

Linear Algebra for ChE

Laboratory 3 : Matrix Operations

Objectives

In the previous laboratory activity, the students have learned and applied the basic principles, and techniques of using Python in creating and representing matrices. However, this laboratory activity focuses on the application of other matrix operations in Python such as the transposition, dot product, determinant, and inverse matrix operations. Through this activity, the students will be able to learn the application of various matrix operations in programming together with its basic operations and techniques. Upon learning its applications, the students will be able to apply their knowledge in translating matrix operations and properties using Python.

Discussion

```
In [ ]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

- Numerical Python or "numpy" in short, supports a broad range of mathematical operations on arrays. It extends Python with sophisticated data formats that ensure fast array and matrix computations, as well as a vast library of elevated numerical methods that work on these matrices and arrays [1].
- Matplotlib.pyplot is a set of algorithms that enable matplotlib to operate similarly to MATLAB. Each pyplot function modifies a figure in a certain way [2].

Transposition

This is known to transpose a matrix wherein the values in the row and column have interchange or switch positions through the codes "np.transpose" and ".T". [4]

$$P = \begin{bmatrix} -3 & 9 & 6 \\ 0 & -7 & -4 \\ -2 & 4 & 8 \end{bmatrix}$$

$$P^T = \begin{bmatrix} -3 & 0 & -2 \\ 9 & -7 & 4 \\ 6 & -4 & 8 \end{bmatrix}$$

```
In [ ]: P = np.array([
    [-3, 9, 6],
    [0, -7, -4],
    [-2, 4, 8]
])
P
```

```
Out[ ]: array([[ -3,  9,  6],
               [  0, -7, -4],
               [ -2,  4,  8]])
```

```
In [ ]: PT1 = np.transpose(P)
PT1
```

```
Out[ ]: array([[ -3,  0, -2],
               [  9, -7,  4],
               [  6, -4,  8]])
```

```
In [ ]: PT2 = P.T
PT2
```

```
Out[ ]: array([[ -3,  0, -2],
               [  9, -7,  4],
               [  6, -4,  8]])
```

```
In [ ]: np.array_equiv(PT1, PT2)
```

```
Out[ ]: True
```

```
In [ ]: R = np.array([
    [9, 11, -7, 6, 8],
    [-21, 4, 0, -3, 9],
    [16, 0, -4, 5, 2],
    [1, 0, 22, 7, 8]
])
R.shape
```

```
Out[ ]: (4, 5)
```

```
In [ ]: np.transpose(R).shape
```

```
Out[ ]: (5, 4)
```

```
In [ ]: R.T.shape
```

```
Out[ ]: (5, 4)
```

```
In [ ]: K = np.array([
    [4, -8, 7],
    [-10, 9, 16],
    [12, 0, -1],
    [5, -13, 8],
    [7, 13, -5],
])
K.shape
```

```
Out[ ]: (5, 3)
```

```
In [ ]: np.transpose(K).shape
```

```
Out[ ]: (3, 5)
```

```
In [ ]: K.T.shape
```

```
Out[ ]: (3, 5)
```

```
In [ ]: KT = K.T
KT
```

```
Out[ ]: array([[ 4, -10, 12,  5,  7],
               [-8,  9,  0, -13, 13],
               [ 7, 16, -1,  8, -5]])
```

Dot Product / Inner Product

In this operation, the product of the two matrices is displayed using three different codes: `np.dot()`, `np.matmul()` and `@` operator. The matrix dot product is more complicated than vector dot products due to the fact that there are rules to follow in the matrix dot product. [3]

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 :

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

$$X \cdot Y = \begin{bmatrix} 1 * -1 + 2 * 2 & 1 * 0 + 2 * 2 \\ 0 * -1 + 1 * 2 & 0 * 0 + 1 * 2 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 2 & 2 \end{bmatrix}$$

```
In [ ]: U = np.array([
        [3,-22,9],
        [14,8,-7],
        [-2,16,25],
    ])
V = np.array([
        [5,10,1],
        [8,8,0],
        [12,33,6]
    ])
```

```
In [ ]: np.array_equiv(U,V)
```

```
Out[ ]: False
```

```
In [ ]: np.dot(U,V)
```


```
Out[ ]: array([[ -53, 151,  57],
               [  50, -27, -28],
               [418, 933, 148]])
```

```
In [ ]: U.dot(V)
```

```
Out[ ]: array([[ -53, 151,  57],
               [  50, -27, -28],
               [418, 933, 148]])
```

```
In [ ]: U @ V
```

```
Out[ ]: array([[ -53, 151,  57],
               [  50, -27, -28],
               [418, 933, 148]])
```

```
In [ ]: np.matmul(U,V) 
```

```
Out[ ]: array([[ -53, 151,  57],
               [  50, -27, -28],
               [418, 933, 148]])
```

```
In [ ]: B = np.array([
        [3,-11,24],
        [50,13, 9]
    ])
C = np.array([
        [-15,0],
        [2,32],
        [9,7]
    ])
```

```
In [ ]: np.dot(B,C)
```

```
Out[ ]: array([[ 149, -184],
               [-643,  479]])
```

```
In [ ]: B.dot(C)
```

```
Out[ ]: array([[ 149, -184],
               [-643,  479]])
```

```
In [ ]: B @ C
```

```
Out[ ]: array([[ 149, -184],
               [-643,  479]])
```

```
In [ ]: np.matmul(B,C)
```

```
Out[ ]: array([[ 149, -184],
               [-643,  479]])
```

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

This pertains to the limitation of multiplying matrices, wherein the number of columns in the first matrix equals the number of rows in the second matrix. To further understand, the product of matrix H and matrix I can be shown since the shape of matrix H is $(4, 5)$ and matrix I has a shape of $(5, 3)$.

$$G = \begin{bmatrix} 4 & 14 & 7 \\ 11 & -5 & 8 \\ 9 & 7 & 13 \end{bmatrix}, H = \begin{bmatrix} 0 & 2 & 0 & 9 & -8 \\ 7 & 8 & -8 & 1 & 0 \\ 19 & -32 & 11 & -10 & -1 \\ 4 & -13 & 0 & 2 & 13 \end{bmatrix}, I = \begin{bmatrix} 3 & 7 & 3 \\ 4 & 4 & 4 \\ 19 & 11 & 14 \\ -13 & 1 & 2 \\ 8 & 0 & 1 \end{bmatrix}, J =$$

So in this case G has a shape of $(3, 3)$, H has a shape of $(4, 5)$, I has a shape of $(5, 3)$ and J has a shape of $(4, 3)$. So the only matrix pairs that is eligible to perform dot product is matrices $I \cdot G$, or $H \cdot I$.

```
In [ ]: G = np.array([
    [4, 14, 7],
    [11, -5, 8],
    [9, 7, 13]
])
H = np.array([
    [0, 2, 0, 9, -8],
    [7, 8, -8, 1, 0],
    [19, -32, 11, -10, -1],
    [4, -13, 0, 2, 13],
])
I = np.array([
    [3, 7, 3],
    [4, 4, 4],
    [19, 11, 14],
    [-13, 1, 2],
    [8, 0, 1],
])
```

```
J = np.array([
    [1,6,9],
    [-17,10,3],
    [5,-14,8],
    [3,2,0]
])
print(G.shape)
print(H.shape)
print(I.shape)
print(J.shape)
```

(3, 3)

(4, 5)

(5, 3)

(4, 3)

In []: H @ I

```
Out[ ]: array([[ -173,   17,   18],
               [ -112,   -6,  -57],
               [ 260,  116,   62],
               [  38,  -22,  -23]])
```

In []: I @ G

```
Out[ ]: array([[ 116,   28,  116],
               [  96,   64,  112],
               [ 323,  309,  403],
               [ -23, -173,  -57],
               [  41,  119,   69]])
```

In []: J @ G

```
Out[ ]: array([[ 151,   47,  172],
               [  69, -267,    0],
               [ -62,  196,   27],
               [  34,   32,   37]])
```

In []: I @ G.T

```
Out[ ]: array([[ 131,   22,  115],
               [ 100,   56,  116],
               [ 328,  266,  430],
               [ -24, -132,  -84],
               [  39,   96,   85]])
```

In []: J @ I.T

```
Out[ ]: array([[ 72,   64,  211,   11,   17],
               [  28,  -16, -171,  237, -133],
               [ -59,   -4,   53,  -63,   48],
               [  23,   20,   79,  -37,   24]])
```

```
In [ ]: D = np.array([
    [3,-4,6,1,9]
])
A = np.array([
```

```

    [-2,7,0,5,-8]
])
print(D.shape)
print(A.shape)

```

```

(1, 5)
(1, 5)

```

```
In [ ]: A @ D.T
```

```
Out[ ]: array([[ -10]])
```

```
In [ ]: D.T @ A
```

```
Out[ ]: array([[ -6,  21,   0,  15, -24],
               [  8, -28,   0, -20,  32],
               [-12,  42,   0,  30, -48],
               [ -2,   7,   0,   5,  -8],
               [-18,  63,   0,  45, -72]])
```

```
In [ ]: A.T @ D
```

```
Out[ ]: array([[ -6,   8, -12,  -2, -18],
               [ 21, -28,  42,   7,  63],
               [  0,   0,   0,   0,   0],
               [ 15, -20,  30,   5,  45],
               [-24,  32, -48,  -8, -72]])
```

Rule 2: Dot Product has special properties

It was known that during the formulation of solutions in Dot Product, there are various distinct properties supposed to be recognized and taken into account.

1. $G \cdot H \neq H \cdot G$
2. $G \cdot (H \cdot I) = (G \cdot H) \cdot I$
3. $G \cdot (H + I) = G \cdot H + G \cdot I$
4. $(H + I) \cdot G = H \cdot G + I \cdot G$
5. $G \cdot I = G$
6. $G \cdot \emptyset = \emptyset$

```
In [76]: G = np.array([
           [4,1,14,7,8],
           [11,23,-5,8,-9],
           [18,-9,7,13,0],
           [5,-6,0,1,2],
           [4,-13,1,2,4]
         ])
H = np.array([
           [0,2,0,9,3],
           [7,8,-8,1,0],
           [19,-32,11,-10,1],
           [18,0,-1,3,-11],
           [7,0,1,-3,4]
         ])

```

```

])
I = np.array([
    [-2,3,7,3,4],
    [1,4,4,4,0],
    [0,19,11,14,-3],
    [-13,1,2,4,0],
    [1,0,-5,7,-6],
])

```

```
In [ ]: np.array_equal(G@H, H@G)
```

```
Out[ ]: False
```

```
In [ ]: np.array_equal(G@(H@I), (G@H)@I)
```

```
Out[ ]: True
```

```
In [ ]: X = H + I
X
```

```
Out[ ]: array([[ -2,   5,   7,  12,   7],
               [  8,  12,  -4,   5,   0],
               [ 19, -13,  22,   4,  -2],
               [  5,   1,   1,   7, -11],
               [  8,   0,  -4,   4,  -2]])
```

```
In [ ]: Y = (G@H) + (G@I)
Y
```

```
Out[ ]: array([[ 365, -143,  307,  190, -93],
               [  35,  404,  -81,  247,  17],
               [  90,  -96,  329,  290, -31],
               [-37,  -46,   52,   45,  20],
               [-51, -147,   88,   17,  -4]])
```

```
In [ ]: np.array_equal(G@X,Y)
```

```
Out[ ]: True
```

```
In [ ]: O = H + I
O
```

```
Out[ ]: array([[ -2,   5,   7,  12,   7],
               [  8,  12,  -4,   5,   0],
               [ 19, -13,  22,   4,  -2],
               [  5,   1,   1,   7, -11],
               [  8,   0,  -4,   4,  -2]])
```

```
In [ ]: P = (H@G) + (I@G)
P
```



```
Out[ ]: array([[ 261, -113,   3,  143,  -9],
               [ 117,  290,  24,  105, -34],
               [ 341, -476, 483,  315, 269],
               [  40,  120,  61,   41,   1],
               [-28,   46,  82,   4,  64]])
```

```
In [ ]: np.array_equal(O@G, P)
```

```
Out[ ]: True
```

```
In [ ]: Q = np.eye(5)
Q
```

```
Out[ ]: array([[1., 0., 0., 0., 0.],
               [0., 1., 0., 0., 0.],
               [0., 0., 1., 0., 0.],
               [0., 0., 0., 1., 0.],
               [0., 0., 0., 0., 1.]])
```

```
In [ ]: np.array_equal(G@Q, G)
```

```
Out[ ]: True
```

```
In [ ]: E = G.dot(np.zeros(G.shape))
E
```

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

```
In [ ]: F = np.zeros(G.shape)
F
```

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

```
In [ ]: np.array_equal(E,F)
```

```
Out[ ]: True
```

```
In [ ]: z_mat = np.zeros(G.shape)
z_mat
```

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

```
In [ ]: a_dot_z = G.dot(np.zeros(G.shape))
a_dot_z
```

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

```
In [ ]: np.array_equal(a_dot_z, z_mat)
```

```
Out[ ]: True
```

```
In [ ]: null_mat = np.empty(A.shape, dtype=float)
null = np.array(null_mat, dtype=float)
print(null)
np.allclose(a_dot_z, null)
```

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

```
Out[ ]: False
```

Determinant

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:

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

In getting the determinant of a matrix, the code used is "np.linalg.det()", especially when the matrix has a shape of more than (2, 2). It is only applicable in square matrices. [5]

```
In [ ]: J = np.array([
        [3,6],
        [9,0]
    ])
np.linalg.det(J)
```

```
Out[ ]: -54.00000000000001
```

```
In [ ]: K = np.array([
        [13,6,9,12],
```

```

    [9,0,5,4],
    [1,2,23,4],
    [0,6,11,18]
])
np.linalg.det(K)

```

Out[]: -13160.000000000005

```

In [ ]: C = np.array([
    [11,-20,8,10,0],
    [19,1,7,51,-8],
    [9,7,21,18,9],
    [1,4,5,0,-1],
    [26,14,-9,0,1]
])
np.linalg.det(C)

```

Out[]: -3143529.9999999963

Inverse

This matrix operation involves steps when doing or solving it manually, but it can be easily done with python programming. The code for the inverse of a matrix is "np.linalg.inv()" and it can also use for matrices with a higher dimension. The expected outcome of the inverse of a matrix is an identity matrix when multiplied by the original matrix. [6]

```

In [ ]: L = np.array([
    [8,9,14],
    [4,9,11],
    [0,3,22]
])

np.array(L @ np.linalg.inv(L), dtype=int)

```

Out[]: array([[1, 0, 0],
 [0, 1, 0],
 [0, 0, 1]])

```

In [ ]: M = np.array([
    [91,5,33,8],
    [6,88,-3,41],
    [50,-39,72,22],
    [1,18,3,55]
])
M_inv = np.linalg.inv(M)
np.array(M @ M_inv,dtype=int)

```

Out[]: array([[1, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 1, 0],
 [0, 0, 0, 1]])

```
In [ ]: M = np.array([
    [91,5,33,8],
    [6,88,-3,41],
    [50,-39,72,22],
    [1,18,3,55]
])
N = np.linalg.inv(M)
N
```

```
Out[ ]: array([[ 0.01536602, -0.00510243, -0.0074448 ,  0.00454649],
               [-0.00192666,  0.01469164,  0.00197274, -0.01146081],
               [-0.01202212,  0.01316151,  0.02062715, -0.01631349],
               [ 0.00100691, -0.0054333 , -0.00163538,  0.02273979]])
```

```
In [ ]: M @ N
```

```
Out[ ]: array([[ 1.00000000e+00, -3.46944695e-17, -4.68375339e-17,
                 -2.77555756e-17],
               [-3.94649591e-17,  1.00000000e+00,  3.79470760e-17,
                 -1.97758476e-16],
               [-8.67361738e-19,  5.72458747e-17,  1.00000000e+00,
                 -7.63278329e-17],
               [-1.60461922e-17,  1.82145965e-17,  2.45029691e-17,
                 1.00000000e+00]])
```

```
In [ ]: S = np.array([
    [7,13,-19,28,1,42,0,5],
    [2,78,1,32,0,0,-4,1],
    [17,2,-2,3,-44,18,25,9],
    [1,-15,22,-8,10,11,34,3],
    [-9,-14,16,7,13,20,35,21],
    [0,3,0,2,1,0,19,0],
    [-8,3,25,37,0,15,2,4],
    [29,8,7,6,-2,-16,27,9],
])
S_inv = np.linalg.inv(S)
np.array(S @ S_inv,dtype=int)
```

```
Out[ ]: array([[1, 0, 0, 0, 0, 0, 0, 0],
               [0, 1, 0, 0, 0, 0, 0, 0],
               [0, 0, 1, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 1, 0, 0, 0],
               [0, 0, 0, 0, 0, 1, 0, 0],
               [0, 0, 0, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 0, 0, 0, 1]])
```

```
In [ ]: squad = np.array([
    [1.25, 1.5, 0.3],
    [0.77, 0.9, 0.9],
    [0.75, 0.25, 2.0]
])
weights = np.array([
    [0.3, 0.3, 0.4]
])
```

```
p_grade = squad @ weights.T
p_grade
```

```
Out[ ]: array([[0.945],
               [0.861],
               [1.1  ]])
```

Activity

Task 1

Prove and implement the remaining 6 matrix multiplication properties. You may create your own matrices in which their shapes should not be lower than (3, 3). In your methodology, create individual flowcharts for each property and discuss the property you would then present your proofs or validity of your implementation in the results section by comparing your result to present functions from NumPy.

```
In [ ]: G = np.array([
    [4,1,14,7,8],
    [11,23,-5,8,-9],
    [18,-9,7,13,0],
    [5,-6,0,1,2],
    [4,-13,1,2,4]
])
H = np.array([
    [0,2,0,9,3],
    [7,8,-8,1,0],
    [19,-32,11,-10,1],
    [18,0,-1,3,-11],
    [7,0,1,-3,4]
])
I = np.array([
    [-2,3,7,3,4],
    [1,4,4,4,0],
    [0,19,11,14,-3],
    [-13,1,2,4,0],
    [1,0,-5,7,-6],
])
```

```
In [ ]: #Test for Commutative Property of Multiplication
A = G @ H
B = H @ G
print(f'G•H:\n{A} \n\nH•G:\n{B}\n')
print("-----")
print(f'Commutative Property:{np.array_equal(A,B)}')
```

G•H:

```
[[ 455 -432  147 -106 -19]
 [ 147  366 -256  223 -96]
 [ 304 -260  136  122 -82]
 [ -10  -38   49   36  12]
 [  -8 -128  117    7    7]]
```

H•G:

```
[[  79  -47   -7   31   12]
 [ -23  257    2   10  -14]
 [-124 -769  504   12  424]
 [  25  152  234   94  106]
 [  47  -36  109   67   66]]
```

Commutative Property:False

```
In [ ]: #Test for Associative Property of Multiplication
C = H @ I
D = G @ H
E = G @ C
F = D @ I
print(f'H•I:\n{C} \n\nG•H:\n{D}\n')
print(f'G•(H•I):\n{E} \n\n(G•H)•I:\n{F}\n')
print("-----")
print(f'Associative Property:{np.array_equal(E,F)}')
```

H•I:

```
[[ -112   17   11   65  -18]
 [  -19  -98   -5  -55   52]
 [   61  128  101   50   37]
 [  -86   38  176  -25  141]
 [   29   37   34   51   1]]
```

G•H:

```
[[ 455 -432  147 -106  -19]
 [ 147  366 -256  223  -96]
 [ 304 -260  136  122  -82]
 [ -10  -38   49   36   12]
 [  -8 -128  117    7    7]]
```

G•(H•I):

```
[[   17  2324  2957  1138  1493]
 [-2923 -2736   603 -1459  1932]
 [-2536  2578  3238  1690  1300]
 [ -474   785   329   732 -259]
 [ -196  1694   698  1179 -425]]
```

(G•H)•I:

```
[[   17  2324  2957  1138  1493]
 [-2923 -2736   603 -1459  1932]
 [-2536  2578  3238  1690  1300]
 [ -474   785   329   732 -259]
 [ -196  1694   698  1179 -425]]
```

Associative Property:True

```
In [ ]: #Test for Left-Side Distributive Property of Multiplication
J = H + I
K = G @ H
L = G @ I
M = G @ J
N = K + L
print(f'H+I:\n{J} \n\nG•H:\n{K}\n \n\nG•I:\n{L}\n')
print(f'G•(H+I):\n{M} \n\nG•H+G•I:\n{N}\n')
print("-----")
print(f'Left Distributive Property:{np.array_equal(M,N)}')
```

H+I:

```
[[ -2  5  7 12  7]
 [  8 12 -4  5  0]
 [ 19 -13 22  4 -2]
 [  5  1  1  7 -11]
 [  8  0 -4  4 -2]]
```

G•H:

```
[[ 455 -432 147 -106 -19]
 [ 147 366 -256 223 -96]
 [ 304 -260 136 122 -82]
 [ -10 -38  49  36 12]
 [ -8 -128 117  7  7]]
```

G•I:

```
[[ -90 289 160 296 -74]
 [-112  38 175  24 113]
 [-214 164 193 168  51]
 [ -27  -8  3  9  8]
 [ -43 -19 -29 10 -11]]
```

G•(H+I):

```
[[ 365 -143 307 190 -93]
 [  35 404 -81 247 17]
 [  90 -96 329 290 -31]
 [ -37 -46  52  45 20]
 [ -51 -147 88 17 -4]]
```

G•H+G•I:

```
[[ 365 -143 307 190 -93]
 [  35 404 -81 247 17]
 [  90 -96 329 290 -31]
 [ -37 -46  52  45 20]
 [ -51 -147 88 17 -4]]
```

Left Distributive Property:True

```
In [ ]: #Test for Right-Side Distributive Property of Multiplication
O = H + I
P = H @ G
Q = I @ G
R = O @ G
S = P + Q
print(f'H+I:\n{O} \n\nH•G:\n{P}\n \n\nI•G:\n{Q}\n')
print(f'(H+I)•G:\n{R} \n\nH•G+I•G:\n{S}\n')
print("-----")
print(f'Right Distributive Property:{np.array_equal(R,S)}')
```


H+I:

```
[[ -2  5  7 12  7]
 [  8 12 -4  5  0]
 [ 19 -13 22  4 -2]
 [  5  1  1  7 -11]
 [  8  0 -4  4 -2]]
```

H•G:

```
[[ 79 -47 -7 31 12]
 [ -23 257  2 10 -14]
 [-124 -769 504 12 424]
 [ 25 152 234 94 106]
 [ 47 -36 109 67 66]]
```

I•G:

```
[[ 182 -66 10 112 -21]
 [ 140  33 22  95 -20]
 [ 465 293 -21 303 -155]
 [ 15 -32 -173 -53 -105]
 [ -75  82 -27 -63 -2]]
```

(H+I)•G:

```
[[ 261 -113  3 143 -9]
 [ 117 290 24 105 -34]
 [ 341 -476 483 315 269]
 [  40 120 61  41  1]
 [ -28  46 82  4 64]]
```

H•G+I•G:

```
[[ 261 -113  3 143 -9]
 [ 117 290 24 105 -34]
 [ 341 -476 483 315 269]
 [  40 120 61  41  1]
 [ -28  46 82  4 64]]
```

Right Distributive Property:True

```
In [ ]: #Test for Multiplicative Identity Property
T = np.eye(5)
U = G @ T
V = G
print(f'G:\n{V} \n\nG•I:\n{U}\n')
print("-----")
print(f'Multiplicative Identity Property:{np.array_equal(U,V)}')
```

```
G:
[[ 4  1 14  7  8]
 [11 23 -5  8 -9]
 [18 -9  7 13  0]
 [ 5 -6  0  1  2]
 [ 4 -13  1  2  4]]
```

```
G•I:
[[ 4.  1. 14.  7.  8.]
 [11. 23. -5.  8. -9.]
 [18. -9.  7. 13.  0.]
 [ 5. -6.  0.  1.  2.]
 [ 4. -13.  1.  2.  4.]]
```

 Multiplicative Identity Property:True

```
In [ ]: #Test for Multiplicative Property of Zero
W = G
X = np.zeros(G.shape)
Y = W @ X
print(f'G:\n{W}\n\nG•0:\n{Y}\n')
print("-----")
print(f'Multiplicative Property of Zero:{np.array_equal(Y,X)}')
```

```
G:
[[ 4  1 14  7  8]
 [11 23 -5  8 -9]
 [18 -9  7 13  0]
 [ 5 -6  0  1  2]
 [ 4 -13  1  2  4]]
```

```
G•0:
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

 Multiplicative Property of Zero:True

Conclusion

Through using Google Colab again, the students were able to expand their knowledge and skills about the fundamental matrix operations. Applying those matrix operations in Python programming has further improved their understanding of the different operations and their importance in Linear Algebra. In addition, the matrix operations discussed were transposition, dot product/inner product that has two rules, determinant and inverse. Different codes were presented, and each of them has shown its unique functions in Python programming as well as translating matrix equations and operations. Moreover, other introduced codes displayed the same results to show that there are different ways of

presenting the matrix operations. The shape of the matrices is also determined, and there are six properties proven in this laboratory report.

On top of that, the application of these matrix operations in Python has helped the students assess their capabilities in solving manually, for it can serve as a checking tool for the acquired answers. It may be known that students can do these matrix operations manually, but with higher dimensions, Python programming is a huge help for solving it. Hence, in this laboratory report, Python programming had played a significant role wherein it simply displayed the results of the matrix operations easily and immediately.

With regards to the real-life application of matrices, it is undeniable that there are a myriad of things that we have right now that apply concepts and principles of matrices. Matrices have many significant applications that have a great impact on the modern world. The impact of matrices and their applications in the mathematical world is widespread because they serve as a foundation for many theories and practices. Various problems relating to science and technology could be solved through the application of matrices because matrices could be utilized to provide rapid approximations to more complex calculations. Matrices also play a vital role in scientific methods and research as they are utilized in plotting graphs, data analysis, and statistics. In relation to the field of health care, these scientific methods and researches are critical to the advancement and eradication of issues in the healthcare systems. It is undeniable that the healthcare industry handles a lot of systematic analysis and data wherein matrices could be a very useful tool to obtain a desirable solution.