  from numpy.linalg import matrix_rank
3  v1 = array([[1,2,3],
4              [2,4,6]])
5  vr1 = matrix_rank(v1)
6  print('Rank : ',vr1)
7
```

```
Console 1/A
Rank :  1
```

TRANSPOSE

A defined matrix can be transposed, which creates a new matrix with the number of columns and rows flipped.

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

$$A^T = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

We can compute it in Python using the following way.

```
1  from numpy import array
2  # define matrix
3  A = array([ [1, 2],
4              [3, 4],
5              [5, 6]])
6  C = A.T
7  print(C)
```

```
Console 1/A
[[1 3 5]
 [2 4 6]]
```

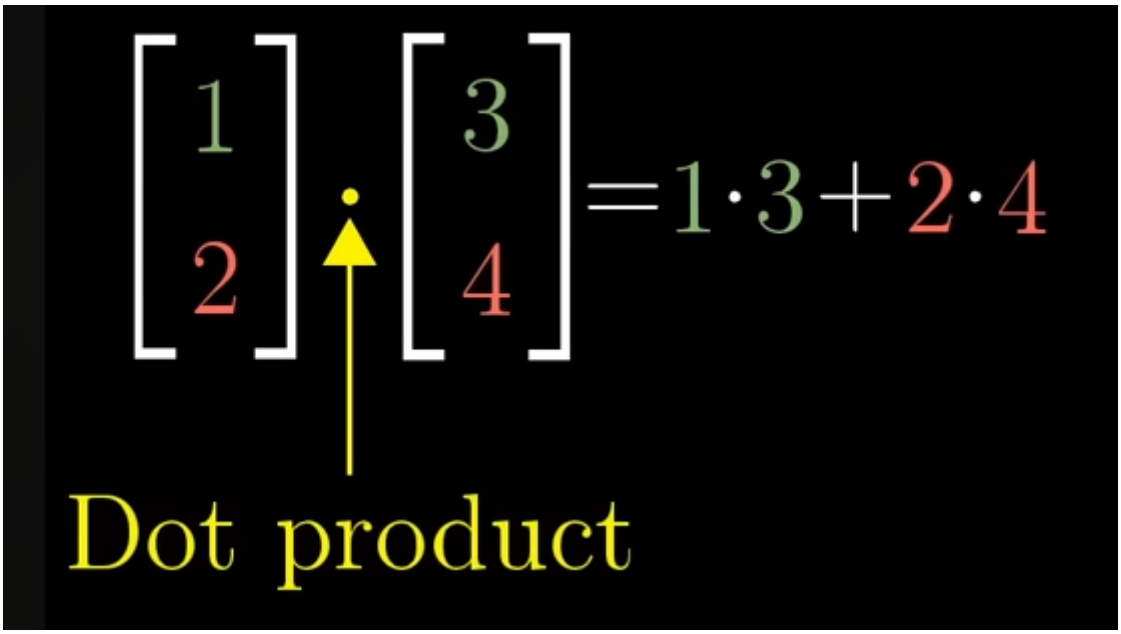
TRACE

The Trace of a matrix is defined as the sum of the main-diagonal elements.

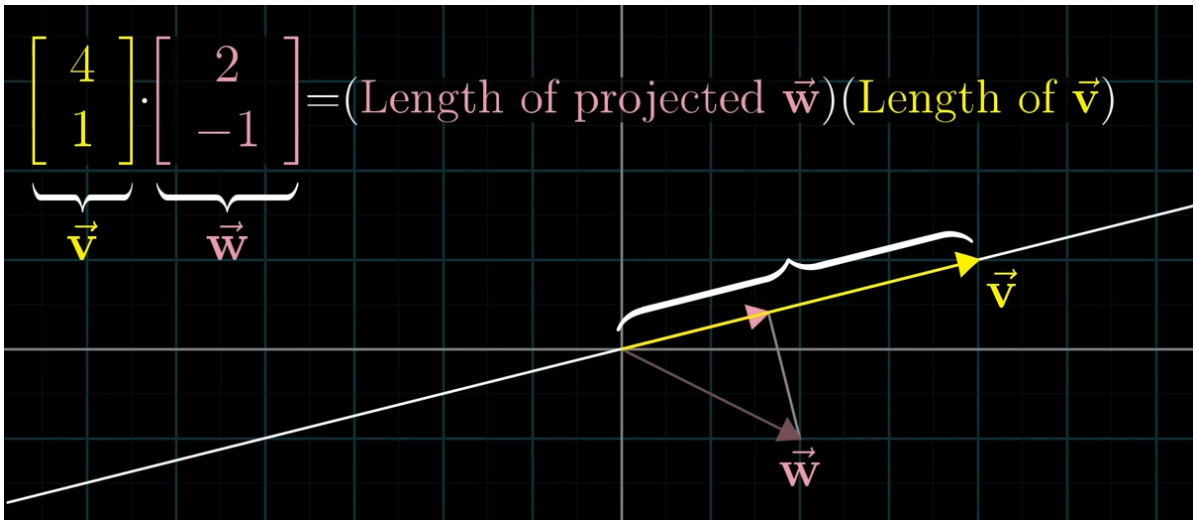
```
1  from numpy import array
2  from numpy import trace
3  A = array([ [1, 2, 3],
4              [4, 5, 6],
5              [7, 8, 9]])
6  B = trace(A)
7  print(B)
```

```
Console 1/A
15
```

THE DOT PRODUCT

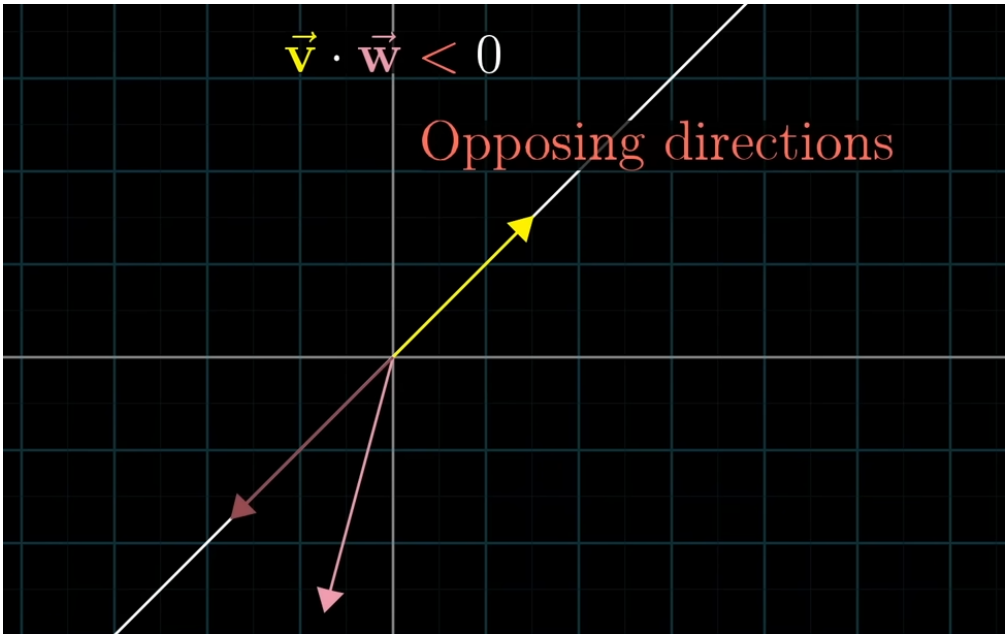


The DOT Product can be seen like the above. To represent it graphically,

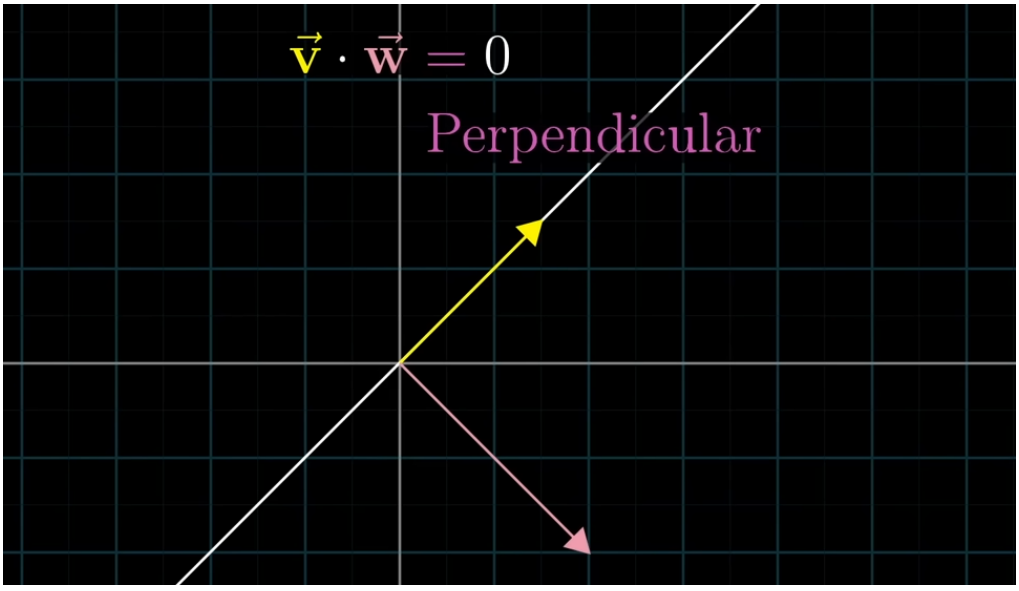


We have to project the second vector over the first vector and find out the product of their lengths.

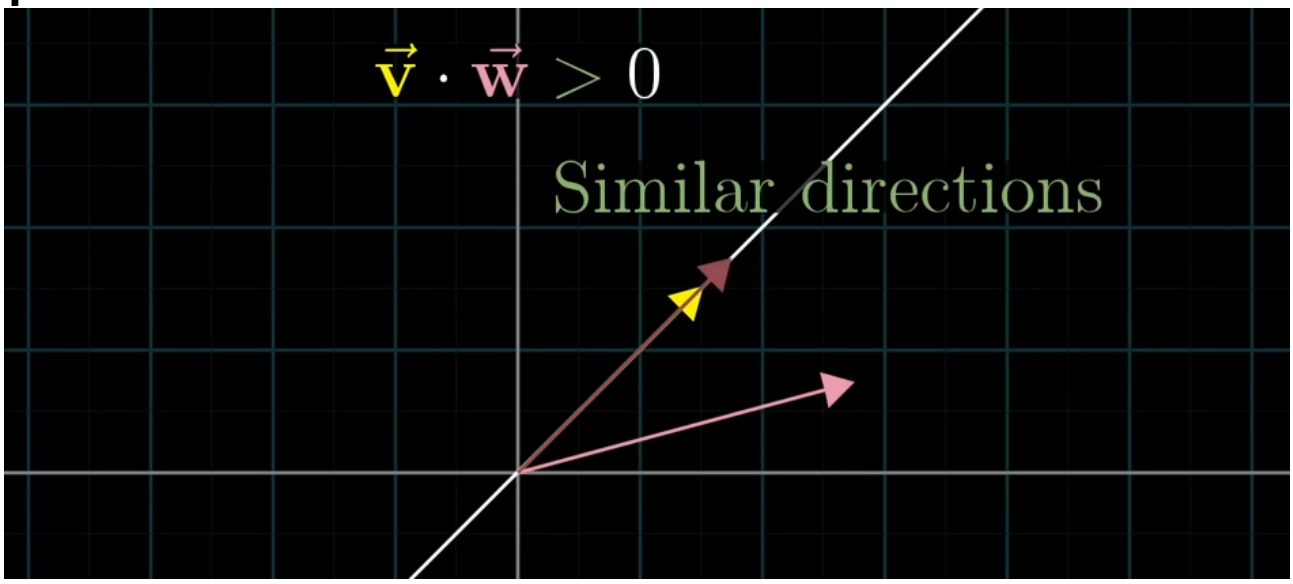
This is nothing but the so-called DOT Product. If these pointers are pointing opposite directions, then the DOT product is negative.



If they are perpendicular to each other, there is no projection and the DOT product is ZERO.

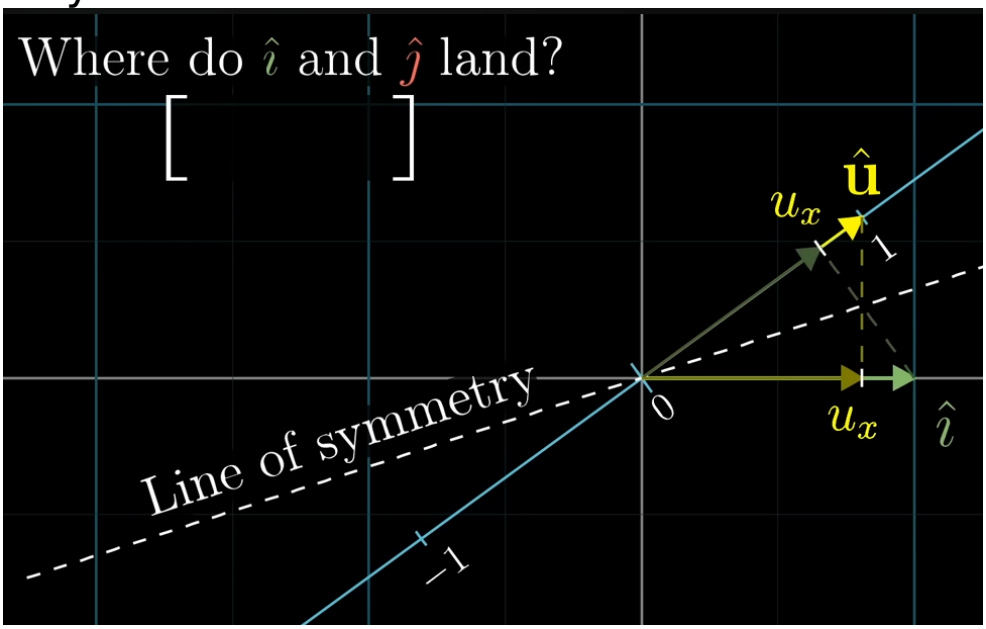


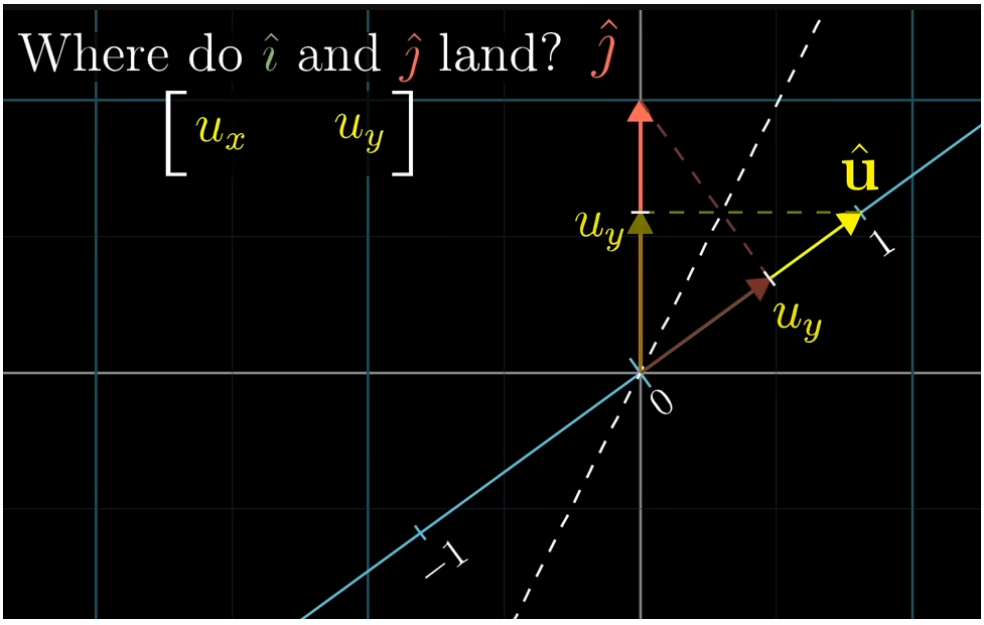
If both of these are in similar directions, the DOT Product is positive.



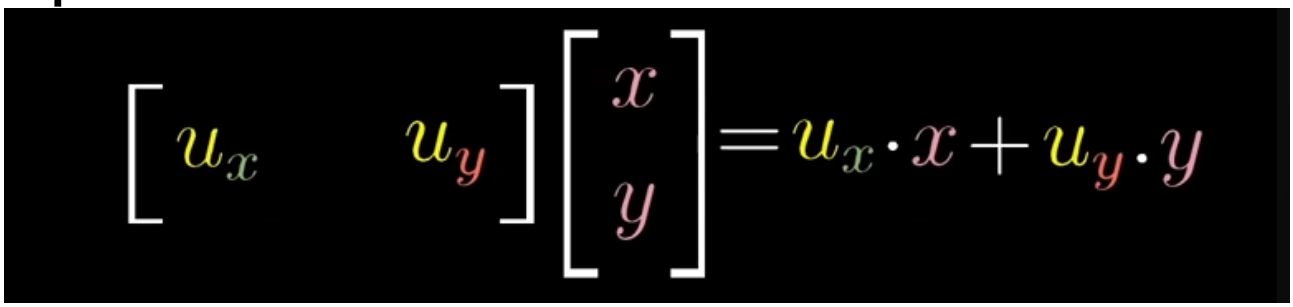
Here, Order doesn’t matter. We can even project the first vector over the second vector. But, why are we projecting these lines to compute the DOT product ? What is the relation of projection and DOT Product ? Well. Let’s imagine the following.

Here, projecting the î over the diagonal line is symmetric to projecting u hat on x axis at u_x. This is the similar case for the u_y.



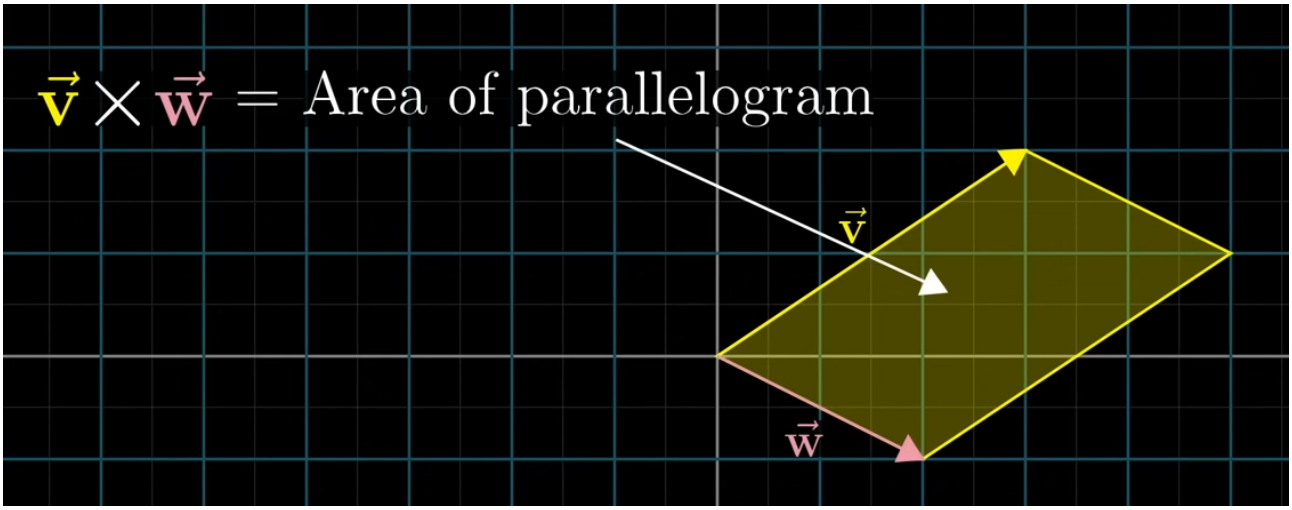


In Essence, The DOT Product is “How much of a is pointing in the same direction as the vector b.” Computationally, taking dot product with unit vector is identical to projecting the vector over the span of that unit vector.

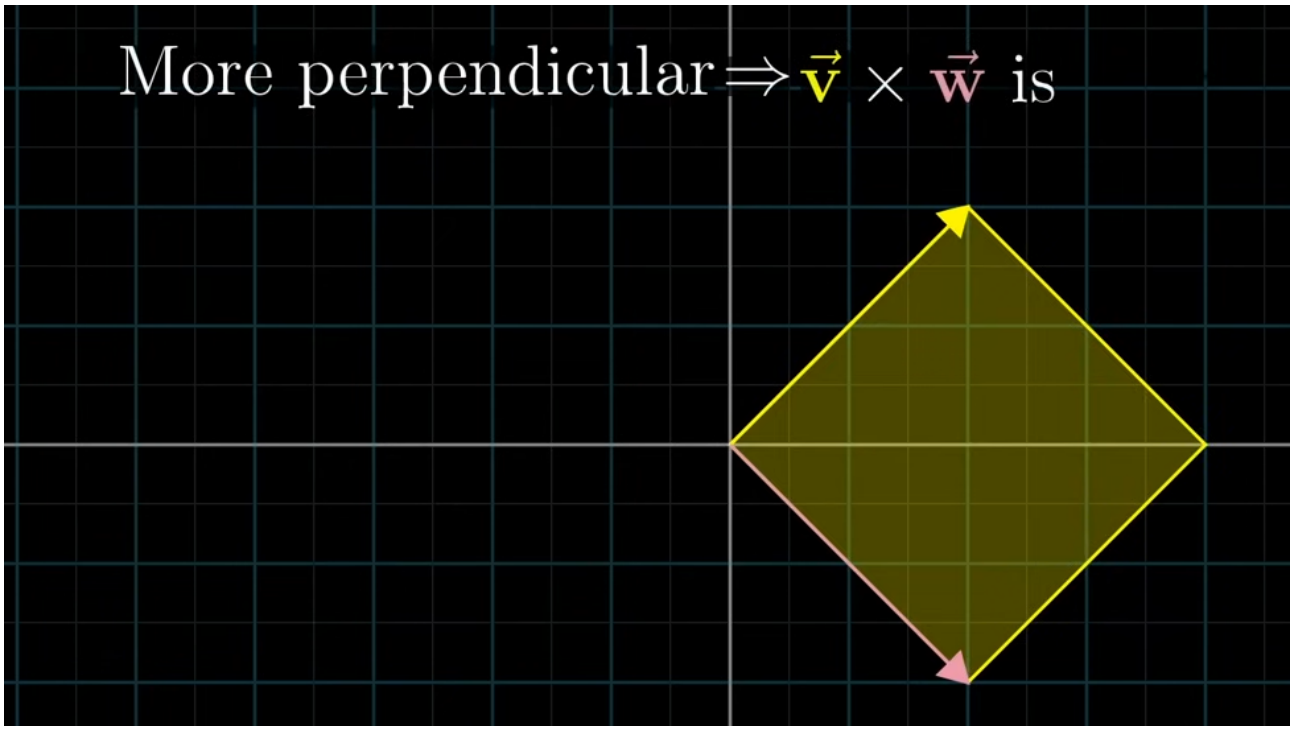


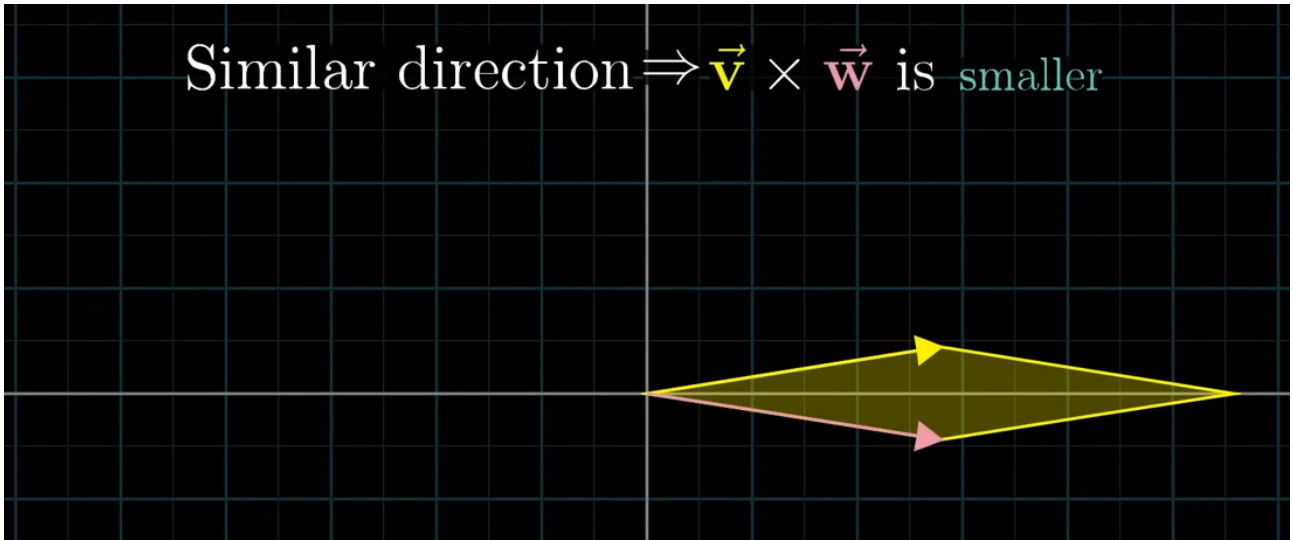
CROSS PRODUCT

The Cross product of two vectors is the area bounded when moving the two vectors in the direction specified by the other vector.

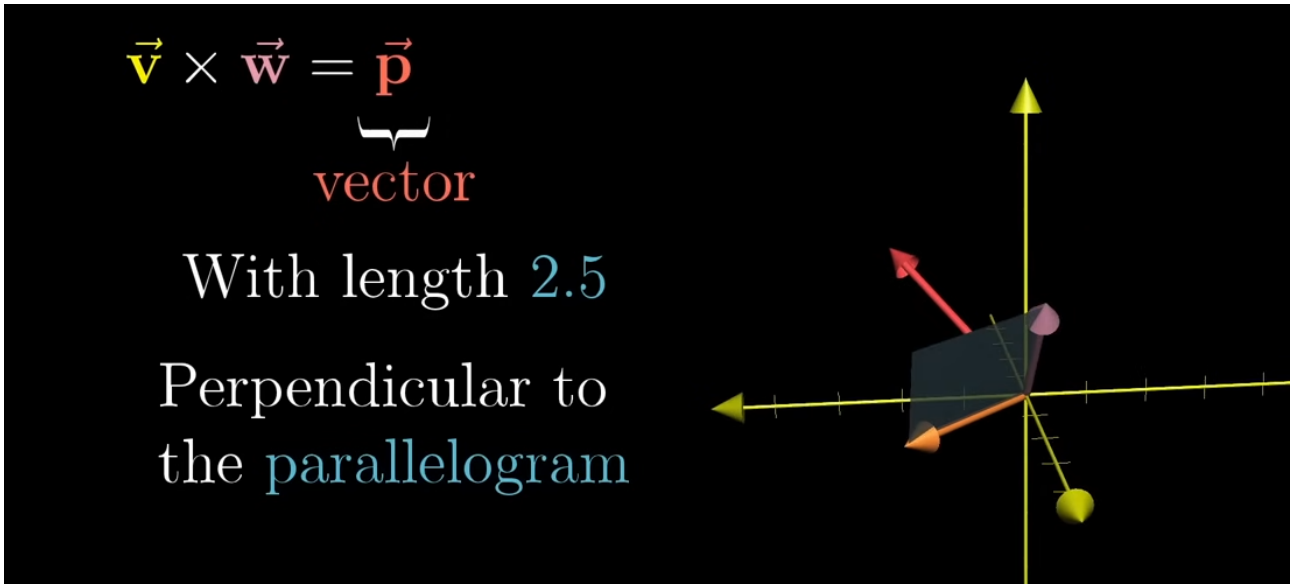


We have to consider the orientation definitely. The Determinant is all about measuring how area is changed when a particular transformation is applied. When v is on the left of w in the space, the orientation is negative.

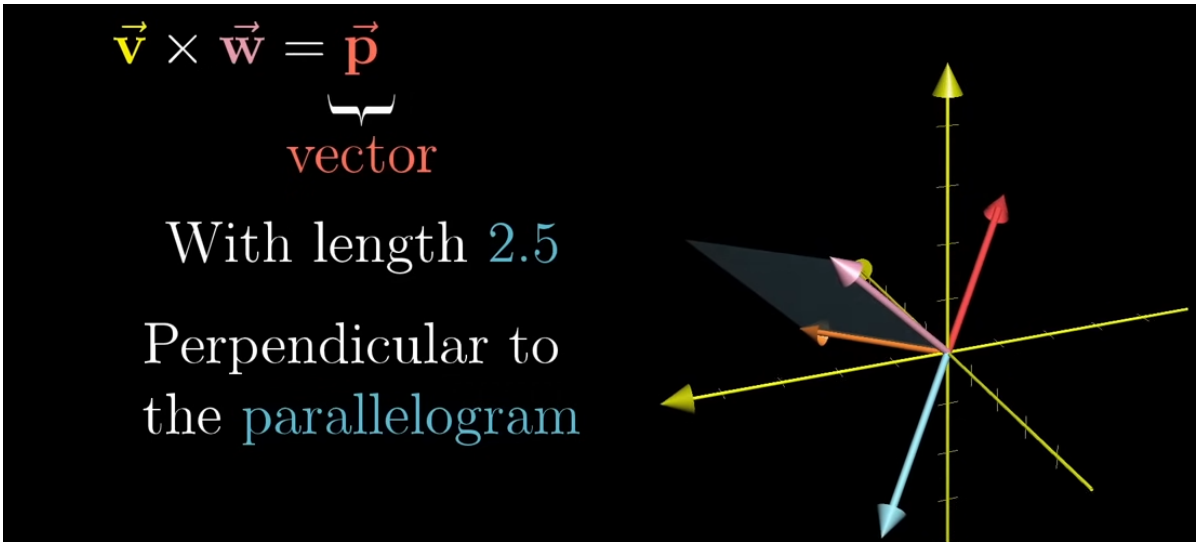




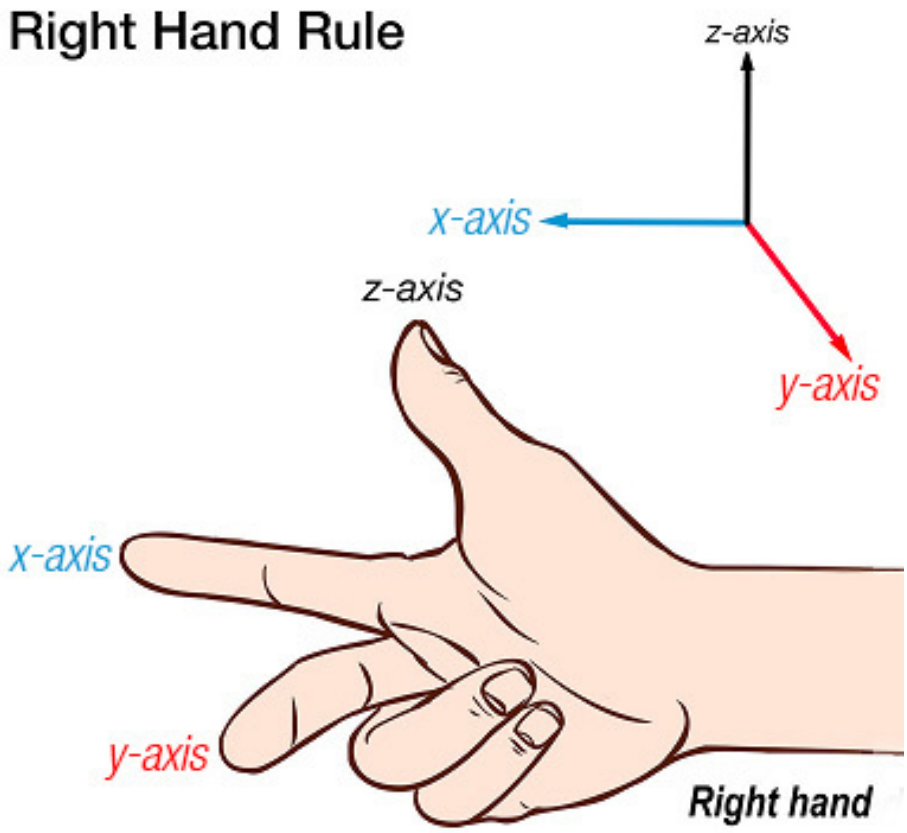
Now, consider one thing. When we cross-product two vectors, the area is found to be 2.5. The newly formed vector is perpendicular to the plane. It’s length is the area of the plane.



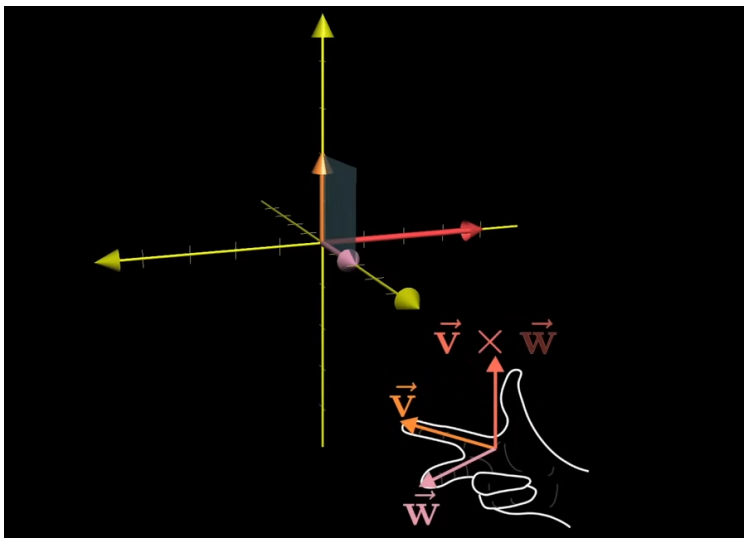
Here, there might be two vectors perpendicular to the plane. (RED and BLUE)



How to find out the direction ?



If we place the index and middle finger in the x and y vector, your toe direction is that of the cross product of x and y.

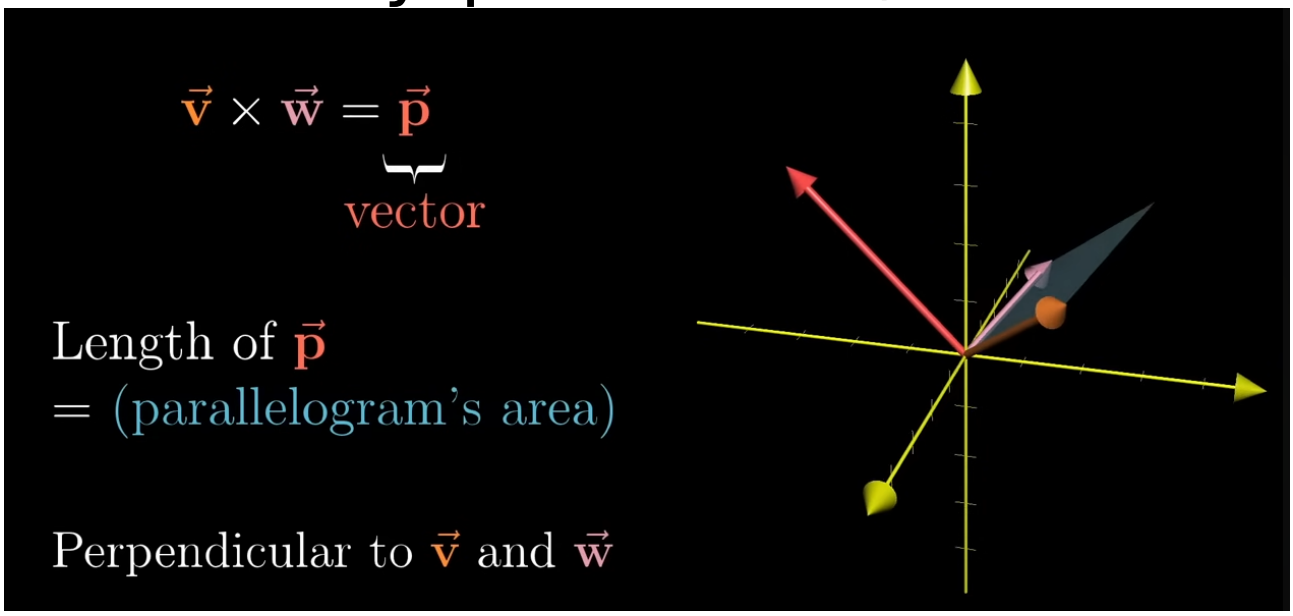


For computation, vector product is found as the following.

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \times \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \det\left(\begin{bmatrix} \hat{i} & v_1 & w_1 \\ \hat{j} & v_2 & w_2 \\ \hat{k} & v_3 & w_3 \end{bmatrix}\right)$$

$$\hat{i}(v_2 w_3 - v_3 w_2) + \hat{j}(v_3 w_1 - v_1 w_3) + \hat{k}(v_1 w_2 - v_2 w_1)$$

Take-away points are,



Performing the linear transformation to form a 1D proportion is same as taking a dot product with a peculiar vector called , DUAL VECTOR.

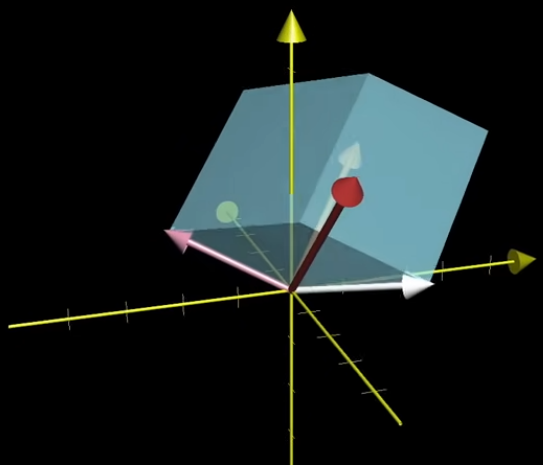
$$\begin{bmatrix}2\\1\end{bmatrix}\cdot\overbrace{\begin{bmatrix}x\\y\end{bmatrix}}^{\vec{\mathbf{v}}} = \overbrace{\begin{bmatrix}2 & 1\end{bmatrix}}^{\text{Transform}}\overbrace{\begin{bmatrix}x\\y\end{bmatrix}}^{\vec{\mathbf{v}}}$$

Dot product

How to find that DUAL Vector ?

What vector $\vec{\mathbf{p}}$ has
the property that

$$\overbrace{\begin{bmatrix}p_1\\p_2\\p_3\end{bmatrix}}^{\vec{\mathbf{p}}}\cdot\begin{bmatrix}x\\y\\z\end{bmatrix} = \det\left(\begin{bmatrix}x & \overbrace{v_1}^{\vec{\mathbf{v}}} & \overbrace{w_1}^{\vec{\mathbf{w}}}\\ y & v_2 & w_2\\ z & v_3 & w_3\end{bmatrix}\right)$$



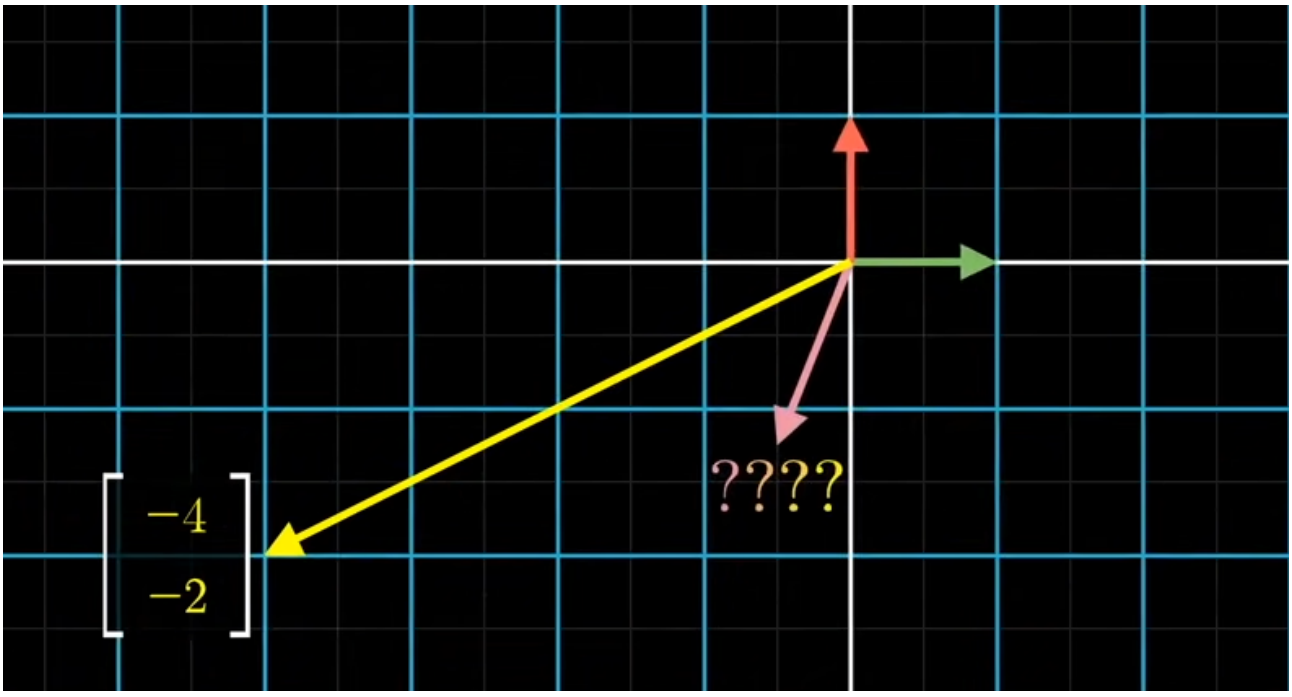
CRAMER'S RULE

We are given a linear system of equations. We have to find out on applying which vector, the transformation given happens.

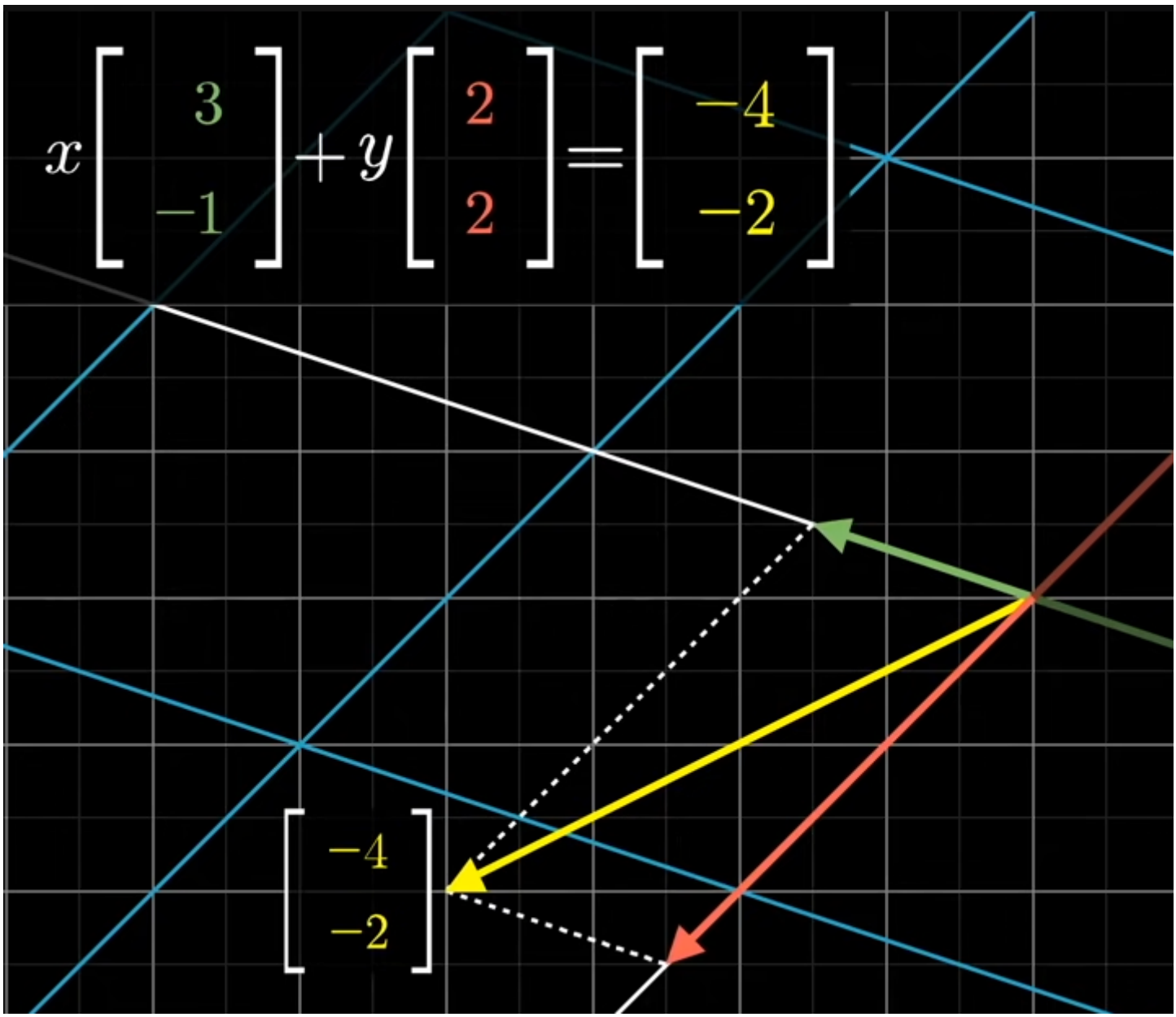
$$3x + 2y = -4$$

$$-1x + 2y = -2$$

$$\begin{bmatrix} 3 & 2 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -4 \\ -2 \end{bmatrix}$$

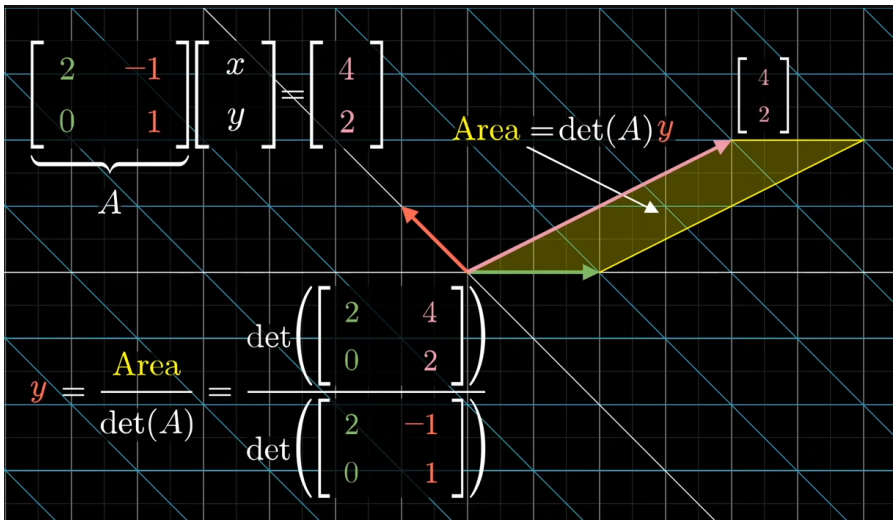


We can rewrite the system as the following.



Let’s assume the area is scaled up. We all know that this scaling-up is by the

determinant. Area gets scaled up by the determinant. Using this fact, y can be found.
Area is the base multiplied by the height perpendicular to that base.



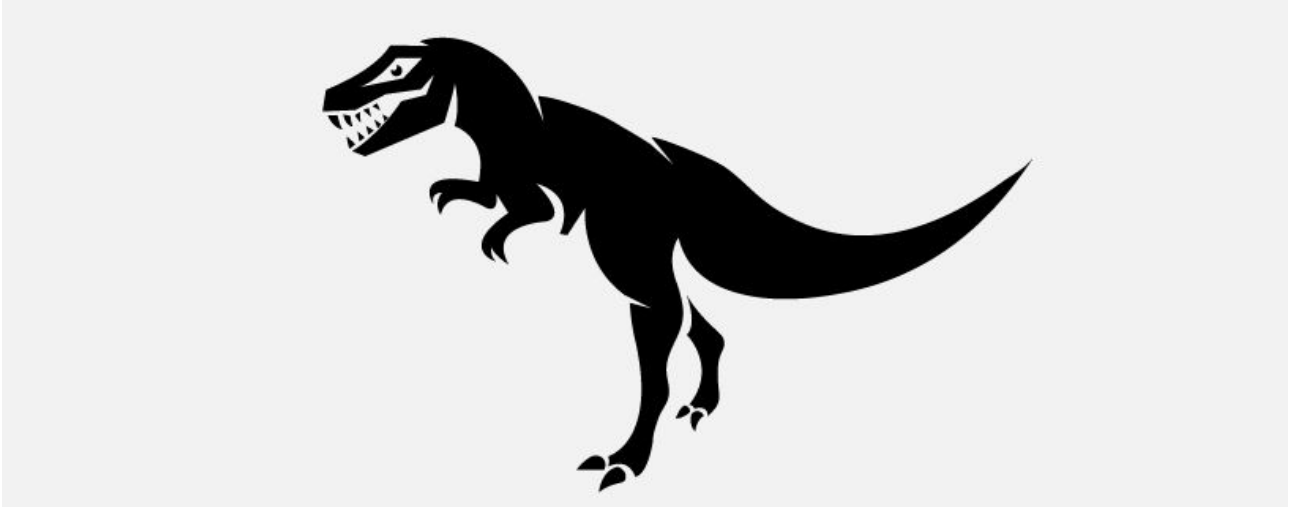
The similar idea can be applied to the x. This is nothing but the Cramer's Rule.

WHAT'S EIGEN

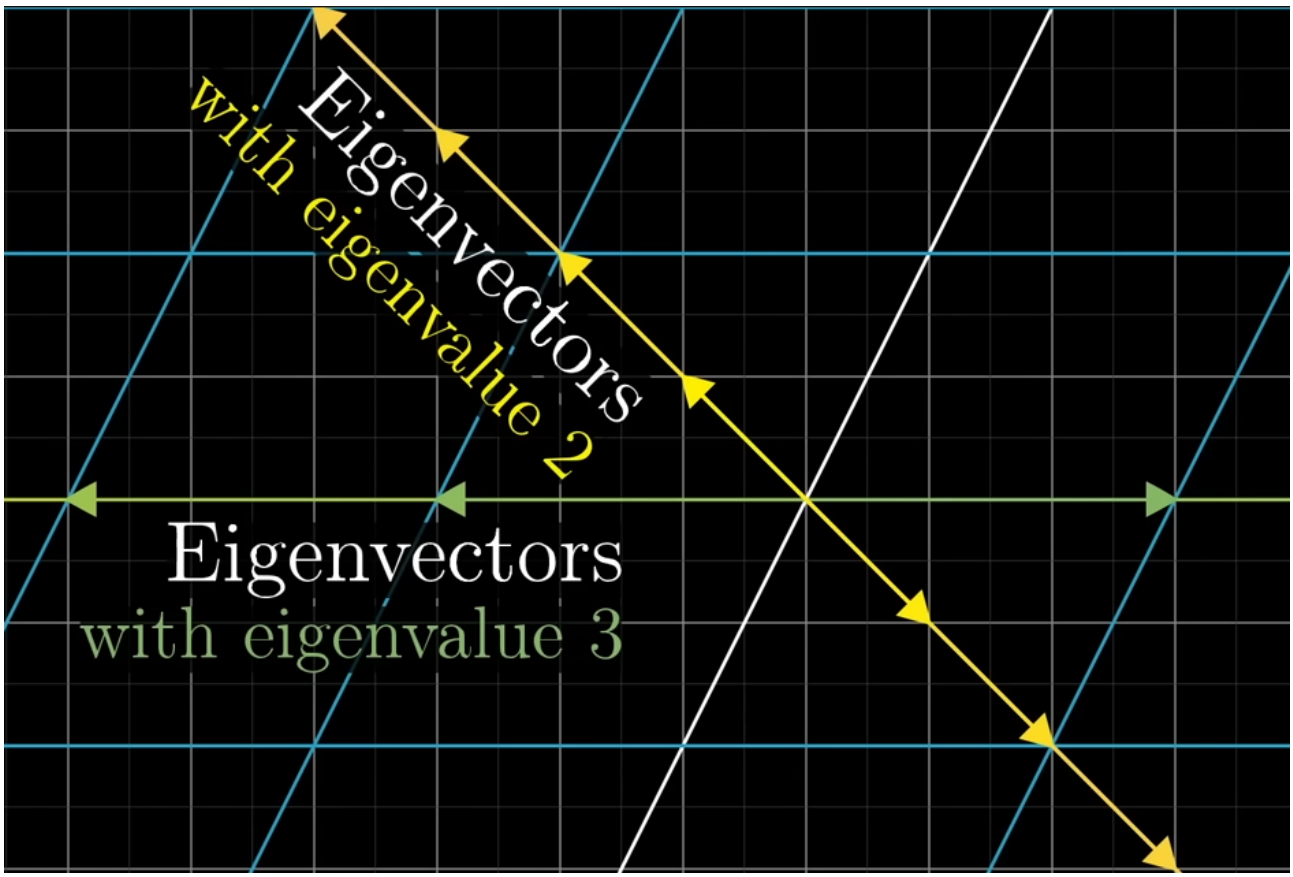
An **eigenvector** of a linear transformation is a nonzero

vector that changes at most by a scalar factor when that linear transformation is applied to it. The corresponding **eigenvalue**, often denoted by lambda, is the factor by which the eigenvector is scaled.

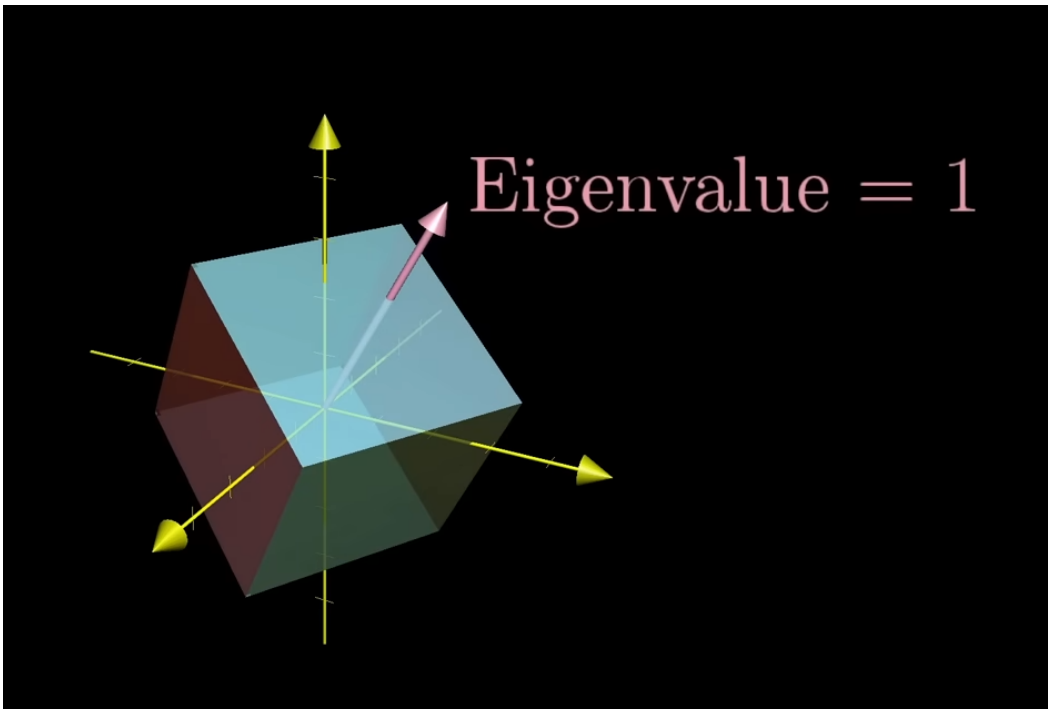
An eigenvector is a vector, which after applying the linear transformation,



stays in the same span (changes by only a scalar factor).



It means, there are some other vectors getting squished or scaled during the transformation is applied.



$$\underbrace{(A - \lambda I)}\vec{\mathbf{v}} = \vec{0}$$

This matrix looks something like

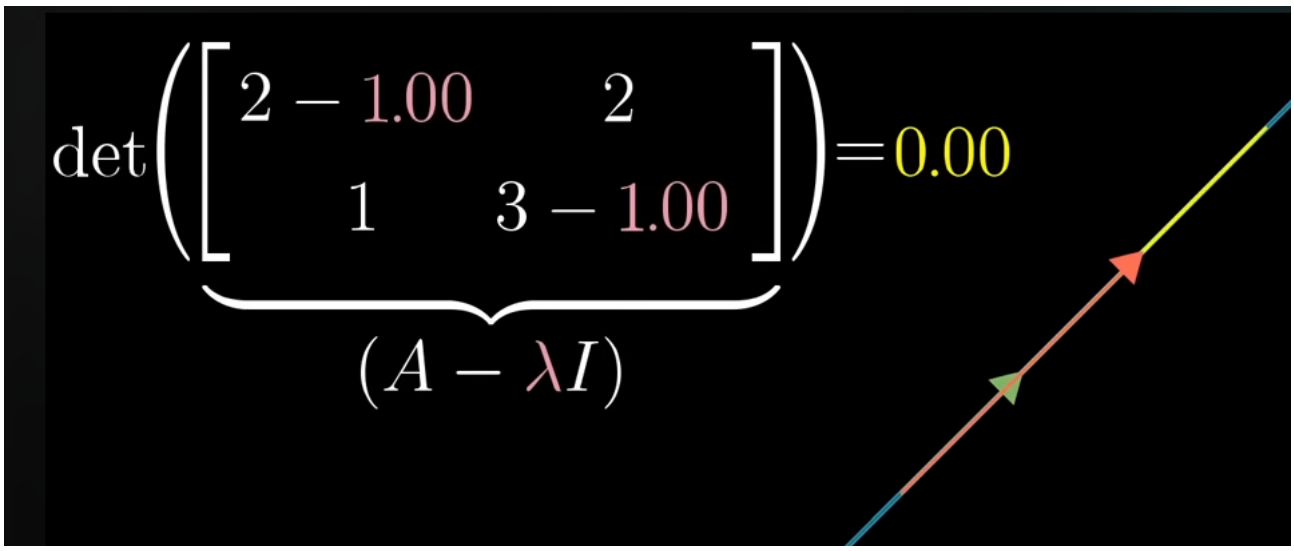
$$\begin{bmatrix} 3-\lambda & 1 & 4 \\ 1 & 5-\lambda & 9 \\ 2 & 6 & 5-\lambda \end{bmatrix}$$

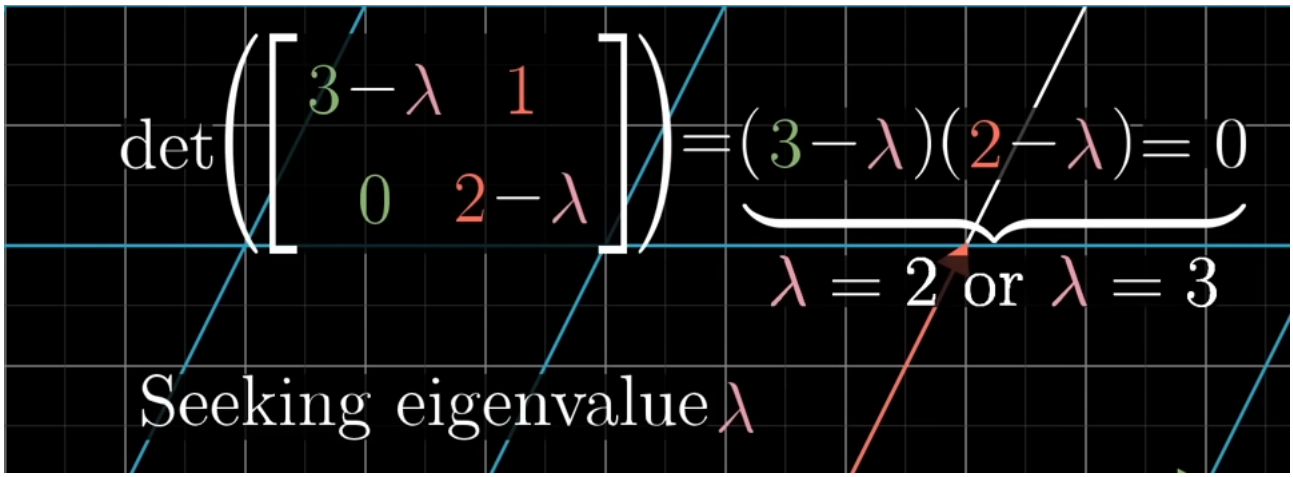
Matrix-vector multiplication

$$\overbrace{A\vec{\mathbf{v}}} = \underbrace{\lambda\vec{\mathbf{v}}}$$

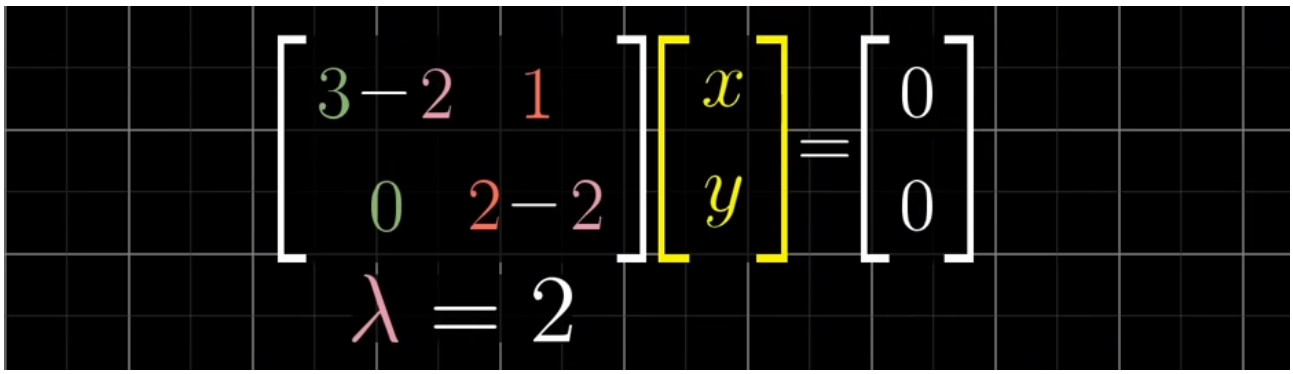
Scalar multiplication

As we can see, we need the vector on which applying the A-λI transformation yields 0. So, we need to find out the λ value for which |A-λI| becomes 0. In this case we got 1 for λ.

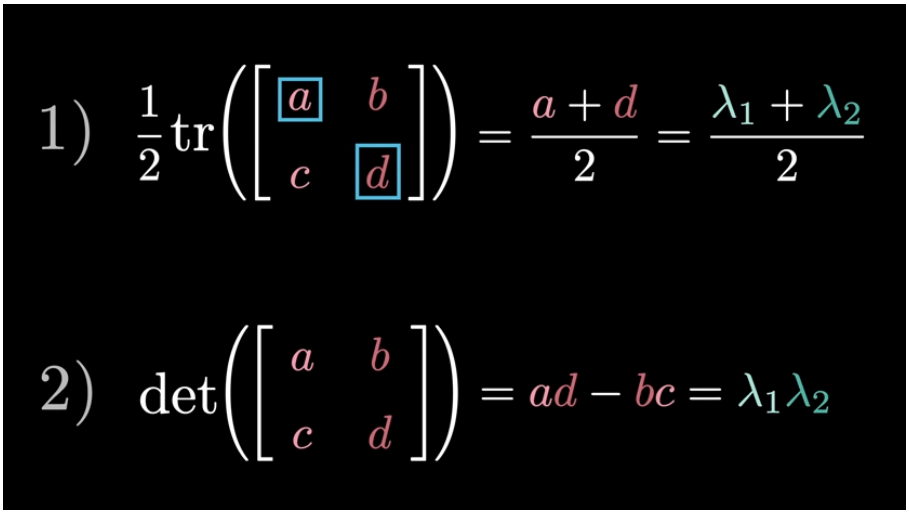


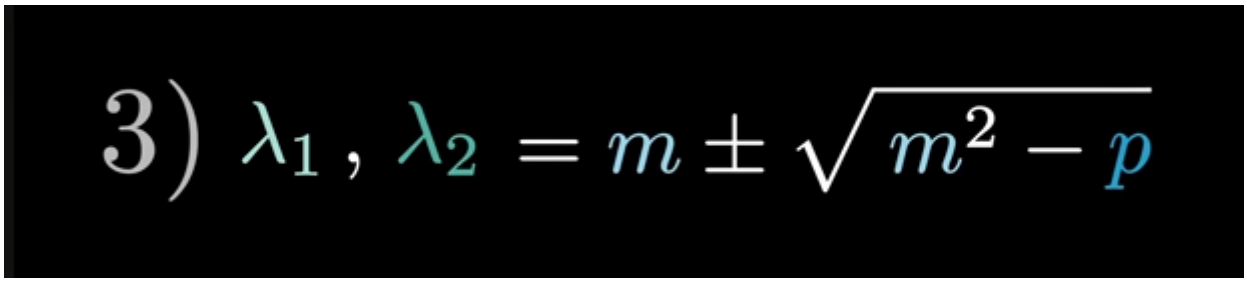


Using the eigen-values found, we can compute the eigen vector as the following.



We also have some formulae to quickly find out the Eigen values.





USES :

Used in Principle Component Analysis (Lowering the data dimension)

TENSOR

A tensor is a generalization of vectors and matrices and is easily understood as a multidimensional array.

```
1  from numpy import array
2  T = array([ [[1,2,3], [4,5,6], [7,8,9]],
3              [[11,12,13], [14,15,16], [17,18,19]],
4              [[21,22,23], [24,25,26], [27,28,29]]])
5  print(T.shape)
6  print(T)
```

```
Console 1/A

In [1]: runfile('/home/ari-pt7127/num.py', wdir='/home/ari-pt7127')
(3, 3, 3)
[[[ 1  2  3]
  [ 4  5  6]
  [ 7  8  9]]

 [[11 12 13]
  [14 15 16]
  [17 18 19]]

 [[21 22 23]
  [24 25 26]
  [27 28 29]]]
```

We can do all the arithmetic operations that we have done over the matrices.

EIGEN - NUMPY

```python
from numpy import array
from numpy.linalg import eig
A = array([ [1, 2, 3],
            [4, 5, 6],
            [7, 8, 9]])
values, vectors = eig(A)
print(values)
print(vectors)
```

```
Console 1/A
In [2]: runfile('/home/ari-pt7127/num.py', wdir='/home/ari-pt7127')
[ 1.61168440e+01 -1.11684397e+00 -3.38433605e-16]
[[-0.23197069 -0.78583024  0.40824829]
 [-0.52532209 -0.08675134 -0.81649658]
 [-0.8186735   0.61232756  0.40824829]]
```

SPARCE MATRIX

A sparse matrix is a matrix that is comprised of mostly zero values. Sparse matrices are distinct from matrices with mostly non-zero values, which are referred to as dense matrices.

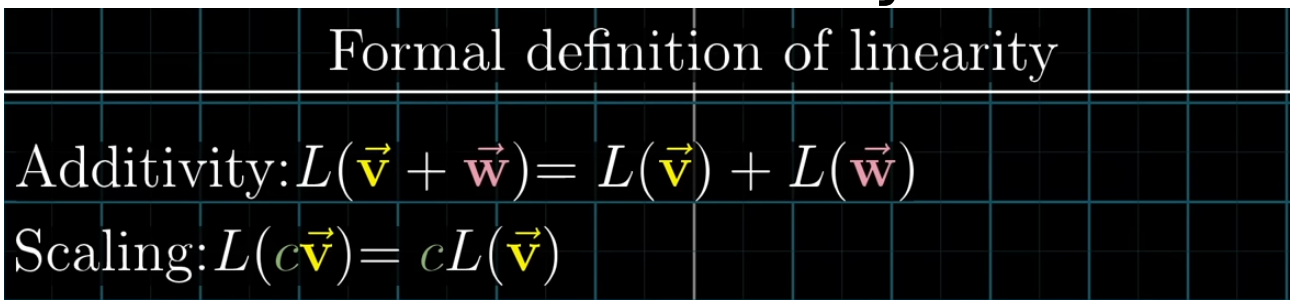
A dense matrix stored in a NumPy array can be converted into a sparse matrix using the CSR representation by calling the csr_matrix() function. In the example below, we define a 3 × 6 sparse matrix as a dense array (e.g. an ndarray), convert it to a CSR sparse representation, and then convert it back to a dense array by calling the todense() function.

```
from numpy import array
from scipy.sparse import csr_matrix
A = array([
    [1, 0, 0, 1, 0, 0],
    [0, 0, 2, 0, 0, 1],
    [0, 0, 0, 2, 0, 0]])
S = csr_matrix(A)
print(S)
B = S.todense()
print(B)
```

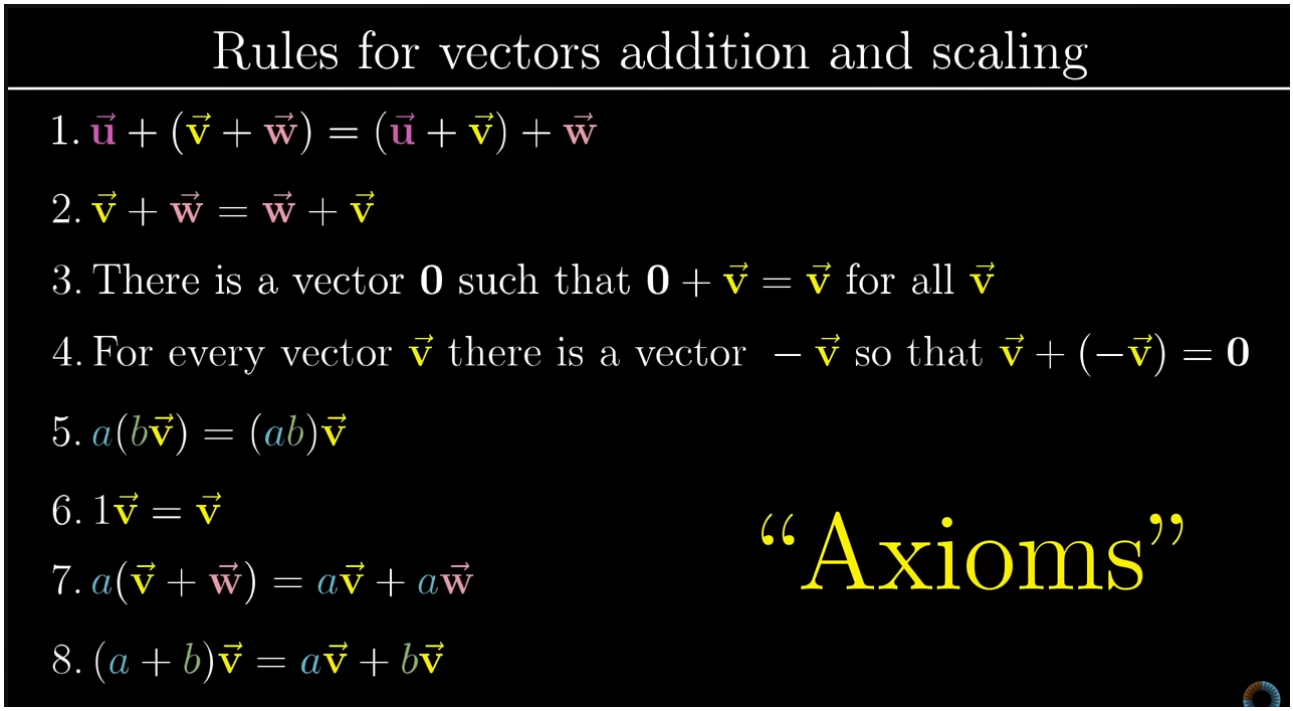
```
Console 1/A
In [3]: runfile('/home/ari-pt7127/num.py', wdir='/home/ari-pt7127')
  (0, 0)    1
  (0, 3)    1
  (1, 2)    2
  (1, 5)    1
  (2, 3)    2
[[1 0 0 1 0 0]
 [0 0 2 0 0 1]
 [0 0 0 2 0 0]]
```

FORMAL DEFINITION

Following are the formal definitions of linearity.



Some of the axioms for vectors addition and scaling are,



FINE