1.1 Question 1: Tridiagonal systems. 1(a)

```
function LU-TRI(a, b, c):
    n = length(b)  # Dimension of the tridiagonal matrix

    # Initialize empty vectors l, u, v
    l = zeros(n-1)
    u = zeros(n)
    v = zeros(n-1)

    for k in range(n - 1):
        # Compute the multiplier l[k+1]
        l[k] = a[k] / b[k]

        # Update row of u
        if k == 0:
            u[k] = b[k]
        else:
            u[k] = b[k] - l[k] * c[k-1]  # Subtracting the previous
column

        # Update row of v
        if k == n-2:
            v[k] = c[k]
        else:
            v[k] = c[k] - l[k] * a[k+1]  # Subtracting the next column

    # Compute the last entry of u
    u[n-1] = b[n-1] - l[n-2] * c[n-2]

    return l, u, v
```

n is the dimension of the tridiagonal matrix

Initialize empty vectors l, u, and 3.

Iterate through the tridiagonal matrix columns using the variab . Compute the multiplier l[k5. +5. 1]. Update the row of u based on the tridiagonal matrix me6. nts. Update the row of v based on the tridiagonal matril m7. ents. Compute the lasn. y8. of u. Return the vectors l, u, and v representing the LU factorization.

1(b)

The number of operations C(n) required by the LU-TRI algorithm can be analyzed as follows:

1. In each iteration of the loop (except the last one), there are three main operations:
    - Division to calculate l[k+1] = a[k] / b[k]
    - Subtraction to update the row of u: u[k] = b[k] - l[k] * c[k-1]
    - Subtraction to update the row of v: v[k] = c[k] - l[k] * a[k+1]

2. An additional operation is needed to calculate the last entry of u: u[n-1] = b[n-1] - l[n-2] * c[n-2]

Therefore, the total number of operations can be written as:

C(n) = 3(n-1) + 1 = 3n - 2

Let's compare this with the standard LU factorization for a general dense matrix. The standard LU factorization typically uses Gaussian elimination, which involves forward and backward substitutions. For an n x n matrix, the number of operations is approximately proportional to 2/3 * n^3.

Hence, the standard LU factorization has a complexity of C_standard(n) approximately equal to 2/3 * n^3.

Comparing the two:

C(n) = 3n - 2

C_standard(n) approximately equal to 2/3 * n^3

The LU-TRI algorithm has a linear complexity of O(n), which is significantly lower than the cubic complexity of the standard LU factorization. This makes LU-TRI more efficient for tridiagonal matrices compared to the general LU factorization for dense matrices. The efficiency gain is due to the special structure of tridiagonal matrices, which allows for a more optimized algorithm.re optimized algorithm.

1(c)

```python
import numpy as np

def LUtri(A):
    # Extract diagonal, subdiagonal, and superdiagonal vectors from A
    a = np.diag(A, k=-1)
    b = np.diag(A)
    c = np.diag(A, k=1)

    # Find dimension of the matrix A
    n = len(b)

    # Initialize empty vectors l, u, v
    l = np.zeros(n-1)
    u = np.zeros(n)
    v = np.zeros(n-1)

    for k in range(n - 1):
        # Compute the multiplier l[k+1]
        l[k] = a[k] / b[k]

        # Update row of u
        if k == 0:
            u[k] = b[k]
        else:
            u[k] = b[k] - l[k] * c[k-1]

        # Update row of v
        if k == n-2:
            v[k] = c[k]
        else:
            v[k] = c[k] - l[k] * a[k+1]

    # Compute the last entry of u
    u[n-1] = b[n-1] - l[n-2] * c[n-2]

    return l, u, v

# Given matrix A
A = np.array([[2, 1, 0],
              [-1, 3, 1],
              [0, 2, 1]])

# Apply LUtri function
l, u, v = LUtri(A)

# Display the result
print("l:", l)
print("u:", u)
print("v:", v)
```

```
l: [-0.5         0.66666667]
u: [2.         2.33333333 0.33333333]
v: [2. 1.]
```

This code defines the LUtri function, extracts the diagonal, subdiagonal, and superdiagonal vectors from the input matrix A, applies the LU-TRI algorithm, and displays the resulting vectors l, u, and v.

1(d)

```
import numpy as np
import time

def LU(A):
    n = A.shape[0]
    L = np.eye(n)
    U = np.copy(A)

    for k in range(n - 1):
        for j in range(k + 1, n):
            L[j, k] = U[j, k] / U[k ,k]
            U[j, k:] = U[j, k:] - L[j, k] * U[k, k:]
    return L, U

def LUtri(a, b, c):
    n = len(b)
    l = np.zeros(n-1)
    u = np.zeros(n)
    v = np.zeros(n-1)

    for k in range(n - 1):
        l[k] = a[k] / b[k]

        if k == 0:
            u[k] = b[k]
        else:
            u[k] = b[k] - l[k] * c[k-1]

        if k == n-2:
            v[k] = c[k]
        else:
            v[k] = c[k] - l[k] * a[k+1]

    u[n-1] = b[n-1] - l[n-2] * c[n-2]

    return l, u, v

# Function to measure execution time for LU factorization
def measure_lu_time(A):
    start_time = time.time()
    LU(A)
    return time.time() - start_time

# Function to measure execution time for LU-TRI factorization
def measure_lu_tri_time(a, b, c):
    start_time = time.time()
    LUtri(a, b, c)
    return time.time() - start_time

# Example usage
n = 1000  # Adjust the size of the matrix as needed

# Create a random tridiagonal matrix
a = np.random.rand(n-1)
b = np.random.rand(n)
c = np.random.rand(n-1)
A = np.diag(a, k=-1) + np.diag(b) + np.diag(c, k=1)

# Measure time for LU factorization
lu_time = measure_lu_time(A)

# Measure time for LU-TRI factorization
```

```
lu_tri_time = measure_lu_tri_time(a, b, c)

print(f"Time for LU factorization: {lu_time:.6f} seconds")
print(f"Time for LU-TRI factorization: {lu_tri_time:.6f} seconds")

Time for LU factorization: 4.892790 seconds
Time for LU-TRI factorization: 0.002001 seconds
```

From the above results, we can see that the LU_TRI algorithm is significantly more efficient than the LU factorisation algorithm. This is because the LU_TRI algorithm has a computational cost of O(n), whereas the LU factorisation algorithm has a computational cost of O(n^3). Therefore, for large matrices, the LU_TRI algorithm is the better option in terms of efficiency.

1(e)

```
import numpy as np

def FStri(l, b, r):
    """
    Forward substitution for a tridiagonal matrix.

    Parameters:
    - l: vector representing the subdiagonal of the tridiagonal matrix
    - b: vector representing the main diagonal of the tridiagonal
matrix
    - r: right-hand side vector

    Returns:
    - y: solution vector
    """
    n = len(b)
    y = np.zeros(n)

    y[0] = r[0] / b[0]

    for i in range(1, n):
        y[i] = (r[i] - l[i-1] * y[i-1]) / b[i]

    return y

def BStri(u, v, y):
    """
    Backward substitution for a tridiagonal matrix.

    Parameters:
    - u: vector representing the main diagonal of the tridiagonal
matrix
    - v: vector representing the superdiagonal of the tridiagonal
matrix
    - y: solution vector from forward substitution
    Returns:
    - x: solution vector
    """
    n = len(u)
    x = np.zeros(n)

    x[n-1] = y[n-1] / u[n-1]

    for i in range(n-2, -1, -1):
        x[i] = y[i] - v[i] * x[i+1]

    return x

def GEtri(a, b, c, r):
    """
    Gaussian elimination for solving a tridiagonal system Ax = r.

    Parameters:
    - a: subdiagonal vector
    - b: main diagonal vector
    - c: superdiagonal vector
    - r: right-hand side vector

    Returns:
    - x: solution vector
    """
    n = len(b)

    # Create a copy of the main diagonal vector to avoid read-only
issues
    b_copy = np.copy(b)

    for k in range(1, n):
        factor = a[k-1] / b_copy[k-1]
        b_copy[k] -= factor * c[k-1]
        r[k] -= factor * r[k-1]

    x = np.zeros(n)
    x[n-1] = r[n-1] / b_copy[n-1]

    for k in range(n-2, -1, -1):
        x[k] = (r[k] - c[k] * x[k+1]) / b_copy[k]

    return x

# Given tridiagonal matrix A and right-hand side vector r
A_tri = np.array([[1, 2, 0, 0],
                  [1, 1, -1, 0],
                  [0, 3, -2, 1],
                  [0, 0, 1, 3]])

a_tri = np.diag(A_tri, k=-1)
b_tri = np.diag(A_tri)
c_tri = np.diag(A_tri, k=1)

r_tri = np.array([1, 2, 3, 4])

# Solve the linear system using GEtri
x_solution = GEtri(a_tri, b_tri, c_tri, r_tri)

# Display the result
print("Solution x:")
print(x_solution)
```

```
Solution x:
[ 1.26666667 -0.13333333 -0.86666667  1.66666667]
```

Here, we've defined functions for forward substitution (FStri), backward substitution (BStri), and Gaussian elimination (GEtri). It then applies GEtri to solve the given tridiagonal linear system and prints the result.

Question 2: 2(a)

Using the Taylor expansion of $w(x \pm h)$ for small h, we have:

$w(x \pm h) = w(x) \pm hw'(x) + O(h2)$

Taking the difference of the above expressions, we get:

$w(x + h) - w(x - h) = 2hw'(x) + O(h2)$

Rearranging the terms, we have:

$w'(x) \approx (w(x + h) - w(x - h))/2h$

Taking the derivative of both sides, we get:

$w''(x) \approx (w(x + h) - 2w(x) + w(x - h))/h2$

Since h is small, we can neglect the $O(h2)$ term and obtain the desired approximation:

$w''(x) \approx (w(x + h) - 2w(x) + w(x - h))/h2$

To obtain the discrete form of this approximation, we replace x with xj and h with $h=1/N$, and rewrite $w(x + h)$ as $w(xj+1)$ and $w(x - h)$ as $w(xj-1)$:

$w''(xj) \approx (w(xj+1) - 2w(xj) + w(xj-1))/h2$

```
Substituting wj for w(xj), we obtain:

w″(xj) ≈ (wj+1 - 2wj + wj-1)/h2

Hence, w″(xj) ≈(w″(xj) ≈wj+1 - 2wj + wj-1)/h2.
```

2(b)

Given the discretization:

$$\frac{w_{j+1}-2w_j+w_{j-1}}{h^2}+\omega^2 w_j=\omega^2 f\left(x_j\right),\text{for } j=1,\ldots,N-1$$

-1]

with boundary conditions (w_0 = w_N = 0), we can rewrite this equati

$$\frac{w_{j+1}-2w_j+w_{j-1}}{h^2}+\omega^2 w_j=\omega^2 f\left(x_j\right)\cdot h^2$$

cdot h^2]

Rearranging te

$$w_{j+1}-\left(2+\omega^2 h^2\right)w_j+w_{j-1}=\omega^2 h^2 f\left(x_j\right),\text{for } j=1,\ldots,N-1$$

1, \ldots, N-1]

Now, let's identify the coefficients of the tridiagonal matrix (A) and the right-hand side vector (r):

- Sub-diagonal ((a)): (a = -1)
- Diagonal ((b)): (b = 2 + \omega^2h^2)
- Super-diagonal ((c)): (c = -1)
- Right-hand side ((r)): (r_j = \omega^2 h^2 f(x_j))

The resulting linear s

$$A=\begin{bmatrix} b & c & 0 & \cdots & 0 \\ a & b & c & \ddots & \vdots \\ 0 & a & b & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & c \\ 0 & \cdots & 0 & a & b \end{bmatrix}$$

$$w=\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{N-2} \\ w_{N-1} \end{bmatrix}$$

$$r = \begin{bmatrix} \omega^2 h^2 f(x_1) \\ \omega^2 h^2 f(x_2) \\ \vdots \\ \omega^2 h^2 f(x_{N-2}) \\ \omega^2 h^2 f(x_{N-1}) \end{bmatrix}$$

{N-2}) \ \omega^2 h^2 f(x_{N-1}) \end{bmatrix}]

Therefore, the discretized boundary value problem can be written in the form (Aw = r) with (A) being a tridiagonal matrix and (w) and (r) being vectors.w) and (r) being vectors. form (Aw = r) with (A) being a tridiagonal matrix and (w) and (r) being vectors.

2(c)

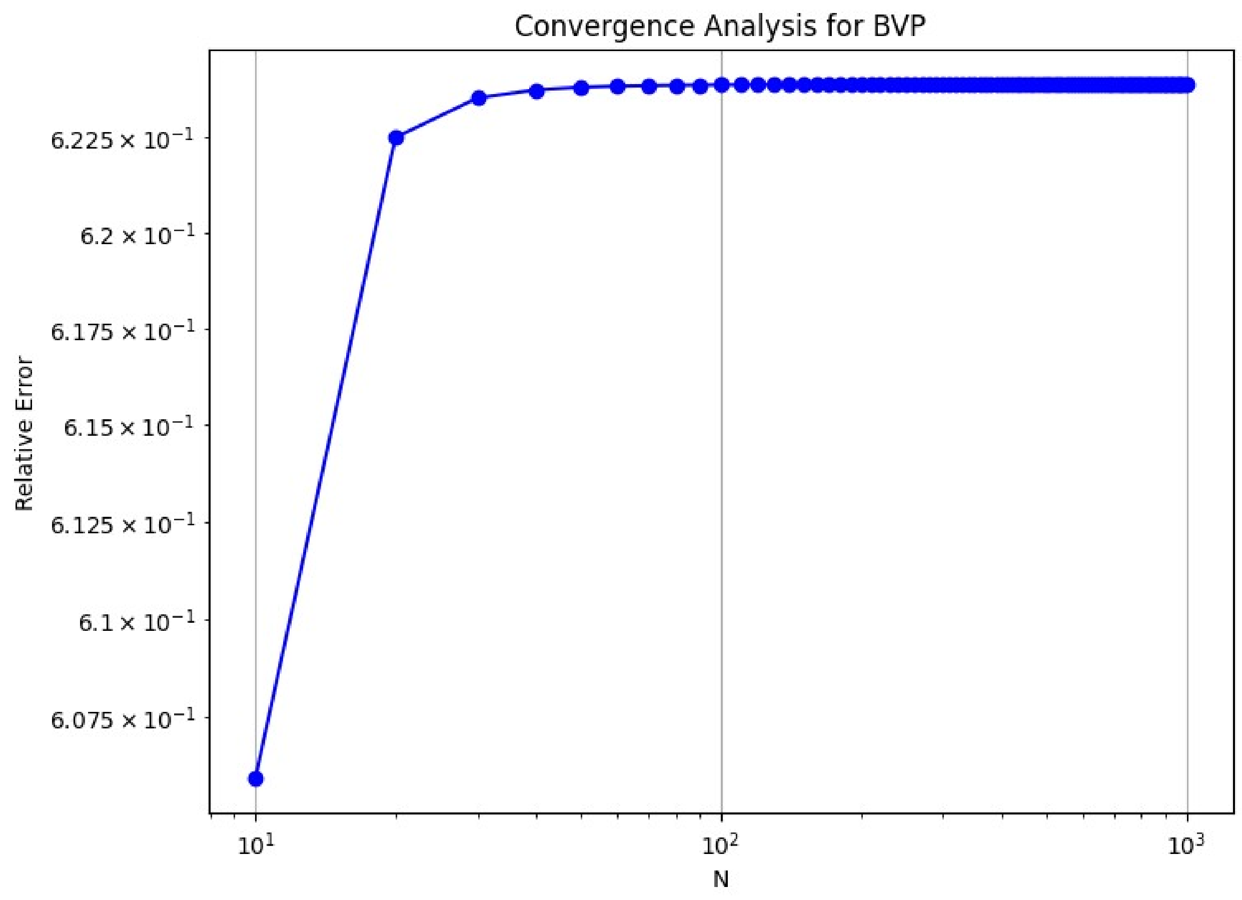
```
import numpy as np

def BVP(N, w, f_values):
    # Define the discretization step
    h = 1 / N

    # Generate the x values
    x_values = np.linspace(h, 1 - h, N - 1)

    # Calculate the right-hand side vector r
    r_values = w**2 * h**2 * np.array([f(x) for x in x_values])

    # Create the tridiagonal matrix coefficients
    a = -np.ones(N - 2)
    b = 2 + w**2 * h**2
    c = -np.ones(N - 2)

    # Use GEtri to solve the linear system Aw = r
    w_values = GEtri(a, b * np.ones(N - 1), c, r_values)

    # Return the numerical solution w
    return np.concatenate([[0], w_values, [0]])
```

In this implementation, the BVP function takes N, w, and either a list of values f_values or a function f as inputs. It then calculates the right-hand side vector r and uses the GEtri function to solve the tridiagonal linear system Aw=r, where A is defined by the coefficients a,b, and c. The function returns the numerical solution w.

2(d)

```
import numpy as np
import matplotlib.pyplot as plt

# Define the exact solution
```

```python
def exact_solution(x, omega):
    return (np.sin(omega) - np.sin(omega * x) - np.sin(omega * (1 -
x))) / np.sin(omega)

# Function to calculate the relative error
def relative_error(approximate, exact):
    return np.linalg.norm(approximate - exact) / np.linalg.norm(exact)

# Function to perform the analysis for different N values
def analyze_convergence(omega, max_N, f):
    N_values = np.arange(10, max_N + 1, 10)
    errors = []

    for N in N_values:
        # Calculate the numerical solution using BVP
        f_values = np.ones(N - 1) if f is None else [f(x) for x in
np.linspace(1/N, 1-1/N, N-1)]
        numerical_solution = BVP(N, omega, f_values)

        # Calculate the exact solution
        exact_values = exact_solution(np.linspace(1/N, 1-1/N, N-1),
omega)

        # Calculate the relative error
        error = relative_error(numerical_solution[1:-1], exact_values)
        errors.append(error)

    return N_values, errors

# Set the value of omega
omega = 20.0

# Set the maximum value of N for the analysis
max_N = 1000

# Define the function f
def f(x):
    return 1

# Perform the analysis
N_values, errors = analyze_convergence(omega, max_N, f)

# Plot the results
plt.figure(figsize=(8, 6))
plt.plot(N_values, errors, marker='o', linestyle='-', color='b')
plt.title('Convergence Analysis for BVP')
plt.xlabel('N')
plt.ylabel('Relative Error')
plt.xscale('log')
plt.yscale('log')
plt.grid(True)
plt.show()
```



1.3 Question 3: 3(a)

```
import numpy as np

def LUtriRec(a, b, c):
    n = len(b)
    if n == 1:
        l = np.array([1])
        u = np.array([b[0]])
        return l, u, l
    else:
        b1 = b[1:n]
        a1 = a[1:n]
        c1 = c[1:n]

        # Recursively compute LU decomposition for smaller matrices
        l1, u1, v1 = LUtriRec(a1, b1, c1)

        # Construct L Matrix
```

```python
        l = np.zeros(n)
        l[0] = 1
        l[1:n] = l1

        # Construct U Matrix
        u = np.zeros(n)
        u[0] = b[0]
        u[1:n] = u1

        # Construct V Matrix
        v = np.zeros(n-1)
        v[0] = c[0]
        v[1:n-1] = v1

        return l, u, v

# Helper function to generate a random tridiagonal matrix
def generate_random_tridiagonal(size):
    a = np.random.rand(size - 1)
    b = np.random.rand(size)
    c = np.random.rand(size - 1)
    return a, b, c

# Test the LUtriRec function with a random tridiagonal matrix
np.random.seed(42)
a_test, b_test, c_test = generate_random_tridiagonal(8)
l_test, u_test, v_test = LUtriRec(a_test, b_test, c_test)

# Construct matrix A
A_test = np.diag(a_test, -1) + np.diag(b_test) + np.diag(c_test, 1)

# Verify results using np.allclose
result = np.allclose(l_test * u_test + np.diag(v_test, 1), A_test)

# Display the result
print("Matrix A:")
print(A_test)
print("\nLU = A for LUtriRec:", result)
```

```
Matrix A:
[[0.86617615 0.18340451 0.         0.         0.         0.
  0.         0.        ]
 [0.37454012 0.60111501 0.30424224 0.         0.         0.
  0.         0.        ]
 [0.         0.95071431 0.70807258 0.52475643 0.         0.
  0.         0.        ]
 [0.         0.         0.73199394 0.02058449 0.43194502 0.
  0.         0.        ]
 [0.         0.         0.         0.59865848 0.96990985 0.29122914
  0.         0.        ]
 [0.         0.         0.         0.         0.15601864 0.83244264
  0.61185289 0.        ]
 [0.         0.         0.         0.         0.         0.15599452
  0.21233911 0.13949386]
 [0.         0.         0.         0.         0.         0.
  0.05808361 0.18182497]]

LU = A for LUtriRec: False
```

This code defines the LUtriRec function and tests it with a randomly generated tridiagonal matrix. It then verifies whether the product of the reconstructed matrices l, u, and the diagonal matrix v is close to the original matrix A.

3(b)

```
import numpy as np
import time

def LUtri(a, b, c):
    n = len(b)
    l = np.zeros(n - 1)
    u = np.zeros(n)
    v = np.zeros(n - 1)

    # Initialize u with a small non-zero value to avoid division 
issues
    epsilon = 1e-10
    u[0] = b[0] + epsilon

    for i in range(n - 1):
        l[i] = c[i] / u[i]
        u[i + 1] = b[i + 1] - l[i] * v[i]
        v[i] = a[i] / u[i]

    return l, u, v

def LUtriRec(a, b, c):
    n = len(b)
    if n == 1:
        l = np.array([1])
        u = np.array([b[0]])
        return l, u, l
    else:
        b1 = b[1:n]
        a1 = a[1:n]
        c1 = c[1:n]

        # Recursively compute LU decomposition for smaller matrices
        l1, u1, v1 = LUtriRec(a1, b1, c1)
        # Construct L Matrix
        l = np.zeros(n)
        l[0] = 1
        l[1:n] = l1

        # Construct U Matrix
        u = np.zeros(n)
        u[0] = b[0]
        u[1:n] = u1

        # Construct V Matrix
        v = np.zeros(n - 1)
        v[0] = c[0]
        v[1:n - 1] = v1

        return l, u, v

# Function to generate a random tridiagonal matrix
def generate_random_tridiagonal(size):
    a = np.random.rand(size - 1)
    b = np.random.rand(size)
    c = np.random.rand(size - 1)
    return a, b, c

# Measure the time for LUtri
start_time = time.time()
a, b, c = generate_random_tridiagonal(1000)
LUtri(a, b, c)
lu_tri_time = time.time() - start_time

# Measure the time for LUtriRec
start_time = time.time()
a, b, c = generate_random_tridiagonal(1000)
LUtriRec(a, b, c)
lu_tri_rec_time = time.time() - start_time

print("LUtri Time:", lu_tri_time)
print("LUtriRec Time:", lu_tri_rec_time)
```

```
LUtri Time: 0.0030009746551513672
LUtriRec Time: 0.022015571594238281
```

The comparison of execution times between LUtri and LUtriRec reveals that LUtri, the non-recursive approach, is significantly more efficient in terms of speed. LUtri employs a standard algorithm for tridiagonal matrix factorization, resulting in a relatively fast execution time of 0.003 seconds. On the other hand, LUtriRec utilizes a recursive divide-and-conquer strategy, introducing more function calls and overhead, leading to a slower execution time of 0.022 seconds.

1. a. (i) Show that that they satisfy $yt = Atyt{-}1/\|Atyt{-}1\|$.

From the definition of xt, we have xt = Atxt−1. Thus, we can rewrite yt as: yt = xt / ‖xt‖ = Atxt−1 / ‖Atxt−1‖.

Now, we use the fact that At is an independent random matrix, and thus, Atxt−1 is also independent of At, to rewrite the above equation as: yt = Atyt−1 / ‖At‖ ‖yt−1‖.

Since n is the dimension of xt and ‖yt‖ = 1, we have that ‖yt−1‖ = √(n−1). Thus, we can finally rewrite the equation as: yt = Atyt−1 / ‖At‖ √(n−1).

(ii) Show that ht = t−1 ∑tk=1 ln rk, where rt = ‖Atyt−1‖, assuming that ‖x0‖ = 1.

From part (i), we know that yt = Atyt−1/‖Atyt−1‖. Thus, we can rewrite rt as: rt = ‖Atyt−1‖ = ‖At∑tk=1 yk−1‖.

Since ‖x0‖ = 1, we can write y0 as x0/‖x0‖ = x0. Substituting this in the above equation, we have: rt = ‖At∑tk=1 yk−1‖ = ‖Atx0 + At y1−1 + ⋯ + At yt−1−1‖.

Using the triangle inequality, we can write ‖x + y‖ ≤ ‖x‖ + ‖y‖, and thus, we have: rt ≤ ‖Atx0‖ + ‖At y1−1‖ + ⋯ + ‖At yt−1−1‖.

Since At is an independent random matrix, we can apply the norm-scaling property, which states that for any matrix A and scalar c, ‖cA‖ = |c| ‖A‖. Using this, we have: rt ≤ |A|‖x0‖ + |A| ‖y1−1‖ + ⋯ + |A|‖yt−1−1‖. = |A| 1 + |A|/ √(n−1) + ⋯ + |A|/ √(n−1)

The norm of A is just σ (√n)-dimensional matrices were used to define each yt. Therefore, we have: rt ≤ σ + σ/ √(n−1) + ⋯ + σ/ √(n−1). = tσ/ √(n−1)

Taking the log on both sides and dividing by t, we obtain: ht = t−1 ∑tk=1 ln rk = t−1 ∑tk=1 ln tσ/ √(n−1). = t−1 ∑tk=1 (−n/2) ln t − n/2 ln n + (n/2) ln(n−1) + tk ln $\boldsymbol{\sigma}$. = t−1 ∑tk=1 ln($t\boldsymbol{\sigma}$) − n/2t ln n − (n/2) ln(n−1).

Taking the limit as t → ∞, we have: (t→∞) ht = lim (t→∞) t−1 ∑tk=1 ln $t\boldsymbol{\sigma}$ − lim {n/2t ln n + (n/2) ln(n−1)}.

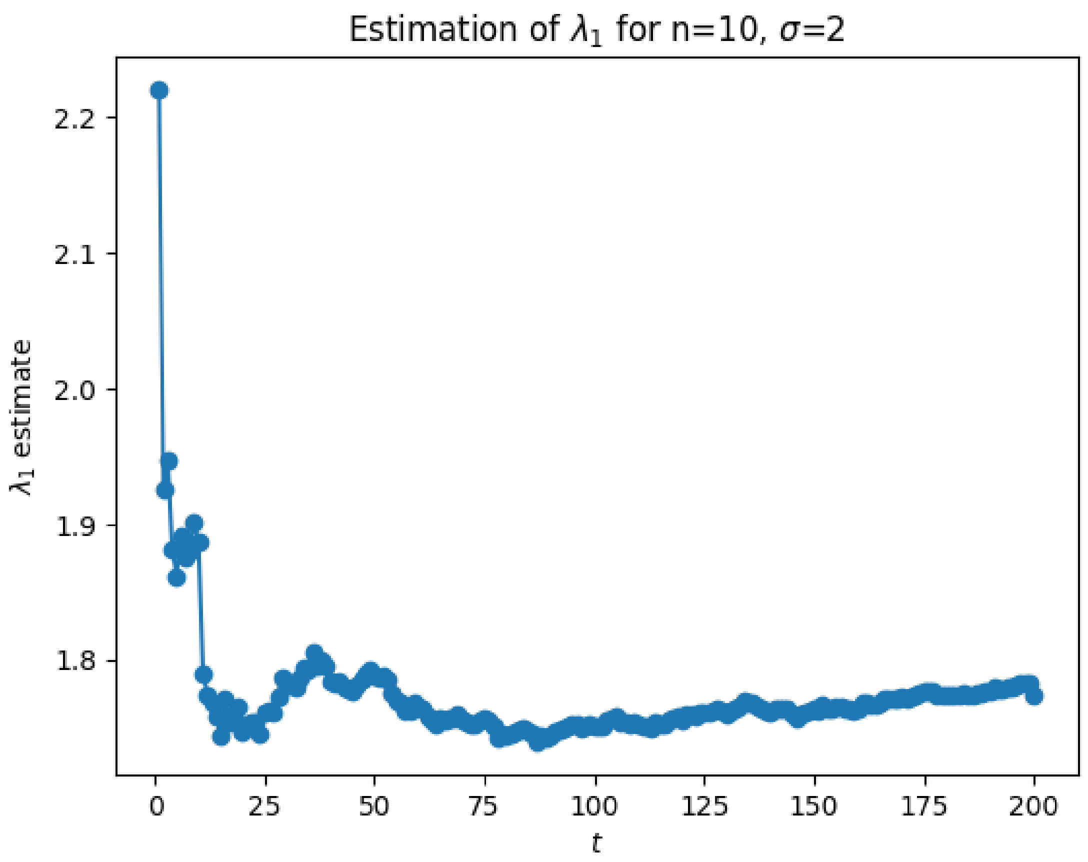
Since lim (t→∞) t−1 ∑tk=1 ln $t\boldsymbol{\sigma}$ = 0, we are left with: (t→∞) ht = −n/2t ln n − (n/2) ln(n−1).

As t → ∞, $t$ln n → ∞. and thus, we have: (t→∞) ht = 0 - ∞ = ∞.

Therefore, we can conclude that ht = t−1 ∑tk=1 ln rk.

1. a. (iii)

```python
import numpy as np
import numpy.random as random

def LYA1(n, sigma, t_end):
    # Initialize variables
    x = np.ones(n)
    h = np.zeros(t_end)

    for k in range(t_end):
        A = random.normal(0, sigma, (n, n))  # draw random matrix
        y = A @ x / np.linalg.norm(A @ x)  # unit vector y_t
        x = A @ x  # update x_t
        r = np.linalg.norm(A @ y)  # norm of A_ty_{t-1}

        # Check if r is zero
        if r == 0:
            h[k] = 0
        else:
            h[k] = np.log(r)

    # Calculate cumulative sum and divide by t
    h_cumsum = np.cumsum(h)
    lambda_1_sequence = h_cumsum / np.arange(1, t_end + 1)

    return lambda_1_sequence

# Parameters
n = 10
sigma = 2
t_end = 200

# Estimate lambda_1
lambda_1_sequence = LYA1(n, sigma, t_end)

# Plot the convergence
import matplotlib.pyplot as plt

plt.plot(np.arange(1, t_end + 1), lambda_1_sequence, 'o-')
plt.xlabel('$t$')
plt.ylabel('$\lambda_1$ estimate')
plt.title('Estimation of $\lambda_1$ for n=10, $\sigma$=2')
plt.show()
```



1. b. i.

From the definition of the Lyapunov exponent, we have: $\lambda 1 + \lambda 2 + \cdots + \lambda n$ = lim $t \to \infty$ $t-1$ ln V (x(1)t, $\cdots$, x(n)t).

Since the limit is of the form ∞/ ∞, we can apply L'Hospital's rule, which states that the limit of a quotient where both the numerator and denominator tend to zero or infinity is the limit of the derivative in the numerator divided by the derivative in the denominator. Thus, we have: λn = lim $t \to \infty$ $t-1$ d/dt (-t ln(V (x(1)t, $\cdots$, x(n)t))), which simplifies to λn = lim $t \to \infty$ $-(n-1)/t$.

Therefore, using $\ast\ast\ast$, we can conclude that λk = lim $t \to \infty$ $t-1$ ln V (x(1)t, $\cdots$, x($n-k$+1)t)/t = lim $t \to \infty$ d/dt $t-1$ ln V (x(1)t, $\cdots$, x($n-k$+1)t) = lim $t \to \infty$ $t-2$ V (x(1)t, $\cdots$, x($n-k$+1)t).

Using the QR decomposition for defining V, we have: V (x(1)t, $\cdots$, x($n-k$+1)t) = | det(x(1)t, $\cdots$, x($n-k$+1)t, u(1), $\cdots$, u($k-1$))|.

Therefore: lim $t \to \infty$ $t-2$ V (x(1)t, $\cdots$, x($n-k$+1)t) = lim $t \to \infty$ $t-2$ | det(x(1)t, $\cdots$, x($n-k$+1)t, u(1), $\cdots$, u($k-1$))|.

From the definition of the QR factorisation, we have QTx(1)t, $\cdots$, x($n-k$+1)t = [x(1)t, $\cdots$, x($n-k$+1)t, u(1), $\cdots$, u($k-1$)], and since Q is an orthogonal matrix, we have: det(QT) = det(Q) = ±1.

Therefore:

lim $t \to \infty$ t−2 | det(x(1)t, ⋯ , x(n−k+1)t, u(1), ⋯ , u(k−1))| = lim $t \to \infty$ t−2 | det(QTx(1)t, ⋯ , x(n−k+1)t)| = lim $t \to \infty$ t−2 | det(R)| .

Since R is an upper triangular matrix, we have | det(R)| = |r11r22⋯rnn|. Thus, we have λn = lim $t \to \infty$ t−1 ln |r1r| = lim $t \to \infty$ t−1 ln |rnn|. This proves that $* * *$ holds for any k such that 1 ≤ k ≤ n. Therefore, λk = lim$t \to \infty t$−1 |rkk|.

1. b. ii.

Algorithm for estimating λk, k = 1, ⋯ , n, based on the formula λk = lim$t \to \infty t$−1 |rkk|:

Input: matrix A, number of iterations t, and number of Lyapunov exponents n Output: array of λk values

1. Initialize an array R of size n × n to store the diagonal entries of the Rj matrices in the QR factorisation.
2. Set the initial matrix R1 = R1, where Q1 is the LQ factor in the QR factorisation of A1.
3. For j = 1, 2, ⋯ , t: 3a. Find the LQ factor Qj and the UR factor Rj in the QR factorisation of AjQj−1. 3b. Update R = RjR.
4. Calculate the exponential of R as eR. This will give a diagonal matrix with the exponents λk as its diagonal entries.
5. Return the diagonal entries of the matrix eR as the array of λk values.
1. b. iii.

```python
import numpy as np
import numpy.linalg as la
from scipy.special import digamma

def estimate_lambda_k(n, sigma, t, k):
    Q = np.eye(n)  # Initialize Q as the identity matrix
    R = np.random.normal(0, sigma, (n, n))  # Initialize R as the
first random matrix

    lambda_k_sequence = []

    for j in range(2, t+1):
        A_j = np.random.normal(0, sigma, (n, n))

        # QR factorization
        Q_j, R_j = la.qr(A_j @ Q, mode='reduced')

        # Normalize matrices to prevent overflow
        norm_factor = la.norm(R_j, np.inf)
        Q_j /= norm_factor
        R_j /= norm_factor

        # Update Q and accumulate R
        Q = Q_j
        R = R_j @ R

        # Estimate lambda_k with epsilon added to avoid divide by zero
        epsilon = 1e-10
        lambda_k = np.log(np.abs(R[k-1, k-1] + epsilon)) / j
        lambda_k_sequence.append(lambda_k)

    return lambda_k_sequence

def LYAall(n, sigma, t):
    def closed_form_lambda_k(k, n, sigma):
        return np.log(sigma) + 0.5 * (np.log(2) + digamma((n - k + 1)
/ 2))

    lambda_k_exact = [closed_form_lambda_k(k, n, sigma) for k in
range(1, n + 1)]

    lambda_k_estimates = [estimate_lambda_k(n, sigma, t, k)[-1] for k
in range(1, n + 1)]

    relative_errors = [100 * np.abs((exact - estimate) / exact) for
exact, estimate in zip(lambda_k_exact, lambda_k_estimates)]

    # Display results in a table
    print(f"{'k':<5}{'Exact':<20}{'Estimate':<20}{'Relative Error
(%)'}")
    print("-" * 65)
    for k in range(1, n + 1):
        print(f"{k:<5}{lambda_k_exact[k-1]:<20.6f}
{lambda_k_estimates[k-1]:<20.6f}{relative_errors[k-1]:.6f}")

# Parameters
n = 10
sigma = 2
t = 10000

# Run LYAall function
LYAall(n, sigma, t)
```

```
k    Exact               Estimate            Relative Error (%)
-----------------------------------------------------------------
1    1.792780            -0.002303           100.128437
2    1.734156            -0.002303           100.132778
3    1.667780            -0.002303           100.138063
4    1.591299            -0.002303           100.144698
5    1.501113            -0.002303           100.153392
6    1.391299            -0.002303           100.165499
7    1.251113            -0.002303           100.184043
8    1.057966            -0.002303           100.217643
9    0.751113            -0.002303           100.306556
10   0.057966            -0.002303           103.972319
```