Work Integrated Learning Programmes Division
M. Tech. in Data Science and Engineering
Assignment 2
DSECL ZC416 - Mathematical Foundations for Data Science

VGT GAYAN

2021FC04612

Q1)

i)

```
import numpy as np


def generate_random_matrix(m,n):
    if m > n:
        rand_gen = np.random.default_rng()
        return np.round_(rand_gen.uniform(low=0, high=10, size=(m,n)),4)
    else:
        print("Error: No: of rows are lesser than no: of columns")
        print(f"Given values: rows = {m}, columns = {n}")
        raise ValueError


def frobenius_norm(matrix):
    m,n = matrix.shape
    square_sum = 0
    for i in range(m):
        for j in range(n):
            square_sum += matrix[i][j]**2

    return round(np.sqrt(square_sum),4)


def main_Q1_i():
    try:
        print("Condition: No: of rows should be greater than no: of columns")
        m = int(input("Enter no: of rows:"))
        n = int(input("Enter no: of columns:"))

        # Generate random matrix
        rand_matrix = generate_random_matrix(m,n)
        print(f"Dimensions of the generated matrix: {rand_matrix.shape}")

        # Calculate frobenius norm
        f_norm = frobenius_norm(rand_matrix)
        print("Frobenius norm: ", f_norm)
    except ValueError:
        print("Invalid Input !!!")


main_Q1_i()
```

```
'''
Input:
    Enter no: of rows: 4
    Enter no: of columns: 3

Output:
    Condition: No: of rows should be greater than no: of columns
    Dimensions of the generated matrix: (4, 3)
    Frobenius norm:  18.7508



Invalid Input:
    Enter no: of rows: 2
    Enter no: of columns: 3

Output:
    Condition: No: of rows should be greater than no: of columns
    Error: No: of rows are lesser than no: of columns
    Given values: rows = 2, columns = 3
    Invalid Input !!!
'''
```

ii)

```
def is_gram_schmidt_applicable(matrix):
    rank = get_rank(matrix)
    rows,cols = matrix.shape
    print(f"Rank = {rank}, Columns = {cols}")
    # Check if no: columns equal to the rank
    if rank == cols:
        print("Columns of the marix are Linearly Independant")
        print("Gram-Schmidt Algorithm can be applied")
        return True
    else:
        print("Columns of the marix are Linearly dependant")
        print("Gram-Schmidt Algorithm cannot be applied")
        return False
```

```python
def get_rank(A):
    A = gauss_ellimination(A)
    rows, cols = A.shape
    rank = min(rows, cols)
    for row in range(rows):
        # Break if current row index is greater than total columns
        if row >= cols:
            break
        # Deduct no of zero rows from rank
        if sum(A[row]) == 0:
            rank -= 1
    return rank
```

```python
def gauss_ellimination(A,pivot_enable=True):
    rows , cols = A.shape
    add_count = 0
    mul_count = 0
    div_count = 0

    ## Forward ellimination process -------------------
    # Do below for each row
    for row in range(rows):
        # Break if current row index is greater than total columns
        if row >= cols:
            break
        # Partial pivotting (if enabled)
        if pivot_enable:
            # Current pivot value
            max_pivot = abs(A[row][row])
            max_pivot_index = row
            # Iterate through pivot column to find the maximum pivot value
            for i in range(row+1,rows):
                if max_pivot < abs(A[i][row]):
                    max_pivot = abs(A[i][row])
                    max_pivot_index = i

            # Do partial pivotting
            if row < max_pivot_index:
                A[row],A[max_pivot_index] = A[max_pivot_index],A[row]

        else:
            if A[row][row] == 0:
                print("Pivot value is zero. Please enable partial pivotting to do
the calculations...")
```

```
                return 0

        ## Apply Row transformation for all the rows below current row
        pivot_element = A[row][row]
        for row2 in range(row+1, rows):
            # Interested element: The element that we set to zero
            interested_element = A[row2][row]

            # Check if interested element if zero and skip
            if interested_element == 0:
                continue

            # Calculate the row multiple value
            row_multiple = interested_element/pivot_element
            div_count += 1
            # Update the interested element to zero
            A[row2][row] = 0

            # Iterated through other columns to update the rest of the values in
row
            for col in range(row+1, cols):
                tmp = A[row2][col] - row_multiple*A[row][col]
                A[row2][col]= tmp
                mul_count += 1
                add_count += 1
    ## End of Forward ellimination process -------------------

    return A
```

Sample output:

```
Rank = 5, Columns = 5

Columns of the marix are Linearly Independant

Gram-Schmidt Algorithm can be applied
```

Q1)

iii)

```
def main_Q1_iii():
    try:
        A = generate_random_matrix(7,5)
        # Keep generating A until the linear independence is obtained.
        while not is_gram_schmidt_applicable(A):
            A = generate_random_matrix(7,5)

        Q = generate_orthogonal_matrix(A)
        print("A: \n", A)
        print("Orthogonal matrix Q: \n", Q)

    except Exception as e:
        print(e)
```

```
def generate_orthogonal_matrix(matrix):
    # Check if Gram-Schmidt Algorithm can be applied to columns of the given
matrix
    # ie: Columns are LI
    if not is_gram_schmidt_applicable(matrix):
        return None

    rows, cols = matrix.shape
    # Initialize orthogonal_matrix
    Q = np.zeros((rows, cols))
    # print(matrix)
    # Apply Gram-Schmidt orthogonalization on each column vector
    for i in range(cols):
        # ith column of input matrix
        x = matrix[:,i].copy()
        # print(x)

        # Calculate ith column of Q
        v = x.copy()
        for j in range(i-1, -1, -1):
            # jth column of Q
            v_j = Q[:,j].copy()

            x_dot_v = _dot_product(x,v_j)
            v_dot_v = _dot_product(v_j,v_j)

            v -= (x_dot_v/v_dot_v)*v_j
```

```
        # Normalize v
        v = v/np.sqrt(_dot_product(v,v))

        # Update ith column in Q
        Q[:,i] = np.round_(v, 4)

    return Q
```

```
def _dot_product(a,b):
    # Check for compatibility
    if a.shape != b.shape:
        print("Given vectors have different dimensions. Hence dot product not 
applicable")
        return None

    r = len(a)
    sum = 0
    for i in range(r):
        sum += a[i]*b[i]

    return round(sum, 4)
```

Sample output:

```
A:
 [[ 8.5419      2.65166667 -4.81908365  1.58279229 -0.60255141]
 [ 8.7775     -0.79304545 -2.48019082 -0.53001549 -0.02286998]
 [ 5.889      -2.91044848  6.31665691 -1.88894196 -1.07871874]
 [ 6.2589      3.00803333  3.84651058 -4.2489002   1.07842197]
 [ 6.0485     -4.33980606 -1.74497858  1.9629225   0.87199594]
 [ 1.6981      1.93426364  6.05146023  6.35207885  0.62609228]
 [ 5.962       0.44134848 -0.03510594  0.70319287 -0.65351835]]
Orthogonal matrix Q:
 [[ 0.0286  0.0553 -0.0389  0.0229 -0.1411]
 [ 0.0294 -0.0165 -0.02   -0.0077 -0.0054]
 [ 0.0197 -0.0607  0.051  -0.0273 -0.2526]
 [ 0.0209  0.0627  0.0311 -0.0615  0.2526]
 [ 0.0202 -0.0905 -0.0141  0.0284  0.2042]
 [ 0.0057  0.0403  0.0489  0.0919  0.1466]
 [ 0.0199  0.0092 -0.0003  0.0102 -0.1531]]
```

Q1) iv)

```
import traceback
def main_Q1_iv():
    try:
        A = generate_random_matrix(7,5)
        # Keep generating A until the linear independence is obtained.
        while not is_gram_schmidt_applicable(A):
            A = generate_random_matrix(7,5)
        print("A: \n",A)

        # QR Decomposition
        Q, R = QR_decomposition(A)
        print("Q: \n",Q)
        print("R: \n",R)

        # QR muliplication
        QR = matrix_multiplication(Q,R)
        print("QR: \n",QR)

        # value of ‖A – (Q.R)‖F
        print("A-QR: \n",A-QR)
        print("‖A-(Q.R)‖F = ",frobenius_norm(A-QR))


    except Exception as e:
        traceback.print_exc()
        print(e)
```

```
def QR_decomposition(A):
    # Generate Q
    Q = generate_orthogonal_matrix(A)
    # Obtain Q transpose
    Q_transpose = matrix_transpose(Q)
    # Obtain R = Q_transpose x A
    R = matrix_multiplication(Q_transpose,A)

    return Q,R
```

```python
def matrix_transpose(A):
    r, c = A.shape
    # Initialize transpose
    A_transpose = np.zeros((c,r))
    for row in range(r):
        for col in range(c):
            A_transpose[col,row] = A[row,col]

    return A_transpose
```

```python
# Perform AxB matrix multiplication
def matrix_multiplication(A,B):
    rA, cA = A.shape
    rB, cB = B.shape
    # Check condition for matrix multiplication
    if cA != rB:
        print("Matrices are not compatible to perform multiplication")
        return None

    # Initialize resultant matrix
    C = np.zeros((rA,cB))
    # Multilplication
    for row in range(rA):
        for col in range(cB):
            # Sum of product
            sop = 0
            for i in range(cA):
                sop += A[row,i]*B[i,col]
            C[row,col] = sop

    return C
```

Sample output:

```
A:
 [[9.     3.1056 7.3745 7.2823 5.2874]
 [1.6211 8.8058 7.2107 4.9337 3.4344]
 [7.1355 6.2748 2.1972 2.8808 1.4069]
 [0.0851 3.5417 7.8682 0.338  4.1927]
 [7.1991 9.0688 1.9857 8.873  2.1186]
```

```
 [8.8316 7.6834 1.0007 8.8622 4.3197]
 [3.2754 1.1262 5.0378 1.7104 2.5548]]
Q:
 [[ 0.5426 -0.4246  0.4836  0.2857 -0.1557]
 [ 0.0977  0.7521  0.2473  0.1589 -0.1832]
 [ 0.4302  0.0464 -0.1426 -0.8551 -0.166 ]
 [ 0.0051  0.3489  0.5715 -0.206   0.4589]
 [ 0.434   0.322  -0.2964  0.2768 -0.4218]
 [ 0.5325  0.049  -0.3344  0.183   0.7137]
 [ 0.1975 -0.155   0.398  -0.0968 -0.1297]]
R:
 [[ 1.65860355e+01  1.35126010e+01  8.08088440e+00  1.45822499e+01
    7.55540092e+00]
 [ 1.77950000e-03  9.95313347e+00  5.04669053e+00  3.89641031e+00
    2.36395589e+00]
 [ 9.19300000e-04 -2.56110000e-04  1.06147187e+01 -3.88548470e-01
    4.54616755e+00]
 [ 1.63110000e-03 -1.37026000e-03 -1.88922000e-03  4.24378024e+00
    1.11922889e+00]
 [-2.01297000e-03 -7.40550000e-04  5.59999999e-07 -3.30690000e-04
    2.09604122e+00]]
QR:
 [[9.00005127 3.10543682 7.37460122 7.28251046 5.28776157]
 [1.62264933 8.8057874  7.209838   4.93348524 3.43421185]
 [7.13420334 6.27627753 2.1985194  2.88068278 1.40674207]
 [0.08447527 3.5413586  7.86870403 0.3374011  4.19276367]
 [7.19994048 9.06938682 1.98541238 8.87332421 2.1184461 ]
 [8.83070551 7.68246995 1.00045151 8.86227857 4.31960991]
 [3.27593526 1.12612978 5.0385785  1.71065344 2.5549553 ]]
A-QR:
 [[-5.12739270e-05  1.63176075e-04 -1.01216884e-04 -2.10457301e-04
   -3.61571397e-04]
 [-1.54932913e-03  1.25961850e-05  8.62004228e-04  2.14761775e-04
    1.88151015e-04]
 [ 1.29666047e-03 -1.47752663e-03 -1.31939764e-03  1.17218592e-04
    1.57929909e-04]
```

```
[ 6.24730085e-04  3.41398662e-04 -5.04026856e-04  5.98901884e-04
 -6.36740560e-05]
[-8.40476026e-04 -5.86820066e-04  2.87623616e-04 -3.24211422e-04
  1.53895804e-04]
[ 8.94490709e-04  9.30045776e-04  2.48491930e-04 -7.85717500e-05
  9.00946260e-05]
[-5.35257929e-04  7.02217520e-05 -7.78495254e-04 -2.53435326e-04
 -1.55300864e-04]]
‖A-(Q.R)‖F =  0.0038
```

Q2) i)

```
def main_Q2_i():
    # 07...0542
    # n1n2n3n4 = 3542 => 35x42
    A = generate_random_matrix(35, 42)
    print("A: \n",A)
    print("l_infinity_norm = ", l_infinity_norm(A))
```

```
def generate_random_matrix(m,n):
        rand_gen = np.random.default_rng()
        return np.round_(rand_gen.uniform(low=0, high=10, size=(m,n)),4)
```

```
def l_infinity_norm(A):
    r,c  = A.shape
    max = 0
    for row in A:
        row_sum = sum(row)
        if max < row_sum:
            max = row_sum
    return max
```

Sample output:

```
A:
 [[3.7537 4.6111 7.4423 ... 9.327  2.128  7.4313]
 [7.9847 4.4052 3.6403 ... 1.7817 7.8962 6.6332]
 [9.4836 5.8936 0.7717 ... 5.7481 6.4143 5.3507]
```

```
 ...
 [4.5869 6.7297 6.0956 ... 9.739  0.9695 9.9127]
 [4.5774 3.7517 1.4167 ... 8.7043 6.8602 6.7845]
 [9.7426 1.7878 2.19   ... 6.5531 6.1136 5.92  ]]
l_infinity_norm =  250.2958
```

Q2) ii)

```python
import math
def vector_l2_norm(a):
    rows = len(a)
    square_sum = 0
    for i in range(rows):
        square_sum += a[i]**2
    return math.sqrt(square_sum)
```

```python
def matrix_transpose(A):
    r, c = A.shape
    # Initialize transpose
    A_transpose = np.zeros((c,r))
    for row in range(r):
        for col in range(c):
            A_transpose[col,row] = A[row,col]

    return A_transpose
```

```python
# Perform AxB matrix multiplication
def matrix_multiplication(A,B):
    rA, cA = A.shape
    rB, cB = B.shape
    # Check condition for matrix multiplication
    if cA != rB:
        print("Matrices are not compatible to perform multiplication")
        return None

    # Initialize resultant matrix
    C = np.zeros((rA,cB))
    # Multilplication
    for row in range(rA):
        for col in range(cB):
            # Sum of product
            sop = 0
            for i in range(cA):
```

```
                sop += A[row,i]*B[i,col]
            C[row,col] = sop

    return C
```

```
def function_fx(A,b,x):
    # Check for dimension compatibility
    r_A, c_A = A.shape
    r_b, c_b = b.shape
    r_x, c_x = x.shape

    if (r_A != r_b) or (c_A != r_x):
        print("Dimension incompatibility in A,b,x")
        return None

    Ax = matrix_multiplication(A,x)
    return 0.5*((vector_l2_norm(Ax - b))**2)
```

```
def gradient_fx(A,b,x):
    # Check for dimension compatibility
    r_A, c_A = A.shape
    r_b, c_b = b.shape
    r_x, c_x = x.shape

    if (r_A != r_b) or (c_A != r_x):
        print("Dimension incompatibility in A,b,x")
        return None

    A_transpose = matrix_transpose(A)
    Ax = matrix_multiplication(A,x)
    Ax_b = Ax - b

    # Return AT(Ax-b)
    return matrix_multiplication(A_transpose, Ax_b)
```

```
def get_step_size(A,b,x):
    A_transpose = matrix_transpose(A)
    g_k = gradient_fx(A,b,x)
    g_k_transpose = matrix_transpose(g_k)

    # Calculate step size
    numerator = matrix_multiplication(g_k_transpose, g_k)
    tmp1 = matrix_multiplication(g_k_transpose,A_transpose)
```

```
    tmp2 = matrix_multiplication(A,g_k)
    denominator = matrix_multiplication(tmp1,tmp2)

    return numerator/denominator
```

```
import pandas as pd

def gradient_descent_algo(A, b):
    # Check for dimension compatibility
    r_A, c_A = A.shape
    r_b, c_b = b.shape
    if r_A != r_b:
        print("No of rows in A and b are different.")
        return None
    # Initial guess for x
    x_k = np.zeros((c_A,1))

    # List to keep estimates of x at each iteration
    x_list = [x_k]
    # List to keep function value at each iteration
    fx_list = [function_fx(A,b,x_k)]

    # Gradient descent iterations
    while True:
        g_k = gradient_fx(A,b,x_k)
        step_size = get_step_size(A,b,x_k)

        # Next guess for x
        x_k_plus_1 = x_k - step_size*g_k

        # Error
        error_l2 = vector_l2_norm(x_k - x_k_plus_1)

        # Update x_k
        x_k = x_k_plus_1

        # Store x_k and f(x_k)
        x_list.append(x_k)
        fx_list.append(function_fx(A,b,x_k))

        # Terminating condition
        if error_l2 < 0.0001:
            break
```

```python
    # Save x_k and f(x_k) values to a file
    df  = pd.DataFrame(data=list(zip(x_list,fx_list)), columns=["x_k", "f(x_k)"])
    print(df)
    df.to_csv("gradient_descent_results.csv")


    # return List of estimates of x
    # (last element is the final estimate of local minima)
    return x_list, fx_list
```

```python
import matplotlib.pyplot as plt

def main_Q2_ii():
    # mobile number: 07...0542
    # n1n2n3n4 = 3542 => 35x42
    A = generate_random_matrix(35, 42)

    # vector b - 35x1
    b = generate_random_matrix(35, 1)

    # call gradient descent algorithm
    x_list, fx_list = gradient_descent_algo(A, b)
    print("Local minima estimate: ", x_list[-1])


    # Plot the graph of f(xk) vs k where k is the iteration number and xk is the current estimate of x at iteration k
    k = range(len(fx_list))
    plt.plot(fx_list)
    plt.ylabel('f(xk)')
    plt.xlabel('Iteration')
    plt.show()
```

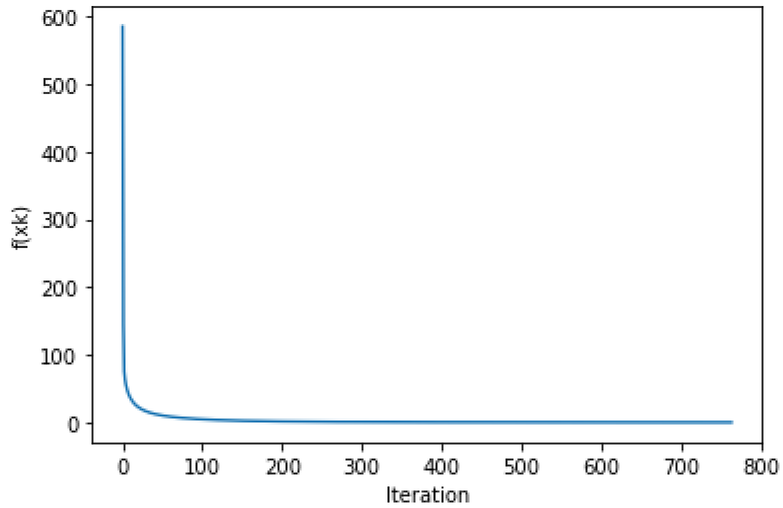
expression for τ = $\tau = \dfrac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_k^T \mathbf{A}^T \mathbf{A} \mathbf{g}_k}$

This is calculated using the function “get_step_size()”.

The values of x_k and $f(x_k)$ should be stored in a file.

This file is dumped in csv table format with the name “gradient_descent_results.csv” in the working directory.

Q2) iii) The graph of f(xk) vs the iteration number



Q3)

i)

Mobile: 0710310542

After replacing 0 by 3 -> 3713313542

polynomial = 3*x^3 - 7*(x^2)*y + x*y^2 - 3*y^3 + 3*x^2 - x*y + 3*y^2 - 5*x + 4*y - 2

ii)

Octave code

```
pkg load symbolic
syms x y real

# 0710310542
# After replacing 0 by 3 -> 3713313542

polynomial = 3*x^3 - 7*(x^2)*y + x*y^2 - 3*y^3 + 3*x^2 - x*y + 3*y^2 - 5*x + 4*y
- 2
#dx = diff(polynomial, x)
dx = 9*x^2 - 14*x*y + y^2 + 6*x - y -5
#dy = diff(polynomial, y)
dy = -7*x^2 + 2*x*y -9*y^2 -x +6*y + 4
```

```
d = solve(dx == 0, dy == 0, x, y)
printf("Critical Points: \n");
printf("x1: %d, y1= %d, z1 = %d \n", double(d{1}.x), double(d{1}.y),
double(pol_f(double(d{1}.x), double(d{1}.y))));
printf("x2: %d, y2= %d, z2 = %d \n", double(d{2}.x), double(d{2}.y),
double(pol_f(double(d{2}.x), double(d{1}.y))));
printf("x3: %d, y3= %d, z3 = %d \n", double(d{3}.x), double(d{3}.y),
double(pol_f(double(d{3}.x), double(d{2}.y))));
printf("x4: %d, y4= %d, z4 = %d \n", double(d{4}.x), double(d{4}.y),
double(pol_f(double(d{4}.x), double(d{3}.y))));
```

```
Critical points:

x1: 0.346324, y1= -0.299694, z1 = -3.70924

x2: 0.823452, y2= 0.502534, z2 = -1.51329

x3: -0.458339, y3= 0.924071, z3 = 2.39568

x4: -0.874078, y4= 0.292158, z4 = 1.66912
```

iii)

```
# Second order derivatives
dxx = 18*x - 14*y + 6
dyy = 2*x - 18*y +6
dxy = -14*x +2*y -1
dyx = -14*x +2*y -1

function res = dxx (x,y)
  res = 18*x - 14*y + 6;
endfunction

function res = dxy (x,y)
  res = -14*x +2*y -1;
endfunction

function res = dyx (x,y)
  res = -14*x +2*y -1;
endfunction

function res = dyy (x,y)
```

```
  res = 2*x - 18*y +6;
endfunction



# Determine whether critical points correspond to a maximum,minimum or a saddle
point.
for i = 1:4,
  # Hessian matrix
  H = [dxx(x(i),y(i)) dxy(x(i),y(i)); dyx(x(i),y(i)) dyy(x(i),y(i))];
  # Eigen values of Hessian matrix
  eig_H = eig(H);

  printf("x: %d, y= %d, z = %d \n", double(x(i)) ,double(y(i)) ,double(z(i)));
  if eig_H(1) > 0 && eig_H(2) > 0,
    disp("Local minimum point\n");
  elseif eig_H(1) < 0 && eig_H(2) < 0,
    disp("Local Maximum point\n");
  else
    disp("Saddle point\n");
  end;
end;
```

```
x: 0.346324, y= -0.299694, z = -3.70924

Saddle point

x: 0.823452, y= 0.502534, z = -1.51329

Saddle point

x: -0.458339, y= 0.924071, z = 2.39568

Saddle point

x: -0.874078, y= 0.292158, z = 1.66912

Saddle point
```

X	Y	Z	Type
0.346324	-0.299694	-3.70924	Saddle point
0.823452	0.502534	-1.51329	Saddle point
-0.458339	0.924071	2.39568	Saddle point
-0.874078	0.292158	1.66912	Saddle point