p.①

Assignment-1

MFML

Name: Peyala Samarasimha Reddy
BITSID: 2023AA05072
Section: 1

Solutions:-

Q1.

a) Code for REF and RREF

```
import numpy as np
# Function to move rows with all zeros to bottom
def Move_rows_with_all_zeros(matrix):
    num_rows = len(matrix)
    num_cols = len(matrix[0])
    # Separate rows with all zeros and non-zero rows
    all_zero_rows = [row for row in matrix if all(element == 0
                        for element in row[:-1])]
    non_zero_rows = [row for row in matrix if any(element != 0
                        for element in row[:-1])]
    # Keep all zero rows at bottom
    result_matrix = non_zero rows + all_zero_rows
    return result_matrix

# Function to swap the rows in row operations
def Swap_rows(matrix, i, j):
    # Swap rows i and j in matrix
    matrix[i], matrix[j] = matrix[j], matrix[i]

# Function to scale rows in row operations
def Scale_row(matrix, i, scale):
    matrix[i] = [entry * scale for entry in matrix[i]]
```

```python
# function to add scaled row in row operations
def Add_scaled_row(matrix, i, j, scale):
    matrix[i] = [entry_i + scale * entry_j for entry_i, entry_j
                 in zip(matrix[i], matrix[j])]

# Function to get the Row echelon form of given A, B matrices
def Get_Row_Echelon_Form(A, B):

    # Combine A, B to get augmeted AB matrix
    AB = [ ]
    for i in range(len(A)):
        row_AB = list(A[i]) + list(B[i])
        AB.append(row_AB)
    matrix = np.array(AB, dtype = np.float64)
    num_rows, num_cols = matrix.shape

    # Perform row operations to get row echelon form
    for i in range(min(num_rows, num_cols - 1)):
        # find first nonzero row and scale it to have a
                                        leading 1
        nonzero_row = next((row for row in range(i, num_rows)
                           if matrix[row][i] != 0), None)
        if nonzero_row is not None:
            swap_rows(matrix, i, nonzero_row)
            pivot_value = matrix[i][i]
            # check for divison by zero
            if pivot_value == 0:
                print("Error: Divison by zero; please
                      enter a different input matrix")
                return matrix
```

```
            scale_row (matrix, i, 1/pivot_value)
            # Eliminate other entries in current column
            for j in range(i+1, num_rows):
                add_scaled_row (matrix, j, i, -matrix[j][i])
    matrix = Move_rows_with_all_zeros (matrix)
    return matrix

# Function to get the Reduced Row echelon Form of given A, B matrices
def Get_Reduced_Row_Echelon_Form(A, B):
    matrix = np.array (Get_Row_Echelon_Form (A,B))
    num_rows, num_cols = matrix.shape
    # make every pivot element to 1
    for i in range (num_rows):
        pivot_col = next((col for col in range(num_cols -1) if
                          matrix[i][col] != 0), None)

        if pivot_col is not None:
            pivot_val = matrix[i][pivot_col]
            scale_row(matrix, i, 1/pivot_val)
            # Make pivot element is only non zero entry in
                                                its column
            for j in range (num_rows):
                if j != 1:
                    add_scaled_row (matrix, j, i,
                                    -matrix[j][pivot_col])

    return matrix
```

Q1. (P. 4)

b) # function to get pivot and non-pivot columns

```
def Get_pivot_columns(matrix):
    num_rows = len(matrix)
    num_cols = len(matrix[0])

    pivot_columns = []
    nonpivot_columns = []
    for row in matrix:
        found_pivot = False
        for col_index, value in enumerate(row[:-1]):
            if value != 0 and not found_pivot:
                pivot_columns.append(col_index)
                found_pivot = True
            elif value != 0 and found_pivot:
                non_pivot_columns.append(col_index)

    # Remove duplicates
    pivot_columns = list(set(pivot_columns))
    nonpivot_columns = list(set(nonpivot_columns))
    return pivot_columns, nonpivot_columns

# function to find particular solution
def Find_particular_solution(row_echelon_form):
    pivot_columns, _ = Get_pivot_nonpivot_columns
                                        (row_echelon_form)

    particular_solution = [0] * len(pivot_columns)
```

(P.5)

fun

def f

```
for i in range (~~last~~ len(pivot_columns)):
    col_index = pivot_column[i]
    pivot_row = next((row_index ~~for~~ for row_index,
            value in enumerate(row_echelon_form) if
                value [col_index] != 0), None)
    if pivotrow is not None:
        particular_solution[i] = row_echelon_form[pivot_row]
                                                    [-1]
variable_names = [f'x{i+1}' for in in pivot_columns]
return dict (zip(variable_names, particular_solution))

# function to find general solution
def find_general_solution (row_echelon_form):
    _, non_pivot_columns = get_pivot_nonpivot_columns(
                                        row_echelon_form)
    num_variables = len(row_echelon_form[0]) - 1
    general_solution_coefficients = []
    for col_index in nonpivot_~~cto~~columns :
        coefficients = [0] * (num_variables)
        pivot_columns_left = [i for i in range(col_index) if
                        i not in nonpivot columns]
        for pivot_col_index in pivot_columns_left :
            pivot_row = next((row_index for row_index,
                value in enumerate(row_echelon_form) if
            value [pivot_col_index] != 0), None)
        coefficients [col_index] = 1
        general_solution_coefficients.append (coefficients)
```

P.6

~~return~~

return np.array(general_solution_coefficients)

#Function to get ~~gen~~ solutions to a given A,B matrices

```
def Find_solutions_for_linear_System(AB):
    pivot_columns = Get_pivot_columns(AB)[0]
    Non_Pivot_columns = Get_pivot_columns(AB)[1]
    print("Pivot cols :", pivot_columns)
    print(" Nonpivot cols :", Non_Pivot_columns)

    #calculate Rank for A and AB
    A = [row[:-1] for row in AB]
    B = [row[:-1] for row in AB]

    rank_A = sum(1 for row in A if any(element != 0 for
                                         element in row))
    rank_AB = sum(1 for row in AB if any(element != 0 for
                                          element in row))

    # check for consistency for system.
    if rank_A < rank_AB:
        print("inconsistent")
    elif rank_A == rank_AB
        print(" System is consistent")
        freevariables_count = len(Non_Pivot_columns)
        if (rank_A == freevariables_count):
            print(" System has unique sol")
            ~~print(Particular~~lar
```

P.7

Q1. c

```
particular_solution = Find_particular_solution(AB)
print ("The particular solution is:", Particular_solution)
else:
    print ("The system has infinitely many solutions:")
    particular_solution = Find_particular_solution(AB)
    print ("The particular solution is:", Particular_solution)
    General_solution = Find_general_solution(AB)
    print ("The General solution is:", General_solution)
```

Q1. c) Using Random 5x7 matrix for A and random B vector

```
A = np.random.rand(5,7)
B = np.random.rand(5,1)
```

generate REF

```
Get_Row_Echelon_Form(A,B)
Get_Reduced_Row_Echelon_Form(A,B)
Find_solution_For_linear_system(Get_Reduced_Row_Echelon_Form(A,B))
```

Output:

input 5x7, Matrix A = $\begin{bmatrix} 0.37 & 0.63 & 0.63 & 0.54 & 0.07 & 0.84 & 0.32 \\ 0.19 & 0.04 & 0.59 & 0.68 & 0.02 & 0.51 & 0.23 \\ 0.65 & 0.17 & 0.69 & 0.79 & 0.94 & 0.14 & 0.34 \\ 0.11 & 0.92 & 0.88 & 0.26 & 0.66 & 0.82 & 0.56 \\ 0.53 & 0.24 & 0.09 & 0.9 & 0.9 & 0.63 & 0.34 \end{bmatrix}$

Matrix B = $\begin{bmatrix} 0.35 \\ 0.73 \\ 0.9 \\ 0.89 \\ 0.78 \end{bmatrix}$

(P.8)

Row Echelon form :-

~~REF(AB) = [1 1.010 0.968 0.393 0.021]~~

REF(AB) =

$$\begin{bmatrix} 1 & 1.010 & 0.968 & 0.393 & 0.021 & 0.989 & 0.457 & 0.531 \\ -5.463 & 1.0 & 4.82 & 4.518 & -1.817 & 5.59 & 6.089 & -1.397 \\ -2.321 & -5.99 & 1.00 & 1.210 & -8.234 & 2.33 & 1.96 & -6.46 \\ 9.164 & 2.588 & 0 & 1.0 & 3.64 & -7.97 & -3.11 & 2.824 \\ 2.49 & 4.84 & 0. & 0 & 1.0 & -9.005 & -1.433 & 7.788 \end{bmatrix}$$

RREF(AB) =

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 2.83 & -0.37 & 0.15 \\ 0 & 1 & 0 & 0 & 0 & -8.50 & 1.69 & 0.28 \\ 0 & 0 & 1 & 0 & 0 & 9.69 & -1.77 & 0.14 \\ 0 & 0 & 0 & 1 & 0 & -6.69 & 2.11 & -0.164 \\ 0 & 0 & 0 & 0 & 1 & -0.09 & -0.143 & 0.77 \end{bmatrix}$$

Solution of System :-

Pivot columns are [0, 1, 2, 3, 4] ← indexes of columns

Non-pivot are [5, 6]

The system is consistent

The system has many solutions,

The particular solution is,

$x_1 = 0.1548$, $x_2 = 0.281$, $x_3 = 0.145$, $x_4 = -0.164$

$x_5 = 0.778$

(P9) The general solution is,

$$\begin{bmatrix} [-2.830 & 8.502 & -9.696 & 8.690 & 0.090 & 1.0 & 0] \\ [0.373 & -1.698 & 1.77 & -2.111 & 0.143 & 0 & 1] \end{bmatrix}$$

Q2.

a) Matrix Decomposition : A = LU

```
# function to find matrix decomposition
def Matrix_Decomposition_LU(A):
    n = len(A)
    L = [[0] * n for _ in range(n)]
    U = [[0] * n for _ in range(n)]
    for i in range(n):
        # upper triangular matrix:
        for k in range(i,n):
            sum = sum (L[i][p] * U[p][k] for p in range(i))
            U[i][k] = A[i][k] - sum

        # lower triangular matrix
        for k in range(i,n):
            for i == k:
                L[i][i] = 1   # Diagonal is 1
            else:
                sum = sum(L[k][p] * U[p][i] for p
                          in range(i))
                L[k][i] = (A[k][i] - sum) / U[i][i]

# verify A=LU
A_reconstructured = [[sum(L[i][p] * U[p][k] for p in
```

```
range (len L[0])) for k in range (len U[0])) for
    i in range (len(L))]
print ("reconstructed Matrix A from LU")
for row in A-reconstructed:
    print (row)
```

Q2.

b) Cholesky's decomposition, generate L & L^T from A and $A = LL^T$

```
import numpy as np
def matrix_multiply (A,B):
    rows_A, cols_A = len(A) len(A[0])
    rows_B, cols_B = len(B), len(B[0])
    result = [[0] * cols_B for _ in range (rows_A)]
    for in in range (rows_A):
        for j in range (cols_B):
            for k in range (cols_A):
                result [i][j] += A[i][k] * B [k][j]
    ~~result res~~
    return result

def transpose_matrix (matrix):
    if not all (len(rows) == len (matrix) for row in matrix):
        print (" Error, not square matrix).
        return
    if not all (matrix[i][j] == matrix [j][i] for in in
```


n= len(

L=np.

#Get

for

#C

L

#

#

if(

p!

```
range (len (matrix)) for j in range(len(matrix)):
    print ("Error: not symmetrix matrix"):
    return

n= len(matrix)
L=np.zeros((n,n))

#Get low triangular matrix.
for i in range (n):
    for j in range (i+1):
        sum_val = sum (L[i][k] * L[j][k] for k in
                       range (j))

        if i==j:
            L[i][j] = np.sqrt(matrix[i][i] - sum_val)

        else:
            L[i][j]= (1.0/ L[j][j] * matrix[i][j] -
                      sum_val))

#Get transpose of lower triangular matrix
LT= transpose_matrix (L)   # L is generated lower Dter matrix

#get reconstructed matrix by L*LT

reconstructered_A = matrix_multiply (L, LT)

#verify the input matrix is equal to reconstructed
     matrix as   A= LLT

if(np.array_equal (matrix, reconstructed_A):
    print ("Yes, it satisfies A=LLT")
```

Q2

c) A = QR decomposition using Gram-Schmidt decomposition generating Q and R.

```
def ~~Ga~~ QR_decomposition(A):
    m,n = A.shape
    # initialize Q and R matrices
    Q = np.zeros((m,n))
    R = np.zeros((n,n))
    for j in range(n):
        # orthogonalization
        v = A[:, j].copy()
        for i in range(j):
            R[i, j] = matrix_multiply(
                         Q[:,i], A[:,j])
            # using defind function previously
            v -= R[i,j] * Q[:,i]
        #Normalization
        norm_v = np.linalg.norm(v)
        if norm_v < 1e-8:
            # handing the case where v is
              very close to a zero vector
            Q[:, j] = 0.0
        else:
            Q[:, j] = v / norm_v
```

P.13

Q.2.

d) Ta

im

#

ro

pr

Q

ol

P. 13

R[j, j] = norm_v
A-reconstructed = matrix_multiply (Q, R) # A = Q*R
return Q, R

return Q and R matrices.

Q.2.

d) Taking random 5x4 matrix and decompose into Q and R.

```
import numpy as np.
# generate random 5x4 matrix
random_matrix = np.random.rand(5,4)
print(random_matrix):
```

$$
= \begin{bmatrix} 0.04 & 0.79 & 0.86 & 0.89 \\ 0.12 & 0.89 & 0.78 & 0.59 \\ 0.33 & 0.91 & 0.55 & 0.24 \\ 0.1 & 1 & 0.77 & 0.86 \\ 0.66 & 0.74 & 0.87 & 0.16 \end{bmatrix}
$$

QR-~~decom~~decomposition(random_matrix).

o/p:

$$
\text{Matrix } Q = \begin{bmatrix} 0.05 & 0.62 & 0.28 & 0.58 \\ 0.16 & 0.48 & 0.28 & -0.79 \\ 0.44 & 0.23 & -0.85 & -0.02 \\ 0.13 & 0.59 & 0.04 & 0.19 \\ 0.87 & -0.32 & 0.35 & 0.09 \end{bmatrix}
$$

$$
\text{Matrix } R = \begin{bmatrix} 0.76 & 1.36 & 1.26 & 0.65 \\ 0 & 1.39 & 1.01 & 1.05 \\ 0 & 0 & 0.27 & 0.3 \\ 0 & 0 & 0 & 0.23 \end{bmatrix}
$$

reviously

P.14

Reconstructed matrix ~~B~~ A from QR:

$$A = QR$$

$$\begin{bmatrix} 0.04 & 0.79 & 0.66 & 0.89 \\ 0.12 & 0.89 & 0.76 & 0.69 \\ 0.33 & 0.901 & 0.35 & 0.24 \\ 0.1 & 1 & 0.77 & 0.86 \\ 0.66 & 0.74 & 0.87 & 0.16 \end{bmatrix}$$

So, $A = QR$ satisfies //

$\Rightarrow$ Observation of R's diagonal elements are:

1) All diagonal elements are positive
2) The magnitude of diagonal elements of R giving the scaling factors during QR decomposition
3) The diagonal elements of R represent the norms of ~~o~~ orthogonalized vectors.