MAT 167 - Project 1

Emily Ng, Honghui Li, Steve Sliva, Xinran Jiang, Yuchen Liu

August 2023

Problem 1

Matrices can be partitioned into several submatrices known as "blocks." Using a blocked approach, matrix-matrix multiplication can be performed with a divide and conquer approach. Supposed we have two matrices,

a) Show that C can be written as

$$C = \begin{pmatrix} M_1 + M_4 + M_5 - M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{pmatrix}$$

Given:

$$M1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M2 = (A_{21} + A_{22})B_{11}$$
$$M3 = A_{11}(B_{12} - B_{22})$$
$$M4 = A_{22}(B_{21} - B_{11})$$
$$M5 = (A_{11} + A_{12})B_{22}$$
$$M6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$M7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

This works for any even shape matrix, so no matter if the A_{xy} is a number or sub-matrix, it follows the rule that we have $A_{xy}(A_{ab} + A_{cd}) = A_{xy} \cdot A_{ab} + A_{xy} \cdot A_{cd}$ (Distributive Property)
1-1) for the first column and first row
$M_1 + M_4 + M_5 - M_7 =$
$A_{11}B_{22}$ + ~~$A_{22}B_{11}$~~ + ~~$A_{22}B_{22}$~~ - ~~$A_{22}B_{21}$~~ - ~~$A_{22}B_{11}$~~ + ~~$A_{11}B_{22}$~~ + ~~$A_{12}B_{22}$~~ + $A_{12}B_{21}$ + ~~$B_{22}A_{12}$~~ - ~~$A_{22}B_{21}$~~ - ~~$A_{22}B_{22}$~~
$= A_{11}B_{11} + A_{12}B_{21}$

1-2) for the first column and second row
$M3 + M5 =$
$A_{11}B_{12}$ - ~~$A_{11}B_{22} + A_{11}B_{22}$~~ $+A_{12}B_{22}$
$= A_{11}B_{12} + A_{12}B_{22}$

2-1) for the second column and first row
$M2 + M4 =$
$A_{21}B_{11}$+~~$A_{22}B11$~~+$A_{22}B_{21}$-~~$A_{22}B_{11}$~~
$= A_{21}B_{11} + A_{22}B_{21}$

2-2) for the second column and second row
$M1 - M2 + M3 + M6 =$
~~$A_{11}B_{22}$~~ + ~~$A_{22}B_{11}$~~ + $A_{22}B_{22}$ - ~~$A_{22}B_{21}$~~-~~$A_{21}B_{11}$~~-~~$A_{22}B11$~~+~~$A_{11}B_{12}$~~ - ~~$A_{11}B_{22}$~~+~~$A_{21}B_{11}$~~+$A_{21}B_{12}$-~~$A_{11}B_{11}$~~-~~$A_{11}B_{12}$~~
$=A_{21}B_{12}+A_{22}B_{22}$
So we can have:

$$C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

b) Implement this method of blocked matrix multiplication and verify your code works by comparing it to the results of your programming language's built-in matrix product routine.

```python
import numpy as np

def matrix_multiply(A, B):
    A11, A12 = A[0]
    A21, A22 = A[1]
    B11, B12 = B[0]
    B21, B22 = B[1]

    M1 = (A11 + A22) * (B11 + B22)
    M2 = (A21 + A22) * B11
    M3 = A11 * (B12 - B22)
    M4 = A22 * (B21 - B11)
    M5 = (A11 + A12) * B22
    M6 = (A21 - A11) * (B11 + B12)
    M7 = (A12 - A22) * (B21 + B22)

    C11 = M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6

    result = np.array([[C11, C12], [C21, C22]])
    return result

# Test the function with example matrices A and B
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
C = matrix_multiply(A, B)

print("Matrix A:")
print(A)
print("Matrix B:")
print(B)
print("Resultant Matrix C:")
print(C)
np.dot(A, B)
```

Listing 1: Python code For question 1b

By this code it should be able to test this function and compare with the built-in function of 'numpy' The out put is

```
Matrix A:
[[1 2]
 [3 4]]
Matrix B:
[[5 6]
 [7 8]]
Resultant Matrix C:
[[19 22]
 [43 50]]
array([[19, 22],
       [43, 50]])
```

C) Implement this method recursively such that it keeps subdividing the problem until the blocks of A and B are just scalars

```python
import numpy as np

def matrix_multiply_recursive(A, B):
    if len(A) == 1:
        return A * B

    n = len(A)
    half_n = n // 2

    A11 = A[:half_n, :half_n]
    A12 = A[:half_n, half_n:]
    A21 = A[half_n:, :half_n]
    A22 = A[half_n:, half_n:]

    B11 = B[:half_n, :half_n]
    B12 = B[:half_n, half_n:]
    B21 = B[half_n:, :half_n]
    B22 = B[half_n:, half_n:]

    M1 = matrix_multiply_recursive(A11 + A22, B11 + B22)
    M2 = matrix_multiply_recursive(A21 + A22, B11)
    M3 = matrix_multiply_recursive(A11, B12 - B22)
    M4 = matrix_multiply_recursive(A22, B21 - B11)
    M5 = matrix_multiply_recursive(A11 + A12, B22)
    M6 = matrix_multiply_recursive(A21 - A11, B11 + B12)
    M7 = matrix_multiply_recursive(A12 - A22, B21 + B22)

    C11 = M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6

    C = np.vstack((np.hstack((C11, C12)), np.hstack((C21, C22))))
    return C

# Test the function with example matrices A and B
A = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]])
B = np.array([[16, 15, 14, 13], [12, 11, 10, 9], [8, 7, 6, 5], [4, 3, 2, 1]])
C = matrix_multiply_recursive(A, B)

print("Matrix A:")
print(A)
print("Matrix B:")
print(B)
print("Resultant Matrix C:")
print(C)
# Perform matrix multiplication using NumPy's built-in function for comparison
C_numpy = np.dot(A, B)
print(C_numpy)
# Check if the results are close
print("Are the results equal?", np.allclose(C, C_numpy))
```

Listing 2: Python code For question 1c

This code will allows recursive run the same process until they converge into 2x2 matrix,
Then we can use a random matrix A and random matrix b to test whether if the function will gives the same answer as the build in function for numpy package
By running the code above we get the output as:

```
Matrix A:
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
```

```
Matrix B:
[[16 15 14 13]
 [12 11 10  9]
 [ 8  7  6  5]
 [ 4  3  2  1]]
Resultant Matrix C:
[[ 80  70  60  50]
 [240 214 188 162]
 [400 358 316 274]
 [560 502 444 386]]
[[ 80  70  60  50]
 [240 214 188 162]
 [400 358 316 274]
 [560 502 444 386]]
Are the results equal? True
```

It shows out output is identical as the build in package , which means that our method works as we expected
We can actually try a larger 64x64 matrix,

```python
# Generate random 64x64 matrices A and B
A = np.random.randint(1, 10, (64, 64))
B = np.random.randint(1, 10, (64, 64))

# Perform matrix multiplication using the recursive function
C_recursive = matrix_multiply_recursive(A, B)

# Perform matrix multiplication using NumPy's built-in function for comparison
C_numpy = np.dot(A, B)

# Check if the results are close
print("Are the results equal?", np.allclose(C_recursive, C_numpy))
```

```
Are the results equal? True
```

D) Numerically compare the asymptotic work for this method and naive matrix multiplication. Here is an exmaple code snippet of very naive matrix multiplication

```python
import numpy as np

def matrix_multiply_naive(A, B):
    if A.shape[1] != B.shape[0]:
        raise ValueError("Number of columns in matrix A must be equal")

    result = np.zeros((A.shape[0], B.shape[1]))

    for i in range(A.shape[0]):
        for j in range(B.shape[1]):
            for k in range(A.shape[1]):
                result[i, j] += A[i, k] * B[k, j]

    return result
```

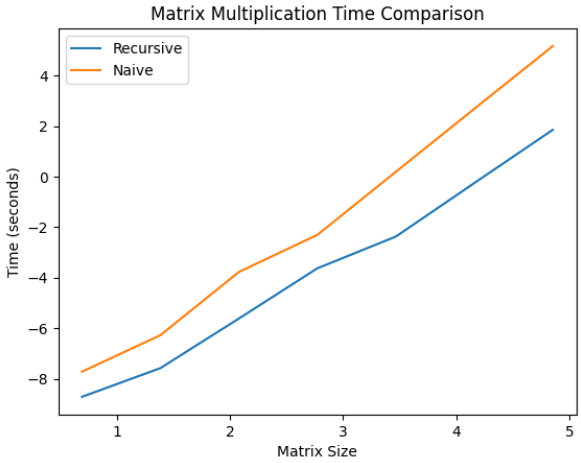

Figure 1: Time of matrix mutiple by two methods

```python
import time
import matplotlib.pyplot as plt
# Compare execution times for different matrix sizes
sizes = [2, 4, 8, 16, 32, 64, 128]
recursive_times = []
naive_times = []

for size in sizes:
    A = np.random.rand(size, size)
    B = np.random.rand(size, size)

    start_time = time.time()
    matrix_multiply_recursive(A, B)
    end_time = time.time()
    recursive_times.append(end_time - start_time)

    start_time = time.time()
    matrix_multiply_naive(A, B)
    end_time = time.time()
    naive_times.append(end_time - start_time)

# Take the natural logarithm of the execution times
ln_recursive_times = np.log(recursive_times)
ln_naive_times = np.log(naive_times)
in_size=np.log(sizes)
# Plot the results
plt.plot(in_size, ln_recursive_times, label='Recursive')
plt.plot(in_size, ln_naive_times, label='Naive')
plt.xlabel('Matrix Size')
plt.ylabel('Time (seconds)')
plt.title('Matrix Multiplication Time Comparison')
plt.legend()
plt.show()
# Estimate the slope using linear regression (numpy.polyfit)
slope_recursive, _ = np.polyfit(in_size, ln_recursive_times, 1)
slope_naive, _ = np.polyfit(in_size, ln_naive_times, 1)
print(slope_recursive)
print(slope_naive)
```

we get the slope , which is the big O of the two method equal to

```
2.841099045760702
3.637288115717954
```

Problem 2

In class we saw (or will see) that the classical Gram-Schmidt process for generating an orthonormal basis isn't numerically stable, so the "modified" Gram-Schmidt method is used instead.

a) Write a routine that takes in a matrix, A and returns a set of orthonormal vectors, Q using classical Gram-Schmidt.

```
# CGS
def proj(v1, v2):
    return (numpy.dot(v2, v1) / numpy.dot(v1, v1)) * v1

def normalize(i):
    normalized_vector = i / numpy.linalg.norm(i)
    return normalized_vector

def cgs(A):
    # Q_t is the transpose of final matrix / basis Q
    # store col vectors as row vectors in Q for easy modification
    Q_t = []

    for i in range(len(A)):
        # v_i
        a_i = A[i]
        v_i = A[i]

        # v_pt: previous v's transpose (row vector)
        for v_pt in Q_t:
            # v_p: previous v's  (col vector)
            v_p = numpy.transpose(v_pt)

            proj_vec = proj(v_p, a_i)
            v_i = v_i - proj_vec

        # append current v_i
        Q_t.append(numpy.transpose(v_i))

    # print(Q_t)

    for i in range(len(Q_t)):
        Q_t[i] = normalize(Q_t[i])

    Q = numpy.transpose(Q_t)
    return Q
```

Listing 3: Python code For question 2a

b) Write a routine that takes in a matrix, A and returns a set of orthonormal vectors, Q using modified Gram-Schmidt.

```python
# MGS
import numpy

def proj(v1, v2):
    return (numpy.dot(v2, v1) / numpy.dot(v1, v1)) * v1

def normalize(i):
    normalized_vector = i / numpy.linalg.norm(i)
    return normalized_vector

def mgs(A):
    # Q_t is the transpose of final matrix / basis Q
    # use transpose of Q so each col vector becoms row vector and easier to access
    Q_t = numpy.transpose(A)

    for i in range(len(Q_t)):
        Q_t[i] = normalize(Q_t[i])
        q_i = numpy.transpose(Q_t[i])

        # subtract q_i's axis from rest q_k, k > i
        for k in range(i + 1, len(Q_t)):
            q_k = numpy.transpose(Q_t[k])

            # subtract projection, already normlized
            q_k = q_k - numpy.dot(q_i, q_k) * q_i

            Q_t[k] = numpy.transpose(q_k)

    Q = numpy.transpose(Q_t)
    return Q
```

Listing 4: Python code For question 2b

c) Recall that the generated matrix, Q is a normal matrix, that is, in exact arithmetic we get, $Q^T - Q = I$. Take your two routines and check how accurate this is, i.e., compute $\|Q^TQ - I\|$and see how close it is to zero. Do this for the matrices,

c-1) A = np.random.random((n, n))
c-2) and A = 0.00001 * np.eye(n) + scipy.linalg.hilbert(n) for a few sizes, n.

Result in table format (n ∈ 3, 5, 7, 9):

```
A = np.random.random((n, n))
 n |        cgs result        |       mgs result
-----------------------------------------------------------
 3 |    1.343782689223772e-15 |    5.822058658345461e-16
 5 |    5.079327213521194e-15 |   1.6425403990754258e-15
 7 |     1.29751252117977e-14 |    5.166999309333753e-15
 9 |    9.295267406282773e-14 |   3.2555245376247796e-14


A = 0.00001 * np.eye(n) + scipy.linalg.hilbert(n)
 n |        cgs result        |       mgs result
-----------------------------------------------------------
 3 |    7.080997315971274e-14 |     6.63317609727746e-15
 5 |    3.165790514913122e-08 |    8.512887360371612e-13
 7 |    8.243819382900969e-06 |   2.8665923273128248e-12
 9 |     0.000335415914015717 |   1.3617858663174204e-11
```

Code:

```
# helper function
def print_result(N, Error):
    print(" n |         cgs result         |         mgs result")
    print("-" * 60)

    for i in range(len(Error)):
        print(f"{N[i]:2} | {Error[i][0]:24} | {Error[i][1]:24}")

# main
def main():
    # random matrix, Q_t*Q - I error test
    N = [3, 5, 7, 9]

    Errors_1 = []
    Errors_2 = []

    for n in N:
        # Random matrix of size n
        A = numpy.random.random((n, n))
        # cgs() and mgs() returens result Q matrix
        result_cgs = cgs(A)
        result_mgs = mgs(A)

        # ||QtQ=I|| for cgs
        error_cgs = numpy.linalg.norm( numpy.dot(numpy.transpose(result_cgs),
    result_cgs) - numpy.identity(n))
        # ||QtQ=I|| for mgs
        error_mgs = numpy.linalg.norm( numpy.dot(numpy.transpose(result_mgs),
    result_mgs) - numpy.identity(n))

        Errors_1.append([error_cgs, error_mgs])

        # Random matrix of size n following the given equation
        A = 0.00001 * numpy.eye(n) + scipy.linalg.hilbert(n)
        result_cgs = cgs(A)
        result_mgs = mgs(A)

        # ||QtQ=I|| for cgs
        error_cgs = numpy.linalg.norm( numpy.dot(numpy.transpose(result_cgs),
    result_cgs) - numpy.identity(n))
        # ||QtQ=I|| for mgs
        error_mgs = numpy.linalg.norm( numpy.dot(numpy.transpose(result_mgs),
    result_mgs) - numpy.identity(n))

        Errors_2.append([error_cgs, error_mgs])


    print("A = np.random.random((n, n))")
    print_result(N, Errors_1)
    print("\n\nA = 0.00001 * np.eye(n) + scipy.linalg.hilbert(n) ")
    print_result(N, Errors_2)

# call main function
main()
```

Listing 5: Python code For question 2c

Problem 3

Suppose we have factored A = QR $\in \mathbb{R}^{nxn}$, where Q is unitary, R is upper triangular. Let $u, v \in \mathbb{R}^n$.

a) Derive a method for computing the QR factorization of $\hat{A} = A + uv^T$ that doesn't involve forming $\hat{A}$ and factoring it. *Hint*: multiply on the left by QT .

Let $\hat{A} = A + uv^T$, $A = QR \in \mathbb{R}^{nxn}$, $Q^TQ = I$, and $uv \in \mathbb{R}^n$.

$$\begin{aligned}
\hat{A} &= A + uv^T \\
Q^T\hat{A} &= Q^TA + Q^Tuv^T \\
Q^T\hat{A} &= R + Q^Tuv^T \\
Q^T\hat{A} &= R + (Q^Tu)v^T
\end{aligned}$$

Let $Q^Tu = w$, since Q^Tu is an inner product, then we have w as col-vector (O(n^2) to get w)

$$Q^T\hat{A} = R + wv^T$$

Apply rotations (denoted G_1) to w to rotate from $\begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_n \end{bmatrix}$ to $\begin{bmatrix} ||w|| \\ 0 \\ \dots \\ 0 \end{bmatrix}$ (O(1) to get each rotation in G_1)

$$\begin{aligned}
G_1Q^T\hat{A} &= G_1(R + wv^T) \\
G_1Q^T\hat{A} &= G_1R + G_1wv^T \\
G_1Q^T\hat{A} &= G_1R + ||w||e_1v^T
\end{aligned}$$

(O(n^2) to apply rotation since they reduced to 2n row operation)

G_1R will produce an upper Hessenberg matrix.
Now let us apply n times Givens rotations (denoted G_2) to the G_1R to ensure that it's upper triangular.

$$\begin{aligned}
\hat{G_1}(Q^T\hat{A}) &= \hat{G_1}R + ||w||e_1v^T \\
\hat{G_2}(\hat{G_1}(Q^T\hat{A})) &= \hat{G_2}(\hat{G_1}R + ||w||e_1v^T) \\
(\hat{G_2}\hat{G_1}Q^T)\hat{A} &= \hat{G_2}(\hat{G_1}R + ||w||e_1v^T) \\
\hat{Q}^T\hat{A} &= \hat{R} \\
\hat{A} &= \hat{Q}\hat{R}
\end{aligned}$$

(O(n^2) to apply rotation since they reduced to 2n row operation)

From above we get:

$$\begin{aligned}
\hat{Q}^T &= \hat{G_2}\hat{G}1Q^T \\
\hat{Q} &= (\hat{G_2}\hat{G}1Q^T)^T \\
\hat{R} &= \hat{G_2}(\hat{G_1}R + ||w||e_1v^T)
\end{aligned}$$

From the above analyze, this method will have overall complexity of O(n^2) witch is less than the naive approach with O(n^3)

b) Write a routine that does this and verify that it works by comparing it to naively computing a new QR of $\hat{A}$. Note that you might be off by a sign. This is fine because a QR decomposition is not unique. Also don't worry about having the most optimized code, just explain in (a) why the update is faster.

Code:

```
import numpy as np
import math


# ================ helper functions ================
def random_invertible_matrix(rank, min_val = -20, max_val = 20):
    while True:
        # Generate a random matrix of size n x n
        random_matrix = np.random.randint(min_val, max_val + 1, size=(rank, rank))

        # Check if the matrix is invertible
        if np.linalg.matrix_rank(random_matrix) == rank:
            return random_matrix

def random_vector(rank, min_val = -20, max_val = 20):
    return np.random.randint(-20, 20 + 1, size=(rank, 1)), np.random.randint(-20, 20
     + 1, size=(rank, 1))

def apply_rotation(matrix, cos, sin, axis_pos):
    # axis_pos used zero-outed row position
    row_a = matrix[axis_pos - 1].copy()
    row_b = matrix[axis_pos].copy()

    # calculate new rows
    # |  C S |
    # | -S C | for rotating clockwise ans zero out 2nd term

    matrix[axis_pos - 1] = cos * row_a + sin * row_b
    matrix[axis_pos] = - sin * row_a + cos * row_b


def apply_givens_rotation_zero_out(matrix, row, col):
    a = matrix[row - 1][col]
    b = matrix[row][col]
    r = math.sqrt(a*a + b*b)

    cos = a/r
    sin = b/r

    row_a = matrix[row - 1].copy()
    row_b = matrix[row].copy()

    matrix[row - 1] = cos * row_a + sin * row_b
    matrix[row] = - sin * row_a + cos * row_b

    return cos, sin


# ================ generate matrix and vector ================
rank = 4
A = random_invertible_matrix(rank)
u, v = random_vector(rank)
v_t = np.transpose(v)
print("Random Matrix A:")
print(A)
print()

# Get original QR decom.
Q, R = np.linalg.qr(A)


# ================ main solution body ================
```

```
# calculate (Q^T)u as w
Q_T = np.transpose(Q)
w = np.matmul(Q_T, u)

# part 1 rotation, rotate u to be [[r] [0] ... [0]]
rank = 4
Q_ht = Q_T.copy()
R_h = R.copy()
for i in range(rank - 1, 0, -1):
    # i is the row position of the term currently zeroing out
    a = w[i - 1][0]
    b = w[i][0]
    r = math.sqrt(a*a + b*b)

    cos = a/r
    sin = b/r

    # apply G (rotation matrix) to w (making sure rotation is correct
    # apply_rotation(w, cos, sin, i)
    # print(np.transpose(w)), tested working

    # rotate w
    w[i - 1][0] = r
    w[i][0] = 0

    # apply G (rotation matrix) to R_hat
    apply_rotation(R_h, cos, sin, i)
    apply_rotation(Q_ht, cos, sin, i)

print("G_1*R Will be upper hesssenberg")
print(R_h)
print()

# add norm(w)e1v_t to R_h
e1 = np.zeros((rank, 1))
e1[0][0] = 1
R_h = R_h + np.matmul(w[0][0] * e1, v_t)

# part 2 rotation, zero out the R_ht to be upper triangular
for i in range(1, rank):
    # zero out one term in R_h and save the rotation parameter c,s
    cos, sin = apply_givens_rotation_zero_out(R_h, i, i-1)

    # apply same rotation to Q_ht
    apply_rotation(Q_ht, cos, sin, i)


Q_h = np.transpose(Q_ht)

# ================ check result ================
# print result
print("Result:")
print("Q_hat")
print(Q_h)
print("R_hat")
print(R_h)

A_hat_result = np.matmul(Q_h, R_h)

A_hat = A + np.matmul(u, v_t)

print()
print("Original A_hat = A + u*v_t")
print(A_hat)
print("Re-composed A_hat = Q_hat * R_hat")
print(A_hat_result)
```

Listing 6: Python code For question 3b

Output:

```
Random Matrix A:
[[-12  -9 -13 -15]
 [-12 -17   0   9]
 [ -8  10  -1  11]
 [ 13 -11  19  13]]

G_1*R Will be upper hesssenberg
[[ 18.41706898  -0.93508297   8.740993    -7.19607331]
 [-13.48375208  -7.87773959 -18.54205917 -21.17589946]
 [ -0.         -22.97970493  10.43947085   9.77154179]
 [  0.           0.           1.34332851  -0.56105418]]

Result:
Q_hat
[[ 0.05910791  0.00967877 -0.75042688  0.65823406]
 [ 0.56574718 -0.69099139 -0.2760349  -0.35533895]
 [ 0.74307093  0.19514135  0.45552741  0.44973351]
 [-0.35253649 -0.69595767  0.39135535  0.48805935]]
R_hat
[[ 4.73709827e+02  4.19134223e+02 -1.55730356e+02 -6.59644327e+01]
 [-0.00000000e+00  2.33345973e+01 -1.42740098e+01 -1.36299233e+01]
 [ 0.00000000e+00  0.00000000e+00  2.08640562e+01  2.09346515e+01]
 [ 0.00000000e+00  0.00000000e+00 -2.22044605e-16 -1.91289693e+00]]

Original A_hat = A + u*v_t
[[  28   25  -25  -21]
 [ 268  221  -84  -33]
 [ 352  316 -109  -43]
 [-167 -164   73   40]]
Re-composed A_hat = Q_hat * R_hat
[[  28.   25.  -25.  -21.]
 [ 268.  221.  -84.  -33.]
 [ 352.  316. -109.  -43.]
 [-167. -164.   73.   40.]]
```

Problem 4

The nth Legendre polynomial $P_n(x)$ is given by:

$$P_n(x) = \frac{1}{2^n} \sum_{k=0}^{\lfloor n/2 \rfloor} (-1)^k \binom{n}{k} \left(x^{n-2k}\right)$$

a) Recall the classical way to compute the eigenvalues of a matrix is to look at the roots of the characteristic polynomial. Find a matrix who's characteristic polynomial is the n-th Legendre polynomial. Hint: The matrix should be upper Hessenberg.

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & \dots & 0 & -c_1 \\ 1 & 0 & 0 & 0 & \dots & 0 & -c_2 \\ 0 & 1 & 0 & 0 & \dots & 0 & -c_3 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 & -c_{n-1} \\ 0 & 0 & 0 & 0 & \dots & 1 & -c_n \end{bmatrix}$$

where c_n is the n^{th} terms of the Legendre polynomial
here we took the first 5^{th} Legendre polynomial as example
The first five Legendre polynomials are given by:

$$\begin{aligned} P_0(x) &= 1 \\ P_1(x) &= x \\ P_2(x) &= \frac{1}{2}(3x^2 - 1) \\ P_3(x) &= \frac{1}{2}(5x^3 - 3x) \\ P_4(x) &= \frac{1}{8}(35x^4 - 30x^2 + 3) \\ P_5(x) &= \frac{1}{8}(63x^5 - 70x^3 + 15x) \end{aligned}$$

where for **2x2 matrix** there are$P_2(x) = \frac{1}{2}(3x^2 - 1) \propto x^2 - \frac{2}{3}$

$$A_{2\times 2} = \begin{bmatrix} 0 & 1 \\ \frac{2}{3} & 0 \end{bmatrix}$$

For 3x3 matrix there are $P_3(x) = \frac{1}{2}(5x^3 - 3x) \propto x^3 - \frac{6}{5}$

$$A_{3\times 3} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & \frac{6}{5} & 0 \end{bmatrix}$$

For 4x4 matrix there are $P_4(x) = \frac{1}{8}(35x^4 - 30x^2 + 3) \propto x^4 - \frac{48}{7}x^2\frac{24}{35}$

$$A_{4\times 4} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{48}{7} & 0 & \frac{24}{35} \end{bmatrix}$$

For 5x5 matrix there are $P_5(x) = \frac{1}{8}(63x^5 - 70x^3 + 15x) \propto x^5 - \frac{560}{63}x^3 + \frac{40}{21}x$

$$A_{5\times 5} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & \frac{40}{21} & 0 & \frac{560}{63} & 0 \end{pmatrix}$$

During our reseach, we found ther are another idea presented by (Golub & Welsch, 1969b), where Any set of orthogonal polynomials, pj(.x)¥-i, satisfies a three term recurrence relationship :

$$P_i(x) = (a_i x + b_j)P_{j-i}(x) - C_j P_{j-2}(x)$$

where

$$j = 1, 2, \ldots n; p - i(x) = 0, p0(x) = l,$$

then we have

$$xp(x) = Tp(x) + \frac{l}{a_N} p_N(x) e_N$$

Where T represents the tridiagonal matrix and $\mathbf{e}^T = (0, 0, \ldots, 0, 1)\mathbf{r}$. Therefore, $p_n(\xi) = 0$ if and only if $\xi - p(\xi) = T(\xi)$ where ξ is an eigenvalue of the tridiagonal matrix T. In reference [12], it is demonstrated that T becomes symmetric when the polynomials are orthonormal. If T is not symmetric, then we can carry out a diagonal similarity transformation that will result in a symmetric tridiagonal matrix J.

$$J = \begin{bmatrix} a_1 & b_1 & 0 & \cdots & 0 \\ b_1 & a_2 & b_2 & \cdots & 0 \\ 0 & b_2 & a_3 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & b_{n-1} \\ 0 & 0 & \cdots & b_{n-1} & a_n \end{bmatrix}$$

A Jacobi matrix, which is a symmetric tridiagonal matrix, corresponds to the characteristic polynomial that is a monic version (with a leading coefficient of 1) of the specific set of orthogonal polynomials being studied. For instance, if we consider the Legendre polynomials, we can start by noting that the monic Legendre polynomials follow a two-term recurrence relation:

$$\hat{P}_{n+1}(x) = x\hat{P}_n(x) - \frac{n^2}{4n^2 - 1}\hat{P}_{n-1}(x)$$

where$\hat{P}_n(x) = \frac{(n!)^2 2^n}{(2n)!} P_n(x)$ is the monic Legendre polynomial.
From this, we can derive an explicit expression for the corresponding Jacobi matrix (here I give the 5-by-5 case):

$$\begin{pmatrix} 0 & \frac{1}{\sqrt{3}} & 0 & 0 & 0 \\ \frac{1}{\sqrt{3}} & 0 & \frac{2}{\sqrt{15}} & 0 & 0 \\ 0 & \frac{2}{\sqrt{15}} & 0 & \frac{3}{\sqrt{35}} & 0 \\ 0 & 0 & \frac{3}{\sqrt{35}} & 0 & \frac{4}{\sqrt{63}} \\ 0 & 0 & 0 & \frac{4}{\sqrt{63}} & 0 \end{pmatrix}$$

$\frac{n}{\sqrt{4n^2-1}}$ in the (n,n+1) and (n+1,n) positions, and 0 elsewhere.
the eigenvector of these two different kinds of matrix will be somewhat identical to each other

b) Write a routine to compute the eigenvalues of a upper Hessenberg matrix using the shifted QR method. It doesn't matter which shift you use but be wary that the Wilkonson shift because it might temporarily become complex before the method converges.

```
def QR_algorithm_Wilkinson_shift(A):
    """The QR algorithm with Wilkinson shift for finding eigenvalues.

    .. code-block:: matlab

        function [myeig tmvec] = QRwilkinson(A)

        %------- Phase 1 -------
        T = mytridiag(A);

        %------- Phase 2 -------
        m = length(A);
        myeig = zeros(m,1);
        % Size of tmvec is not known in advance.
        % Estimating an initial size.
        tmvec = zeros(m*8,1);
        mu = 0;
        counter = 0;

        while (m > 1)
            counter = counter + 1;
            muMat = diag(mu * ones(m,1));
            [Q,R] = myQR(T - muMat);
            T = R*Q + muMat;
            tmvec(counter) = abs(T(m,m-1));
            delta = (T(m-1,m-1) - T(m,m)) / 2;
            mu = T(m,m) - sign(delta) * T(m,m-1) / ...
                    (abs(delta) + norm([delta T(m,m-1)]));
            if (tmvec(counter) < 1e-8)
                myeig(m) = T(m,m);
                m = m - 1;
                T = T(1:m,1:m);
            end
        end

        myeig(1) = T;
        tmvec = tmvec(1:counter);

        end
```

c) Use your routine from (b) to compute the roots the Legendre polynomials for n = 1, 2, 3, 4, 5. Compare your approximate roots to the actual roots (they can be found easily online).

Here we tried n=5

```python
    # Define the matrix elements
elements = [
    [0, 1/np.sqrt(3), 0, 0, 0],
    [1/np.sqrt(3), 0, 2/np.sqrt(15), 0, 0],
    [0, 2/np.sqrt(15), 0, 3/np.sqrt(35), 0],
    [0, 0, 3/np.sqrt(35), 0, 4/np.sqrt(63)],
    [0, 0, 0, 4/np.sqrt(63), 0]
]

matrix = np.array([
    [0, 1, 0, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 0, 1, 0],
    [0, 0, 0, 0, 1],
    [0, 40/21, 0, 560/63, 0]
])
# Convert the list of lists to a NumPy matrix
A_matrix = np.array(elements)

B_matrix = np.array(elements)
eigenvalues_shifted_qr = QR_algorithm_Wilkinson_shift(A_matrix)
print("Eigenvalues from Shifted QR Method:", eigenvalues_shifted_qr)
eigenvalues_shifted_qr = QR_algorithm_Wilkinson_shift(B_matrix)
print("Eigenvalues from Shifted QR Method:", eigenvalues_shifted_qr)
```

```
Eigenvalues from Shifted QR Method: [-9.06179846e-01 -5.38469310e-01  1.78162900e-17  9.06179846e-01
  5.38469310e-01]
Eigenvalues from Shifted QR Method: [-9.06179846e-01 -5.38469310e-01  1.78162900e-17  9.06179846e-01
  5.38469310e-01]
```

We can compare this with the roots online:

Number of points, n	Points, x_i		Weights, w_i	
1	0		2	
2	$\pm\frac{1}{\sqrt{3}}$	±0.57735...	1	
3	0		$\frac{8}{9}$	0.888889...
	$\pm\sqrt{\frac{3}{5}}$	±0.774597...	$\frac{5}{9}$	0.555556...
4	$\pm\sqrt{\frac{3}{7}-\frac{2}{7}\sqrt{\frac{6}{5}}}$	±0.339981...	$\frac{18+\sqrt{30}}{36}$	0.652145...
	$\pm\sqrt{\frac{3}{7}+\frac{2}{7}\sqrt{\frac{6}{5}}}$	±0.861136...	$\frac{18-\sqrt{30}}{36}$	0.347855...
5	0		$\frac{128}{225}$	0.568889...
	$\pm\frac{1}{3}\sqrt{5-2\sqrt{\frac{10}{7}}}$	±0.538469...	$\frac{322+13\sqrt{70}}{900}$	0.478629...
	$\pm\frac{1}{3}\sqrt{5+2\sqrt{\frac{10}{7}}}$	±0.90618...	$\frac{322-13\sqrt{70}}{900}$	0.236927...

Figure 2: Gauss–Legendre quadrature

References

[1] Hbldh. (n.d.). *b2ac/b2ac/eigenmethods/qr_algorithm.py at master · hbldh/b2ac. GitHub.*
https://github.com/hbldh/b2ac/blob/master/b2ac/eigenmethods/qr_algorithm.py

[2] Wikipedia contributors. (2023). *Gauss–Legendre quadrature. Wikipedia.*
https://en.wikipedia.org/wiki/Gauss%E2%80%93Legendre_quadrature

[3] Golub, G. H., & Welsch, J. H. (1969). Calculation of Gauss quadrature Rules. *Mathematics of Computation, 23*(106), 221.
https://doi.org/10.2307/2004418