Welcome to Linear Algebra Review

by R.J. Surio ©2021

Linear Algebra is one of the fundamental mathematics for Artificial Intelligence Development and also Computer Vision. We will see the applications of Linear Algebra in advanced mathematical techniques such as optimization, vectorized programming, and matrix manipulation.

In this module, we are going to review some of the useful equations needed in data science to refresh our skills in Python programming. We are going to cover:

- Vectors
- Operations with Vectors
- Matrices
- Tensors
- Determinants
- System of Linear Equations
- Visualizing of Vectors
- Modulus of a Vector
- The Sigmoid function

1. Vectors

NumPy or Numerical Python is a package or library that allows programmers to code and model computations and see them in action. You can check the NumPy documentation on how to use their APIs.

```
## You can install NumPy in your local machine by doing the following line without the "!"
!pip install numpy
## But in Google Colab NumPy is already installed in your session.
import numpy as np
print(f'NumPy library version: {np.__version__}')
```

```
Requirement already satisfied: numpy in d:\anaconda\lib\site-packages (1.18.5)
NumPy library version: 1.18.5
```

Defining Vectors, Matrices, and Tensors

Vectors, Matrices, and Tensors are the fundamental objects in Linear Algebra programming. We'll be defining each of these objects specifically in the Computer Science/Engineering perspective since it would be much confusing if we consider their Physics and Pure Mathematics definitions.

Scalars

Scalars are numerical entities that are represented by a single value.

```
x = np.array(8) # Example of scalar
y= np.array(4)
z= np.array(3)
scalar = x+y+z
scalar
```

```
15
```

Vectors

Vectors are array of numerical values or scalars that would represent any feature space. Feature spaces or simply dimensions or the parameters of an equation or a function.

```
a = np.array([4,2,1]) # Collection of scalars therefore a vector
a
```

```
array([4, 2, 1])
```

Matrices

Matrices are array of vectors or a multi-dimensional array for features for an equation or function.

```
b = np.array([
    [3,1,1],
    [4,5,6], # Collection of vectors
    [2,2,4]
])
b

# Accessing using index

b[1] # Prints the 2nd vector
b[1,0] # Prints the first element of 2nd vector
```

```
4
```

Tensors

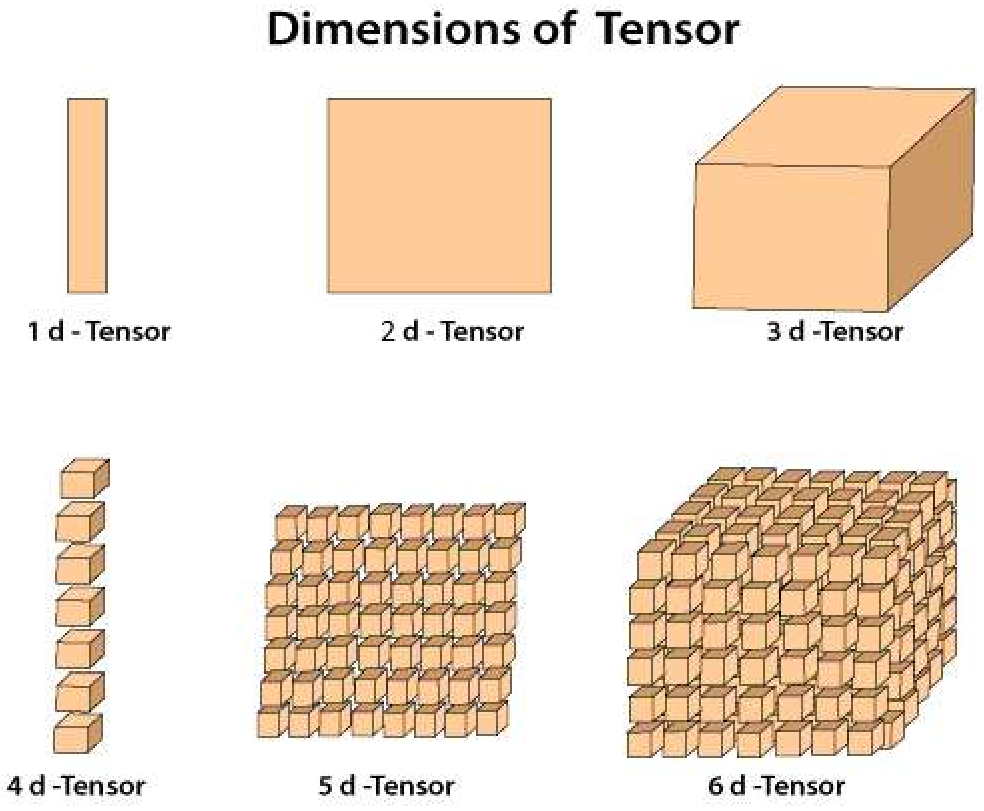
Tensors are an array of matrices. Tensors have dimensions, tensors can have alternate names depending on what dimension they are in. 1D tensors can be considered as vectors, 2D tensors can be considered are matrices, and 3D onwards are called high dimensional tensors.

```
 c = np.array([
    [[3,1,1],[4,5,6],[2,2,4]], # Collection of matrices
    [[4,1,5],[6,2,1],[5,1,1]]
])

c
c[1,0,2] # Accessing the 2nd matrix then the first vecter and its third element
```

```
     5
```

Here's a visual representation of the data types that we are going to use.



Describing Tensors

Describing tensors is very important if we want to perform basic to advanced operations with them. The fundamental ways in describing tensors are knowing their shape, size, and dimensions.

Shapes

The shape of a tensor tells us how many rows and columns are there in an axis.

```
""" c = np.array([
    [[3,1,1],[4,5,6],[2,2,4]],
    [[4,1,5],[6,2,1],[5,1,1]]
])
    """
c.shape # To check the shape of the array
```

```
(2, 3, 3)
```

Dimensions

In NumPy the dimension of a tensor is also called axes.

```
c.ndim # To check the dimension of the array
#b.ndim
```

```
3
```

Sizes

The size of a tensor/ vector/ matrix is simply the total number of elements in it.

```
d = np.array([c,c]) # A 4d-tensor
d.size # To check the number of elements on the array
```

```
36
```

```
d
```

```
array([[[[3, 1, 1],
         [4, 5, 6],
         [2, 2, 4]],

        [[4, 1, 5],
         [6, 2, 1],
         [5, 1, 1]]],


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

        [[4, 1, 5],
         [6, 2, 1],
         [5, 1, 1]]]])
```

Types of Matrices

The notation and use of matrices are probably one of the fundamentals of modern computing. Matrices are also handy representations of complex equations or multiple inter-related equations from 2-dimensional equations to even hundreds and thousands of them.

Let's say for example you have A and B as the system of equations.

$$A = \begin{cases} x + y \\ 4x - 10y \end{cases}$$

$$B = \begin{cases} x + y + z \\ 3x - 2y - z \\ -x + 4y + 2z \end{cases}$$

We could see that A is a system of 2 equations with 2 parameters. While B is a system of 3 equations with 3 parameters. We can represent them as matrices as:

$$A = \begin{bmatrix} 1 & 1 \\ 4 & -10 \end{bmatrix}$$

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

We'll represent the system of linear equations as a matrix. The entities or numbers in matrices are called the elements of a matrix. These elements are arranged and ordered in rows and columns which form the list/array-like structure of matrices. And just like arrays, these elements are indexed according to their position with respect to their rows and columns. This can be represented just like the equation below. Whereas A is a matrix consisting of elements denoted by $a_{i,j}$. Denoted by i is the number of rows in the matrix while j stands for the number of columns.
Do note that the $size$ of a matrix is $i \times j$.

$$A = \begin{bmatrix} a_{(0,0)} & a_{(0,1)} & \dots & a_{(0,j-1)} \\ a_{(1,0)} & a_{(1,1)} & \dots & a_{(1,j-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(i-1,0)} & a_{(i-1,1)} & \dots & a_{(i-1,j-1)} \end{bmatrix}$$

We already gone over some of the types of matrices as vectors but we'll further discuss them in this laboratory activity. Since you already know how to describe vectors using shape, dimensions and size attributes, we'll use them to analyze these matrices.

```
def describe_mat(matrix):
    print(f'Matrix:\n{matrix}\n\nShape:\t{matrix.shape}\nRank:\t{matrix.ndim}\n')
```

Matrices according to shape

Row and Column Matrices

Row and column matrices are common in vector and matrix computations. They can also represent row and column spaces of a bigger vector space. Row and column matrices are represented by a single column or single row. So with that being, the shape of row matrices would be $1 \times j$ and column matrices would be $i \times 1$.

```
## Declaring a Row Matrix

f = np.array([[3,4,1]])
describe_mat(f) # The '1' in shape represents the row
```

```
Matrix:
[[3 4 1]]

Shape:  (1, 3)
Rank:   2
```

```
## Declaring a Column Matrix
## Should have 1 element on each row

col = np.array([
                [4],
                [2],
                [1]
])

describe_mat(col) # The '3' in shape represents rows while the '1' is the column
```

```
Matrix:
[[4]
 [2]
 [1]]

Shape:  (3, 1)
Rank:   2
```

Square Matrices

Square matrices are matrices that have the same row and column sizes. We could say a matrix is square if $i = j$. We can tweak our matrix descriptor function to determine square matrices.

```
## Square matrices only exist on rank 2 or 2nd dimension

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

```
describe_mat(sq)
```

```
Matrix:
[[3 4]
 [1 2]]

Shape:  (2, 2)
Rank:   2
```

Matrices according to element values

Empty Matrix

An empty Matrix is a matrix that has no elements. It is always a subspace of any vector or matrix.

```
nul = np.empty((1,2)) # Empty declaration - has a small value considered as 0
nul

empt = np.array([],[]) # Nothing inside
empt
```

```
array([], dtype=[])
```

Zero/Null Matrix

A zero matrix can be any rectangular matrix but with all elements having a value of 0. In most texts, the zero matrix is denoted as $\emptyset$.
Check out: `numpy.zeros`

```
z = np.zeros((4,4)) # For creation of matrix will all zero values
z
```

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

```
## Another way to do matrix of specific element
m = np.full((4,4),0)
m
```

```
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

Ones Matrix

A ones matrix, just like zero matrices, can be any rectangular matrix but all of its elements are 1s instead of 0s.

Check out: [numpy.ones]

```
m = np.ones((3,3))
m
```

```
    array([[1., 1., 1.],
           [1., 1., 1.],
           [1., 1., 1.]])
```

```
n = np.full((3,3),1)
n
```

```
    array([[1, 1, 1],
           [1, 1, 1],
           [1, 1, 1]])
```

Diagonal Matrix

Check out: [numpy.diag]

```
diag = np.diag([3,1,1]) # Elements in diagonal
diag
```

```
    array([[3, 0, 0],
           [0, 1, 0],
           [0, 0, 1]])
```

Identity Matrix

An identity matrix is a special diagonal matrix in which the values at the diagonal are ones. In most texts, the identity matrix is denoted as I.
Check out:

- [numpy.eye]
- [numpy.identity]

```
I = np.eye(3) # Always a square matrix
I

describe_mat(I)
```

```
Matrix:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

Shape:  (3, 3)
Rank:   2
```

Scalar Matrix

Since scalars cannot be explicitly operated with matrices, one workaround is to convert scalars into matrices. This is done by a matrix with all diagonal values equal to the original scalar.

```
scal = 3*np.identity(4) # The scalar'3' was multiplied into ones resulting to matrix
scal
```

```
array([[3., 0., 0., 0.],
       [0., 3., 0., 0.],
       [0., 0., 3., 0.],
       [0., 0., 0., 3.]])
```

Upper Triangular Matrix

An upper triangular matrix is a matrix that has no values below the diagonal.

```
upper = np.array([
    [2,3,1],
    [0,3,5],  # Example
    [0,0,1]
])
upper
```

```
array([[2, 3, 1],
       [0, 3, 5],
       [0, 0, 1]])
```

Lower Triangular Matrix

A lower triangular matrix is a matrix that has no values above the diagonal.

```
lower = np.array([
    [5,0,0],
    [4,6,0],  # Example
    [1,4,2]
])

lower
```

```
array([[5, 0, 0],
       [4, 6, 0],
       [1, 4, 2]])
```

Matrix / Tensor Algebra

Moving forward with matrices, vectors, and tensors. We'll try to see them in action using the commonly used operations for tensors. We will now dwell on the concepts and applications of Tensor Algebra

Arithmetic / Element-wise Operations

Check out:

- `numpy.add`
- `numpy.sum`
- `numpy.subtract`
- `numpy.multiply`
- `numpy.square`
- `numpy.divide`

Always remember that the array should have equal shapes when doing operations

```
g = np.array([
    [4,1],
    [3,4]
])
k = np.array([
    [9,5],
    [7,8]
])
```

```
## Addition

g+k # can be np.add(g,k)

#np.add(g,k)
g+3 # Broadcasting - only possible in computational in Python not on actual
```

```
array([[7, 4],
       [6, 7]])
```

```
## Subtraction
```

```
g-k

#np.subtract(g,k)
```

```
     array([[-5, -4],
            [-4, -4]])
```

```
## Multiplication

g*k

#np.multiply
```

```
     array([[36,  5],
            [21, 32]])
```

```
## Division

## To avoid division with zero add a very small number
alpha = 10**-6
g/(k+alpha)
```

```
     array([[0.4444444 , 0.19999996],
            [0.42857137, 0.49999994]])
```

Transpose of a Matrix

One of the fundamental operations in matrix algebra is Transposition. The transpose of a matrix is done by flipping the values of its elements over its diagonals. With this, the rows and columns from the original matrix will be switched. So for a matrix A its transpose is denoted as A^T. So for example:

$$A = \begin{bmatrix} 1 & 2 & 5 \\ 5 & -1 & 0 \\ 0 & -3 & 3 \end{bmatrix}$$
$$A^T = \begin{bmatrix} 1 & 5 & 0 \\ 2 & -1 & -3 \\ 5 & 0 & 3 \end{bmatrix}$$

This can now be achieved programmatically by using `np.transpose()` or using the `T` method. Check out:

- [np.transpose]

```
L = np.array([
    [3,4,6],
    [2,4,4],
    [6,5,3]])

print(L)
print("------------")
print(np.transpose(L))
```

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

Vector Product

The inner product of a vector is the sum of the products of each element of the vectors. So given vectors H and G below:

$$H = \begin{bmatrix} 1 \\ 3 \\ 6 \end{bmatrix}, G = \begin{bmatrix} 5 \\ 2 \\ 1 \end{bmatrix}$$

We first take the element-wise product of the vectors:

$$H * G = \begin{bmatrix} 5 \\ 6 \\ 6 \end{bmatrix}$$

Then we take the sum of the products, making it the inner product of a vector:

$$H \cdot G = 17$$

You can solve for the inner product using an explicit function, `np.inner()` or the `@` operator. Check out:

- [`np.inner`]

```
def inner (vec,vec2):
    dot = 0
    for element in range(len(vec)):
        dot = vec[element]*vec2[element] + dot
    return (dot)


vect_1 = [2, 5, 2, 5, 14]
vect_2 = [5, 2, 4, 8, 10]

print("---------------------------------------------------------")
print("Using the function created")
print("---------------------------------------------------------")
print("The inner product is: ", inner(vect_1,vect_2))

print("---------------------------------------------------------")
print("For comparison with np.inner()")
print("---------------------------------------------------------")

vect_1 = np.array([2, 5, 2, 5, 14])
vect_2 = np.array([5, 2, 4, 8, 10])

print("The inner product is : ", np.inner(vect_1,vect_2))

print("---------------------------------------------------------")
print("For comparison with @ ")
print("---------------------------------------------------------")
print(vect_1 @ vect_2)

print("---------------------------------------------------------")
print("For comparison with np.sum(?*?) ")
print("---------------------------------------------------------")
np.sum(vect_1*vect_2)
```

```
---------------------------------------------------------
Using the function created
---------------------------------------------------------
The inner product is:  208
---------------------------------------------------------
For comparison with np.inner()
---------------------------------------------------------
The inner product is :  208
---------------------------------------------------------
For comparison with @
---------------------------------------------------------
208
---------------------------------------------------------
For comparison with np.sum(?*?)
---------------------------------------------------------
208
```

In matrix dot products, we are going to get the sum of products of the vectors by row-column pairs. So if we have two matrices X and Y:

$$X = \begin{bmatrix} x_{(0,0)} & x_{(0,1)} \\ x_{(1,0)} & x_{(1,1)} \end{bmatrix}, Y = \begin{bmatrix} y_{(0,0)} & y_{(0,1)} \\ y_{(1,0)} & y_{(1,1)} \end{bmatrix}$$

The dot product will then be computed as:

$$X \cdot Y = \begin{bmatrix} x_{(0,0)} * y_{(0,0)} + x_{(0,1)} * y_{(1,0)} & x_{(0,0)} * y_{(0,1)} + x_{(0,1)} * y_{(1,1)} \\ x_{(1,0)} * y_{(0,0)} + x_{(1,1)} * y_{(1,0)} & x_{(1,0)} * y_{(0,1)} + x_{(1,1)} * y_{(1,1)} \end{bmatrix}$$

So if we assign values to X and Y:

$$X = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}, Y = \begin{bmatrix} -1 & 0 \\ 2 & 2 \end{bmatrix}$$

Check out:

- `np.dot`

```
vect_1 = np.array([2, 5, 2, 5, 14])
vect_2 = np.array([5, 2, 4, 8, 10])

np.dot(vect_1,vect_2)
```

```
    208
```

In matrix dot products there are additional rules compared with vector dot products. Since vector dot products were just in one dimension, there are fewer restrictions. Since now we are dealing with Rank 2 vectors we need to consider some rules:

Rule 1: The inner dimensions of the two matrices in question must be the same.

So given a matrix A with a shape of (a, b) where a and b are any integers. If we want to do a dot product between A and another matrix B, then matrix B should have a shape of (b, c) where b and c are any integers. So for given the following matrices:

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

So in this case A has a shape of $(3, 2)$, B has a shape of $(3, 2)$ and C has a shape of $(2, 3)$. So the only matrix pairs that is eligible to perform dot product is matrices $A \cdot C$, or $B \cdot C$.

$A = (n, k), B = (k, m)$ The column space size on A or (k) in A must equal to row space size on B or (k) in B; The outer numbers would be the dimention of the new matrix after the dot product or the n x m

```
A = np.array([
    [3,2,1,4],
    [4,5,1,5],
    [1,1,0,5],
    [1,4,2,3]
])

B = np.array([
    [4,1,6],
    [4,1,9],
    [1,4,8],
    [2,5,1]
])

shapeA = np.shape(A)
shapeB = np.shape(B)

print(shapeA)
print(shapeB)

CheckIfEqual = shapeA[1] == shapeB[0]
#shapeA[1] is equal to the column space size of A
#shapeB[0] is equal to the row space size of B
print(CheckIfEqual)

New_Matrix = A @ B
print(New_Matrix)
#The dimention on the New_Matrix is n x m or 4 x 3
#Therefore n x m(from property) == 4 x 3 (from New_Matrix)
```

```
(4, 4)
(4, 3)
True
[[29 29 48]
 [47 38 82]
 [18 27 20]
 [28 28 61]]
```

Rule 2: Dot Product has special properties

Dot products are prevalent in matrix algebra, this implies that it has several unique properties and it should be considered when formulation solutions:

1. $A \cdot B \neq B \cdot A$
2. $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
3. $A \cdot (B + C) = A \cdot B + A \cdot C$
4. $(B + C) \cdot A = B \cdot A + C \cdot A$
5. $A \cdot I = A$
6. $A \cdot \emptyset = \emptyset$

1. $A \cdot B \neq B \cdot A$

```
A = np.array([
    [3,2,1,4],
    [4,5,1,5],
    [1,1,0,5],
    [1,4,2,3]
])
```

```
B = np.array([
    [4,1,6,1],
    [3,2,9,4],
    [1,4,8,5],
    [2,5,1,2]
])


A @ B == B @ A
```

```
array([[False, False, False, False],
       [False, False, False, False],
       [False, False, False, False],
       [False, False, False, False]])
```

2. $A \cdot (B \cdot C) = (A \cdot B) \cdot C$

```
A = np.array([
    [3,2,1,4],
    [4,5,1,5],
    [1,1,0,5],
    [1,4,2,3]
])

B = np.array([
    [4,1,6,1],
    [4,1,9,4],
    [1,4,8,5],
    [2,5,1,2]
])

C = np.array([
    [1,1,0,2],
    [0,1,1,1],
    [1,0,1,4],
    [1,2,4,5]
])

A @ (B @ C) == (A @ B) @ C
```

```
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

3. $A \cdot (B + C) = A \cdot B + A \cdot C$

```
A = np.array([
    [3,2,1,4],
    [4,5,1,5],
    [1,1,0,5],
    [1,4,2,3]
])

B = np.array([
    [4,1,6,1],
    [4,1,9,4],
    [1,4,8,5],
    [2,5,1,2]
])

C = np.array([
    [1,1,0,2],
    [0,1,1,1],
    [1,0,1,4],
    [1,2,4,5]
])

A @ (B + C) == A @ B + A @ C
```

```
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

4. $(B + C) \cdot A = B \cdot A + C \cdot A$

```
A = np.array([
    [3,2,1,4],
    [4,5,1,5],
    [1,1,0,5],
    [1,4,2,3]
])

B = np.array([
    [4,1,6,1],
    [4,1,9,4],
    [1,4,8,5],
    [2,5,1,2]
])

C = np.array([
    [1,1,0,2],
    [0,1,1,1],
    [1,0,1,4],
    [1,2,4,5]
])

(B+C) @ A == B @ A + C @ A
```

```
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

5. $A \cdot I = A$

```
A = np.array([
    [3,2,1,1],
    [4,5,1,4],
    [1,1,0,5],
    [2,3,4,1]
])

#np.eye(4) same as
I = np.array([
    [1,0,0,0],
    [0,1,0,0],
    [0,0,1,0],
    [0,0,0,1]

])

A @ I #Same with I @ A
A

A @ I == A
```

```
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

6. $A \cdot \emptyset = \emptyset$

```
A = np.array([
    [3,2,1,3],
    [4,5,1,3],
    [1,1,0,1],
    [3,5,1,4]
])
```

```
emptyset = np.full((4,4),0)


A @ emptyset == emptyset
```

```
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

Determinants

A determinant is a scalar value derived from a square matrix. The determinant is a fundamental and important value used in matrix algebra.

The determinant of some matrix A is denoted as $det(A)$ or $|A|$. So let's say A is represented as:

$$A = \begin{bmatrix} a_{(0,0)} & a_{(0,1)} \\ a_{(1,0)} & a_{(1,1)} \end{bmatrix}$$

We can compute for the determinant as:

$$|A| = a_{(0,0)} * a_{(1,1)} - a_{(1,0)} * a_{(0,1)}$$

So if we have A as:

$$A = \begin{bmatrix} 1 & 4 \\ 0 & 3 \end{bmatrix}, |A| = 3$$

But you might wonder how about square matrices beyond the shape $(2,2)$? We can approach this problem by using several methods such as co-factor expansion and the minors method. This can be taught in the lecture of the laboratory but we can achieve the strenuous computation of high-dimensional matrices programmatically using Python. We can achieve this by using `np.linalg.det`.

```
## Determines the squareness, linearity, and linear dependence of your matrix or equation

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

det_X = np.linalg.det(X)
det_X
```

```
-17.999999999999996
```

2.6 Matrix Inverse

The inverse of a matrix is another fundamental operation in matrix algebra. Determining the inverse of a matrix let us determine if its solvability and its characteristic as a system of linear equation. Another use of the inverse matrix is solving the problem of divisibility between matrices. Although element-wise division exist but dividing the entire concept of matrices does not exists. Inverse matrices provide a related operation that could have the same concept of "dividing" matrices.

Now to determine the inverse of a matrix we need to perform several steps. So let's say we have a matrix M:

$$M = \begin{bmatrix} 1 & 7 \\ -3 & 5 \end{bmatrix}$$

First, we need to get the determinant of M.

$$|M| = (1)(5) - (-3)(7) = 26$$

Next, we need to reform the matrix into the inverse form:

$$M^{-1} = \frac{1}{|M|} \begin{bmatrix} m_{(1,1)} & -m_{(0,1)} \\ -m_{(1,0)} & m_{(0,0)} \end{bmatrix}$$

So that will be:

$$M^{-1} = \frac{1}{26} \begin{bmatrix} 5 & -7 \\ 3 & 1 \end{bmatrix} = \begin{bmatrix} \frac{5}{26} & \frac{-7}{26} \\ \frac{3}{26} & \frac{1}{26} \end{bmatrix}$$

For higher-dimension matrices, you might need to use co-factors, minors, adjugates, and other reduction techniques. To solve this programmatically we can use `np.linalg.inv`.

To validate the wether if the matric that you have solved is really the inverse, we follow this dot product property for a matrix M:

$$M \cdot M^{-1} = I$$

```
M = np.array([
    [3,4],
    [5,1]
])

M_inv = np.linalg.inv(M)
print(M_inv)

# To check
M @ M_inv # It means its correct since it yields to relatively small number
```

```
[[-0.05882353  0.23529412]
 [ 0.29411765 -0.17647059]]
array([[ 1.00000000e+00, -2.22044605e-16],
       [ 0.00000000e+00,  1.00000000e+00]])
```

System of Linear Equations

Solving linear equations is one of the fundamental skills of higher engineering mathematics. Aside from solving them, we must be skilled enough to spot them in the wild as well.

Given an equation:

$$B = \begin{cases} x + y + z = 1 \\ 3x - 2y - z = 4 \\ -x + 4y + 2z = -3 \end{cases}$$

We can represent it in matrix form considering the linear combination of the equations. We can also think of its dot product form:

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

We can make a general form for this equation by putting our matrices and vectors as variables. So let's say that the matrix $\begin{bmatrix} 1 & 1 & 1 \\ 3 & -2 & -1 \\ -1 & 4 & 2 \end{bmatrix}$ is X and $\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$ is the vector r then the answer $\begin{bmatrix} 1 \\ 4 \\ -3 \end{bmatrix}$ as Y. So we'll have:

$$Xr = Y$$

Our goal is to solve for r so we can solve it algebraically by multiplying both sides with the inverse of X, so we'll get:

$$X^{-1}Xr = X^{-1}Y$$
$$Ir = X^{-1}Y$$
$$r = X^{-1}Y$$

We'll take $r = X^{-1}Y$ as the $\mathbf{vectorized}$ equation as our formula in solving for the vector r or

So given an equation:

$$A = \begin{cases} 6a + 4b + 3c = 30000 \\ 2a + 2b + 3c = 18000 \\ 4a + 2b + 1c = 15000 \end{cases}$$

```
quantities = np.array([
    [6,4,3],
    [2,2,3],
    [4,2,1]
])
cost = np.array([
    [30000],
    [18000],
    [15000]
])

prices = np.linalg.inv(quantities) @ cost

print('The price of first item is: PHP {:.2f}'.format(float(prices[0])))
print('The price of 2nd item: PHP {:.2f}'.format(float(prices[1])))
print('The price of third item: PHP {:.2f}'.format(float(prices[2])))
```

```
The price of first item is: PHP 1500.00
The price of 2nd item: PHP 3000.00
The price of third item: PHP 3000.00
```

Visualizing Vectors

So far I know you have been experiencing mathematical exhaustion due to all these mathematical expressions. Allow me to show you a tad more interesting side of Linear Algebra.

Undoubtedly, one of the most interesting and frustrating parts of Data Analysts and Data Scientist is visualizing data. Although we will be visualizing more on matrices and tensors. So, bear with me here and I'll try to spark a bit of interest in you guys.

```
### If you haven't installed it use:
#!pip install matplotlib

import matplotlib.pyplot as plt
```

2D Cartersian Plots

Check out:

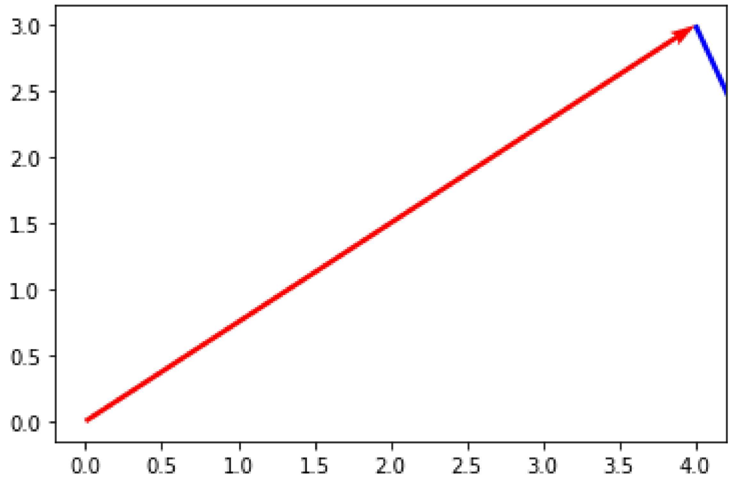
- `matplotlib.pyplot.xlim`
- `matplotlib.pyplot.ylim`
- `matplotlib.pyplot.quiver`
- `matplotlib.pyplot.grid`
- `matplotlib.pyplot.show`

```
A = np.array([4, 3])
B = np.array([2, -5])

# plt.xlim(-15, 15)
# plt.ylim(-15, 15)
plt.quiver(0,0, A[0], A[1], angles='xy', scale_units='xy',scale=1, color='red') # Red --> A
plt.quiver(A[0], A[1], B[0], B[1], angles='xy', scale_units='xy',scale=1, color='b') # Blue -
R = A + B
print(R)
```

```
# plt.grid()
# plt.show()
```

```
[ 6 -2]
```



Practice 1: Modulus of a Vector

The modulus of a vector or the magnitude of a vector can be determined using the Pythagorean theorem. Given the vector A and its scalars denoted as a_n where n is the index of the scalar. So if we have:

$$A = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

We can compute the magnitude as:

$$||A|| = \sqrt{a_1^2 + a_2^2} = \sqrt{1^2 + 2^2} = \sqrt{5}$$

So if we have a matrix with more parameters such as:

$$B = \begin{bmatrix} 2 \\ 5 \\ -1 \\ 0 \end{bmatrix}$$

We can generalize the Pythagorean theorem to compute for the magnitude as:

$$||B|| = \sqrt{b_1^2 + b_2^2 + b_3^2 + \ldots + b_n^2} = \sqrt{\sum_{n=1}^{N} b_n^2}$$

And this equation is now called a Euclidian distance or the Euclidean Norm.

Code Time:

```
import math
```

```
def modulus(vec):
    raisesum = 0
    for element in range(len(vec)):  # Accesing the values
        raisesum += vec[element] **2 # raising each number to 2
        sqr = math.sqrt(raisesum)    # Squaring
    return(sqr)

vect1 = [2, 5, 2, 5, 14] # These are the values of vectors
print("Vector's magnitude for vect1 = ", modulus(vect1))
```

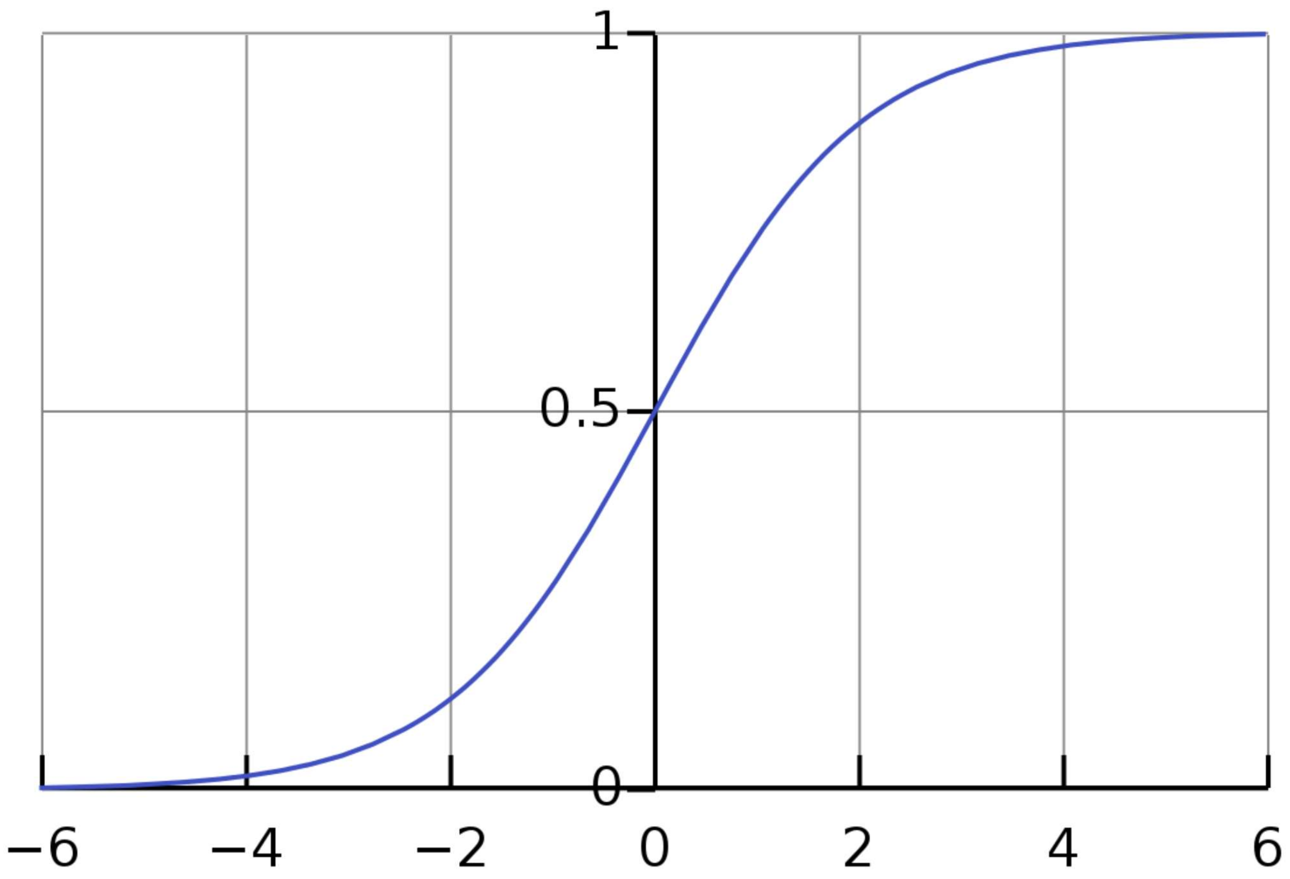
```
Vector's magnitude for vect1 =  15.937377450509228
```

About the code

The function above calculated the magnitude of the vector with 5 elements. The variable ‘raisesum’ was set to 0 so that the assignment of this variable is possible at the same time that it was used as referenced. A loop statement was created to access every element inside the vector created and inside this loop statement are the operations that will compute the modulus. The operations include the summation of the squared elements from the vectors. The sum of these elements was assigned to the ‘raisesum’ variable and the square root of this was assigned to ‘sqr’ which was returned afterward and printed.

Practice 2: The Sigmoid

The sigmoid function is one of the popular Activation Functions which we will discuss later on. The sigmoid is a bounded, differentiable, real function in which its range would be any value from 0 to 1. It is widely used in binary classifications.



If we were to check the equation characterizing this curve in textbooks or journals it would be:

$$sig(x) = \frac{1}{1+e^{-x}}$$

or

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

So let's try to translate this in NumPy. You might want to use `numpy.exp` for this function.
If you want to read more about the sigmoid function click here.

Code Time:

Standard Function

```
def Sigmoid(x):
    return 1/(1 + np.exp(-x)) # Transformed equation
Sigmoid(3)                    # Using the function
```

```
0.9525741268224334
```

Lambda Function

```
x =  3
s = lambda x: 1/(1 + np.exp(-x))
print(s(x))
```

```
     0.9525741268224334
```

About the code

The equation of sigmoid was transformed into standard and lambda function. The standard function was used by calling the function with value '3' as its argument that passed into the function as its parameter and returns the result of the equation. Meanwhile, the lambda function is the more optimized version of the standard function since the declaration of a function is not needed but still does the job the same as with the standard function.