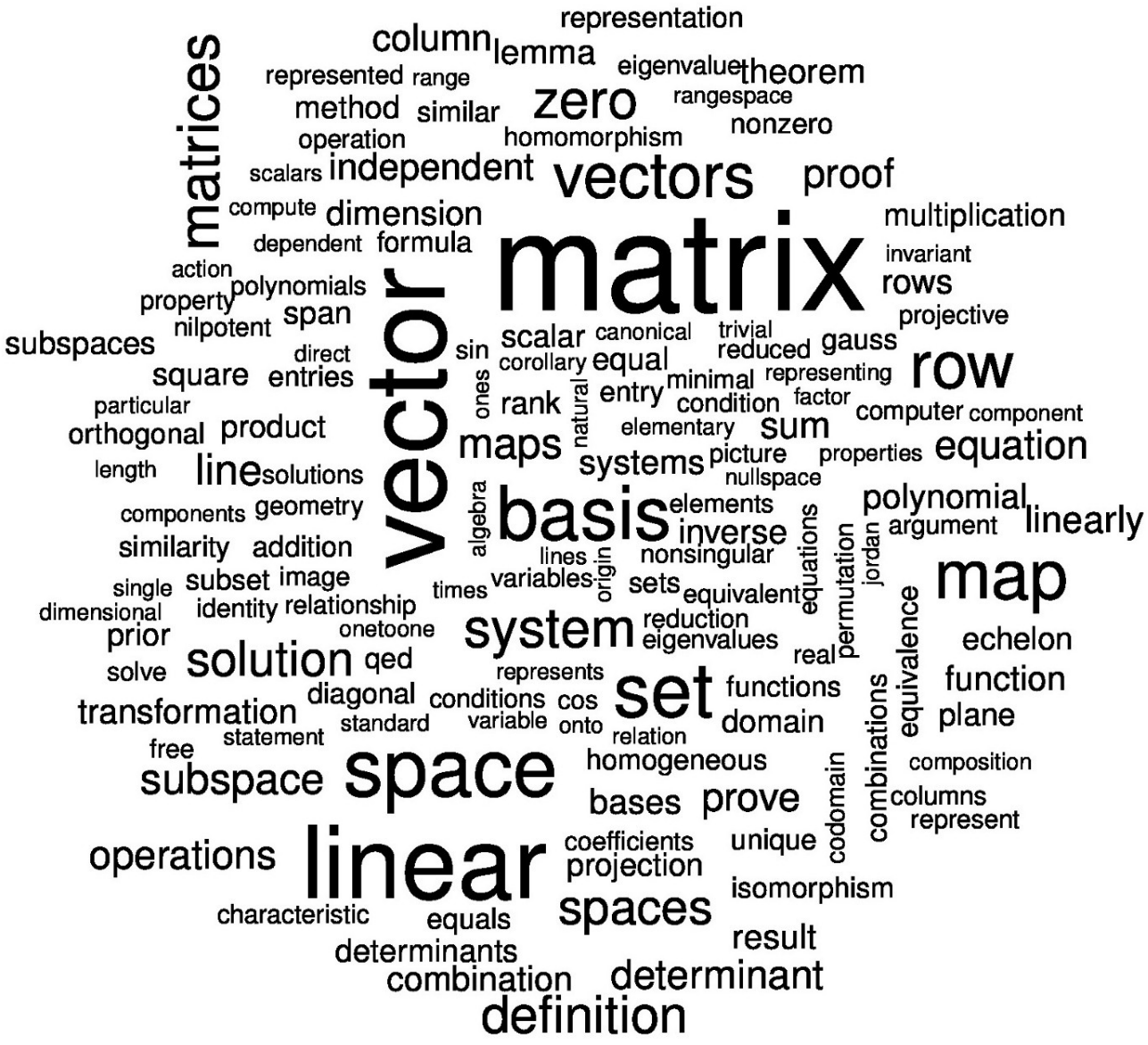
Linear Algebra and Numpy

Akshay Pal Follow
Aug 10, 2019 · 24 min read



Linear Algebra

‘d like to present the blog post on the linear algebra and how it benefits to Machine

I Learning and Deep Learning. so, the goal of this post is to help the beginners in the field of Machine learning and Deep learning. basically, we have a focus on the Numpy Library, who don't know about the Numpy Library. it is a Python Package basically stands for Numerical Python. I can assume that you are familiar with the basic Python Programming. So, Let's start.

What is Linear Algebra?

Linear algebra is a branch of mathematics that is widely used throughout science and engineering. Yet because linear algebra is a form of continuous rather than discrete mathematics, many computer scientists have little experience with it. A good understanding of linear algebra is essential for understanding and working with many machine learning algorithms, especially deep learning algorithms.
In this Module, we learn about Linear algebra topic called Matrix and their Operations i.e Vectors, Matrix Multiplication, determinant, trace and many more.
And learn how to implement these in the Programming way. so, for this, we will be using the Python Package called Numpy.

let's familiarise with numpy and their installation.

Numpy -

Numpy

NumPy is the fundamental package for scientific computing with Python. adding

support for large, multi-dimensional arrays and matrices, along with a large collection of high mathematics functions to operate on these arrays.

> *Numpy Contains—*
> *-> a powerful N-dimensional array object.*
> *-> sophisticated (broadcasting) functions.*
> *-> tools for integrating C/C++ and Fortran code.*
> *-> useful **linear algebra**, Fourier transform, and random number capabilities.*

Installation of Numpy —

Numpy Installation via pip

```
python -m pip install numpy
```

Numpy Installation for Anaconda Jupyter Notebook user

```
conda install -c anaconda numpy
```

To use the Numpy library in the program all you need to do in to import it.

```
import numpy as np
```

don't confuse with np it just a method of the renaming of module **numpy** into **np**.

Now, the system is set up for writing the programs and learn Mathematics with hands-on Implementation.
So, Take a cup of coffee, wear your headphones and enjoy coding.

Let's familiarize with the basics of Numpy —

Basic of Numpy -

Note: an array can be referred to as a matrix.

a) Create an Array and Print an Array —

```
In [1]: numpy_matrix = np.array([1,2,3])
In [2]: print(numpy_matrix)
Out[2]: [1 2 3]

In [3]: numpy_matrix = np.array([(1,2,3),(4,5,6)])
In [4]: print(numpy_matrix)
Out[4]: [[1 2 3]
         [4 5 6]]
```

b) Finding the Number of array dimensions —

```
In [5]: numpy_matrix = np.array([2,3,4])
In [6]: print(numpy_matrix.ndim)
Out[6]: 1
```

```
In [7]: numpy_matrix2 = np.array([(1,2,3),(2,3,4)])
In [8]: print(numpy_matrix2.ndim)
Out[8]: 2
```

c) Finding the shape of an Array —

```
In [9]: numpy_matrix = np.array([(1, 2, 3, 4), (5, 6, 7, 8)])
In [10]: print(numpy_matrix.shape)
Out[10]: (2, 4)
```

d) Reshaping of an Array —

```
In [11]: numpy_matrix = np.array([(1, 2, 3, 4), (5, 6, 7, 8)])
In [12]: print(numpy_matrix)
Out[12]:[[1 2 3 4]
         [3 4 5 6]]


In [13]: numpy_matrix = numpy_matrix.reshape(4, 2)
In [14]: print(numpy_matrix)

Out[14]: [[1, 2]
          [3, 4]
          [5, 6]
          [7, 8]]
```

e) Some Mathematical Operation —

```
# Addition -

In [15]: a = np.array([(1,2,3,4),(3,4,5,6)])
In [16]: b = np.array([(1,2,3,4),(3,4,5,6)])
In [17]: print(a+b)
Out[17]: [[ 2  4  6  8]
          [ 6  8 10 12]]

# Substraction -

In [18]: a = np.array([(1,2,3,4),(3,4,5,6)])
In [19]: b = np.array([(1,2,3,4),(3,4,5,6)])
```

```
In [20]: print(a-b)
Out[20]: [[0 0 0 0]
          [0 0 0 0]]
```

 # Multiplication -

```
In [21]: a = np.array([(1,2,3,4),(3,4,5,6)])
In [22]: b = np.array([(1,2,3,4),(3,4,5,6)])
In [23]: print(a*b)
Out[23]: [[ 1  4  9 16]
          [ 9 16 25 36]]
```

 # Divide -

```
In [24]: a = np.array([(1,2,3,4),(3,4,5,6)])
In [25]: b = np.array([(1,2,3,4),(3,4,5,6)])
In [26]: print(a/b)
Out[26]: [[1. 1. 1. 1.]
          [1. 1. 1. 1.]]
```

 # Remainder -

```
In [27]: a = np.array([(1,2,3,4),(3,4,5,6)])
In [28]: b = np.array([(1,2,3,4),(3,4,5,6)])
In [29]: print(a%b)
Out[29]: [[0 0 0 0]
          [0 0 0 0]]
```

 # Squareroot -

```
In[30]: a = np.array([(1,2,3,4),(3,4,5,6),(7,8,9,10)])
In[31]: print(np.sqrt(a))
Out[31]: [[1.         1.41421356 1.73205081 2.        ]
          [1.73205081 2.         2.23606798 2.44948974]
          [2.64575131 2.82842712 3.         3.16227766]]
```

f) Finding the mean, median, variance and standard deviation of an array:

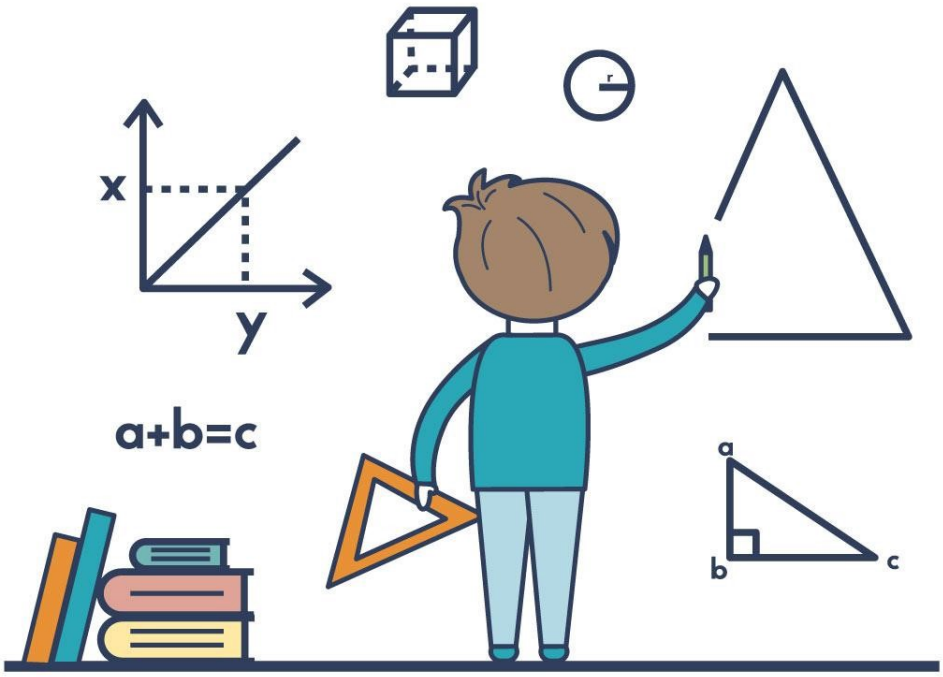
```
In[31]:a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])

In[31]: print(np.mean(a))
Out[31]: 5.0

In[31]: print(np.median(a))
Out[31]: 5.0

In[31]: print(np.var(a))
```

```
Out[31]: 6.666666666666667

In[31]: print(np.std(a))
Out[31]: 2.581988897471611
```

for more practice, click here

Now, we come to the main part i.e. Linear Algebra and we first discuss the Vectors.

Vectors -



Vectors(Mathematics)

Vectors and vector spaces are fundamental to *linear algebra*, and they're used in many machine learning models. Vectors describe spatial lines and planes, enabling you to perform calculations that explore relationships in multi-dimensional space.

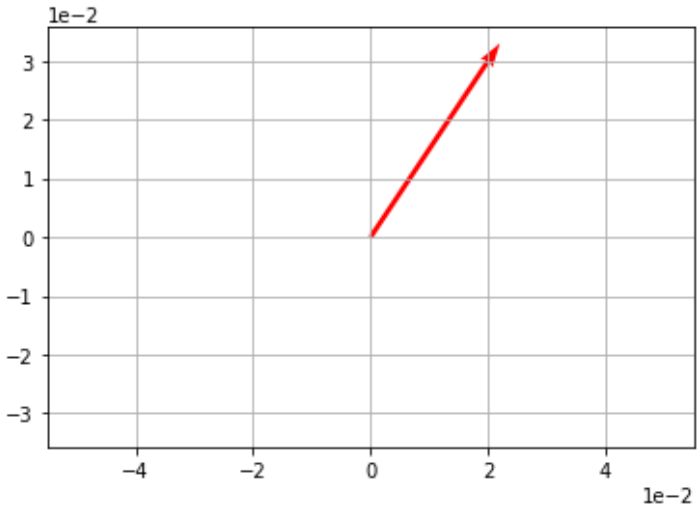
What is Vector -

A vector is a numeric element that has both *magnitude* and *direction*. The magnitude represents a distance (for example, "2 miles") and the direction indicates which way the

vector is headed (for example, “East”). Vectors are defined by an n-dimensional coordinate that describes a point in space that can be connected by a line from an arbitrary origin.

Our vector can be written as $\mathbf{v}=(2,3)$, Run the below code in your cell to visualize the vector $\mathbf{v}$ (which remember is described by the coordinate $(2,3)$).
Make sure you install the matplotlib Python Package.

```
import numpy as np
import matplotlib.pyplot as plt

# We'll use a numpy array for our vector
v = np.array([2,3])

# and we'll use a quiver plot to visualize it.
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, *v, scale=10, color='r')
plt.show()
```



Output

Calculating Magnitude —

Calculating the magnitude of the vector from its cartesian coordinates requires measuring the distance between the arbitrary starting point and the vector head point.

For a two-dimensional vector, we're actually just calculating the length of the hypotenuse in a right-angled triangle — so we could simply invoke Pythagorean theorem and calculate the square root of the sum of the squares of its components. So, here we can calculate the magnitude of a vector by two methods -

Code 1 (simple)-by using math module

```
import math
v = np.array([2,3])
vMag = math.sqrt(v[0]**2 + v[1]**2)
print (vMag)
```

Output — 3.605551275463989

Code 2 (By using *linalg* library provided by numpy) -
In Python, *numpy* provides a linear algebra library named **linalg** that makes it easier to work with vectors — you can use the **norm** function in the following code to calculate the magnitude of a vector.

```
import numpy as np

v = np.array([2,3])
vMag = np.linalg.norm(v)
print (vMag)
```

the output is the same as above

Calculating Direction -

To calculate the direction, or *amplitude*, of a vector from its cartesian coordinates, you must employ a little trigonometry. We can get the angle of the vector by calculating the *inverse tangent*; sometimes known as the *arctan* (the *tangent* calculates an angle as a ratio — the inverse tangent, or **tan-1**, expresses this in degrees).
n any right-angled triangle, the tangent is calculated as the *opposite* over the *adjacent*. In a two dimensional vector, this is the *y value over the x* value, so for our **v** vector (2,3):

$=>\tan(\theta) = 3/2$

$=>\theta$ = tan-1(3/2) = 56.309932474020215

Run the following Python code in your cell to confirm this:

```
import math
import numpy as np

v = np.array([2,1])
vTan = v[1] / v[0]
print ('tan = ' + str(vTan))
vAtan = math.atan(vTan)
# atan returns the angle in radians, so convert to degrees
print('inverse-tan = ' + str(math.degrees(vAtan)))
```

and you will see the output —
tan = 1.5
inverse-tan = 56.309932474020215

There is an added complication, however, because if the value for *x or y* (or both) is negative, the orientation of the vector is not standard, and a calculator can give you the wrong tan-1 value from this method. To ensure you get the correct direction for your vector, use the following rules:

> *Both* x and y *is positive: Use the tan-1 value.*
>
> x is negative, y *is positive: Add 180 to the tan-1 value.*
>
> *Both* x and y *is negative: Add 180 to the tan-1 value.*
>
> x is positive, y *is negative: Add 360 to the tan-1 value.*

In the previous Python code, we used *math.**atan** function to calculate the inverse tangent from a numeric tangent. The *numpy* library includes a similar **arctan** function. When working with numpy arrays, you can also use the *numpy.**arctan2** function to return the inverse tangent of an array-based vector in radians*, and you can use the *numpy.**degrees** function to convert this to degrees. The **arctan2** function automatically makes the necessary adjustment for negative x* and *y* values.

```
import numpy as np

v = np.array([2,3])
print ('v: ' + str(np.degrees(np.arctan2(v[1], v[0]))))

s = np.array([-3,2])
print ('s: ' + str(np.degrees(np.arctan2(s[1], s[0]))))
```

output -

v: 56.309932474020215

s: 146.30993247402023

Vector Addition -

So far, we've worked with one vector at a time. What happens when you need to add two vectors.

let's take an example —

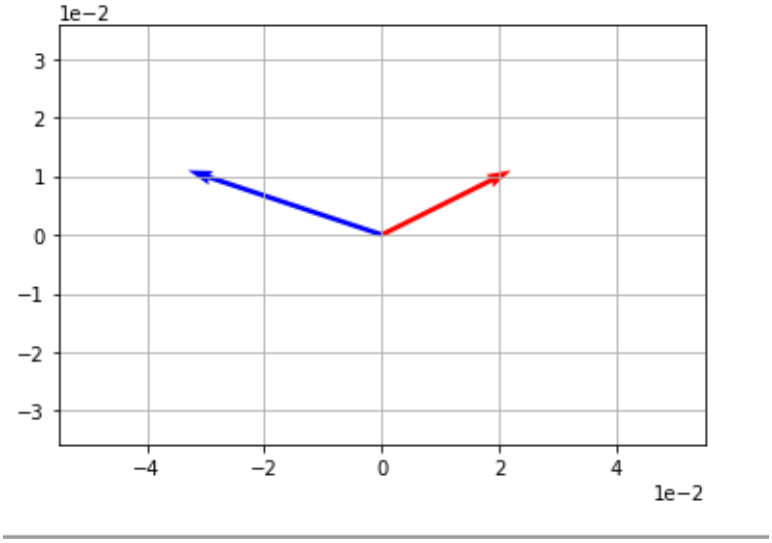
v = (2,1)

s = (-3,1)

addition of v and s give (-1,2)

Run the cell below to create **v and plot it together with s.**

```
import math
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

v = np.array([2,1])
s = np.array([-3,1])
print (s)

# Plot v and s
vecs = np.array([v,s])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver
```



Plotting of v and s vectors

now, we add the v and s vectors

```
z = v + s
print(z)
```

output -

[-1 2]

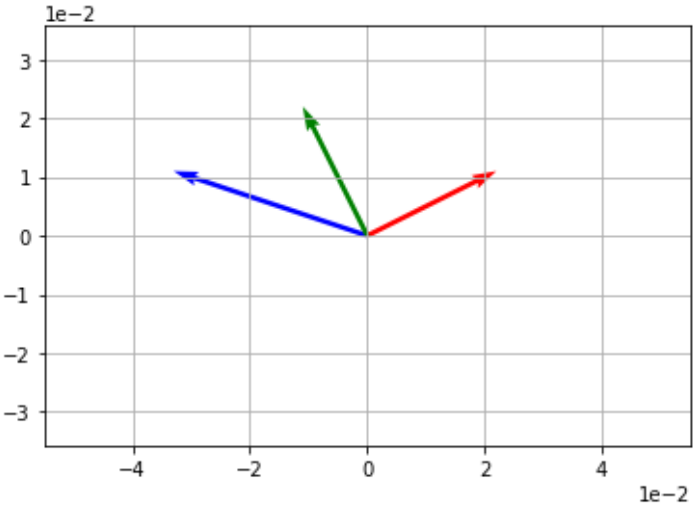
now, let see how this is looking like (Plotting)

```
vecs = np.array([v,s,z])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['r', 'b', 'g'],
scale=10)
plt.show()
```

Vector Multiplic

Vector multiplicatic

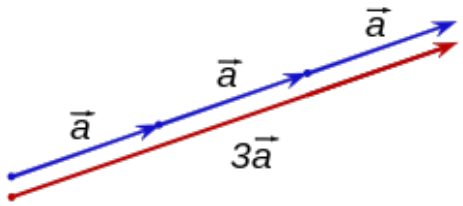
- Scalar Multiplic
- Dot Product Multiplication
- Cross Product Multiplication



Plotting of s, v, and their summation

here, the blue arrow is vector s, the red arrow is vector v and the green arrow is them

Scalar Multiplication -



Let's start with *scalar* multiplication — in other words, multiplying a vector by a single numeric value.

suppose I want to multiply vector v by 3, which I could write like this: w = 3v.

so, basically, value 3 multiply to all elements of the vector. for example -

=> v = (-1,1)

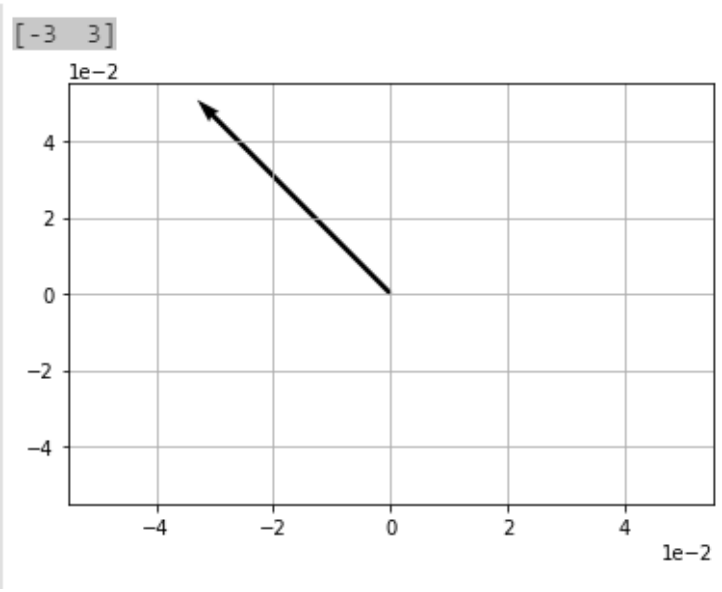
=> w = 3v

=> w = (-3,3)

let's do it by running the code -

```
import numpy as np
import matplotlib.pyplot as plt
import math
```

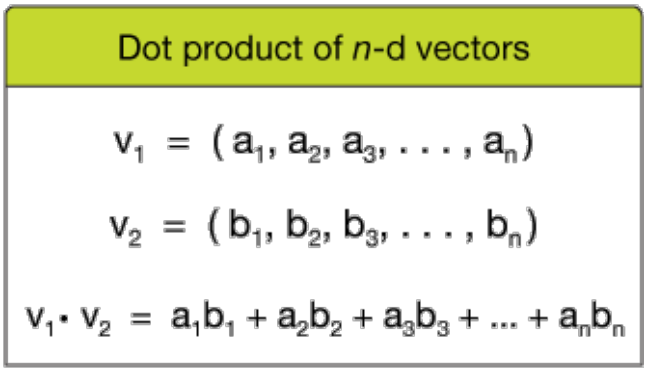
```
v = np.array([-1,1])

w = 3 * v
print(w)

# Plot w
origin = [0], [0]
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, *w, scale=10)
plt.show()
```



output

The same approach is taken for scalar division.
Try it for yourself — to calculate a new vector named **b** based on the following definition: b = v/2
code of this problem is here

Dot Product Multiplication -



To get a scalar product, we calculate the *dot product*. This takes a similar approach to multiply a vector by a scalar, except that it multiplies each component pair of the vectors and sums the results. To indicate that we are performing a dot product operation, we use the • operator:

$\vec{v} \cdot \vec{s} = (v1 \cdot s1) + (v2 \cdot s2) \ldots + (vn \cdot sn)$

So for vectors **v (2,3) and s (-3,1)**, our calculation looks like this:
$\vec{v} \cdot \vec{s} = (2 \cdot -3) + (3 \cdot 1) = -6 + 3 = -3$

In Python, you can use the *numpy.* dot function to calculate the dot product of two vector arrays.

```
import numpy as np

v = np.array([2,3])
s = np.array([-3,1])
d = np.dot(v,s)
print (d)
```

you will the same output as we calculated: -3

In Python 3.5 and later, we can also use the @ operator to calculate the dot product:

```
import numpy as np

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

```
d = v @ s
print (d)
```

The Cosine Rule -

$$\overline{u} \cdot \overline{v} = \left\|\overline{u}\right\|\left\|\overline{v}\right\| \cos\,\theta$$

$$\frac{\overline{u} \cdot \overline{v}}{\left\|\overline{u}\right\|\left\|\overline{v}\right\|} = \cos\,\theta$$

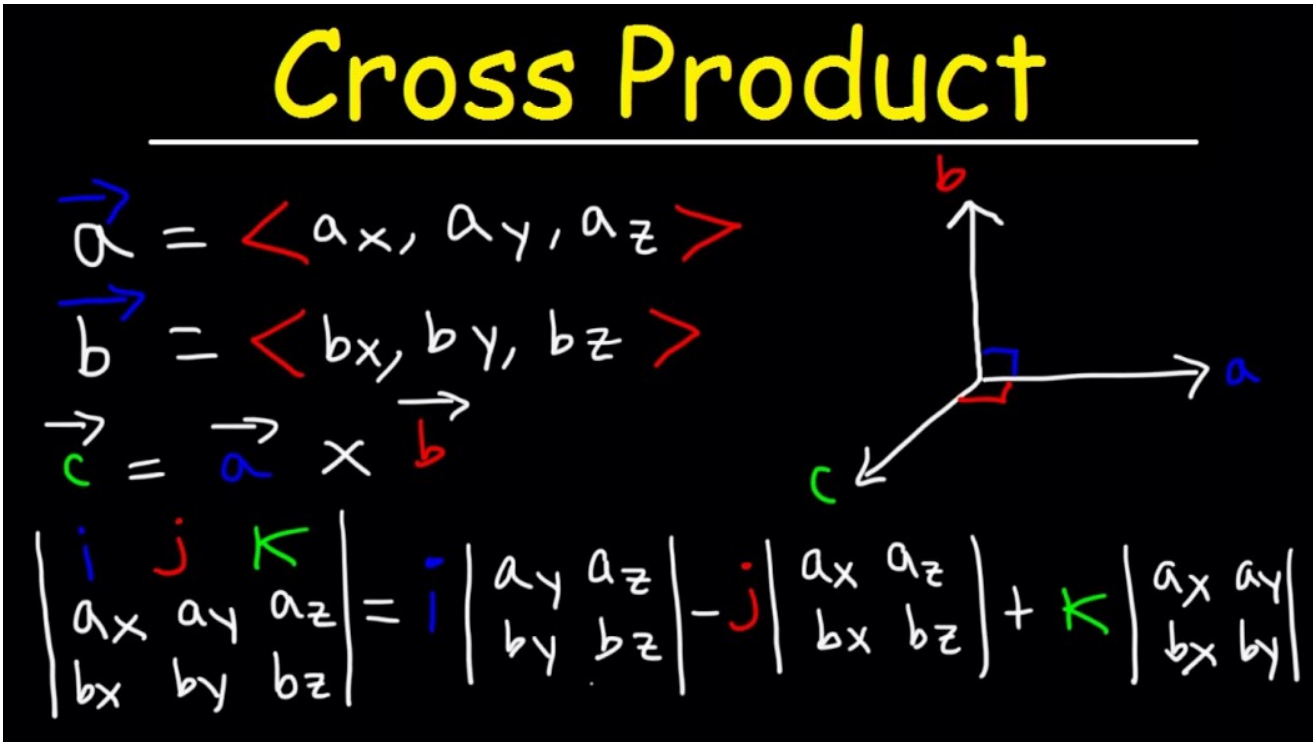
Cosine rule

A useful property of vector dot product multiplication is that we can use it to calculate the cosine of the angle between two vectors.
Here's that calculation in Python:

```
import math
import numpy as np

# define our vectors
v = np.array([2,3])
s = np.array([-3,1])

# get the magnitudes
vMag = np.linalg.norm(v)
sMag = np.linalg.norm(s)

# calculate the cosine of theta
cos = (v @ s) / (vMag * sMag)

# so theta (in degrees) is:
theta = math.degrees(math.acos(cos))

print(theta)
```

Match your output with theta = 105.25511870305779

Cross Product Multiplication -



Cross Product

To get the *vector product* of multiplying two vectors together, you must calculate the *cross product*. The result of this is a new vector that is at right angles to both the other vectors in 3D Euclidean space. This means that the cross-product only really makes sense when working with vectors that contain three components.

In Python, we can use the *numpy.*cross-function to calculate the cross product of two vector arrays:

```
import numpy as np
p = np.array([2,3,4])
q = np.array([1,2,-2])
r = np.cross(p,q)
print(r)
```

Matrices -

A matrix is an array of numbers that are arranged into rows and columns.
So, we already learn how to create a matrix(Array) and their basic operation in section Basic of Numpy. here, we learn more advanced of Matrix i.e Transpose, Inverse, Eigen Vector, etc. but before we start more advanced let's do a simple program to create a matrix.

In Python, we can define a matrix as a 2-dimensional *numpy.*array, like this:

```
import numpy as np

A = np.array([[1,2,3],
              [4,5,6]])
print (A)
```

The output of this code is as you expect -

```
[[1 2 3]
 [4 5 6]]
```

but we can also use *numpy.*matrix type, which is a specialist **subclass** of an array:

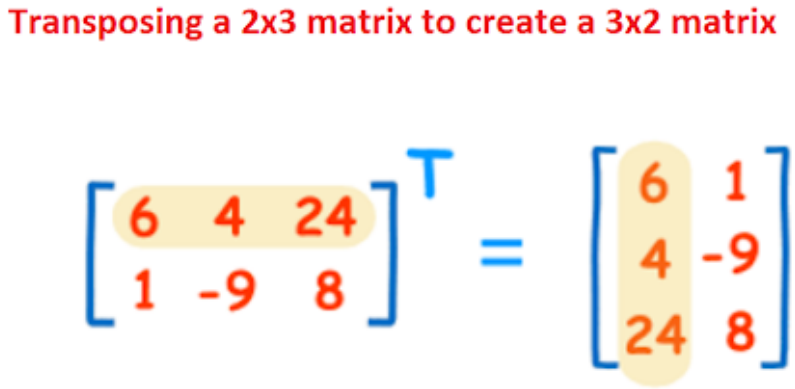
```
import numpy as np

M = np.matrix([[1,2,3],
               [4,5,6]])
print (M)
```

The output of this code is the same as np.array() give

There are some differences in behavior between ***array*** and ***matrix*** types — particularly with regards to **multiplication** (which we'll explore later). You can use either, but most experienced Python programmers who need to work with both vectors and matrices tend to prefer the ***array*** type for consistency.
We also prefer an array.

Matrix Transposition -

You can *transpose* a matrix, that switches the orientation of its rows and columns. You indicate this with a superscript **T**, like this:



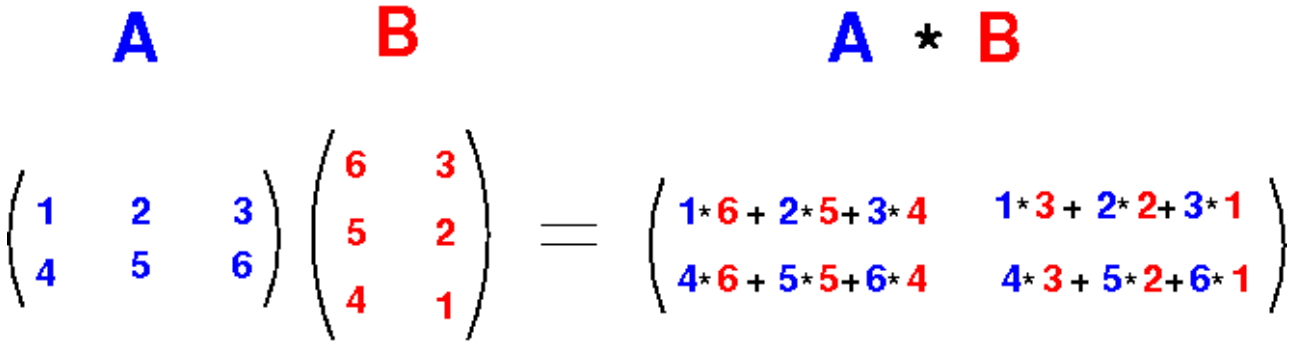
In Python, both *numpy.*array and numpy.matrix have a **T** function:

```
import numpy as np

A = np.array([[1,2,3],
              [4,5,6]])
print(A.T)
```

Output -

```
[[1 4]
 [2 5]
 [3 6]]
```

Matrix Multiplication -



To multiply two matrices together, you need to calculate the *dot product* of rows and columns. This means multiplying each of the elements in each row of the first matrix by each of the elements in each column of the second matrix and adding the results. We perform this operation by applying the *RC* rule — always multiplying ***R*ows by *C*olumns**. For this to work, the number of ***columns*** in the first matrix must be the same as the number of ***rows*** in the second matrix so that the matrices are *conformable* for the dot product operation.

In Python, we can use *numpy.***dot function** or the @ operator to multiply matrices and two-dimensional arrays.

```
import numpy as np

A = np.array([[1,2,3],
              [4,5,6]])
B = np.array([[9,8],
              [7,6],
              [5,4]])
print(np.dot(A,B))
print(A @ B)
```

Output -

```
[[ 38  32]
 [101  86]]

[[ 38  32]
 [101  86]]
```

As you can see both ***np.dot(A, B)*** and ***A@B*** give the same output. so, you can anyone for multiplication.

Now, here is one case where there is a difference in behavior between ***numpy.array*** and ***numpy.matrix*****,** You can also use a regular multiplication operator with a matrix, but not with an array:

```
import numpy as np

A = np.matrix([[1,2,3]
               ,[4,5,6]])
B = np.matrix([[9,8],
               [7,6],
               [5,4]])
print(A * B)
```

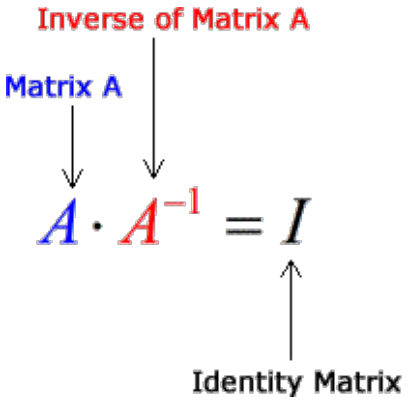
You can compare the output of this code with the upper one that both are same.

Now, you already read about the property of the matrices that commutative law is not applicable in matrix multiplication.so, let's proof this one.

```
import numpy as np

A = np.array([[2,4],
              [6,8]])
B = np.array([[1,3],
              [5,7]])
print(A @ B)
print(B @ A)
```

Analyze your output that commutative law is applicable or not.

The inverse of Matrix -



So, how do you calculate the inverse of a matrix? For a 2x2 matrix

1)Either we use the Gauss — Jordan method(row or column elementary operation)
2)you can follow this formula:

$$\mathbf{B} = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$ If AD - BC ≠ 0, then B has an inverse, denoted B^{-1}

$$\mathbf{B^{-1}} = \frac{1}{AD-BC}\begin{bmatrix} D & -B \\ -C & A \end{bmatrix}$$

In Python, we can use *numpy.linalg.**inv** function to get the inverse of a matrix in an array* or *matrix* object:

```
import numpy as np

B = np.array([[6,2],
              [1,2]])

print(np.linalg.inv(B))
```

Additionally, the *matrix* type has an ***I*** method that returns the inverse matrix:

```
import numpy as np

B = np.matrix([[6,2],
               [1,2]])

print(B.I)
```

For larger matrices, the process to calculate the inverse is more complex.

Steps to calculate the Inverse of a Matrix:

- Step 1: calculating the Matrix of Minors,
- Step 2: then turn that into the Matrix of Cofactors,

- Step 3: then the Adjugate(also called Adjoint), and
- Step 4: multiply that by 1/Determinant.

As you can see, this can get pretty complicated. But the program to calculate is not so complicated.

```
import numpy as np

B = np.array([[6,2,1],
              [1,2,1],
              [2,3,4]])

print(np.linalg.inv(B))
```

So, till now we have learned multiplication, Inverse of the matrices, etc.
Let's done a problem to Solve the Systems of Equations with Matrices.
Can you do yourself?
here, is the problem —

$$2x + 4y = 18$$
$$6x + 2y = 34$$

you have to solve these equations with the help of Matrices.
Hint — your equation looks like this:

$$A = \begin{bmatrix} 2 & 4 \\ 6 & 2 \end{bmatrix} \quad X = \begin{bmatrix} x \\ y \end{bmatrix} \quad B = \begin{bmatrix} 18 \\ 34 \end{bmatrix}$$

Problem code solution is here

Transformations

Matrices and vectors are used together to manipulate spatial dimensions. This has a lot of applications, including the mathematical generation of 3D computer graphics, geometric modeling, and the training and optimization of machine learning algorithms. We're not going to cover the subject exhaustively here, but we'll focus on a few key concepts that are useful to know when you plan to work with *machine learning*.

Linear Transformations -

We can manipulate a vector by multiplying it with a matrix. The matrix acts as a function that operates on an input vector to produce a vector output. Specifically, matrix

multiplications of vectors are *linear transformations* that transform the input vector into the output vector.
for example, consider this matrix ***A*** and vector ***v***:

$$A = \begin{bmatrix} 2 & 3 \\ 5 & 2 \end{bmatrix} \quad \vec{v} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

We can define a transformation ***T*** like this: $T(\vec{v}) = A\vec{v}$

To perform this transformation, we simply calculate the dot product by applying the *RC* rule; multiplying each row of the matrix by the single column of the vector.
Here's the calculation in Python:

```
import numpy as np

v = np.array([1,2])
A = np.array([[2,3],
              [5,2]])

t = A@v
print (t)
```

In this case, both the input vector and the output vector have 2 components — in other words, the transformation takes a 2-dimensional vector and produces a new 2-dimensional vector; which we can indicate like this:

$$T : \mathbb{R}^2 \rightarrow \mathbb{R}^2$$

Note that the output vector may have a different number of dimensions from the input vector; so the matrix function might transform the vector from one space to another — or in notation:

$$\mathbb{R}^n \text{ -> } \mathbb{R}^m.$$

Here it is in Python:

```
# our transformation transforms the vector from 2-D space to 3-D
space

import numpy as np
v = np.array([1,2])
A = np.array([[2,3],
              [5,2],
              [1,1]])

t = A@v
print (t)
```

Transformations of Magnitude and Amplitude -

When we multiply a vector by a matrix, we transform it in at least one of the following two ways:

- Scale the length (*magnitude*) of the matrix to make it longer or shorter.
- Change the direction (*amplitude*) of the matrix.

For example, consider the following matrix and vector:

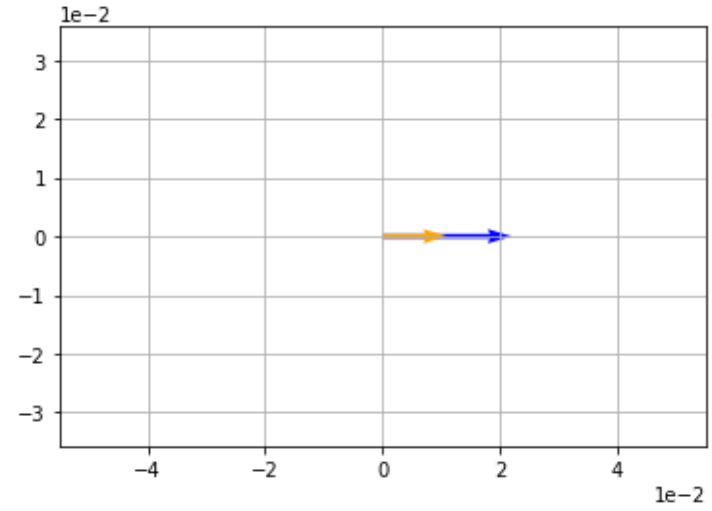
$$A = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \quad \vec{v} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

As before, we transform the vector ***v*** by multiplying it with the matrix ***A*.**
In this case, the resulting vector has changed in length (*magnitude*) but has not changed its direction (*amplitude*).
Let's visualize that in Python:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

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

```
                [0,2]])

t = A@v
print (t)

# Plot v and t
vecs = np.array([t,v])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'],
scale=10)
plt.show()
```

```
[2 0]
```



The output of the above code

The original vector ***v*** is shown in orange, and the transformed vector ***t*** is shown in blue — note that ***t*** has the same direction (*amplitude*) as ***v*** but a greater length (*magnitude*).

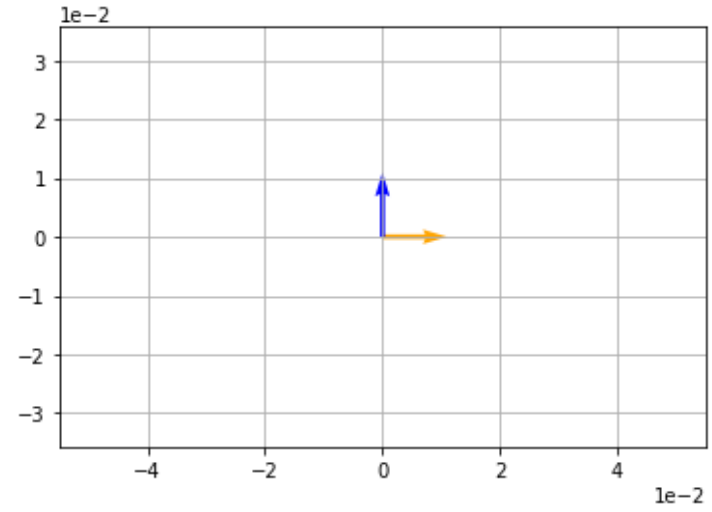
Now let's use a different matrix to transform the vector ***v***:

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

This time, the resulting vector has been changed to a different amplitude but has the same magnitude.

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

v = np.array([1,0])
A = np.array([[0,-1],
              [1,0]])

t = A@v
print (t)

# Plot v and t
vecs = np.array([v,t])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['orange', 'blue'],
scale=10)
plt.show()
```

```
[0 1]
```



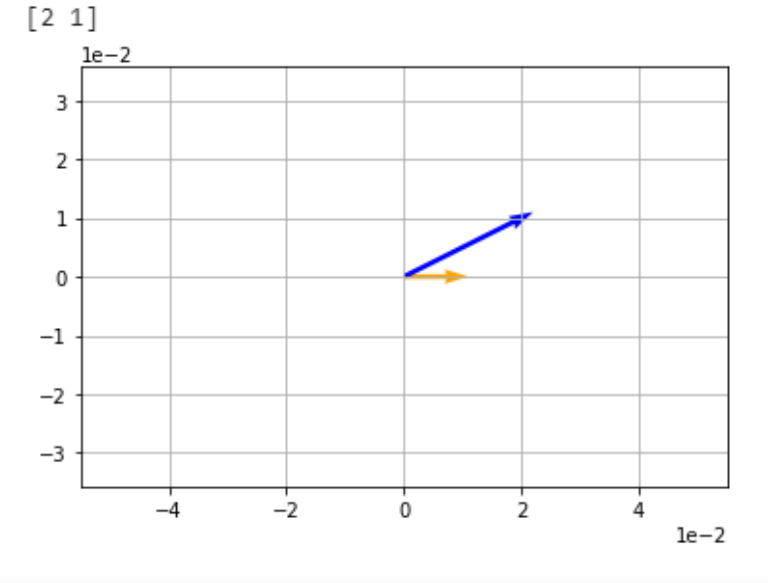
The output of the above code

Now let's see change the matrix one more time:

$$\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

Now our resulting vector has been transformed into a new amplitude *and* magnitude — the transformation has affected both direction and scale.

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

v = np.array([1,0])
A = np.array([[2,1],
              [1,2]])

t = A@v
print (t)

# Plot v and t
vecs = np.array([v,t])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['orange', 'blue'],
scale=10)
plt.show()
```



The output of the above code.

Affine Transformations -

An Affine transformation multiplies a vector by a matrix and adds an offset vector,

sometimes referred to as *bias*; like this:

$$T(\vec{v}) = A\vec{v} + \vec{b}$$

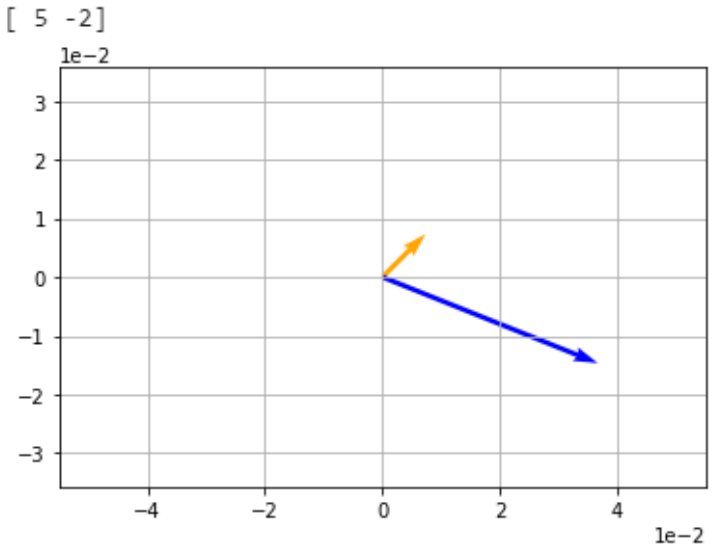
For example:

$$\begin{bmatrix} 5 & 2 \\ 3 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -2 \\ -6 \end{bmatrix} = \begin{bmatrix} 5 \\ -2 \end{bmatrix}$$

This kind of transformation is actually the basis of linear regression, which is a core foundation for machine learning. The matrix defines the *features*, the first vector is the *coefficients*, and the bias vector is the *intercept*.

here's an example of an Affine transformation in Python:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

v = np.array([1,1])
A = np.array([[5,2],
              [3,1]])
b = np.array([-2,-6])

t = A@v + b
print (t)

# Plot v and t
vecs = np.array([v,t])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['orange', 'blue'],
scale=15)
plt.show()
```



The output of the above code

Eigenvectors a

So we can see that ınge its direction, length, or both. When the transformation only affects scale (in other words, the output vector has a different magnitude but the same amplitude as the input vector), the matrix multiplication for the transformation is the equivalent operation as some scalar multiplication of the vector.

For example, earlier we examined the following transformation that dot-multiplies a vector by a matrix:

$$\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$

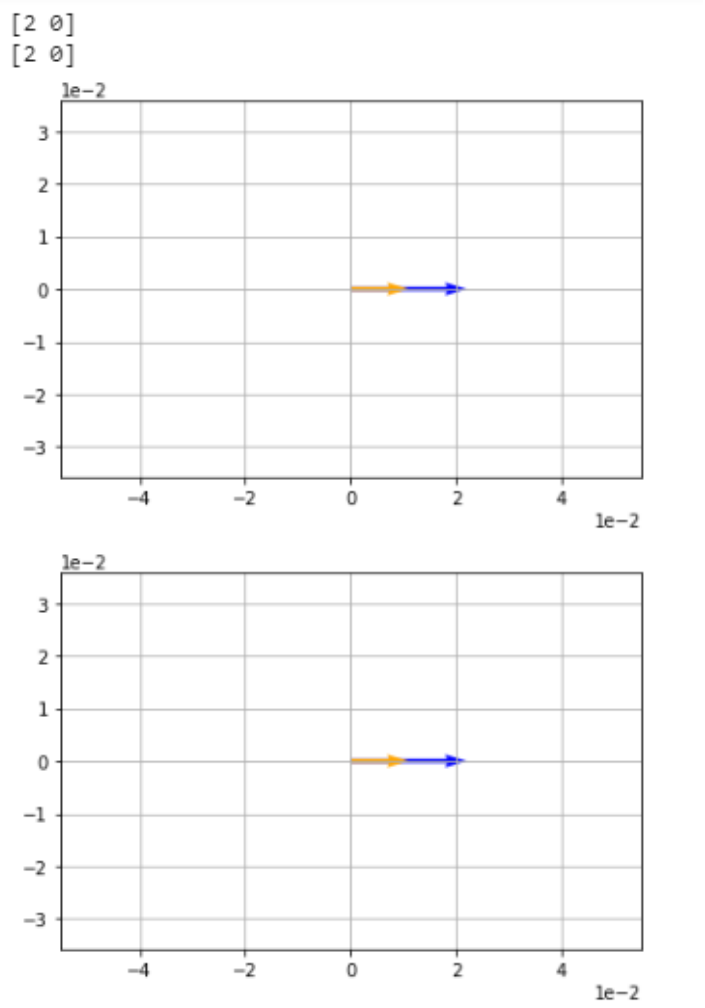
We can achieve the same result by multiplying the vector by the scalar value ***2***:

$$2 \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$

The following python performs both of this calculation and shows the results, which are identical.

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

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

t1 = A@v
print (t1)
t2 = 2*v
print (t2)

fig = plt.figure()
a=fig.add_subplot(1,1,1)
# Plot v and t1
vecs = np.array([t1,v])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'],
scale=10)
plt.show()
a=fig.add_subplot(1,2,1)
# Plot v and t2
vecs = np.array([t2,v])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'],
scale=10)
plt.show()
```



The output of the above code

In cases like these, where a matrix transformation is the equivalent of a scalar-vector multiplication, the scalar-vector pairs that correspond to the matrix are known respectively as eigenvalues and eigenvectors. We generally indicate eigenvalues using the Greek letter lambda (λ), and the formula that defines eigenvalues and eigenvectors with respect to a transformation is:

$$T(\vec{v}) = \lambda\vec{v}$$

Where the vector $\boldsymbol{v}$ is an eigenvector and the value $\boldsymbol{\lambda}$ is an eigenvalue for transformation $\boldsymbol{T}$.

When the transformation $\boldsymbol{T}$ is represented as matrix multiplication, as in this case where the transformation is represented by matrix $\boldsymbol{A}$:

$$T(\vec{v}) = A\vec{v} = \lambda\vec{v}$$

where $\boldsymbol{\lambda}$ is a scalar value called the 'eigenvalue'. This means that the linear transformation A on vector $\vec{v}$ is completely defined by $\boldsymbol{\lambda}$.

We can rewrite the equation as follows:

$$A\vec{v} - \lambda\vec{v} = 0$$
$$\Rightarrow \vec{v}(A - \lambda I) = 0,$$

Where, I is the identity matrix of the same dimensions as A.

A matrix can have multiple eigenvector-eigenvalue pairs, and you can calculate them manually. However, it's generally easier to use a tool or programming language. For example, in Python, you can use the ***linalg.eig*** function, which returns an array of eigenvalues and a matrix of the corresponding eigenvectors for the specified matrix.

Here's an example that returns the eigenvalue and eigenvector pairs for the following matrix:

$$\boldsymbol{A} = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}$$

```
import numpy as np
A = np.array([[2,0],
              [0,3]])
eVals, eVecs = np.linalg.eig(A)
print(eVals)
print(eVecs)
```

Output -

```
[2. 3.]
```

```
[[1. 0.]
 [0. 1.]]
```

So there are two eigenvalue-eigenvector pairs for this matrix, as shown here:

$$\lambda_1 = 2, \vec{v_1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \lambda_2 = 3, \vec{v_2} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Let's verify that multiplying each eigenvalue-eigenvector pair corresponds to the dot-product of the eigenvector and the matrix. Here's the first pair:

$$2 \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix} \quad \textit{and} \quad \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$

So far so good. Now let's check the second pair:

$$3 \times \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \end{bmatrix} \quad \textit{and} \quad \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \end{bmatrix}$$

So our eigenvalue-eigenvector scalar multiplications do indeed correspond to our matrix-eigenvector dot-product transformations.

Here's the equivalent code in Python, using the ***eVals*** and ***eVecs*** variables you generated in the previous code cell:

```
vec1 = eVecs[:,0]
lam1 = eVals[0]

print('Matrix A:')
print(A)
print('-------')

print('lam1: ' + str(lam1))
print ('v1: ' + str(vec1))
```

```
print ('Av1: ' + str(A@vec1))
print ('lam1 x v1: ' + str(lam1*vec1))

print('-------')

vec2 = eVecs[:,1]
lam2 = eVals[1]

print('lam2: ' + str(lam2))
print ('v2: ' + str(vec2))
print ('Av2: ' + str(A@vec2))
print ('lam2 x v2: ' + str(lam2*vec2))
```

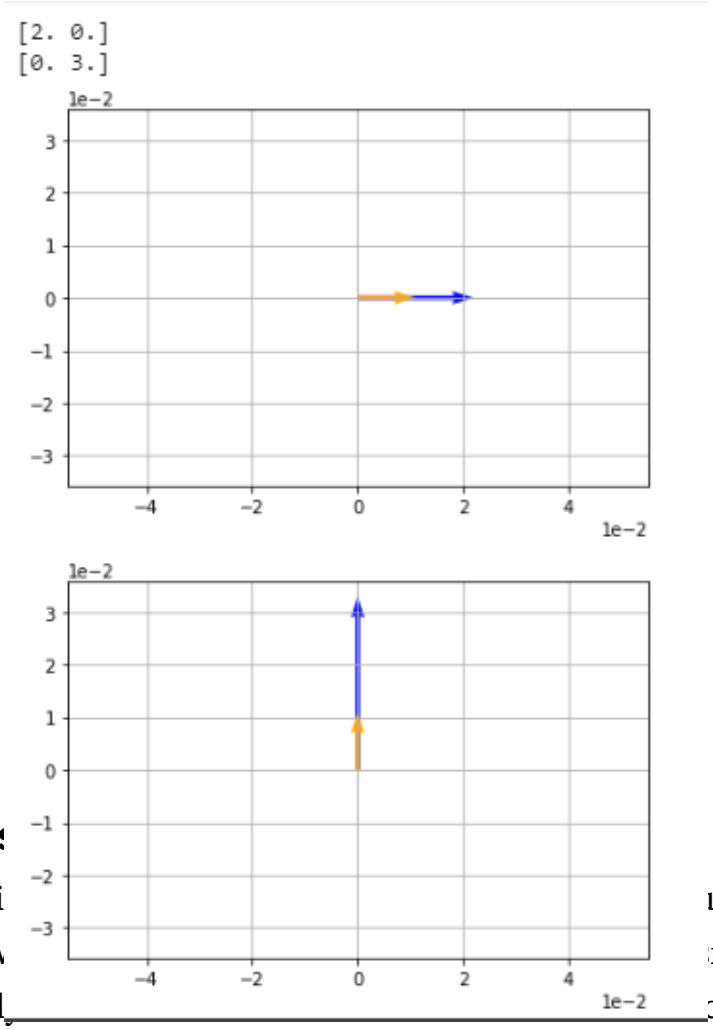
The output of this code -

```
Matrix A:
[[2 0]
 [0 3]]
-------
lam1: 2.0
v1: [1. 0.]
Av1: [2. 0.]
lam1 x v1: [2. 0.]
-------
lam2: 3.0
v2: [0. 1.]
Av2: [0. 3.]
lam2 x v2: [0. 3.]
```

You can use the following code to visualize these transformations:

```
t1 = lam1*vec1
print (t1)
t2 = lam2*vec2
print (t2)

fig = plt.figure()
a=fig.add_subplot(1,1,1)
# Plot v and t1
vecs = np.array([t1,vec1])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'],
```

```
scale=10)
plt.show()
a=fig.add_subplot(1,2,1)
# Plot v and t2
vecs = np.array([t2,vec2])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'],
scale=10)
plt.show()
```



Eigen decompos

So we've learned a li ı may be wondering what use they are. W formation matrices. Recall that previousl ctor changes its magnitude, amplitude, or both. Without getting too technical about it, we need to remember that vectors can exist in any spatial orientation, or *basis*; and the same transformation can be applied in different *bases*.

Similarly, can you examine the following matrix transformation:

We can decompose a matrix using the following formula:

$$\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$

Where $\boldsymbol{A}$ is a transformation that can be applied to a vector in its current base, $\boldsymbol{Q}$ is a matrix of eigenvectors that defines a change of basis, and $\boldsymbol{\Lambda}$ is a matrix with eigenvalues on the diagonal that defines the same linear transformation as $\boldsymbol{A}$ in the base defined by $\boldsymbol{Q}$.

Let's look at these in some more detail. Consider this matrix:

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

$\boldsymbol{Q}$ is a matrix in which each column is an eigenvector of $\boldsymbol{A}$; which as we've seen previously, we can calculate using Python:

```
import numpy as np

A = np.array([[3,2],
              [1,0]])

l, Q = np.linalg.eig(A)
print(Q)
```

So for matrix $\boldsymbol{A}$, $\boldsymbol{Q}$ is the following matrix:

$$\boldsymbol{Q} = \begin{bmatrix} 0.96276969 & -0.48963374 \\ 0.27032301 & 0.87192821 \end{bmatrix}$$

$\boldsymbol{\Lambda}$ is a matrix that contains the eigenvalues for $\boldsymbol{A}$ on the diagonal, with zeros in all other elements; so for a 2x2 matrix, Λ will look like this:

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

In our Python code, we've already used the ***linalg.eig*** function to return the array of eigenvalues for ***A*** into the variable ***l***, so now we just need to format that as a matrix:

```
L = np.diag(l)
print (L)
```

So ***Λ*** is the following matrix:

$$\Lambda = \begin{bmatrix} 3.56155281 & 0 \\ 0 & -0.56155281 \end{bmatrix}$$

Now we just need to find ***Q-1***, which is the inverse of ***Q***:

```
Qinv = np.linalg.inv(Q)
print(Qinv)
```

The inverse of ***Q*** is:

$$Q^{-1} = \begin{bmatrix} 0.89720673 & 0.50382896 \\ -0.27816009 & 0.99068183 \end{bmatrix}$$

So what does that mean? Well, it means that we can decompose the transformation of *any* vector multiplied by a matrix ***A*** into the separate operations ***QΛQ-1***:

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

To prove this, let's take vector ***v***:

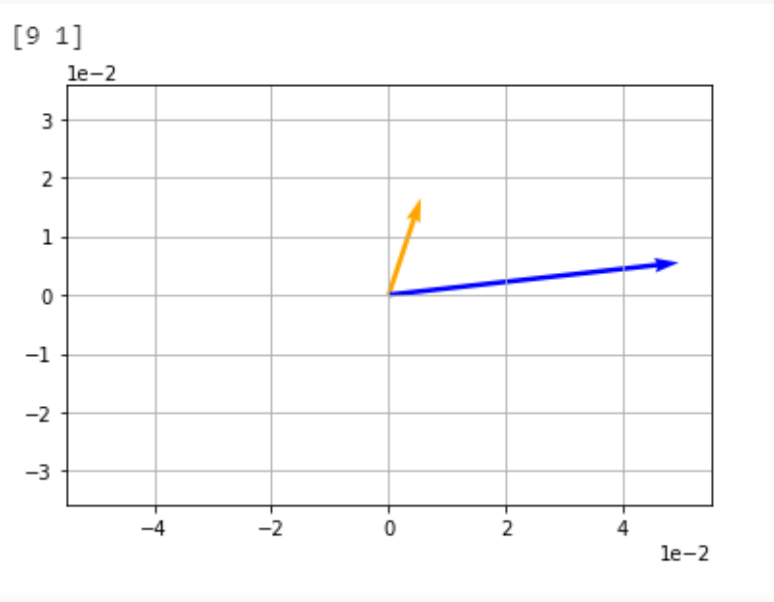
$$\vec{v} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

Our matrix transformation using ***A*** is:

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

So let's show the results of that using Python:

```
import matplotlib.pyplot as plt
%matplotlib inline
v = np.array([1,3])
t = A@v

print(t)

# Plot v and t
vecs = np.array([v,t])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['orange', 'b'],
scale=20)
plt.show()
```



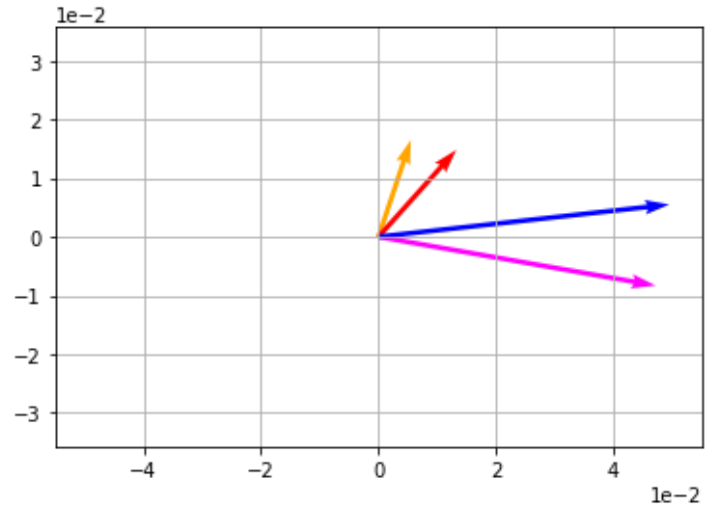
The output of the above code

And now, let's do the same thing using the ***QΛQ-1*** sequence of operations:

```
import math
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

t = (Q@(L@(Qinv)))@v

# Plot v and t
vecs = np.array([v,t])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['orange', 'b'],
scale=20)
plt.show()
```

And you can see that the output of this code is the same as above.
So ***A*** and ***QΛQ-1*** are equivalent.
If we view the intermediary stages of the decomposed transformation, you can see the transformation using ***A*** in the original base for ***v*** (orange to blue) and the transformation using ***Λ*** in the change of basis described by ***Q*** (red to magenta):

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

t1 = Qinv@v
t2 = L@t1
t3 = Q@t2

# Plot the transformations
vecs = np.array([v,t1, t2, t3])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['orange', 'red',
'magenta', 'blue'], scale=20)
plt.show()
```



Rank of a Matr

The **rank** of a squa he matrix. A **full rank** matrix has th nsion of the matrix. A **rank-de** ıensions. The inverse of a rank d

Consider the follo

The code

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

So from this visualization, it should be clear that the transformation ***Av*** can be performed by changing the basis for ***v*** using ***Q*** (from orange to red in the above plot) applying the equivalent linear transformation in that base using ***Λ*** (red to magenta), and switching back to the original base using ***Q-1*** (magenta to blue).

Let's find its eigenvalues (***Λ***):

```
import numpy as np
A = np.array([[1,2],
              [4,3]])
```

```
l, Q = np.linalg.eig(A)
L = np.diag(l)
print(L)
```

$$\Lambda = \begin{bmatrix} -1 & 0 \\ 0 & 5 \end{bmatrix}$$

This matrix has full rank. The dimensions of the matrix are 2. There are two non-zero eigenvalues.

Now consider this matrix:

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

Note that the second and third columns are just scalar multiples of the first column. Let's examine its eigenvalues:

```
B = np.array([[3,-3,6],
              [2,-2,4],
              [1,-1,2]])
lb, Qb = np.linalg.eig(B)
Lb = np.diag(lb)
print(Lb)
```

$$\Lambda = \begin{bmatrix} 3 & 0 & 0 \\ 0 & -6 \times 10^{-17} & 0 \\ 0 & 0 & 3.6 \times 10^{-16} \end{bmatrix}$$

Note that the matrix has only 1 non-zero eigenvalue. The other two eigenvalues are so extremely small as to be effectively zero. This is an example of a rank-deficient matrix; and as such, it has no inverse.

Inverse of a Square Full Rank Matrix -

We can calculate the inverse of a square full rank matrix by using the following formula:

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

Let's apply this to matrix ***A***:

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

Let's find the matrices for ***Q***, ***Λ-1***, and ***Q-1***:

```
import numpy as np
A = np.array([[1,2],
              [4,3]])

l, Q = np.linalg.eig(A)
L = np.diag(l)
print(Q)
Linv = np.linalg.inv(L)
Qinv = np.linalg.inv(Q)
print(Linv)
print(Qinv)
```

$$A^{-1} = \begin{bmatrix} -0.70710678 & -0.4472136 \\ 0.70710678 & -0.89442719 \end{bmatrix} \cdot \begin{bmatrix} -1 & -0 \\ 0 & 0.2 \end{bmatrix} \cdot \begin{bmatrix} -0.94280904 & 0.47140452 \\ -0.74535599 & -0.74535599 \end{bmatrix}$$

Let's calculate that in Python:

```
Ainv = (Q@(Linv@(Qinv)))
print(Ainv)
```

That gives us the result:

$$A^{-1} = \begin{bmatrix} -0.6 & 0.4 \\ 0.8 & -0.2 \end{bmatrix}$$

We can apply ***np.linalg.inv*** function directly to ***A*** to verify this:

```
print(np.linalg.inv(A))
```

This will give you the same A inverse value as above.

Let's end this here! All the code of this discussion is here.
So, congrats you do it perfectly.

I know it a little bit longer. but Hope I have given some idea about Numpy and the Linear Algebra and how it Apply in the field like Machine Learning, Deep Learning, and Computer Vision.

If you ❤ this article, Just give a clap.
please connect with me on LinkedIn, Github, Facebook, Twitter, Quora, and Instagram