MAT128B Final Report

June 15, 2023

1 Problem 1

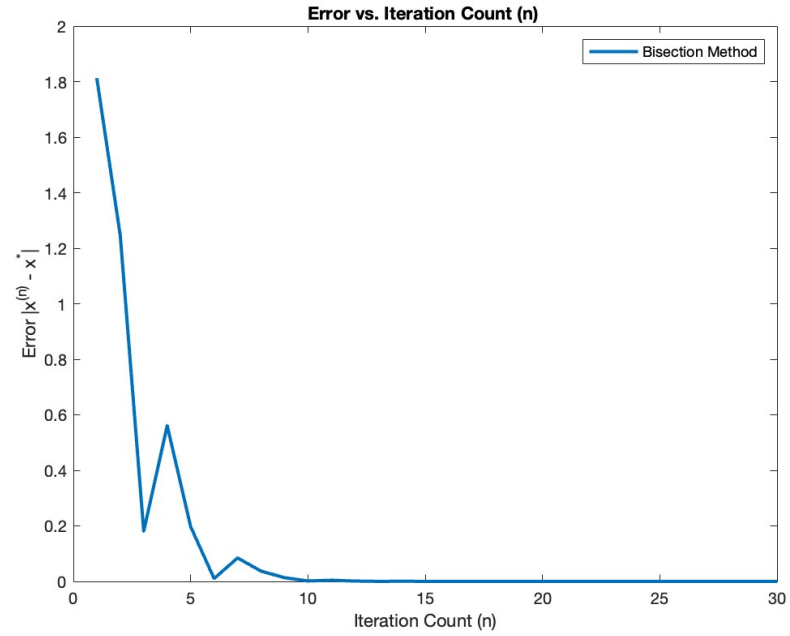

Figure 1: A graph of Iteration Count (n) vs. Error $|x^{(n)} - x^*|$ for Bisection method.

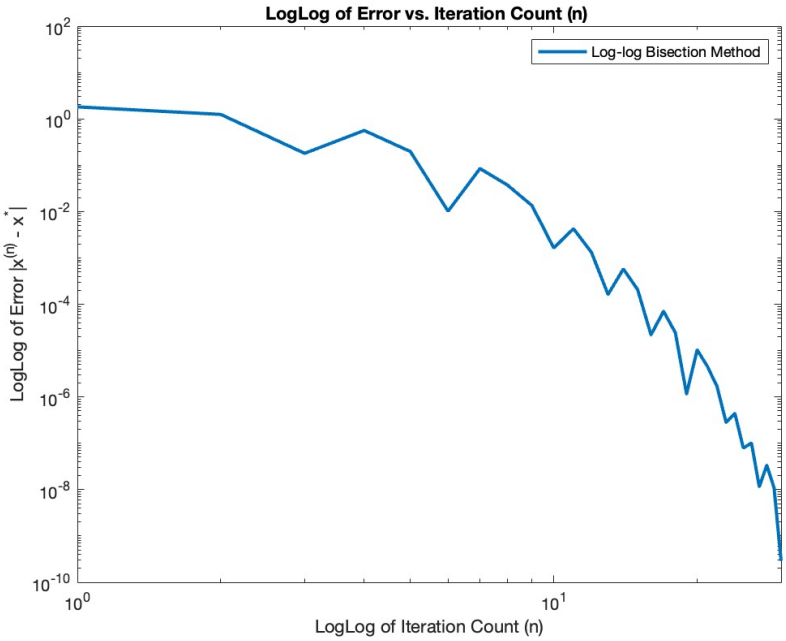

Figure 2: A graph of Iteration Count (n) vs. Error $|x^{(n)} - x^*|$ for Bisection method.

2 Problem 2

Method Name	Average Time	Average Error
Gaussian Elimination	0.0025862	4.1849e-15
LU Factorization	0.0013189	2.4212e-15
Jacobi Method	0.012405	4.9257e-09
Gauss-Siedel	0.0068125	7.3343e-10

Table 1: One table containing Average time and Average Error for Gaussian Elimination, LU Factorization, Jacobi Method, and Gauss-Siedel Method .

3 Problem 3

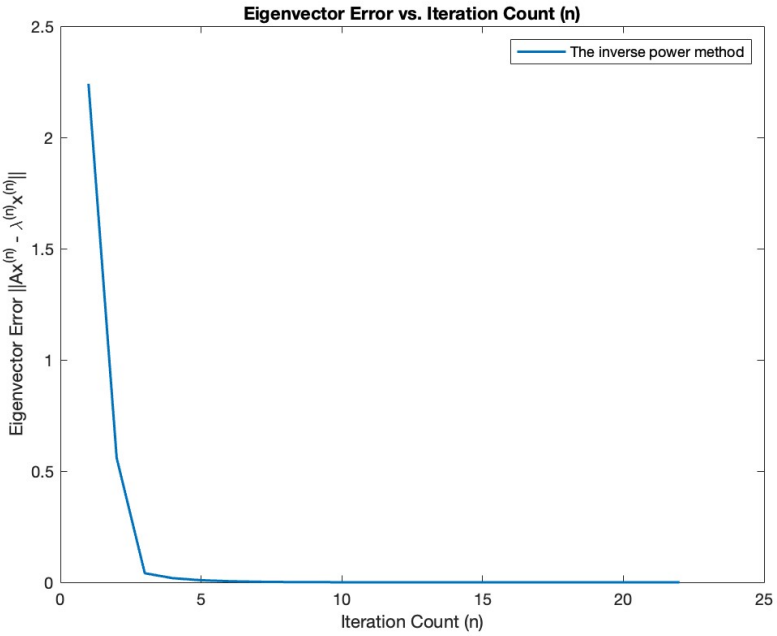

Figure 3: A graph showing the eigenvector error $||Ax^{(n)} - \lambda^{(n)}x^{(n)}||$ on the y-axis and n on the x-axis for the inverse power method.

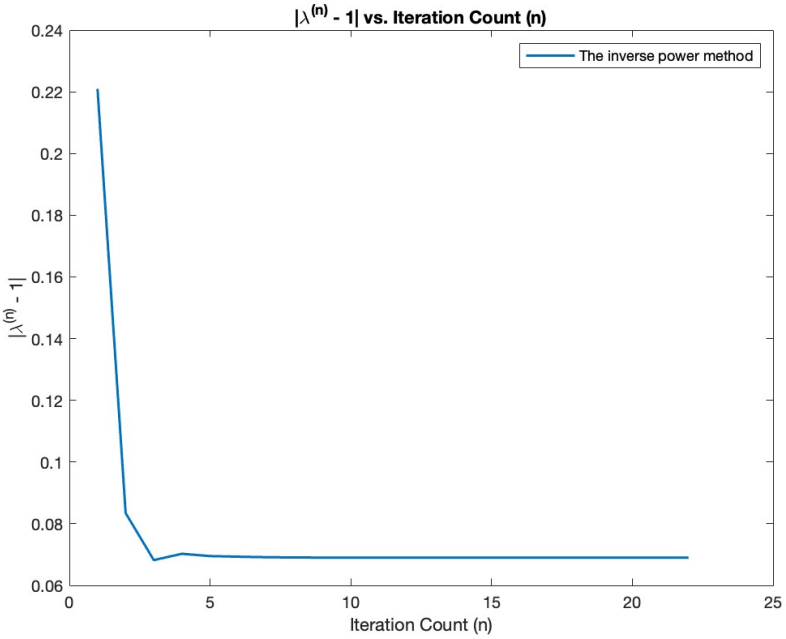

Figure 4: A graph showing $|\lambda^{(n)} - 1|$ on the y-axis and n on the x-axis for the inverse power method.

4 Problem 4

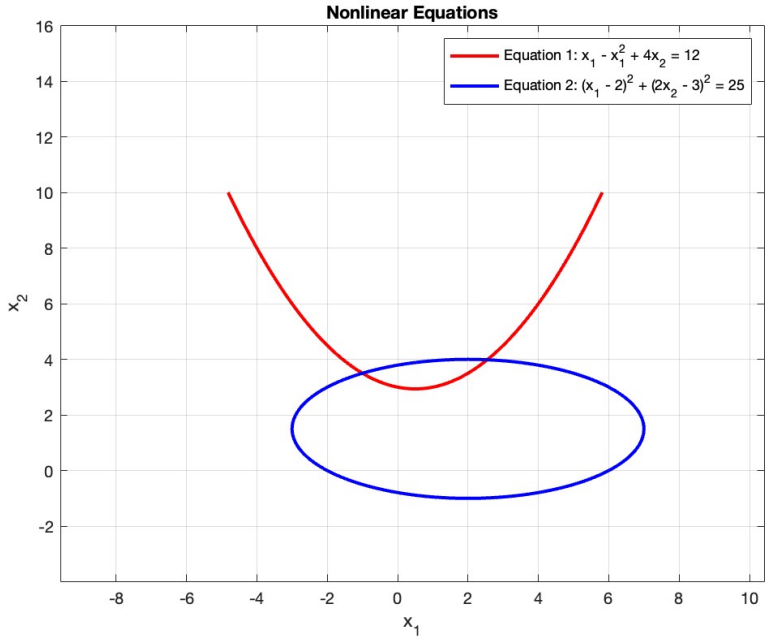

Figure 5: A graph of Nonlinear Equations with x_1 vs. x_2 for Equation 1: $x_1 - x_1^2 + 4x_2 = 12$ and Equation 2: $(x_1 - 2)^2 + (2x_2 - 3)^2 = 25$.

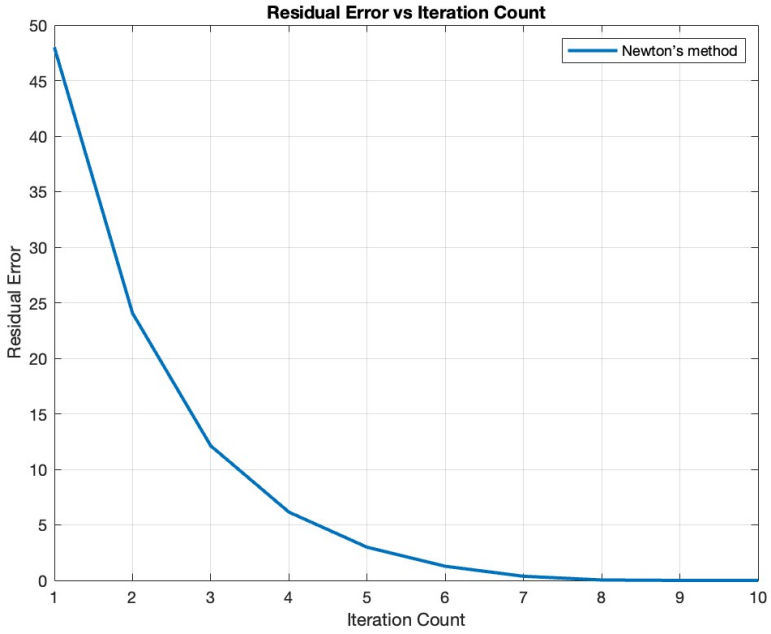

Figure 6: A graph showing the residual error (y-axis) vs iteration count (x-axis) for Newton's method.

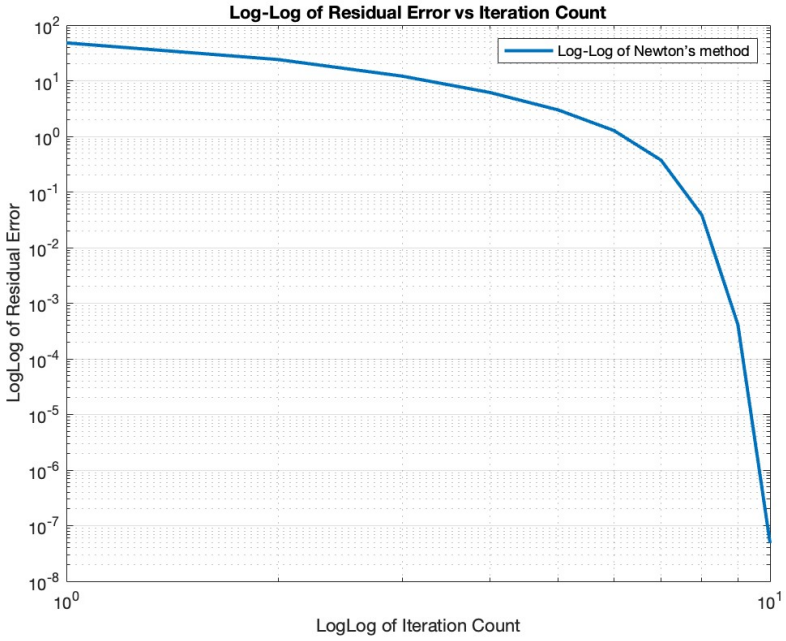

Figure 7: A Log-Log graph showing log-log the residual error (y-axis) vs log-log iteration count (x-axis) for Newton's method.

5 Academic Integrity

On my personal integrity as a student and member of the UCD community, I have not given nor received any unauthorized assistance on this assignment.

6 Appendix

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% Problem 1 %%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clc;clear;close all;
a = 0;
b = 30;
tol = 1e-8;
N0 = 1000;

[p1_bisec, abs_err1_bisec] = BisectionMethod(a, b, tol, N0);

iteration_count = 1:length(abs_err1_bisec);
plot(iteration_count, abs_err1_bisec, '-', 'MarkerFaceColor',
    [0 0.447 0.741], 'LineWidth', 2)
legend('Bisection Method', 'TextColor', 'black')
xlabel('Iteration Count (n)')
ylabel('Error |x^{(n)} - x^*|')
title('Error vs. Iteration Count (n) ')

figure;
loglog(iteration_count, abs_err1_bisec, '-', 'MarkerFaceColor
   ', [0 0.447 0.741], 'LineWidth', 2)
legend('Log-log Bisection Method', 'TextColor', 'black')
xlabel('LogLog of Iteration Count (n)')
ylabel('LogLog of Error |x^{(n)} - x^*|')
title('LogLog of Error vs. Iteration Count (n) ')
function f = my_fun(x)
f = (x^(7/5) - 1) / 9 - 3; % define f(x)
end

function [p_bisec, abs_err_bisec] = BisectionMethod(a, b, tol
   , N0)
    i = 1;
    FA = my_fun(a);

    abs_err_bisec = [];
    while i <= N0
        p_bisec = a + (b - a) / 2;
        FP = (p_bisec^(7/5) - 1)/9 - 3;
        abs_err_bisec(end + 1) = abs(FP);

        if abs(FP) < tol || (b - a) / 2 < tol
            return
        end
```

```
        i = i + 1;

        if FA * FP > 0
            a = p_bisec;
            FA = FP;
        else
            b = p_bisec;
        end
    end

    if i > N0
        fprintf('Method failed after N0 iterations, N0=%d',
          N0);
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% Problem 2 %%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clc;clear;close all;
% Set the parameters
n = 100;  % Size of the matrix
N = 1000;  % Maximum number of iterations
tol = 1e-8;  % Tolerance
num_vectors = 25;  % Number of random vectors

% Initialize variables to store results
avg_time_gaussian = 0;
avg_time_lu = 0;
avg_time_jacobi = 0;
avg_time_gauss_seidel = 0;
avg_residual_gaussian = 0;
avg_residual_lu = 0;
avg_residual_jacobi = 0;
avg_residual_gauss_seidel = 0;

for i = 1:num_vectors
    % Generate a random vector b
    n = 100;
    b = randn(n, 1);
    R = randn (n , n );
    I = eye(n);
    P = I(randperm(n), :);
    A = 7 * I + (1/10) * (P + R);

    % Gaussian elimination without pivoting
    tic;
```

```
    [U, b_transformed] = GaussElimWithoutPivot(A, b);
    x_gaussian = BackwardSubstitution(U, b_transformed);
    time_gaussian = toc;
    residual_gaussian = norm(A * x_gaussian - b);
    avg_time_gaussian = avg_time_gaussian + time_gaussian;
    avg_residual_gaussian = avg_residual_gaussian +
       residual_gaussian;

    % LU factorization
    [L, U] = luFactorization(A);

    tic;
    x_lu = luSolve(L, U, b);
    time_lu = toc;
    residual_lu = norm(A * x_lu - b);
    avg_time_lu = avg_time_lu + time_lu;
    avg_residual_lu = avg_residual_lu + residual_lu;

    % Jacobi method
    x0 = zeros(n, 1);  % Initial approximation
    tic;
    [x_jacobi, ~, ~, error_jacobi] = Jacobi(A, b, tol, N, x0)
       ;
    time_jacobi = toc;
    residual_jacobi = norm(A * x_jacobi - b);
    avg_time_jacobi = avg_time_jacobi + time_jacobi;
    avg_residual_jacobi = avg_residual_jacobi +
       residual_jacobi;

    % Gauss-Seidel method
    tic;
    [x_gauss_seidel, ~, ~, error_gauss_seidel] = GaussSeidel(
       A, b, tol, N, x0);
    time_gauss_seidel = toc;
    residual_gauss_seidel = norm(A * x_gauss_seidel - b);
    avg_time_gauss_seidel = avg_time_gauss_seidel +
       time_gauss_seidel;
    avg_residual_gauss_seidel = avg_residual_gauss_seidel +
       residual_gauss_seidel;
end

% Compute average values
avg_time_gaussian = avg_time_gaussian / num_vectors;
avg_time_lu = avg_time_lu / num_vectors;
avg_time_jacobi = avg_time_jacobi / num_vectors;
avg_time_gauss_seidel = avg_time_gauss_seidel / num_vectors;
avg_residual_gaussian = avg_residual_gaussian / num_vectors;
avg_residual_lu = avg_residual_lu / num_vectors;
```

```
avg_residual_jacobi = avg_residual_jacobi / num_vectors;
avg_residual_gauss_seidel = avg_residual_gauss_seidel /
   num_vectors;

% Display the results in a table
methods = {'Gaussian Elimination.', 'LU Factorization', '
   Jacobi Method', 'Gauss-Seidel'};
avg_times = [avg_time_gaussian, avg_time_lu, avg_time_jacobi,
    avg_time_gauss_seidel];
avg_residuals = [avg_residual_gaussian, avg_residual_lu,
   avg_residual_jacobi, avg_residual_gauss_seidel];
table(methods', avg_times', avg_residuals', 'VariableNames',
   {'Method', 'Average Time', 'Average Error'})


%-----Gaussian elimination without pivoting------
% this function is on the augmented matrix A | b
% and returns the upper-triangular matrix U
% and returns the transformed right-hand side vector b
% A: the matrix of coefficients
% b: the right-hand side vector
% U: the upper triangular matrix

function [U, b] = GaussElimWithoutPivot(A, b)
    n = size(A, 1);
    U = A;
    % Gaussian elimination without pivoting
    for k = 1:n-1
        for p = k:n
            if U(p, k) ~= 0
                break;
            end
            fprintf("No unique solution exists.");
            return;
        end
        for i = k+1:n
            factor = U(i,k) / U(k,k);
            U(i,k:n) = U(i,k:n) - factor * U(k,k:n);
            b(i) = b(i) - b(k) * factor;
        end
    end
end


%--------------- Back Substitution-------------
% this function is to solve the system Ux = b for x, where
% U: the upper triangular matrix
% b: the right-hand side vector
```

```
% x: the solution vector
function x = BackwardSubstitution(U,b)

    n = size(U,1);
    if size(b, 1) ~= n
        error('Matrix dimensions are inconsistent.')
    end
    x = zeros(n,1);

    for i = n:-1:1
        x(i) = b(i);
        for j = i+1:n
            x(i) = x(i) - U(i,j)*x(j);
        end
        x(i) = x(i)/U(i,i);
    end

end


%--------------- Forward Substitution--------------
% this function is to solve the system Ly = b for y, where
% L: the lower-triangular matrix
% b: the right-hand side vector
% y: the solution vector

function x = ForwardSubstitution(L, b)
    n = size(L,1);
    x = zeros(n,1);
    x(1,1) = b(1,1)/L(1,1);
    for i = 2:n
        sum = 0;
        for j = 1:i-1
            sum = sum + L(i,j)*x(j,1);
        end
        x(i,1) = (b(i,1)-sum)/L(i,i);
    end
end

%--------LU factorization-------------------
function [L, U] = luFactorization(A)
% this function is to solve the LU factorization of a nxn
   matrix A
% A: the nxn matrix
% L: the lower-triangular matrix
% U: the upper-triangular matrix
    n = size(A,1);
    % Initialize L and U matrices
```

```
    L = eye(n);
    U = A;
    % Gaussian elimination without pivoting
    for k = 1:n-1
        if U(k,k) == 0
           disp('Factorization impossible');
        end
        for i = k+1:n
            factor = U(i,k) / U(k,k);
            L(i,k) = factor;
            U(i,k:n) = U(i,k:n) - factor * U(k,k:n);
        end
    end
end

%----------------luSolver----------------
function x2 = luSolve(L, U, b)
% this function is to solve the system LUx = b for x, where
% L: the lower-triangular matrix
% U: the upper-triangular matrix
% b: the right-hand side vector
% x: the solution vector
    y = ForwardSubstitution(L, b);
    x2 = BackwardSubstitution(U, y);
end

%---------------Jacobi--------------
function [x, k, t,error] = Jacobi(A, b, tol, N, x0)
    % A: Matrix
    % b: Matrix
    % tol: Tolerance
    % N: Maximum iteration
    % x0:Initial approximation
    k = 1;
    error=[];
    [n, m] = size(A);
    x = zeros(n, 1);
    t = zeros(n, N);

    while k <= N
        x0 = x;
        error(end+1)=norm(A*x-b);% When generating this first
            two graph, commented out errors
        for i = 1:n
            sum = 0;
            for j = 1:n
                if j ~= i
                    sum = sum + A(i, j) * x0(j);
```

```
                end
            end
            x(i) = (1 / A(i, i)) * (b(i) - sum);
        end


        if norm(x - x0) < tol
            error(end+1)=norm(A*x-b);% When generating this
               first two graph, commented out errors
            break;
        end

        k = k + 1;
    end
end

%---------------GaussSeidel--------------

function [x, k, t,error] = GaussSeidel(A, b, tol, N, x0)
    % A: Matrix
    % b: Matrix
    % tol: Tolerance
    % N: Maximum iteration
    % x0:Initial approximation
    k = 1;
    error=[];
    [n, m] = size(A);
    x = zeros(n, 1);
    t = [];  % Store all iterations of x

    while k <= N
        x0 = x;
        error(end+1)=norm(A*x-b); % When generating this
           first two graph, commented out errors
        for i = 1:n
            sum = 0;
            for j = 1:i-1
                sum = sum + A(i, j) * x(j);
            end
            for j = i+1:n
                sum = sum + A(i, j) * x0(j);
            end
            x(i) = (b(i) - sum) / A(i, i);
        end


        if norm(x - x0) < tol
```

```
            error(end+1)=norm(A*x-b); % When generating this
               first two graph, commented out errors
            break
        end
        k = k + 1;
    end
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%% Problem 3 %%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clc;clear;close all;

A = -4 * eye(100);
R = (1/2) * randn(100, 100);
B = R + R';
A = A + B;
[n, ~] = size(A);
N = 1000;
x0 = ones(n, 1);
tol = 1e-8;
[ev_error, eigenvalue_error, k] = InversePowerMethod(A, x0,
   tol, N);

% Plot the eigenvector error
figure;
plot(1:length(ev_error), ev_error, '-', 'MarkerFaceColor', [0
    0.447 0.741], 'LineWidth', 1.5);
xlabel('Iteration Count (n)');
ylabel('Eigenvector Error ||Ax^{(n)} - \lambda^{(n)}x^{(n)}||
   ');
title('Eigenvector Error vs. Iteration Count (n)');
legend('The inverse power method');

% Plot the eigenvalue difference
figure;
plot(1:length(eigenvalue_error), eigenvalue_error, '-', '
   MarkerFaceColor', [0 0.447 0.741], 'LineWidth', 1.5);
xlabel('Iteration Count (n)');
ylabel('|\lambda^{(n)} - 1|');
title('|\lambda^{(n)} - 1| vs. Iteration Count (n)');
legend('The inverse power method');
```

```
function [ev_error, eigenvalue_error, k] = InversePowerMethod
   (A, x, tol, N)
    n = size(A, 1);
    q = 1; % Find the eigenvalue which is closest to 1.
    k = 1;
    mu = q; % Set mu(0) to 1 for the first iteration

    ev_error = [];
    eigenvalue_error = [];

    for p = 1:n
        if abs(x(p)) == norm(x, inf)
            xp=x(p);
            break ;
        end
    end
    %[~, i] = max(abs(x));
    %xp = x(i);
    x = x / xp;

    while k <= N
        [L, U] = LU_Fac_Fun(A - q * eye(n));
        sol = Forward_Sub_Fun(L, x);
        y = Backward_Sub_Fun(U, sol);
        [~, i] = max(abs(y)); % Find the index of the maximum
            absolute value in the vector y
        yp = y(i); % Find the element with the maximum
           absolute value in y
        prev_mu = mu;
        mu = 1 / yp + q;
        err = abs(prev_mu - mu);
        x = y / yp; % Update x
        eVector_err = norm(A * x - mu * x); % ||Ax - lambda*x
           ||
        ev_error = [ev_error eVector_err];
        eigenvalue_error = [eigenvalue_error abs(mu - 1)];
        if err < tol
            return
        end
        k = k + 1;
    end

    error('Maximum number of iterations exceeded');
end

% -------------- Back Substitution --------------
% This function solves the system Ux = b for x, where
```

```
% U: the upper triangular matrix
% b: the right-hand side vector
% x: the solution vector
function x = Backward_Sub_Fun(U, b)
    n = size(U, 1);
    if size(b, 1) ~= n
        error('Matrix dimensions are inconsistent.');
    end
    x = zeros(n, 1);

    for i = n:-1:1
        x(i) = b(i);
        for j = i+1:n
            x(i) = x(i) - U(i, j) * x(j);
        end
        x(i) = x(i) / U(i, i);
    end
end

% -------------- Forward Substitution --------------
% This function solves the system Ly = b for y, where
% L: the lower-triangular matrix
% b: the right-hand side vector
% y: the solution vector
function x = Forward_Sub_Fun(L, b)
    n = size(L, 1);
    x = zeros(n, 1);
    x(1) = b(1) / L(1, 1);
    for i = 2:n
        sum = 0;
        for j = 1:i-1
            sum = sum + L(i, j) * x(j);
        end
        x(i) = (b(i) - sum) / L(i, i);
    end
end

% -------- LU factorization -------------------
function [L, U] = LU_Fac_Fun(A)
    % This function performs the LU factorization of an nxn
       matrix A
    % A: the nxn matrix
    % L: the lower-triangular matrix
    % U: the upper-triangular matrix
    n = size(A, 1);
    % Initialize L and U matrices
    L = eye(n);
    U = A;
```

```
    % Gaussian elimination without pivoting
    for k = 1:n-1
        if U(k, k) == 0
            disp('Factorization impossible');
        end
        for i = k+1:n
            factor = U(i, k) / U(k, k);
            L(i, k) = factor;
            U(i, k:n) = U(i, k:n) - factor * U(k, k:n);
        end
    end
end

%----------------luSolver----------------
function x2 = luSolve(L, U, b)
% this function is to solve the system LUx = b for x, where
% L: the lower-triangular matrix
% U: the upper-triangular matrix
% b: the right-hand side vector
% x: the solution vector
    y = ForwardSubstitution(L, b);
    x2 = BackwardSubstitution(U, y);
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% Problem 4 %%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clc;clear;close all;
% Define the grid of x1 and x2 values
[x1, x2] = meshgrid(-10:0.1:10);

% Evaluate the equations
f1 = x1 - x1.^2 + 4*x2 - 12;
f2 = (x1 - 2).^2 + (2*x2 - 3).^2 - 25;

% Plot the equations
figure;
contour(x1, x2, f1, [0 0], 'r', 'LineWidth', 2);
hold on;
contour(x1, x2, f2, [0 0], 'b', 'LineWidth', 2);
xlabel('x_{1}');
ylabel('x_{2}');
legend('Equation 1: x_{1} - x_{1}^2 + 4x_{2} = 12 ', '
   Equation 2: (x_{1} - 2)^2 + (2x_{2} - 3)^2 = 25');
```

```
title('Nonlinear Equations');
grid on;
hold off;

% Set the initial guess, tolerance, and maximum number of
   iterations
x0 = [0; 0];
TOL = 1e-6;
N = 100;

% Call the Newton's method function
[x, k, err] = Newton(x0, TOL, N);

% Plot the residual error vs iteration count
figure;
plot(1:k, err, '-', 'MarkerFaceColor', [0 0.447 0.741], '
   LineWidth', 2);
xlabel('Iteration Count');
ylabel('Residual Error');
title('Residual Error vs Iteration Count');
legend('Newtons method');
grid on;

% Plot the residual error using a log-log scale
figure;
loglog(1:k, err, '-', 'MarkerFaceColor', [0 0.447 0.741], '
   LineWidth', 2);
xlabel('LogLog of Iteration Count');
ylabel('LogLog of Residual Error');
title('Log-Log of Residual Error vs Iteration Count');
legend('Newtons method');
grid on;


% ----- Newton's Method --------
function [x, k, err] = Newton(x, TOL, N)
    k = 1;
    err = [];

    while k <= N
        F_x = my_fun(x(1), x(2));
        J_x = Jacobian(x(1), x(2));

        [L, U] = LUFactorization(J_x);
        L_v = [L, -F_x];
        L_v_sol = ForSubstitution(2, L_v);
        U_y = [U, L_v_sol];
```

```
        y = BackSubstitution(2, U_y);
        x = x + y;

        err(end+1) = norm(y, Inf);

        if norm(y) < TOL
            return;
        end

        k = k + 1;
    end

    fprintf("Maximum number of iterations exceeded for Newton
       's Method\n");
end

function F = my_fun(x1, x2)
    f1 = x1 - x1.^2 + 4*x2 - 12;
    f2 = (x1 - 2).^2 + (2*x2 - 3).^2 - 25;
    F = [f1; f2];
end

function J = Jacobian(x1, x2)
    f11 = 1 - 2*x1;
    f12 = 4;
    f21 = 2*(x1 - 2);
    f22 = 4*(2*x2-3);
    J = [f11, f12; f21, f22];
end

% ----- Back Substitution -------
function x = BackSubstitution(n, b)
    x = zeros(n, 1);
    x(n, 1) = b(n, n+1) / b(n, n);

    for i = n-1:-1:1
        sum = 0;

        for j = i+1:n
            sum = sum + b(i, j) * x(j, 1);
        end

        x(i, 1) = (b(i, n+1) - sum) / b(i, i);
    end
end

% ----- Forward Substitution -------
function x = ForSubstitution(n, b)
```

```
    x = zeros(n, 1);
    x(1, 1) = b(1, n+1) / b(1, 1);

    for i = 2:n
        sum = 0;

        for j = 1:i-1
            sum = sum + b(i, j) * x(j, 1);
        end

        x(i, 1) = (b(i, n+1) - sum) / b(i, i);
    end
end

%--------LU factorization-------------------
function [L, U] = LUFactorization(A)
    n = size(A,1);
    % Initialize L and U matrices
    L = eye(n);
    U = A;
    % Gaussian elimination without pivoting
    for k = 1:n-1
        if U(k,k) == 0
            disp('Factorization impossible');
        end
        for i = k+1:n
            factor = U(i,k) / U(k,k);
            L(i,k) = factor;
            U(i,k:n) = U(i,k:n) - factor * U(k,k:n);
        end
    end
end
```