# PES University, Bangalore

(Established under Karnataka Act No. 16 of 2013)

**APRIL 2022: IN SEMESTER ASSESSMENT (ISA) B.TECH. IV SEMESTER**

**UE20MA251- LINEAR ALGEBRA**

# Assignments

Session: Jan-May 2022

**Name of the Student : Sriram R**

**SRN : PES1UG20CS435**

**Branch : Computer Science and Engineering**

**Semester & Section : Semester IV Section H**

**FOR OFFICE USE ONLY:**

**Marks Allotted : / 05**

Name of the Course Instructor :**Prof. Jyothi R**

Signature of the Course Instructor : ____________________

# Linear Algebra Assignment 1 (I & II included)

**Sriram Radhakrishna PES1UG20CS435 Section : 'H'**

## Python code :

1. Hill cipher encryption using matrix inversion :

```
# implementation of basic cryptographic techniques to appreciate matrix
inversion

keyMatrix = [[0] * 3 for i in range(3)]
messageVector = [[0] for i in range(3)]
Cipher = [[0] for i in range(3)]
cipherMatrix = [[0] for i in range(3)]
DTextMatrix = [[0] for i in range(3)]


def transposeMatrix(m):
    return list(map(list,zip(*m)))

def getMatrixMinor(m,i,j):
    return [row[:j] + row[j+1:] for row in (m[:i]+m[i+1:])]

def getMatrixDeternminant(m):

    # base case for 3x3 matrix
    if len(m) == 2:
        return m[0][0]*m[1][1]-m[0][1]*m[1][0]

    determinant = 0

    for c in range(len(m)):
        determinant += ((-
1)**c)*m[0][c]*getMatrixDeternminant(getMatrixMinor(m,0,c))
        return abs(determinant)

def getmodInverse(a, m):
    for x in range(1, m):
        if (((a%m) * (x%m)) % m == 1):
            return x
    return -1

def getMatrixInverse(m):
    determinant = getMatrixDeternminant(m)

    if(getmodInverse(determinant,26) != -1):
        det = getmodInverse(determinant,26)

        # special case for 2x2 matrix:
        if len(m) == 2:
            return [[m[1][1]/determinant, -1*m[0][1]/determinant],[-
1*m[1][0]/determinant, m[0][0]/determinant]]
```

```
        # find matrix of cofactors
        cofactors = []
        for r in range(len(m)):
            cofactorRow = []

            for c in range(len(m)):
                minor = getMatrixMinor(m,r,c)
                cofactorRow.append(((-1)**(r+c)) *
                getMatrixDeternminant(minor))
                cofactors.append(cofactorRow)

        cofactors = transposeMatrix(cofactors)
        for r in range(len(cofactors)):
            for c in range(len(cofactors)):
                if (cofactors[r][c] < 0):
                    cofactors[r][c] = (cofactors[r][c] + 26) % 26
                cofactors[r][c] = (cofactors[r][c] * det) % 26

        return cofactors

    else:
        print('Non-invertible matrix. Decryption not possible')
        exit

def getKeyMatrix(key):
    k = 0
    for i in range(3):
        for j in range(3):
            keyMatrix[i][j] = ord(key[k]) % 65
            k += 1
    return keyMatrix

def encrypt(messageVector):
    for i in range(3):
        for j in range(1):
            cipherMatrix[i][j] = 0
            for x in range(3):
                cipherMatrix[i][j] += (keyMatrix[i][x] *
messageVector[x][j])
            cipherMatrix[i][j] = cipherMatrix[i][j] % 26

def decrypt(inverse,Cipher):
    for k in range(3):
        for l in range(1):
            DTextMatrix[k][l] = 0
            for y in range(3):
                DTextMatrix[k][l] += (inverse[k][y] * Cipher[y][l])
            DTextMatrix[k][l] = DTextMatrix[k][l] % 26

def HillCipher(message, key):

    # encryption

    keyMatrix = getKeyMatrix(key)

    for i in range(3):
        messageVector[i][0] = ord(message[i]) % 65

    encrypt(messageVector)

    CipherText = []
```

```
    for i in range(3):
        CipherText.append(chr(cipherMatrix[i][0] + 65))

    print('Cipher text : ', ''.join(CipherText))
    Ciphertext = ''.join([str(elem) for elem in CipherText])

    # decryption
    inverse = getMatrixInverse(keyMatrix)

    for i in range(3):
        Cipher[i][0] = ord(Ciphertext[i]) % 65

    if(inverse):
        decrypt(inverse,Cipher)
        Text = []
        for i in range(3):
            Text.append(chr(int(DTextMatrix[i][0]) + 65))

        print('Message vector : ', ''.join(Text))

# Driver Code
def main():
    message = input('Input a 3 Letter message(All in capital Letters): ')
    key = input('Input a 9 Letter key(All in Capital Letters): ')
    HillCipher(message, key)

# main
if __name__ == '__main__':
    main()
```

2. Implementation of Markov Chains for :

   - Population migration distribution between two Indian states :

```
import numpy as np
import random as rm

state = ["S", "T"]
transitionName = [["SS", "ST"], ["TS", "TT"]]
transitionMatrix = [[0, 100], [250, 0]]

if len(transitionMatrix) == 2:
    print("Move forward.")
else:
    print("Transition matrix error")


def pop_mig(transition):

    activityToday = "S"
    print("Start state: " + activityToday)
    activityList = [activityToday]
    i = 0
    prob = 0

    while i != transition:
```

```
        if activityToday == "S":
            change = np.random.choice(transitionName[0], replace=True)
            if change == "SS":
                prob = prob + 0
                activityList.append("S")
                pass
            else:
                prob = prob + 400
                activityToday = "ST"
                activityList.append("T")

        elif activityToday == "T":
            change = np.random.choice(transitionName[1], replace=True)
            if change == "TS":
                prob = prob + 500
                activityList.append("S")
                pass
            else:
                prob = prob + 0
                activityToday = "TT"
                activityList.append("T")

        else:
            return -1;

        i += 1

    print("Possible states : " + str(activityList))
    print("End state after " + str(transition) + " transition: " +
activityToday + ", current population of " + str(transition) + " : " +
str(prob))

pop_mig(1)
```

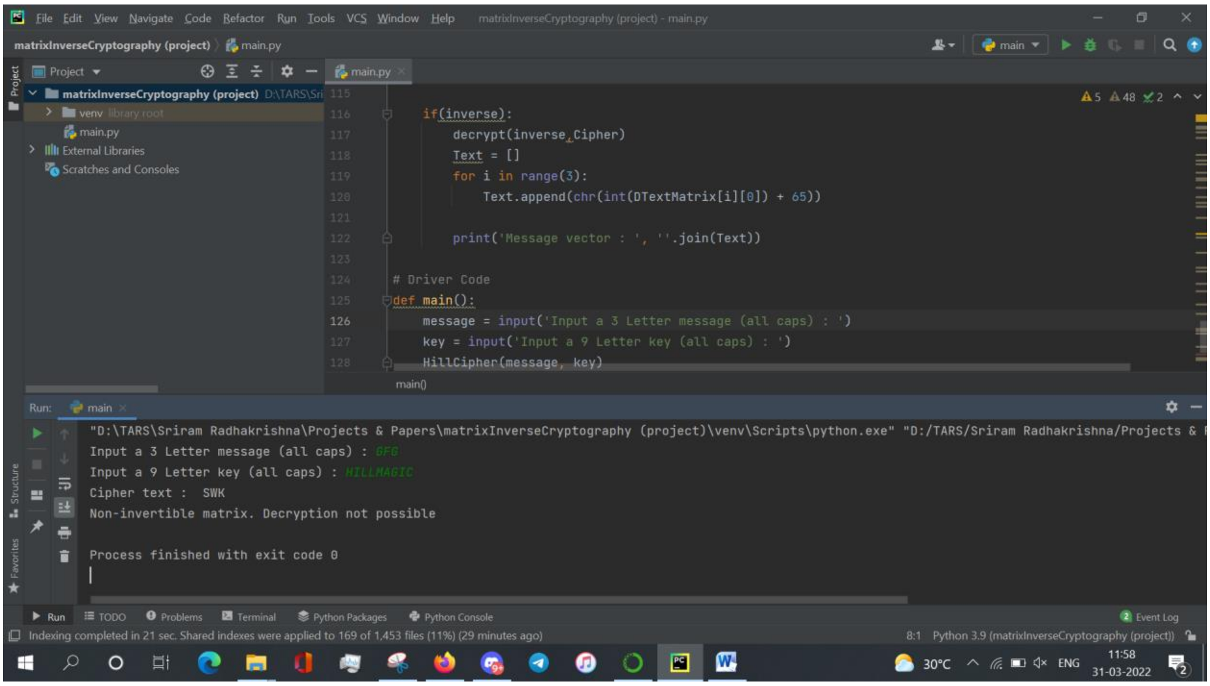
- Vote changing pattern of three political parties from one election to the next :

```
import numpy as np

states = ["A", "B", "C"]
transitionName = [["AA", "AB", "AC"], ["BA", "BB", "BC"], ["CA", "CB",
"CC"]]

if len(transitionName) == 3:
    print("move forward")
else:
    print("N/A")


def vote_change(elections):

    activityToday = "A"
    print("Start state: " + activityToday)

    activityList = [activityToday]
```

```
    i = 0

    while i != elections:

        if activityToday == "A":

            change = np.random.choice(transitionName[0], replace=True)

            if change == "AA":
                activityList.append("A")
                pass
            elif change == "AB":
                activityToday = "B"
                activityList.append("B")
            else:
                activityToday = "C"
                activityList.append("C")

        elif activityToday == "B":

            change = np.random.choice(transitionName[1], replace=True)
            if change == "BB":
                activityList.append("B")
                pass
            elif change == "BC":
                activityToday = "C"
                activityList.append("C")
            else:
                activityToday = "A"
                activityList.append("A")

        elif activityToday == "C":

            change = np.random.choice(transitionName[2], replace=True)

            if change == "CC":
                activityList.append("C")
                pass
            elif change == "CA":
                activityToday = "A"
                activityList.append("A")
            else:
                activityToday = "B"
                activityList.append("B")

        i += 1

    print("Possible states: " + str(activityList))
    print("After " + str(elections) + " elections the votes shifted to
party " + activityToday)

vote_change(4)
```

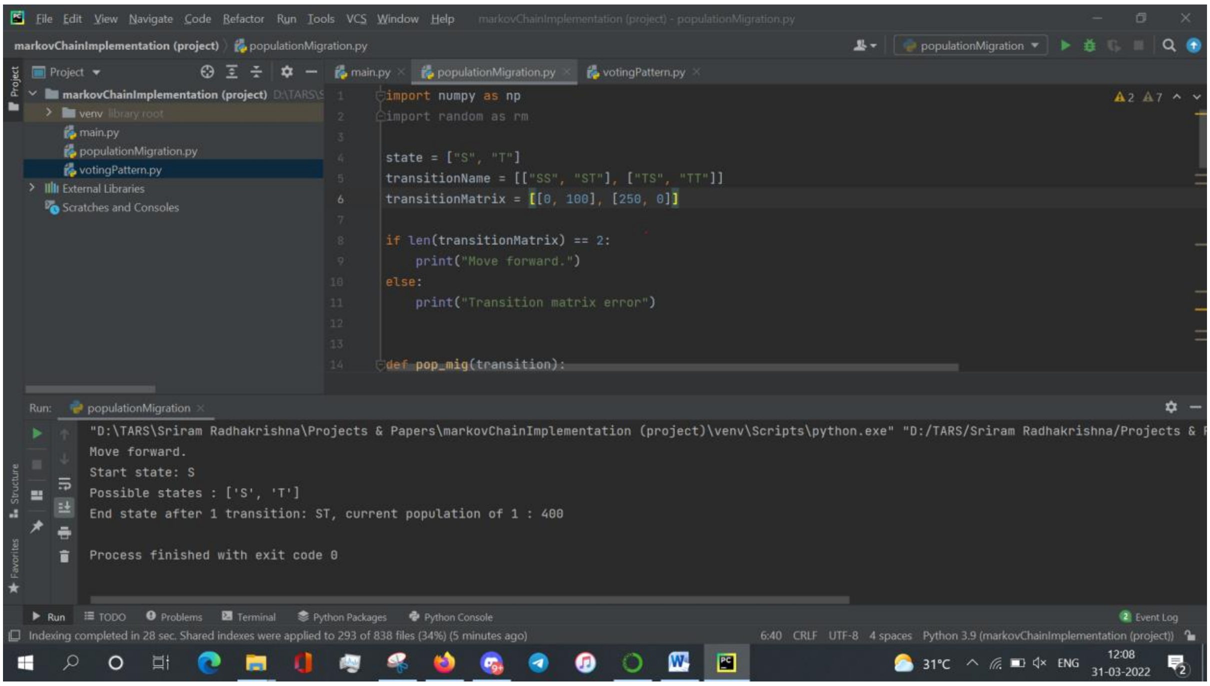
**Output screenshots :**

1. Hill cipher encryption using matrix inversion :

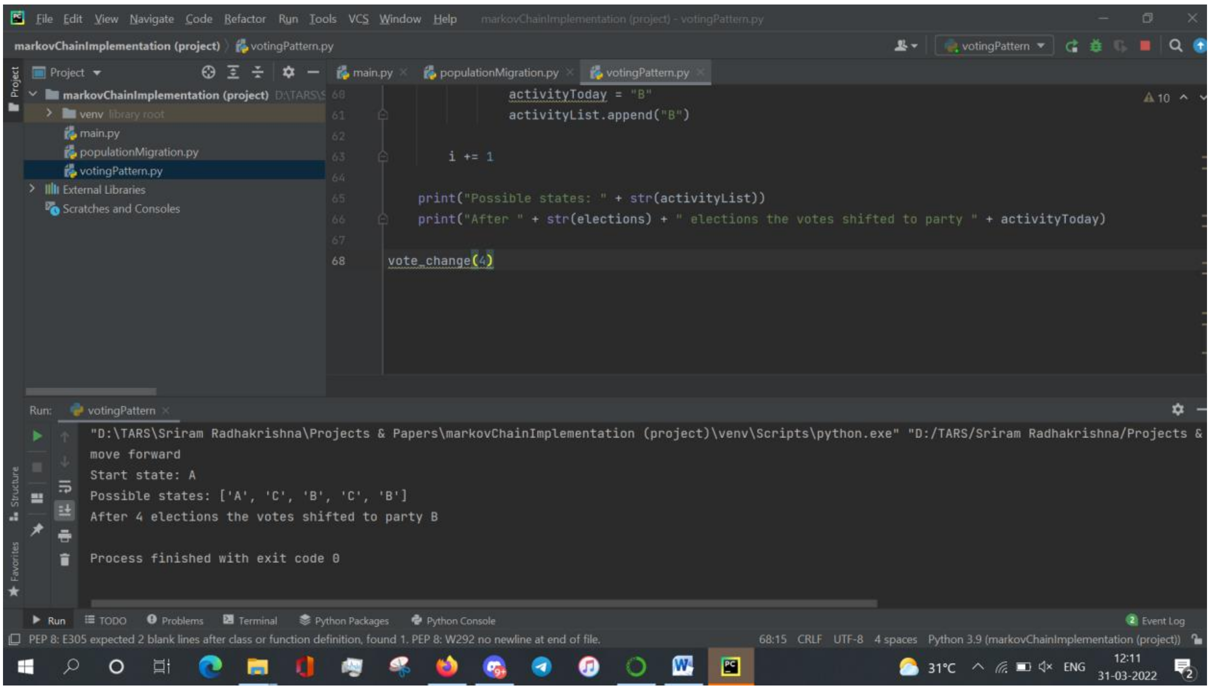



1. Implementation of Markov Chains for :

- Population migration distribution between two Indian states :



- Vote changing pattern of three political parties from one election to the next :



**Linear Algebra Assignment 3**

**Sriram Radhakrishna PES1UG20CS435 Section : 'H'**

**Note : Markov chains and Hill Cipher was submitted in the same document for assignment 1 instead of splitting them and submitting separately. This document was submitted in the form for assignment 2 but is actually assignment 3.**

**Python code (executed on processing IDE) :**

1. Translation :

```
def setup():
    size(200, 200)
    background(255)
    noStroke()

    # draw the original position in gray
    fill(192)
    rect(20, 20, 40, 40)

    # draw a translucent red rectangle by changing the coordinates
    fill(255, 0, 0, 128)
    rect(20 + 60, 20 + 80, 40, 40)

    # draw a translucent blue rectangle by translating the grid
    fill(0, 0, 255, 128)
    pushMatrix()
    translate(60, 80)
    rect(20, 20, 40, 40)
    popMatrix()
```

2. Rotation :

```
def setup():
    size(200, 200)
    background(255)
    smooth()
    fill(192)
    noStroke()
    rect(40, 40, 40, 40)

    pushMatrix()
    # move the origin to the pivot point
    translate(40, 40)

    # then pivot the grid
    rotate(radians(45))

    # and draw the square at the origin
    fill(0)
    rect(0, 0, 40, 40)
    popMatrix()
```

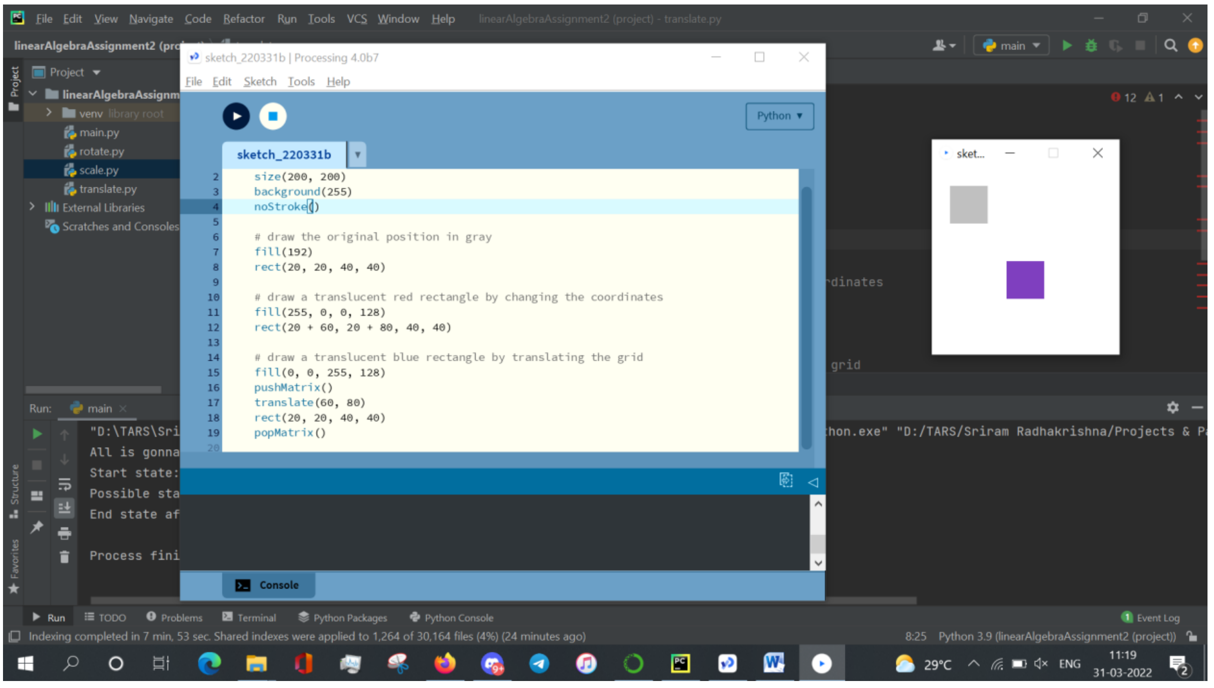
3. Scaling :

```
def setup():
    size(200, 200)
```

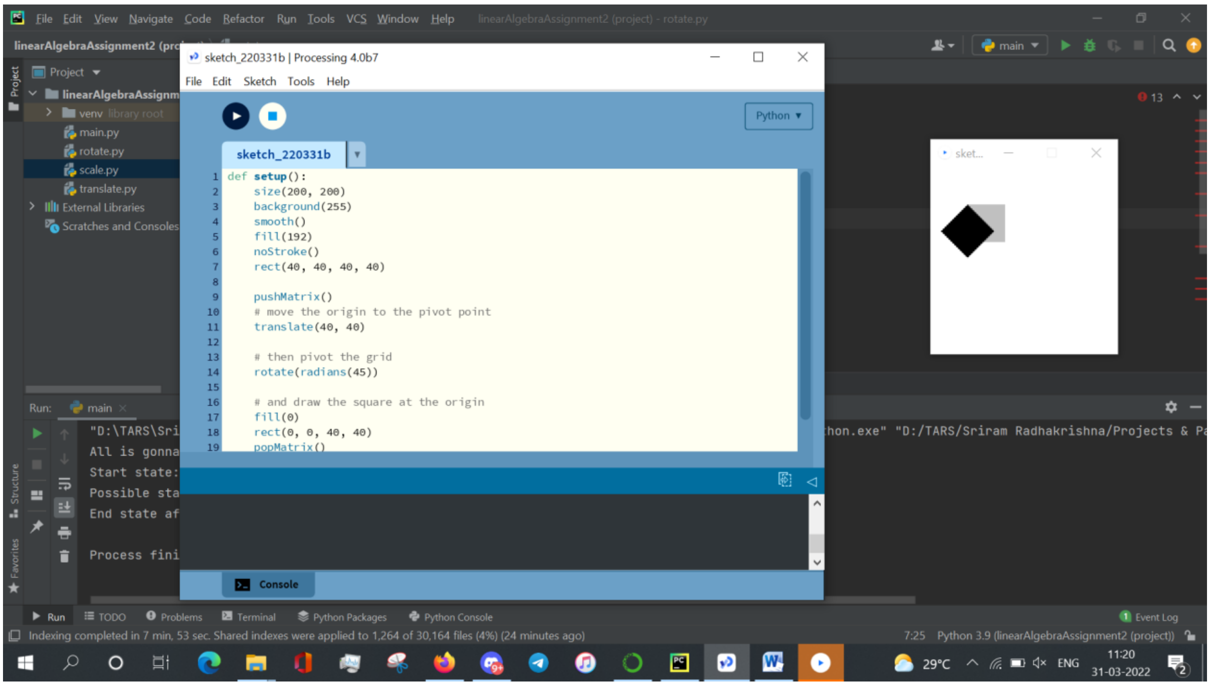
```
background(255)


stroke(128)

rect(20, 20, 40, 40)


stroke(0)

pushMatrix()

scale(2.0)

rect(20, 20, 40, 40)

popMatrix()
```

**Output screenshots :**

1. Translation :




2. Rotation :

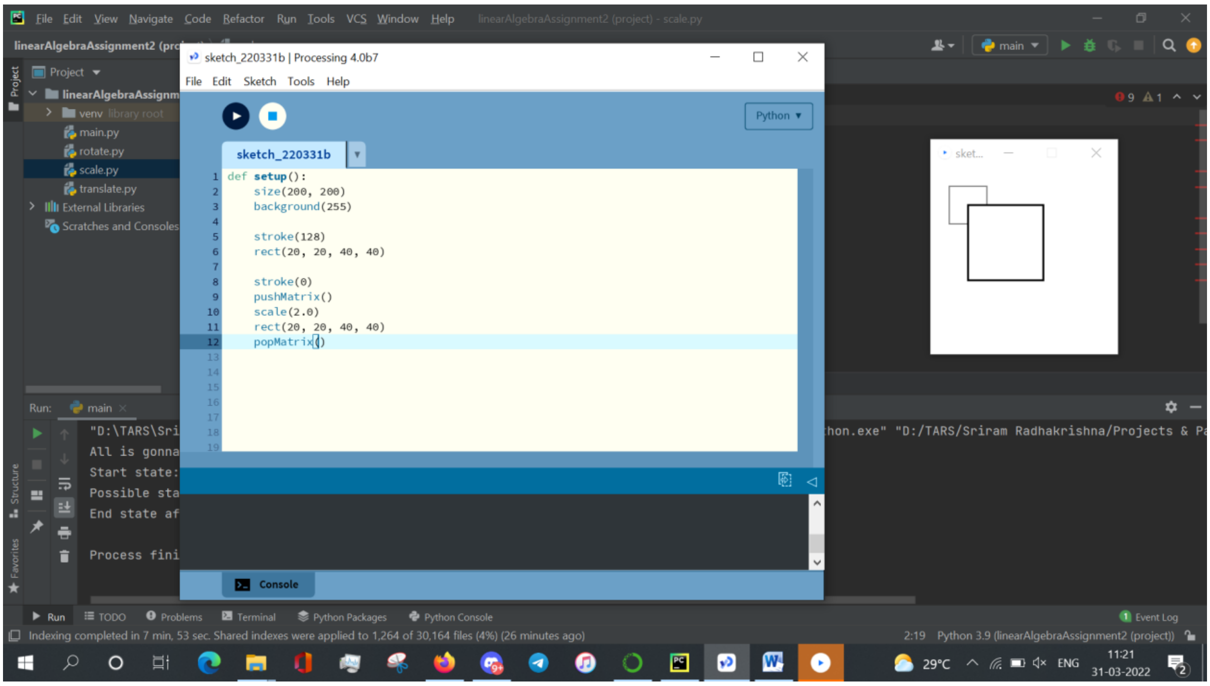

sketch_220331b | Processing 4.0b7
def setup():
    size(200, 200)
    background(255)
    smooth()
    fill(192)
    noStroke()
    rect(40, 40, 40, 40)

    pushMatrix()
    # move the origin to the pivot point
    translate(40, 40)

    # then pivot the grid
    rotate(radians(45))

    # and draw the square at the origin
    fill(0)
    rect(0, 0, 40, 40)
    popMatrix()
Console

3. Scaling :


sketch_220331b | Processing 4.0b7
def setup():
    size(200, 200)
    background(255)

    stroke(128)
    rect(20, 20, 40, 40)

    stroke(0)
    pushMatrix()
    scale(2.0)
    rect(20, 20, 40, 40)
    popMatrix()
Console

**Linear Algebra Assignment 4**

**Sriram Radhakrishna PES1UG20CS435 Section : 'H'**

**Principal Component Analysis applied on MNIST Dataset**

**Python code (executed on a kaggle notebook, space separated based on cell content) :**

```
# initial library import
import numpy as np
import pandas as pd
import seaborn as sns

# data import
data = pd.read_csv('../input/mnist-data/train.csv')
data.head()

# dropping unnecessary labels
label = data['label'] # save label data for later use
data.drop('label', axis = 1, inplace = True)
data.head()

# scaling data to have a mean of 0 and standard deviation of 1
from sklearn.preprocessing import StandardScaler
data_standardized = StandardScaler().fit_transform(data)
data_standardized

# covariance matrix to determine dimensional relationships
covMatrix = np.matmul(data_standardized.T ,data_standardized)
covMatrix

# eigenvalue & eigenvector calculation to determine principal
components
from scipy.linalg import eigh
```

```
values, vector = eigh(covMatrix,eigvals=(782,783))
vector = vector.T
values

# projecting vector on standardized data
projectedData = np.matmul(vector, data_standardized.T)
projectedData

# preparing stacked data for visualization
reducedData = np.vstack((projectedData, label)).T
reducedData = pd.DataFrame(reducedData, columns = ['pca_1', 'pca_2',
'label'])

# data visualization
sns.FacetGrid(reducedData, hue = 'label', size = 8).map(sns.scatterplot,
'pca_1', 'pca_2').add_legend()

# visualization of what the dataset actually represents
import matplotlib.pyplot as plt

index = 1234 # random index chosen for representation purposes
fig_data = np.array(data.iloc[index]).reshape(28,28)
plt.imshow(fig_data, interpolation = None, cmap = 'gray')
plt.show()
print('Digit represented : ', label[index])
```

**Output screenshots :**

[32]:

```python
# initial library import
import numpy as np
import pandas as pd
import seaborn as sns
```

**Principal component analysis of a dataset :**

- Unlike what the name suggests, it is a dimension reduction technique for easier data processing.
- In this notebook, we'll demonstrate the same by converting an of 784 dimensions from the MNIST dataset into a 2D visualization.

[33]:

```python
# data import
data = pd.read_csv('../input/mnist-data/train.csv')
data.head()
```

[33]:

| | label | pixel0 | pixel1 | pixel2 | pixel3 | pixel4 | pixel5 | pixel6 | pixel7 | pixel8 | ... | pixel774 | pixel775 | pixel776 | pixel777 | pixel778 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 |
| **2** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 |
| **3** | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 |
| **4** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 |

5 rows × 785 columns

[34]:

```python
# dropping unnecessary labels
label = data['label'] # save label data for later use
data.drop('label', axis = 1, inplace = True)
data.head()
```

[34]:

| | pixel0 | pixel1 | pixel2 | pixel3 | pixel4 | pixel5 | pixel6 | pixel7 | pixel8 | pixel9 | ... | pixel774 | pixel775 | pixel776 | pixel777 | pixel77 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | |
| **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | |
| **2** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | |
| **3** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | |
| **4** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | |

5 rows × 784 columns

**Data standardization :**

PCA gives more emphasis to variables with high variance. Therefore, if the dimensions are not scaled, we will get inconsistent results. For example, the value for one variable might lie in the range 50-100 and the other one 5-10. In this case, PCA will give more weight to the first variable. Such issues can be resolved by standardizing the dataset before applying PCA.

[35]:
```
# scaling data to have a mean of 0 and standard deviation of 1
from sklearn.preprocessing import StandardScaler
data_standardized = StandardScaler().fit_transform(data)
data_standardized
```

[35]:
```
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., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

[36]:
```
# covariance matrix to determine dimensional relationships
covMatrix = np.matmul(data_standardized.T ,data_standardized)
covMatrix
```

[36]:
```
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., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

[37]:
```
# eigenvalue & eigenvector calculation to determine principal components
from scipy.linalg import eigh
values, vector = eigh(covMatrix,eigvals=(782,783))
vector = vector.T
values
```

[37]:
```
array([1222652.44613786, 1709211.41082575])
```

[38]:
```
# projecting vector on standardized data
projectedData = np.matmul(vector, data_standardized.T)
projectedData
```

[38]:
```
array([[-5.2264454 ,  6.03299601, -1.70581328, ...,  7.07627667,
        -4.34451279,  1.55912058],
       [-5.14047772, 19.29233234, -7.64450341, ...,  0.49539137,
         2.30724011, -4.80767022]])
```

[39]:
```
# preparing stacked data for visualization
reducedData = np.vstack((projectedData, label)).T
reducedData = pd.DataFrame(reducedData, columns = ['pca_1', 'pca_2', 'label'])
```

**Visualization using FacetGrid :**

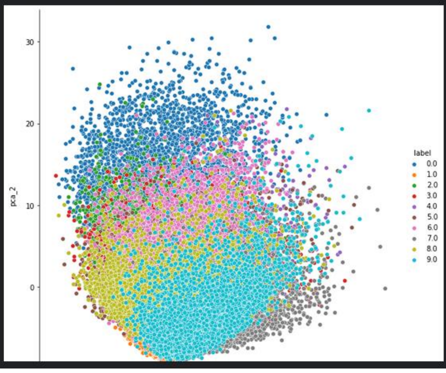
FacetGrid is used for plotting conditional relationships. The basic workflow is to initialize the FacetGrid object with the dataset, and the variables used to structure the grid. Then one or more plotting functions can be applied to each subset by calling FacetGrid.map() or FacetGrid.map_dataframe(), and then the other customizations can also be done.

+ Code  + Markdown

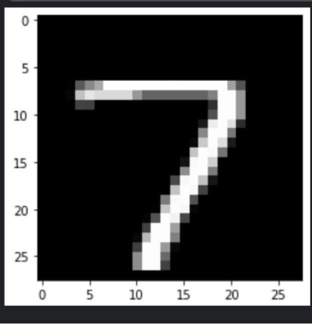
[40]:
```
# data visualization
sns.FacetGrid(reducedData, hue = 'label', size = 8).map(sns.scatterplot, 'pca_1', 'pca_2').add_legend()
```

```
/opt/conda/lib/python3.7/site-packages/seaborn/axisgrid.py:337: UserWarning: The `size` parameter has been renamed to `height`; please update your code.
  warnings.warn(msg, UserWarning)
```

[40… `<seaborn.axisgrid.FacetGrid at 0x7fa5c8a0bad0>`




[41]:
```
# visualization of what the dataset actually represents
import matplotlib.pyplot as plt

index = 1234 # random index chosen for representation purposes
fig_data = np.array(data.iloc[index]).reshape(28,28)
plt.imshow(fig_data, interpolation = None, cmap = 'gray')
plt.show()
print('Digit represented : ', label[index])
```



**Linear Algebra Assignment 5**

**Sriram Radhakrishna PES1UG20CS435 Section : 'H'**

**Applications of Linear Algebra on Page Rank Algorithm**

**Python code (executed on IDLE) :**

```
import numpy as np
import scipy as sc
import pandas as pd
from fractions import Fraction

def display_format(my_vector, my_decimal):
    return np.round((my_vector).astype(np.float), decimals=my_decimal)

my_dp = Fraction(1,3)
Mat = np.matrix([[0,0,1],
[Fraction(1,2),0,0],
[Fraction(1,2),1,0]])
Ex = np.zeros((3,3))
Ex[:] = my_dp
beta = 0.7
Al = beta * Mat + ((1-beta) * Ex)
r = np.matrix([my_dp, my_dp, my_dp])
r = np.transpose(r)
previous_r = r

for i in range(1,100):
  r = Al * r
  print(display_format(r,3))

  if (previous_r==r).all():
    break
```

previous_r = r

print ("Final:\n", display_format(r,3))
print ("sum", np.sum(r))

**Output screenshots :**