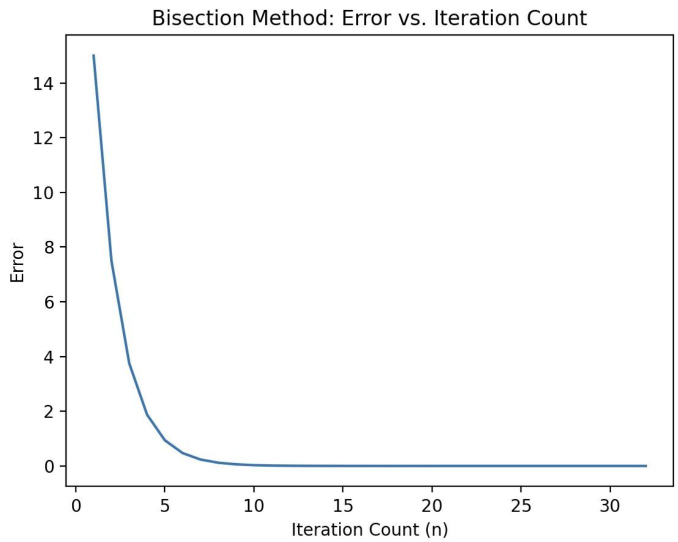
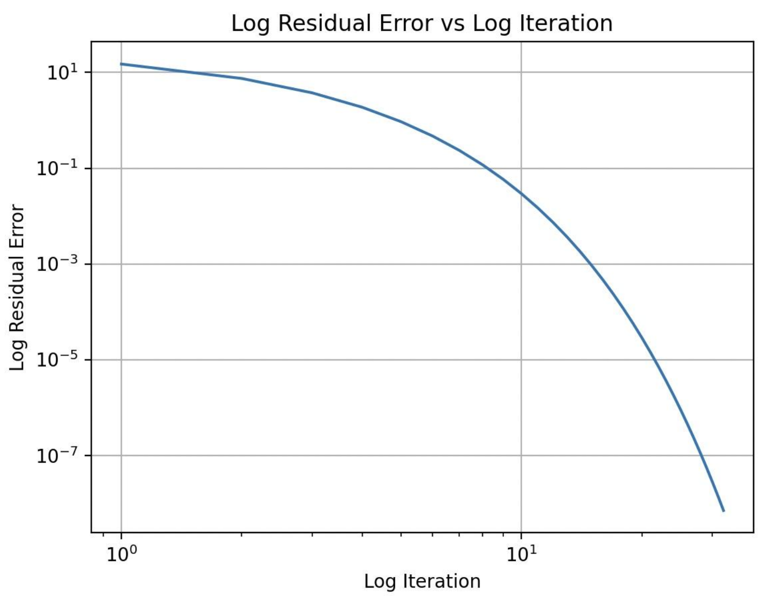
128B Final

Joe Lenning

Questions 1:






```
import matplotlib.pyplot as plt
#PART 1
def f1(x):
   f1x = (x ** (7/5) -1) / 9
   return f1x
```

```python
def bisection_method(func, a, b, error_accept, iterations):
   fa = func(a)
   i = 1
   errors = []
   while i <= iterations:
       p = a + ((b - a) / 2)
       fp = func(p)
       error = (b - a) / 2
       errors.append(error)
       if fp == 0 or error < error_accept:
           return p, errors
       i = i + 1
       if fa * fp > 0:
           a = p
           fa = fp
       else:
           b = p
   return 'Failed', errors

solution, errors = bisection_method(f1, 0, 30, 1e-8, 100)

# Plotting the error vs. iteration count
iteration_count = range(1, len(errors) + 1)
plt.plot(iteration_count, errors)
plt.xlabel('Iteration Count (n)')
plt.ylabel('Error')
plt.title('Bisection Method: Error vs. Iteration Count')
plt.show()

# Plot the log of the residual error versus log of the iteration
plt.loglog(iteration_count, errors)
plt.xlabel('Log Iteration')
plt.ylabel('Log Residual Error')
plt.title('Log Residual Error vs Log Iteration')
plt.grid(True)
plt.show()

print('Approximate solution:', solution)
print('Final error:', errors[-1])
```

Question 2:

Method	Average Time	Average Residual Error
Gauss Elimination	.01103	4.5852158e-15
LU Factorization	.0001113	4.805441202e-15

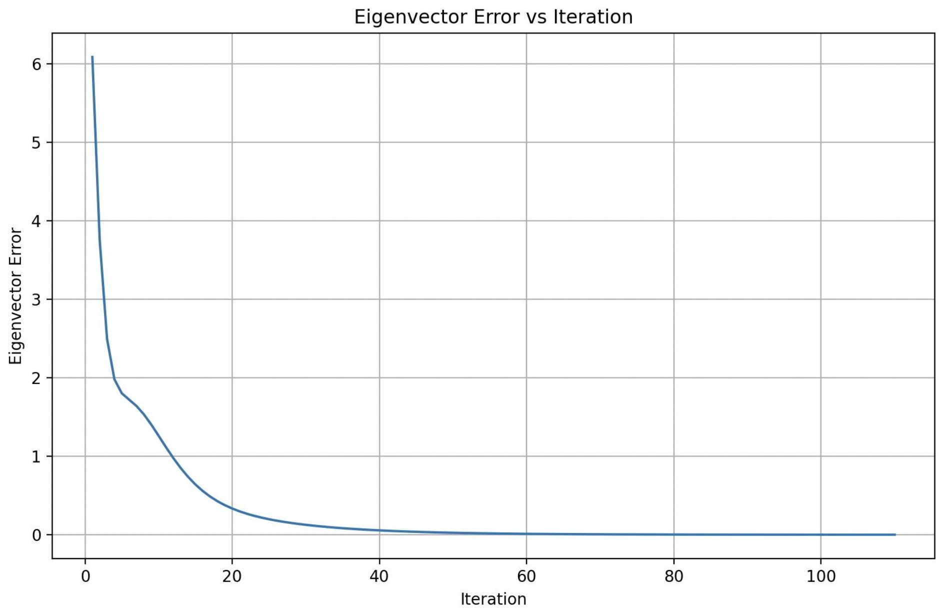
Jacobi	.002911	5.1263515795e-9
Gauss - Seidel	.00210	1.6536581-9

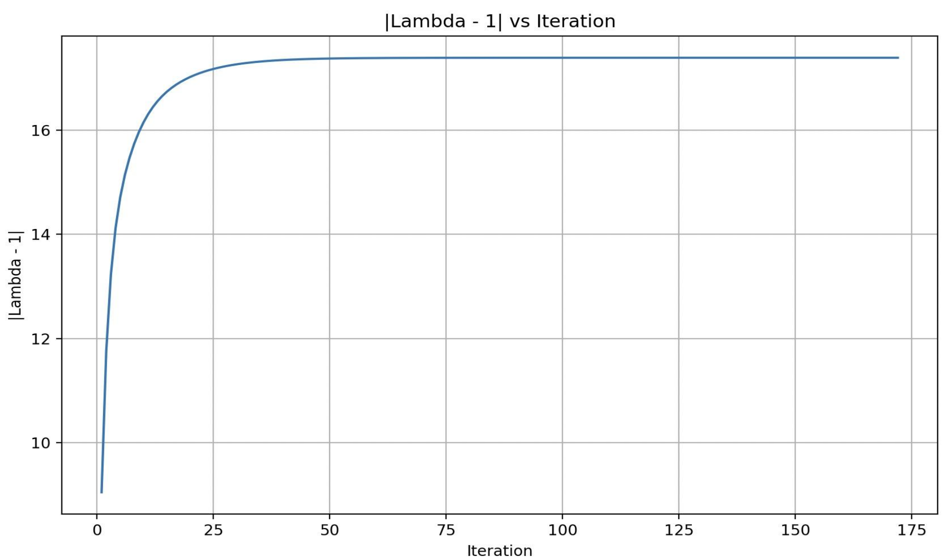
```
import numpy as np
import time
import scipy.linalg as la

# Step 1: Generate matrices and vectors
n = 100
I = np.eye(n)
P = I[np.random.permutation(n), :]
R = np.random.normal(0, 1, (n, n))
A = 7 * I + 1 / 10 * (P + R)
b_list = [np.random.normal(0, 1, n) for _ in range(25)]

# Step 2: Perform LU factorization of matrix A
start_time_lu = time.time()
lu_p, lu_l, lu_u = la.lu(A)
lu_factorization_time = time.time() - start_time_lu

# Step 3a: Gaussian elimination
def gaussian_elimination(A, b):
   augmented_matrix = np.column_stack((A, b))
   n = augmented_matrix.shape[0]

   for pivot_row in range(n):
       for row in range(pivot_row + 1, n):
           factor = augmented_matrix[row, pivot_row] /
augmented_matrix[pivot_row, pivot_row]
           augmented_matrix[row, pivot_row:] -= factor *
augmented_matrix[pivot_row, pivot_row:]

   x = np.zeros(n)
   for i in range(n - 1, -1, -1):
       x[i] = (augmented_matrix[i, -1] - np.dot(augmented_matrix[i, i+1:-1],
x[i+1:])) / augmented_matrix[i, i]

   return x

gaussian_times = []
gaussian_residuals = []
for b in b_list:
   start_time_gaussian = time.time()
   x_gaussian = gaussian_elimination(A, b)
   gaussian_time = time.time() - start_time_gaussian
   gaussian_residual = np.linalg.norm(np.dot(A, x_gaussian) - b)
   gaussian_times.append(gaussian_time)
   gaussian_residuals.append(gaussian_residual)
```

```
# Step 3b: LU factorization (using precomputed factorization)
lu_times = []
lu_residuals = []
for b in b_list:
   start_time_lu_solve = time.time()
   x_lu = np.linalg.solve(A, b)
   lu_time = time.time() - start_time_lu_solve
   lu_residual = np.linalg.norm(np.dot(A, x_lu) - b)
   lu_times.append(lu_time)
   lu_residuals.append(lu_residual)

# Step 3c: Jacobi method
def jacobi_method(A, b, max_iterations=1000, tolerance=1e-8):
   n = A.shape[0]
   x = np.zeros(n)
   x_prev = np.zeros(n)
   iteration = 0

   while iteration < max_iterations:
       for i in range(n):
           x[i] = (b[i] - np.dot(A[i, :i], x_prev[:i]) - np.dot(A[i, i+1:],
x_prev[i+1:])) / A[i, i]

       if np.linalg.norm(x - x_prev) < tolerance:
           break

       x_prev = x.copy()
       iteration += 1

   return x

jacobi_times = []
jacobi_residuals = []
for b in b_list:
   start_time_jacobi = time.time()
   x_jacobi = jacobi_method(A, b)
   jacobi_time = time.time() - start_time_jacobi
   jacobi_residual = np.linalg.norm(np.dot(A, x_jacobi) - b)
   jacobi_times.append(jacobi_time)
   jacobi_residuals.append(jacobi_residual)

# Step 3d: Gauss-Seidel method
def gauss_seidel_method(A, b, max_iterations=1000, tolerance=1e-8):
   n = A.shape[0]
   x = np.zeros(n)
   x_prev = np.zeros(n)
   iteration = 0
```

```python
    while iteration < max_iterations:
        for i in range(n):
            x[i] = (b[i] - np.dot(A[i, :i], x[:i]) - np.dot(A[i, i+1:],
x_prev[i+1:])) / A[i, i]

        if np.linalg.norm(x - x_prev) < tolerance:
            break

        x_prev = x.copy()
        iteration += 1

    return x

gauss_seidel_times = []
gauss_seidel_residuals = []
for b in b_list:
    start_time_gauss_seidel = time.time()
    x_gauss_seidel = gauss_seidel_method(A, b)
    gauss_seidel_time = time.time() - start_time_gauss_seidel
    gauss_seidel_residual = np.linalg.norm(np.dot(A, x_gauss_seidel) - b)
    gauss_seidel_times.append(gauss_seidel_time)
    gauss_seidel_residuals.append(gauss_seidel_residual)

# Step 4: Compute average time and residual error for each method
avg_gaussian_time = np.mean(gaussian_times)
avg_gaussian_residual = np.mean(gaussian_residuals)
avg_lu_time = np.mean(lu_times)
avg_lu_residual = np.mean(lu_residuals)
avg_jacobi_time = np.mean(jacobi_times)
avg_jacobi_residual = np.mean(jacobi_residuals)
avg_gauss_seidel_time = np.mean(gauss_seidel_times)
avg_gauss_seidel_residual = np.mean(gauss_seidel_residuals)

# Step 5: Display the results
print("Method\t\tAverage Time\t\tAverage Residual Error")
print("-----------------------------------------------------")
print(f"Gaussian\t{avg_gaussian_time}\t{avg_gaussian_residual}")
print(f"LU Factorization\t{avg_lu_time}\t{avg_lu_residual}")
print(f"Jacobi\t\t{avg_jacobi_time}\t{avg_jacobi_residual}")
print(f"Gauss-Seidel\t{avg_gauss_seidel_time}\t{avg_gauss_seidel_residual}")
```

Question 3:



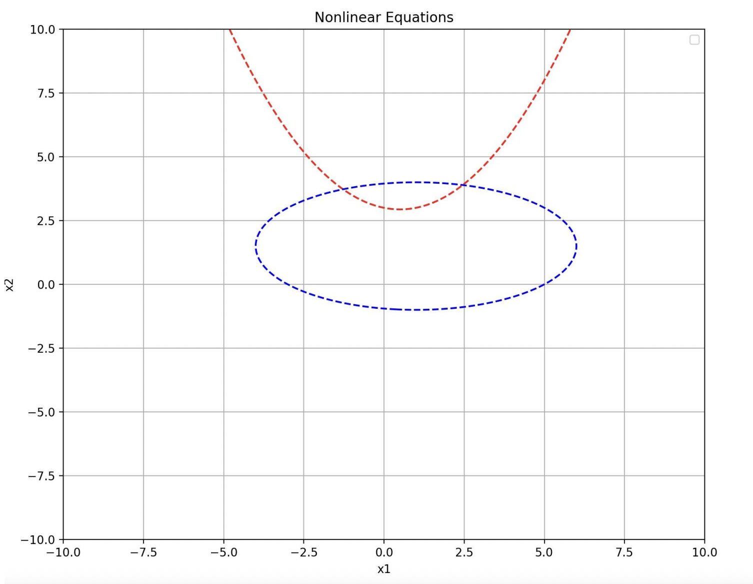


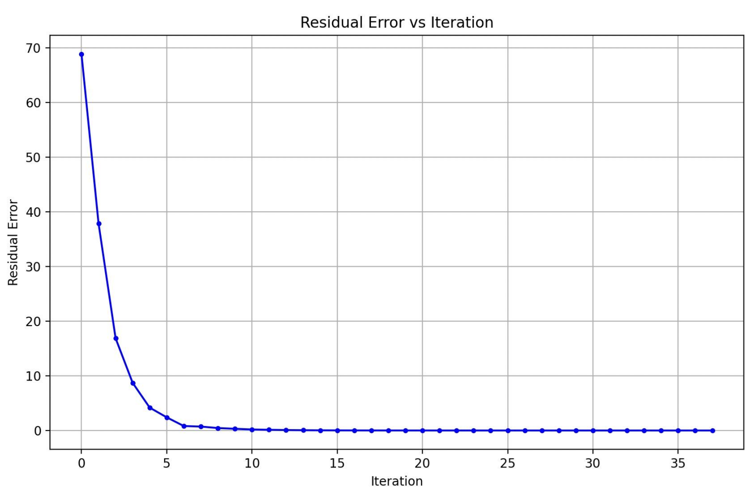

```
import numpy as np
import matplotlib.pyplot as plt

matrix_size = 100
I = np.eye(matrix_size)
R = 1/2 * np.random.randn(matrix_size, matrix_size)
B = R + R.T
A = (-4 * I) + B
```

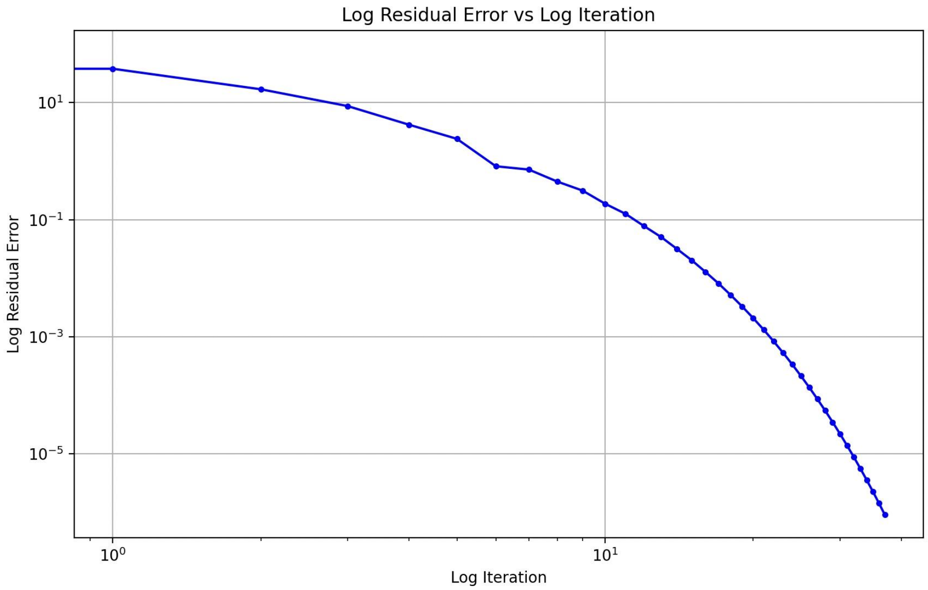
```python
def power_method(A, x0, tolerance):
   x = np.random.rand(matrix_size)
   x /= np.linalg.norm(x)

   lambda_prev = 0
   lambda_current = np.dot(x.T, A.dot(x))
   iteration = 0
   eigenvector_errors = []
   lambda_errors = []

   while np.abs(lambda_current - lambda_prev) >= tolerance:
       x = A.dot(x)
       x /= np.linalg.norm(x)

       lambda_prev = lambda_current
       lambda_current = np.dot(x.T, A.dot(x))

       eigenvector_error = np.linalg.norm(A.dot(x) - lambda_current * x)
       eigenvector_errors.append(eigenvector_error)

       lambda_error = np.abs(lambda_current - 1)
       lambda_errors.append(lambda_error)

       iteration += 1

   return lambda_current, x, eigenvector_errors, lambda_errors

eigenvalue, eigenvector, eigenvector_errors, lambda_errors = power_method(A, 0,
1e-8)

# Plot eigenvector error
plt.figure(figsize=(10, 6))
iterations = np.arange(1, len(eigenvector_errors) + 1)
plt.plot(iterations, eigenvector_errors)
plt.xlabel('Iteration')
plt.ylabel('Eigenvector Error')
plt.title('Eigenvector Error vs Iteration')
plt.grid(True)
plt.show()

# Plot lambda error
plt.figure(figsize=(10, 6))
iterations = np.arange(1, len(lambda_errors) + 1)
lambda_errors_abs = np.abs(np.array(lambda_errors) - 1)
plt.plot(iterations, lambda_errors_abs)
plt.xlabel('Iteration')
plt.ylabel('|Lambda - 1|')
plt.title('|Lambda - 1| vs Iteration')
plt.grid(True)
```

```
plt.show()
```

Question 4:








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

def f1(x, y):
   return x - x ** 2 + 4 * y - 12

def f2(x, y):
   return (x - 1) ** 2 + (2 * y - 3) ** 2 - 25

def f1_derivative_x(x, y):
   return 1 - 2 * x

def f1_derivative_y(x, y):
   return 4

def f2_derivative_x(x, y):
   return 2 * x - 2

def f2_derivative_y(x, y):
   return 4 * y - 6

# Generate x and y values
x1 = np.linspace(-10, 10, 400)
x2 = np.linspace(-10, 10, 400)
X1, X2 = np.meshgrid(x1, x2)

# Evaluate the equations
Z1 = f1(X1, X2)
Z2 = f2(X1, X2)
```

```python
# Create the plot
plt.figure(figsize=(10, 8))
plt.contour(X1, X2, Z1, levels=[0], colors='red', linestyles='dashed')
plt.contour(X1, X2, Z2, levels=[0], colors='blue', linestyles='dashed')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Nonlinear Equations')
plt.grid(True)
plt.legend(['Equation 1: x1 - x2/2 + 4x2 = 12', 'Equation 2: (x1 - 2)^2 + (2x2
- 3)^2 = 25'])
plt.show()

def newtonMethod(x0, y0, tol, max_iterations):
    start_time = time.time()
    iterations = 0
    errors = []
    for i in range(max_iterations):
        J = np.array([[f1_derivative_x(x0, y0), f1_derivative_y(x0, y0)],
                      [f2_derivative_x(x0, y0), f2_derivative_y(x0, y0)]])
        F = np.array([-f1(x0, y0), -f2(x0, y0)])
        delta = np.linalg.solve(J, F)
        x = x0 + delta[0]
        y = y0 + delta[1]
        my_error = np.linalg.norm([x - x0, y - y0])
        errors.append(my_error)
        if my_error < tol:
            end_time = time.time()
            elapsed_time = end_time - start_time
            return x, y, errors, iterations, elapsed_time
        x0 = x
        y0 = y
        iterations += 1
    end_time = time.time()
    elapsed_time = end_time - start_time
    return x, y, errors, iterations, elapsed_time

# Set the initial guess, tolerance, and maximum iterations
initial_guess = (0, 0)
tolerance = 1e-6
max_iterations = 1000

# Call the Newton's method function
x_solution, y_solution, errors, num_iterations, elapsed_time =
newtonMethod(initial_guess[0], initial_guess[1], tolerance, max_iterations)

# Print the results
print("Newton's Method:")
print("Solution (x, y):", (x_solution, y_solution))
```

```python
print("Number of iterations:", num_iterations)
print("Elapsed time:", elapsed_time, "seconds")

# Plot the residual error versus iteration
iterations = np.arange(num_iterations + 1)
plt.figure(figsize=(10, 6))
plt.plot(iterations, errors, 'b.-')
plt.xlabel('Iteration')
plt.ylabel('Residual Error')
plt.title('Residual Error vs Iteration')
plt.grid(True)
plt.show()

# Plot the log of the residual error versus log of the iteration
plt.figure(figsize=(10, 6))
plt.loglog(iterations, errors, 'b.-')
plt.xlabel('Log Iteration')
plt.ylabel('Log Residual Error')
plt.title('Log Residual Error vs Log Iteration')
plt.grid(True)
plt.show()
```