**Q1.**

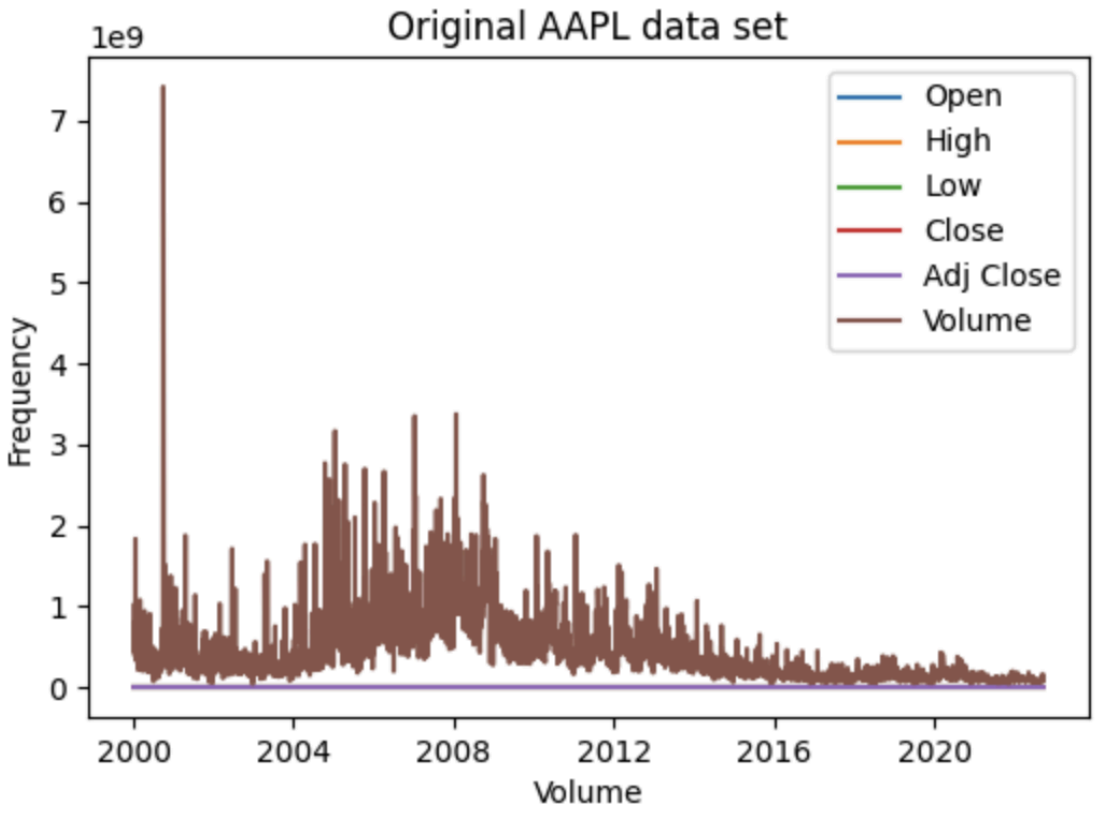
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

class datapreprocessing:  2 usages  new *
    def __init__(self,data):  new *
        self.data = data
        self.normalized_data = None
        self.standardized_data = None
        self.iqr_data=None

    def Normalized(self):  1 usage  new *
        self.normalized_data = (self.data - self.data.min())/(self.data.max()-self.data.min())

    def Standardized(self):  1 usage  new *
        self.standardized_data = (self.data - self.data.mean())/(self.data.std())

    def IQR(self):  1 usage  new *
        Q1 = self.data.quantile(0.25)
        Q3 = self.data.quantile(0.75)
        IQR = Q3-Q1
        self.iqr_data = (self.data - Q1)/IQR

    def Show_original(self):  1 usage  new *
        plt.figure(figsize=(6,4))
        for column in self.data.columns:
            plt.plot( *args: self.data[column],label=column)
        plt.title("Original AAPL data set")
        plt.xlabel(column)
        plt.ylabel('Frequency')
        plt.legend()
        plt.show()
```

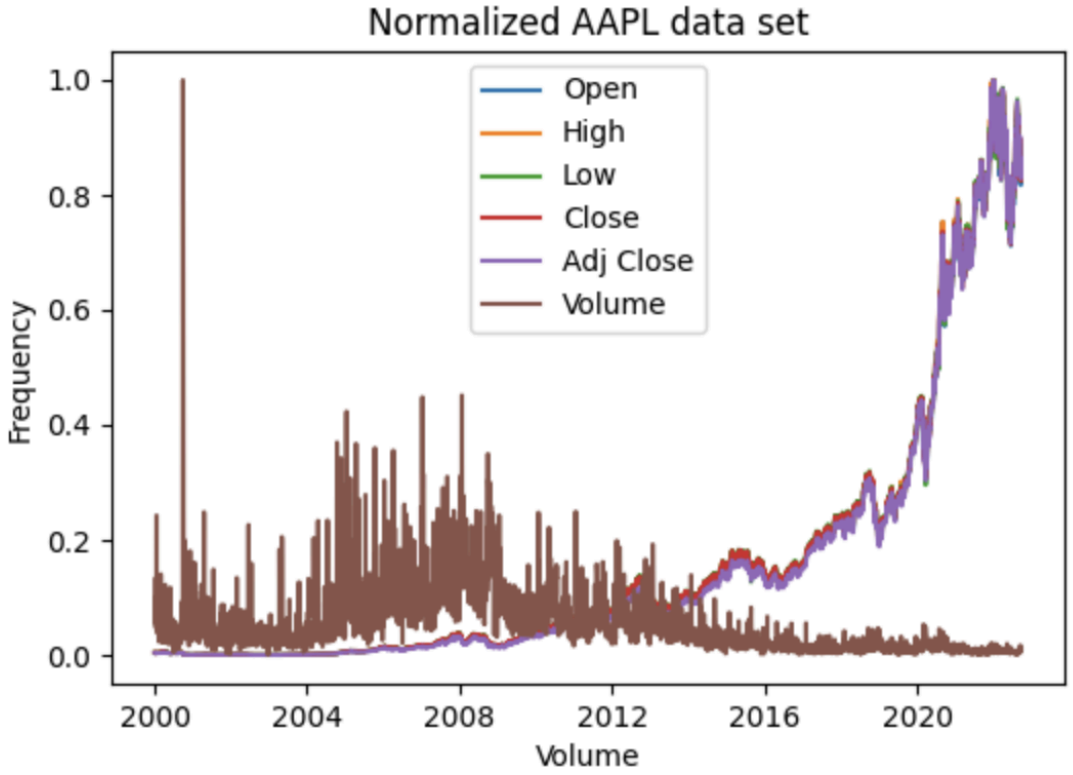
```
34        def Show_normalized(self):  1 usage  new *
35            if self.normalized_data is None:
36                self.Normalized()
37                plt.figure(figsize=(6, 4))
38                for column in self.normalized_data.columns:
39                    plt.plot( *args: self.normalized_data[column], label=column)
40                plt.title("Normalized AAPL data set")
41                plt.xlabel(column)
42                plt.ylabel('Frequency')
43                plt.legend()
44                plt.show()
45
46        def Show_standardized(self):  1 usage  new *
47            if self.standardized_data is None:
48                self.Standardized()
49                plt.figure(figsize=(6, 4))
50                for column in self.standardized_data.columns:
51                    plt.plot( *args: self.standardized_data[column], label=column)
52                plt.title("Standardized AAPL data set")
53                plt.xlabel(column)
54                plt.ylabel('Frequency')
55                plt.legend()
56                plt.show()
57
58        def Show_IQR(self):  1 usage  new *
59            if self.iqr_data is None:
60                self.IQR()
61                plt.figure(figsize=(6, 4))
62                for column in self.iqr_data.columns:
63                    plt.plot( *args: self.iqr_data[column], label=column)
64                plt.title("IQR transformation AAPL data set")
65                plt.xlabel(column)
66                plt.ylabel('Frequency')
67                plt.legend()
68                plt.show()
```

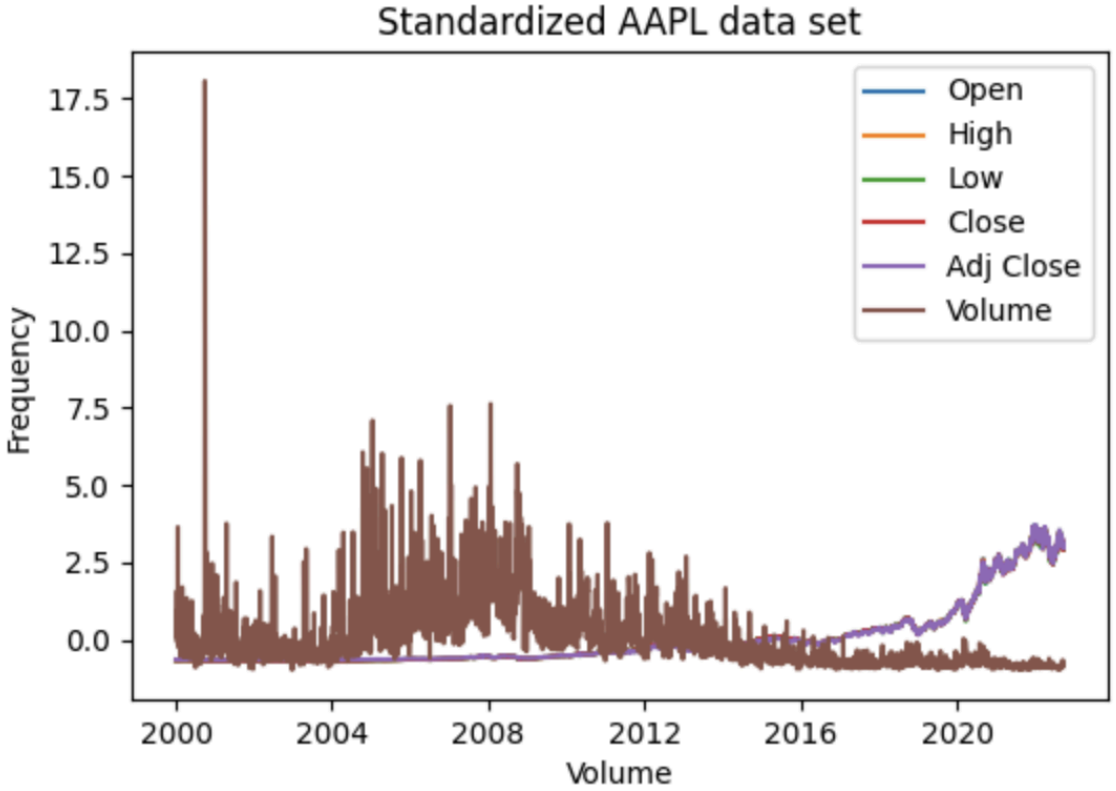
**Q2.**
**Code:**

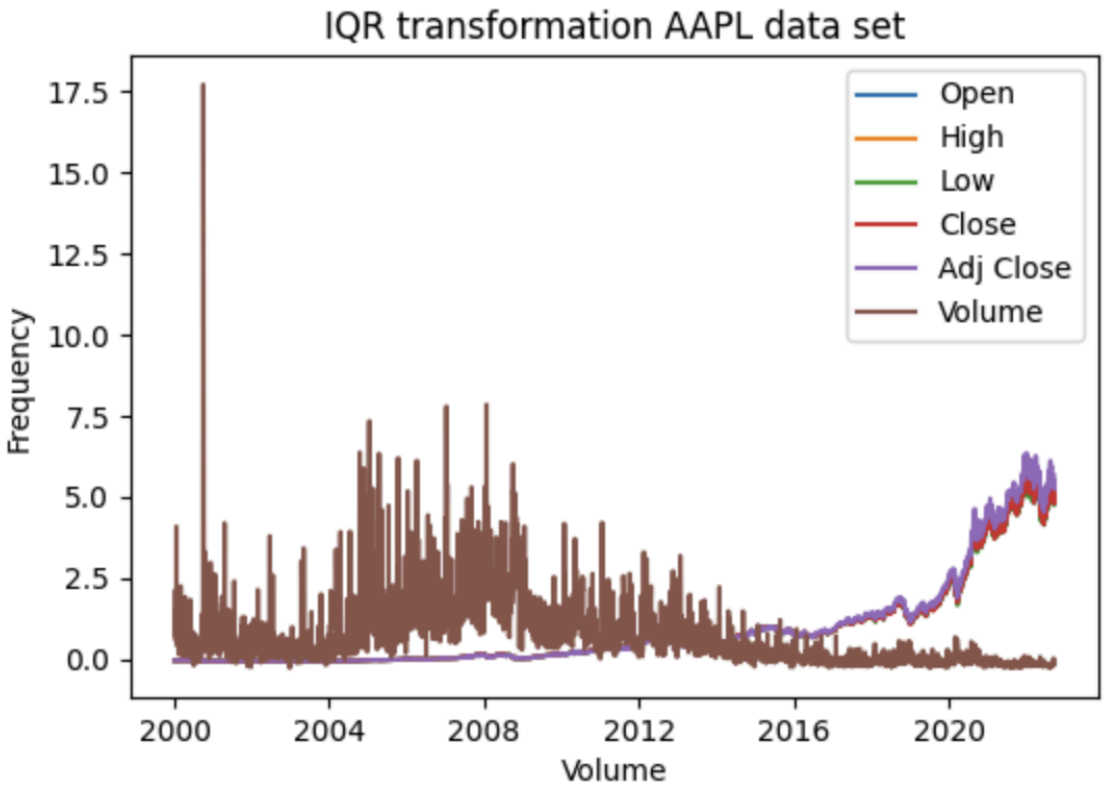
```
#%%---------------Q2---------------
import pandas as pd
from fontTools.misc.bezierTools import epsilon
from pandas_datareader import data
import matplotlib.pyplot as plt
import yfinance as yf
from scipy.special import delta

from Home_Work_3_Q1 import datapreprocessing
apple_stock = yf.download( tickers: 'AAPL',start="2000-01-01",end="2022-09-25")
df = datapreprocessing(apple_stock)
df.Show_original()
df.Show_normalized()
df.Show_standardized()
df.Show_IQR()
```

**Output:**




Normalized AAPL data set
Open
High
Low
Close
Adj Close
Volume
Frequency
1.0
0.8
0.6
0.4
0.2
0.0
2000
2004
2008
2012
2016
2020
Volume

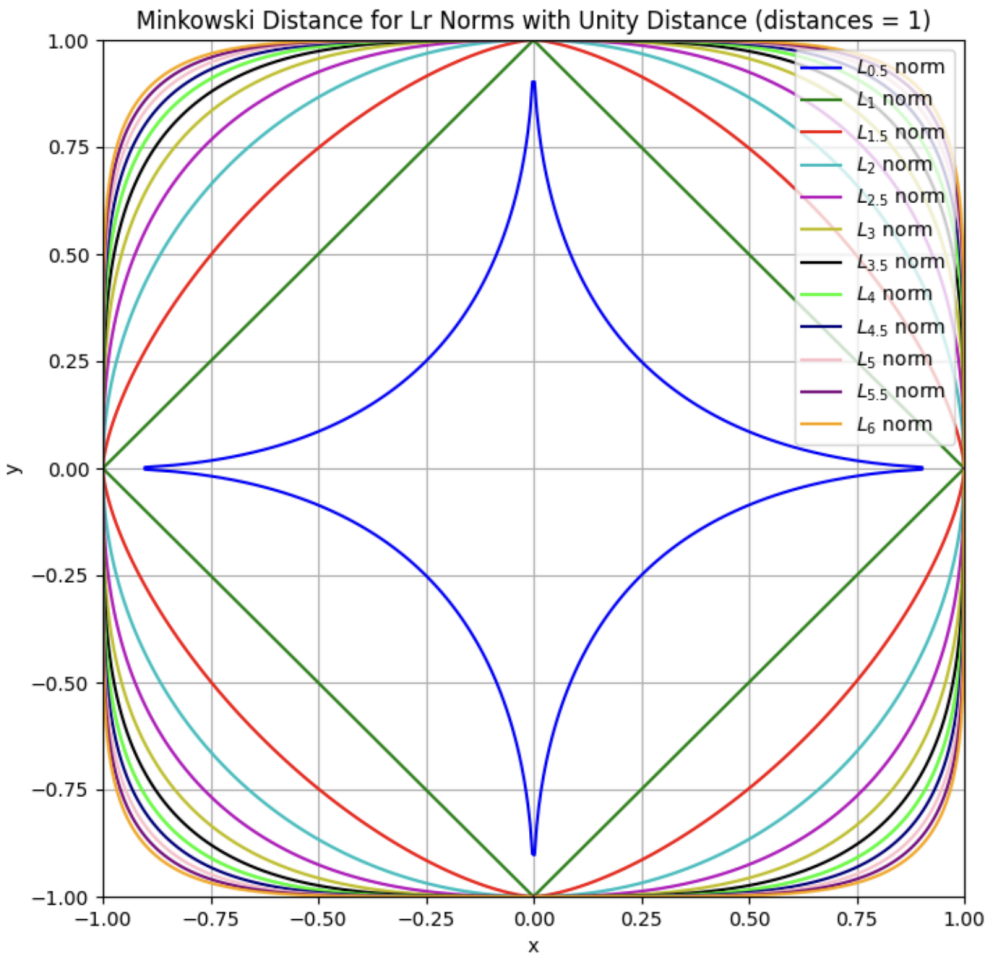

Standardized AAPL data set
Open
High
Low
Close
Adj Close
Volume
Frequency
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
2000
2004
2008
2012
2016
2020
Volume




**Q3.Code:**

```
#%%---------------Q3----------------
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
r_values = [0.5,1,1.5,2,2.5,3,3.5,4,4.5,5,5.5,6]
colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k', 'lime', 'navy', 'pink',
'purple', 'orange']
legend_handles = []
x = np.linspace(-1, stop: 1, num: 400)
y = np.linspace(-1, stop: 1, num: 400)
X,Y = np.meshgrid( *xi: x,y)
plt.figure(figsize=(8,8))
for r,color in zip(r_values, colors):
    Z=(np.abs(X)**r+np.abs(Y)**r)**(1/r)
    plt.contour( *args: X,Y,Z,levels=[1],colors=[color])
    legend_handles.append(plt.Line2D( xdata: [0], ydata: [0], linestyle='-', color=color,
label=f'$L_{{{r}}}$ norm'))
plt.title('Minkowski Distance for Lr Norms with Unity Distance (distances = 1)')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(handles=legend_handles)
plt.grid(True)
plt.show()
```

**Output:**


Minkowski Distance for Lr Norms with Unity Distance (distances = 1)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x
$L_{0.5}$ norm
$L_1$ norm
$L_{1.5}$ norm
$L_2$ norm
$L_{2.5}$ norm
$L_3$ norm
$L_{3.5}$ norm
$L_4$ norm
$L_{4.5}$ norm
$L_5$ norm
$L_{5.5}$ norm
$L_6$ norm

**Q4.**

4) The Manhattan distance $L_1$-norm, between two points $(x_1, y_1)$ and $(x_2, y_2)$

$$d((x_1,y_1),(x_2,y_2)) = |x_1 - x_2| + |y_1 - y_2|$$

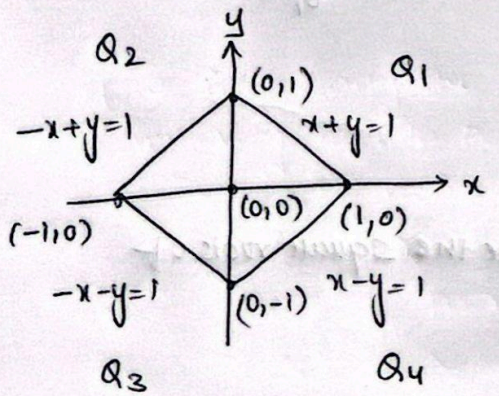
To find all points at a distance 1 unit from origin $(0,0)$ under $L_1$-no

$$|x-0| + |y-0| = 1$$

$$|x| + |y| = 1$$

considering all quadrants :-

| First quadrant | second quadrant | 3rd quadrant | 4th quadrant |
|---|---|---|---|
| $x \geq 0 \quad y \geq 0$. | $x \leq 0 \quad y \geq 0$ | $x \leq 0 \quad y \leq 0$ | $x \geq 0 \quad y \leq 0$ |
| $x + y = 1$ | $-x + y = 1$ | $-x - y = 1$ | $x - y = 1$ |




plotting above 4 lines:-

$$x + y = 1$$

$$-x + y = 1$$

$$-x - y = 1$$

$$x - y = 1$$

we can see they form diamond shape centered $(0,0)$. The vertices of the diamond are $(1,0)$ $(0,1)$ $(-1,0)$ and $(0,-$

∴ Therefore we have proven that $L_1$ minkowski distance with unity distance forms a diamond shape.

Q5.

5) $L_\infty$ Minskowski distance is defined as,

$D(x,y) = \max(|x_1-y_1|, |x_2-y_2| \dots |x_n-y_n|)$

In 2D, $D(x,y) = \max(|x_1-y_1|, |x_2-y_2|)$

→ Let us take a point P(a,b)

Now we need to find q(x,y) such that $L_\infty$ minskowski distance between points P,q = 1

→ we look into all cases where $\max(|x-a|, |y-b|) = 1$.

$$\max(|x-a|, |y-b|) = 1$$

which refers as

$|x-a|$ or $|y-b|$ or both $|x-a|, |y-b| = 1$

① if $|x-a| = 1$

$x-a = 1$ $\quad x-a = -1$

$x = a+1$ $\quad x = a-1$

② if $|y-b| = 1$

$y-b = 1$ $\quad b-y = 1$

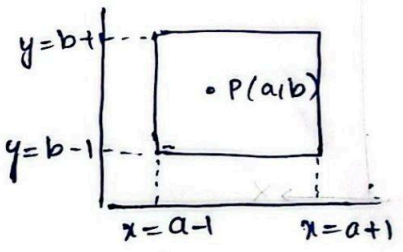
$y = b+1$ $\quad y = b-1$

we derive at four possible points

$(a+1, b)$ $(a-1, b)$ $(a, b+1)$ $(a, b-1)$

The above 4 points are at distance of 1 unit from P(a,b)

They form a square with centre P(a,b).




Hence proved that $L_\infty$ distance a square in 2 dimensions

**Q6.**

**6.a)**

```python
#%%---------------Q6--------------
import numpy as np
from prettytable import PrettyTable
import matplotlib.pyplot as plt
import pandas as pd
np.random.seed(5808)
x = np.random.normal( loc: 1,np.sqrt(2), size: 1000)
epsilon= np.random.normal( loc: 2,np.sqrt(3), size: 1000)
y = x+ epsilon
X = np.vstack((x,y)).T
n = X.shape[0]
X_centered = X- X.mean(axis=0)
cov_matrix = (1/(n-1))*(np.dot(X_centered.T,X_centered))
cov_matrix_table = PrettyTable()
cov_matrix_table.field_names =["Covariance Matrix Element","Value"]
for i in range(cov_matrix.shape[0]):
    for j in range(cov_matrix.shape[1]):
        element_name = f'Cov(x{i+1},x{j+1})'
        value = "{:.2f}".format(cov_matrix[i,j])
        cov_matrix_table.add_row([element_name,value])
print("Estimated Covariance Matrix:")
print(cov_matrix_table)
```

```
Estimated Covariance Matrix:
+--------------------------+-------+
| Covariance Matrix Element | Value |
+--------------------------+-------+
|         Cov(x1,x1)       |  2.09 |
|         Cov(x1,x2)       |  1.97 |
|         Cov(x2,x1)       |  1.97 |
|         Cov(x2,x2)       |  4.41 |
+--------------------------+-------+
```

***Justification*** of why the diagonal elements of calculated covariance matrix $\sum$ is correct:
We know the diagonal elements of the covariance matrix represent the variance of the individual elements.
For x we know (given in question) it is 2 and for y when we calculate we get the variance as 4.41 (see the below attached code)and even when compare those values which we got in the covariance Cov(x1,x1) which is almost equal to 2 and Cov(x2,x2) which is almost equal to 4.41 below the python code to calculate the variance of y.

```python
mean_x=1
variance_x=2
sample_size=1000
np.random.seed(5808)
x = np.random.normal(mean_x,np.sqrt(variance_x),sample_size)
mean_epsilon = 2
variance_epsilon = 3
epsilon = np.random.normal(mean_epsilon,np.sqrt(variance_epsilon),sample_size)
y = x + epsilon
variance_y = np.var(y)
print(f"Variance of y: {variance_y:.2f}")
```

```
Variance of y: 4.41


>>>
```

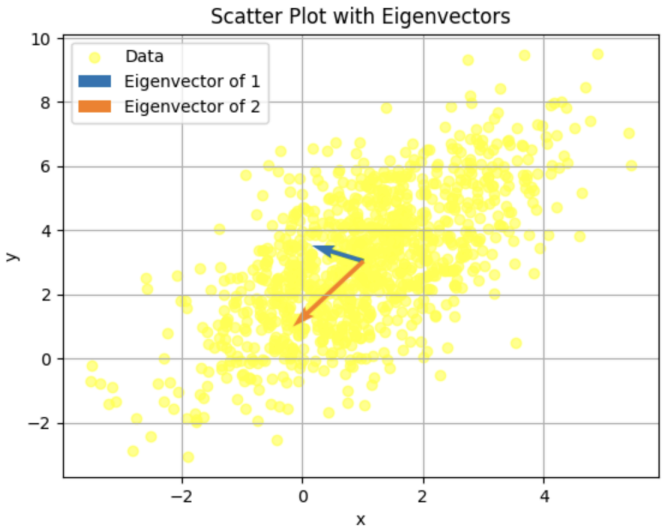
**6.b)**

```python
eigen_values, eigen_vectors = np.linalg.eig(cov_matrix)
eigen_table = PrettyTable()
eigen_table.field_names = ["Eigen value","Eigen vector"]
for i in range(len(eigen_values)):
    eigenvalue_formatted = "{:.2f}".format(eigen_values[i])
    eigenvector_formatted = np.array2string(eigen_vectors[:,i],formatter={'float_kind' : lambda x: "{:.2f}".format(x)})
    eigen_table.add_row([eigenvalue_formatted,eigenvector_formatted])
print("Eigen values & Eigen Vectors: ")
print(eigen_table)
```

```
+--------------------------+-------+
Eigen values & Eigen Vectors:
+-------------+--------------+
| Eigen value |  Eigen vector |
+-------------+--------------+
|     0.97    |  [-0.87 0.50] |
|     5.54    | [-0.50 -0.87] |
+-------------+--------------+
```

**6.c)**

```python
plt.scatter(x, y, label='Data', alpha=0.5,color='yellow')
for i in range(len(eigen_values)):
    plt.quiver(
      *args: X.mean(axis=0)[0],
           X.mean(axis=0)[1],
           eigen_vectors[0, i] *np.sqrt(eigen_values[i]),
           eigen_vectors[1, i] *np.sqrt(eigen_values[i]),
           angles='xy',
           scale_units='xy', scale=1,
           label=f'Eigenvector of {i+1}',color=f'C{i}')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Scatter Plot with Eigenvectors')
plt.legend()
plt.grid(True)
plt.show()
```

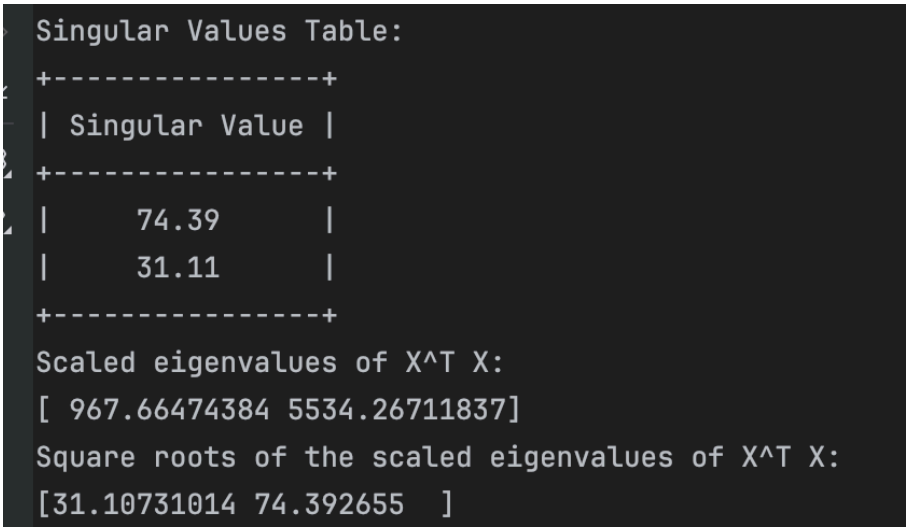



If I were to drop one feature (x or y), I would typically drop the feature corresponding to the smaller variance. Retaining the dimension with higher variance (larger eigenvalue) preserves more information. **So I will drop feature x in this case**.

- Eigenvector corresponding to the maximum eigenvalue represents the direction of maximum variance in the data. In other words, it points along the axis of the data's spread. This direction is often considered the "principal component" of the data.
- Eigenvector corresponding to the minimum eigenvalue represents the direction of minimum variance.

**6.d)**

```
88  eigen_values_scaled = eigen_values * (n - 1)
89  sqrt_eigenvalues_scaled = np.sqrt(eigen_values_scaled)
90  singular_values = np.linalg.svd(X_centered,compute_uv=False)
91  singular_values_table=PrettyTable()
92  singular_values_table.field_names = ["Singular Value"]
93  for i in range(len(singular_values)):
94      singular_value_formatted = "{:.2f}".format(singular_values[i])
95      singular_values_table.add_row([singular_value_formatted])
96  print("Singular Values Table:")
97  print(singular_values_table)
98  print("Scaled eigenvalues of X^T X:")
99  print(eigen_values_scaled)
100 print("Square roots of the scaled eigenvalues of X^T X:")
101 print(sqrt_eigenvalues_scaled)
```




- The **singular values** of a matrix (X_centered) are **the square roots of the eigenvalues** of the matrix ((X_centered) ^ T)*(X_centered) This is because, in Singular Value Decomposition (SVD), the matrix X is decomposed into $U\Sigma V^T$, and the matrix $X^TX = V\Sigma^2T$. The eigenvalues of $X^TX$ are the squares of the singular values, which explains why the singular values are the

square roots of these eigenvalues.

6)

d. $\tilde{X} = U \Sigma V^T$

$U, V$ are orthogonal matrix ($U^T U = I$, $V^T V = I$)

$\tilde{X}^T \cdot \tilde{X} = (U \Sigma V^T)^T \cdot (U \Sigma V^T)$

$= ((V^T)^T \Sigma^T \cdot U^T) \cdot (U \Sigma V^T)$

$= V \cdot \Sigma^T \cdot U^T \cdot U \cdot \Sigma \cdot V^T$

$= (V \cdot V^T) \cdot (U \cdot U^T)(\Sigma^T \cdot \Sigma)$

$= I \cdot I \, (\Sigma^T \Sigma) = \Sigma^2$

$\tilde{X}^T \cdot \tilde{X} = \Sigma^2$ ($\Sigma$ is diagonal matrix)

↳ singular value $\sigma_i^2$

↓

$\lambda_i$ eigen values of $\tilde{X}^T \cdot \tilde{X}$

∴ singular values of $\tilde{X}$ are the square roots of the eigen values of $\tilde{X}^T \cdot \tilde{X}$.

**6.e)**

```
96    df = pd.DataFrame({'x': x, 'y': y})
97    correlation_matrix = df.corr()
98    correlation_matrix = correlation_matrix.round(2)
99    print("Correlation Matrix:")
100   print(correlation_matrix)
101   mean_x = np.mean(x)
102   mean_y = np.mean(y)
103   numerator_x_y = np.sum((x - mean_x) * (y - mean_y))
104   denominator_x = np.sqrt(np.sum((x - mean_x) ** 2))
105   denominator_y = np.sqrt(np.sum((y - mean_y) ** 2))
106   correlation_coefficient = numerator_x_y / (denominator_x * denominator_y)
107   print(f'Sample Pearson Correlation Coefficient between x and y (r):{ correlation_coefficient:.2f}')
```

```
Correlation Matrix:
      x     y
x  1.00  0.65
y  0.65  1.00
Sample Pearson Correlation Coefficient between x and y (r):0.65
```

**Justification:(attached image below)**

mathematical relation between covariance and correlation ρ = σxy / (σx * σy)

- ρ (rho) is the Pearson correlation coefficient.
- σxy is the covariance between X and Y.
- σx and σy are the standard deviations of X and Y, respectively.

6.e) co-relation coefficient (ρ) $= \frac{\sigma_{xy}}{\sigma_x \times \sigma_y}$

co-variance matrix $= \begin{bmatrix} 2.09 & 1.97 \\ 1.97 & 4.41 \end{bmatrix}$ (derived in 6.a)

co-relation matrix $= \begin{bmatrix} 1.00 & 0.65 \\ 0.65 & 1.00 \end{bmatrix}$

$$\rho(x,x) = \frac{cov(x,x)}{\sigma_x \cdot \sigma_x} = \frac{\sigma_x^2}{\sigma_x^2} = 1$$

$$\rho(y,y) = \frac{cov(y,y)}{\sigma_y \cdot \sigma_y} = \frac{\sigma_y^2}{\sigma_y^2} = 1$$

$$\rho(x,y) = \frac{cov(x,y)}{\sigma_x \cdot \sigma_y} = \frac{1.97}{\sqrt{2.09}\sqrt{4.41}} \approx 0.65$$

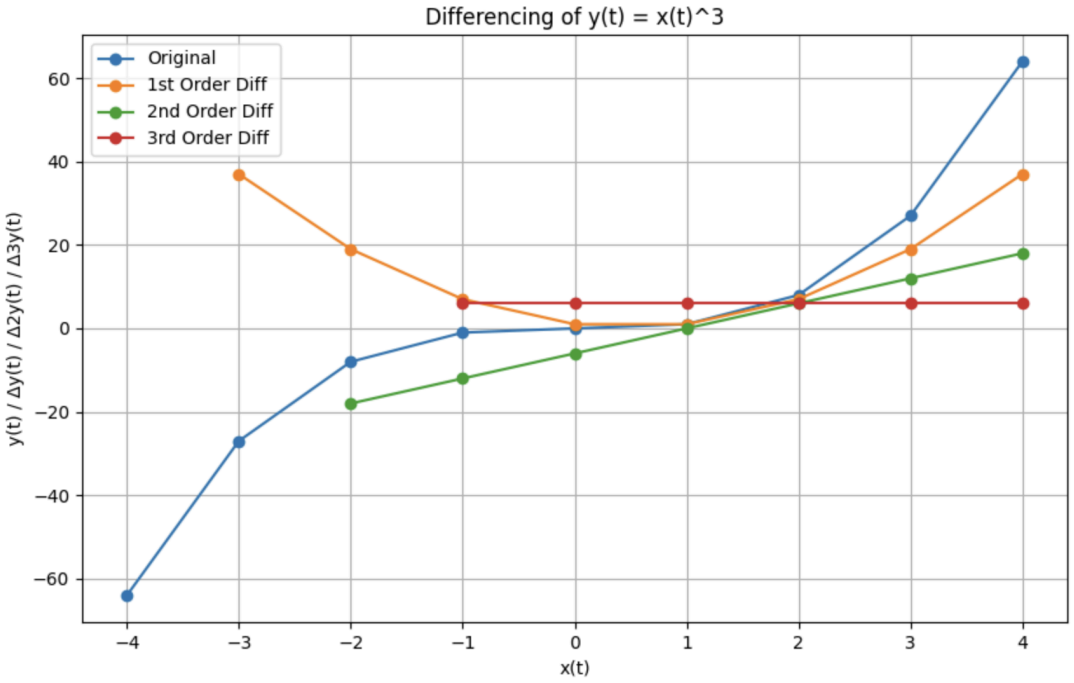
This matches the value in the correlation matrix, confirming that the correlation coefficient matrix is the normalized form of covariance matrix, using the standard deviations to scale the co-variance.

∴ The correlation matrix is derived by normalizing the co-variance matrix

## Q.7)

```python
#%%--------------------Q7----------------
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from prettytable import PrettyTable
x_t = np.arange(-4,5,1)
y_t = x_t ** 3

def first_order_diff(y):  1 usage  new *
    return np.diff(y,n=1)
def second_order_diff(y):  1 usage  new *
    return np.diff(y,n=2)
def third_order_diff(y):  1 usage  new *
    return np.diff(y,n=3)

delta_y_t = first_order_diff(y_t)
delta2_y_t = second_order_diff(y_t)
delta3_y_t = third_order_diff(y_t)

first_order_diff = np.pad(delta_y_t.astype(float),(1,0),mode='constant',constant_values=(np.nan,))
second_order_diff = np.pad(delta2_y_t.astype(float),(2,0),mode='constant',constant_values=(np.nan,))
third_order_diff = np.pad(delta3_y_t.astype(float),(3,0),mode='constant',constant_values=(np.nan,))

data = {'x(t)': x_t, 'y(t)': y_t, 'Δy(t)': first_order_diff, 'Δ2y(t)':second_order_diff, 'Δ3y(t)': third_order_diff}
df = pd.DataFrame(data)
print(df)
plt.figure(figsize=(10, 6))
plt.plot( *args: x_t, y_t, label='Original', marker='o')
plt.plot( *args: x_t, first_order_diff, label='1st Order Diff', marker='o')
plt.plot( *args: x_t, second_order_diff, label='2nd Order Diff', marker='o')
plt.plot( *args: x_t, third_order_diff, label='3rd Order Diff', marker='o')
plt.xlabel('x(t)')
plt.ylabel('y(t) / Δy(t) / Δ2y(t) / Δ3y(t)')
plt.title('Differencing of y(t) = x(t)^3')
plt.legend()
plt.grid(True)
plt.show()
```

```
   x(t)  y(t)  Δy(t)  Δ2y(t)  Δ3y(t)
0    -4   -64    NaN     NaN     NaN
1    -3   -27   37.0     NaN     NaN
2    -2    -8   19.0   -18.0     NaN
3    -1    -1    7.0   -12.0     6.0
4     0     0    1.0    -6.0     6.0
5     1     1    1.0     0.0     6.0
6     2     8    7.0     6.0     6.0
7     3    27   19.0    12.0     6.0
8     4    64   37.0    18.0     6.0


>>>
```




$$\Delta y(t) = y(t+1) - y(t)$$

$$\Delta^2 y(t) = \Delta y(t+1) - \Delta y(t)$$

$$\Delta^3 y(t) = \Delta^2 y(t+1) - \Delta^2 y(t)$$

- This is the cubic function $y(t)=x^3$ (blue colored graph) which curves sharply on either side of x=0 with negative values for negative x(t) and positive values for positive x(t).
- 1st Order Diff (orange colored graph) shows the rate of change of the cubic function. It starts off with large values (because the cubic curve is steep) and decreases toward 0 at x=0, then increases again for positive x(t).
- 2nd Order Diff (green colored graph) shows how the rate of change (slope) is itself changing. The negative values show that the slope is decreasing as x approaches 0, and the positive values show that the slope increases after 0 as it is a 2nd order diff of cubic function hence it's a linear graph.
- 3rd Order Diff (red coloured graph) is constant, indicating that the change in acceleration is uniform and 3rd order diff of $x^3$ is 6.