Imports

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

a) Function to generate tridiagonal matrix

```
def generate_tridiagonal_matrix(n):

    matrix = [[0] * n for _ in range(n)]

    # Set main diagonal elements to -2
    for i in range(n):
        matrix[i][i] = -2

    # Set upper diagonal elements to 1
    for i in range(n - 1):
        matrix[i][i + 1] = 1

    # Set lower diagonal elements to 1
    for i in range(1, n):
        matrix[i][i - 1] = 1

    return matrix
```

Example usage:

```
n = 4
tridiagonal_matrix = generate_tridiagonal_matrix(n)
for row in tridiagonal_matrix:
    print(row)
```

```
[-2, 1, 0, 0]
[1, -2, 1, 0]
[0, 1, -2, 1]
[0, 0, 1, -2]
```

b) Solve $Ax=b$ using LU decomposition

In LU decomposition, we decompose the system of equation $Ax=b$ as $LUx=b$. Then we can solve the original problem in two sets as $Ly=b$ (with forward substitution) and $Ux=y$ (with backward substituion).

Functions

```
def lu_decompose(A):
    n = len(A)
    L = np.zeros((n, n))  # Initialize L matrix with zeros
```

```
    U = np.zeros((n, n))  # Initialize U matrix with zeros

    for i in range(n):

        # Upper triangular matrix
        for k in range(i, n):
            sum_ = 0
            for j in range(i):
                sum_ += L[i][j] * U[j][k]
            U[i][k] = A[i][k] - sum_

        # Lower triangular matrix
        for k in range(i, n):
            if i == k:
                L[i][i] = 1  # Diagonal elements of L are 1
            else:
                sum_ = 0
                for j in range(i):
                    sum_ += L[k][j] * U[j][i]
                L[k][i] = (A[k][i] - sum_) / U[i][i]

    return L, U

def forward_substitution(L, b):
    n = len(b)
    y = np.zeros(n)

    # Forward substitution
    for i in range(n):
        y[i] = (b[i] - np.dot(L[i,:i], y[:i])) / L[i,i]

    return y

def backward_substitution(U, y):
    n = len(y)
    x = np.zeros(n)

    # Backward substitution
    for i in range(n-1, -1, -1):
        x[i] = (y[i] - np.dot(U[i,i+1:], x[i+1:])) / U[i,i]

    return x
```

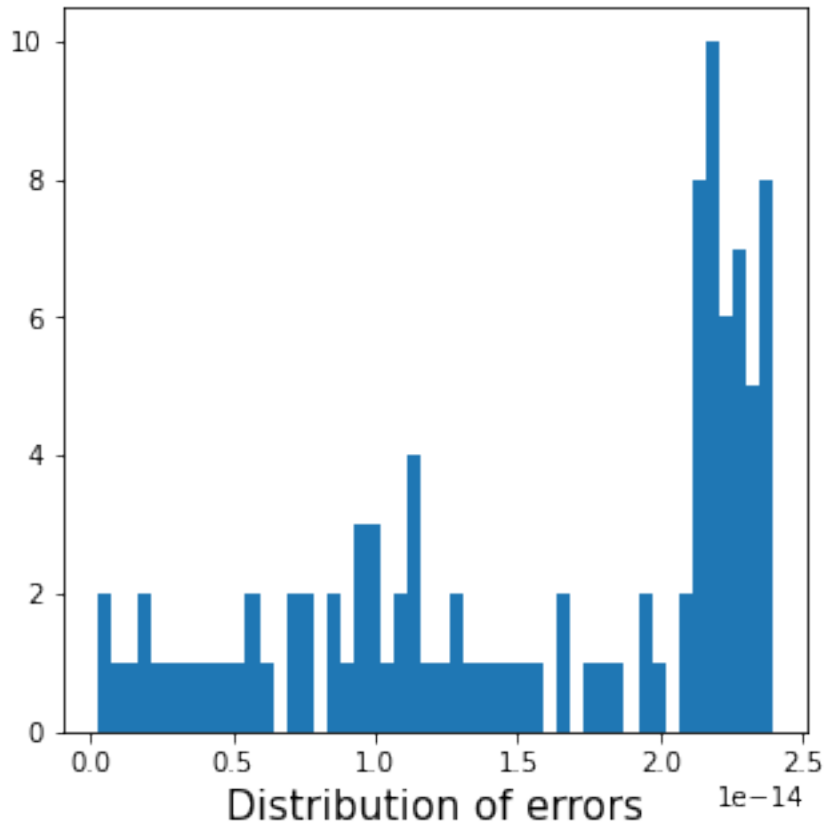
Example Usage

```
n = 100
A = generate_tridiagonal_matrix(n)
x_original  = np.random.normal(0, 1, n)  # generate a random vector
solution from the gaussian distribution with
                                          # mean 0 and standarad
```

```
deviation 1

b = np.dot(A, x_original)                    # multiply A*x to get b

L,U = lu_decompose(A)

# Solve Ly = b using forward substitution
y = forward_substitution(L, b)

# Solve Ux = y using backward substitution
x = backward_substitution(U, y)              # recovered solution
```

Compare the error in the recovered solution

```
store_errors = []
for j in range(len(x_original)):
    error = np.abs(x_original[j] - x[j])
    store_errors.append(error)

plt.figure(figsize = (5,5))
plt.hist(store_errors, bins = 50)
plt.xlabel('Distribution of errors', fontsize = 15)
plt.show()
```



Remark : We see that the error in the recovered solution is the order of 10^{-14}

c) Solve $Ax=b$ using Gauss Seidel function

The Gauss-Seidel method is an iterative technique used to solve a system of linear equations. It's an improvement over the Gauss elimination method for solving linear systems. Here's how it works:

1. Initial Guess: Start with an initial guess for the solution vector x_0.
2. Iterative Procedure: Repeat until convergence:

   Update each component of the solution vector iteratively using the current values of other components. At each iteration kk, update the iith component of the solution vector x(k)x(k) using the formula:

   $x_i^{(k)} = \frac{1}{A_{ii}}\left(b_i - \sum_{j=1}^{i-1} A_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} A_{ij} x_j^{(k-1)}\right)$ where A is the coefficient matrix, b is the constant vector, and n is the number of equations.
3. Convergence Criteria: Check for convergence by comparing the new solution $x(k)$ with the previous solution $x(k-1)$. Typically, convergence is determined by checking whether the change in the solution vector is below a predefined tolerance level.
4. Termination: Stop the iteration when the convergence criteria are met, and the solution vector has sufficiently converged

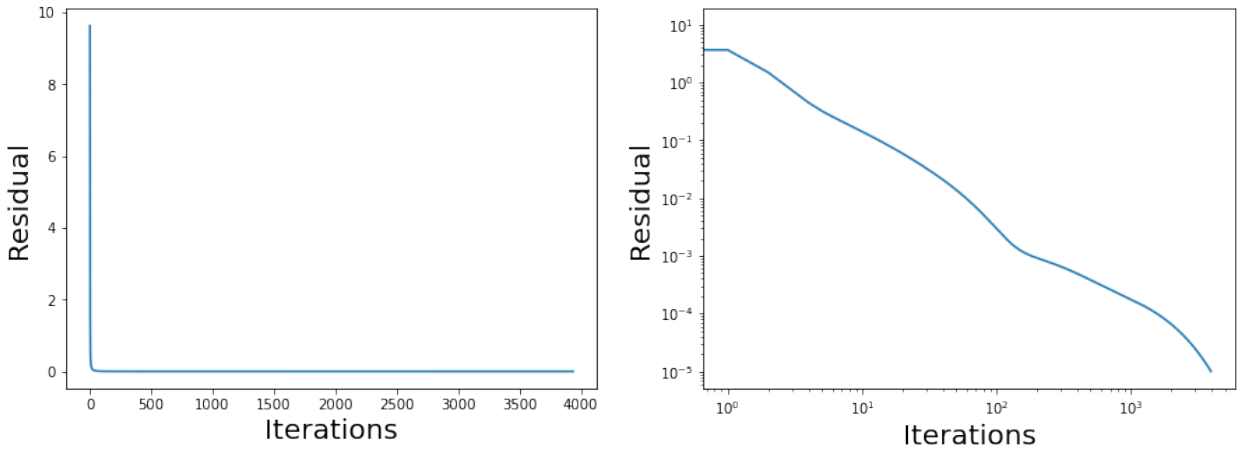
```
def gauss_seidel(A, b, x0, tol, max_iter=100000):
    n = len(b)
    x = x0  # Initial guess

    for _ in range(max_iter):
        x_new = np.zeros(n)
        for i in range(n):
            # Compute the new value of x[i]
            x_new[i] = (b[i] - np.dot(A[i,:i], x_new[:i]) -
np.dot(A[i,i+1:], x[i+1:])) / A[i,i]

        # Check for convergence
        if np.linalg.norm(x_new - x) < tol:
            return x_new

        x = x_new

    raise ValueError("Gauss-Seidel method did not converge within the
maximum number of iterations.")

# Example usage:
```

```
A = np.array(A)
b = np.array(b)
x0 = np.zeros(n)
x_gs = gauss_seidel(A, b, x0, tol = 1e-5)
```

d) Modify the Gauss-Seidel function to display tolerance at each iteration.

```
def mod_gauss_seidel(A, b, x0, tol, max_iter=100000):

    n = len(b)
    x = x0  # Initial guess
    norm_t = []
    for iter_ in range(max_iter):
        x_new = np.zeros(n)
        for i in range(n):
            # Compute the new value of x[i]
            x_new[i] = (b[i] - np.dot(A[i,:i], x_new[:i]) -
np.dot(A[i,i+1:], x[i+1:])) / A[i,i]

        # Check for convergence
        norm = np.linalg.norm(x_new - x)
        norm_t.append(norm)
        #print(f'iteration: {iter_}, residual: {norm}')
        if norm < tol:
            return x_new, norm_t

        x = x_new

    raise ValueError("Gauss-Seidel method did not converge within the
maximum number of iterations.")

# Example usage:
A = np.array(A)
b = np.array(b)
x_gs, residual = mod_gauss_seidel(A, b, x0,tol = 1e-5)

plt.figure(figsize = (15,5))

plt.subplot(121)

plt.plot(residual)
plt.xlabel('Iterations', fontsize = 20)
plt.ylabel('Residual', fontsize = 20)

plt.subplot(122)

plt.loglog(residual)
plt.xlabel('Iterations', fontsize = 20)
```

```
plt.ylabel('Residual', fontsize = 20)

plt.show()
```



e) Gauss-Seidel and LU decomposition for pre-defined system

```
A = np.load('A.npy')
b = np.load('b.npy')

## LU decomposition

L,U = lu_decompose(A)

# Solve Ly = b using forward substitution
y = forward_substitution(L, b)

# Solve Ux = y using backward substitution
x_lu = backward_substitution(U, y)

## Gauss Seidel

x_gs, residual = mod_gauss_seidel(A, b, x0,tol = 1e-5)
```

```
/tmp/ipykernel_18587/3806670321.py:10: RuntimeWarning: overflow
encountered in true_divide
  x_new[i] = (b[i] - np.dot(A[i,:i], x_new[:i]) - np.dot(A[i,i+1:],
x[i+1:])) / A[i,i]

---------------------------------------------------------------------
-----
ValueError                                Traceback (most recent call
last)
Input In [77], in <cell line: 3>()
      1 ## Gauss Seidel
```

```
----> 3 x_gs, residual = mod_gauss_seidel(A, b, x0,tol = 1e-5)

Input In [63], in mod_gauss_seidel(A, b, x0, tol, max_iter)
     17         return x_new, norm_t
     19     x = x_new
---> 21 raise ValueError("Gauss-Seidel method did not converge within
the maximum number of iterations.")

ValueError: Gauss-Seidel method did not converge within the maximum
number of iterations.
```