Problem 1. For an $n \times n$ matrix A, where

$$A_{1,1} = 2, \quad A_{1,2} = -1, \quad A_{n,n} = 2, \quad A_{n-1,n} = -1$$

and

$$A_{i,i} = 2, \quad A_{i,i+1} = -1, \quad A_{i,i-1} = -1, \quad \forall i \neq 1, n$$

and $A_{i,j} = 0$ otherwise. Compute its LU factorization with MATLAB/Python. Can the LU factorization be obtained faster than $O(n^3)$ complexity? If so, what would the algorithm be?

Solution: The matrix A is the following form:

$$A = \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 & 0 & 0 \\ -1 & 2 & -1 & \ddots & \vdots & \vdots & \vdots \\ 0 & -1 & 2 & \ddots & -1 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & 2 & -1 & 0 \\ 0 & \cdots & -1 & 2 & -1 & \ddots & \vdots \\ 0 & \cdots & 0 & -1 & 2 & -1 & \\ 0 & \cdots & 0 & 0 & -1 & 2 & \end{bmatrix}$$

The code is as follows:

```
A = generate_tridiagonal_matrix(5);
[L_A, U_A] = LU_decomp(A);


function A = generate_tridiagonal_matrix(n)
    % Generates an n x n symmetric tridiagonal matrix with:
    %  2 on the diagonal and -1 on the sub- and super-diagonals

    e = ones(n,1);
    A = 2*diag(e) - diag(e(1:end-1),1) - diag(e(1:end-1),-1);
end

function[L, U] = LU_decomp(A)
    n_tuple = size(A);
    assert(n_tuple(1) == n_tuple(2));
    n = n_tuple(1);
    U = A;
    L = eye(n_tuple(1), n_tuple(2));
    for j = 1:n
        for i = j+1:n
            if U(j,j) == 0
                error("entry in U is 0");
            end
            L(i,j) = U(i,j) / U(j,j);
            U(i, j:n) = U(i,j:n) - L(i,j) * U(j,j:n);
```

```
            end
        end
    end
```

Note that given the matrix is a tridiagonal matrix, we know that we can exploit the structure of the matrix to get a faster run time. Let m denote the main diagonal, l denote the lower diagnol and u denote the upper diagnol. We can see that it suffices to only use a singular for loop and and to construct the LU decomposition as we do not have to iterate through the columns of the matrix, other along the main diagnol. The pseudocode is as follows:

Algorithm 1 LU Decomposition of a Symmetric Tridiagonal Matrix

Require: Main diagonal $a[1..n]$, sub/super-diagonal $b[1..n-1]$
Ensure: Subdiagonal $l[1..n-1]$ of L, diagonal $u[1..n]$ of U

$u[1] \leftarrow a[1]$
for $i = 2$ to n **do**
  $l[i-1] \leftarrow b[i-1]/u[i-1]$
  $u[i] \leftarrow a[i] - l[i-1] \cdot b[i-1]$
end for
return l, u

$\square$

Problem 2. Implement a gradient descent method (with fixed step size) to solve

$$\min_{x\in\mathbb{R}^n} \frac{1}{2}(x - x_*)^T A(x - x_*)$$

where $x_* = [1, 2, \ldots, n]^T$. What is the complexity per step of gradient descent? (Show the complexity as n grows using a plot). Furthermore, by repeating the experiment for different n, extract the rate of convergence and show its dependency on n using big-O notation.

Solution:

```
n_vals = 10:10:400;
tolerance = 1.5e-5;

iters_vec = zeros(length(n_vals), 1);
total_flops_vec = zeros(length(n_vals), 1);
rel_error_mat = cell(length(n_vals), 1);

for i = 1:length(n_vals)
    disp("grad");
    disp(i)
    n = n_vals(i);
    [num_iter, total_flops, rel_errors] = grad_descent(n,
       tolerance);
    total_flops_vec(i) = total_flops;
    iters_vec(i) = num_iter;
    rel_error_mat{i} = rel_errors;
end

rho_vec = zeros(length(n_vals), 1);  % Store convergence
   rates

for i = 1:length(n_vals)
    rel_errors = rel_error_mat{i};
    k = 0:length(rel_errors)-1;
    log_err = log(rel_errors(:));
    coeffs = polyfit(k', log_err, 1);
    rho_vec(i) = coeffs(1);
end

figure;

% Plot 1: Total FLOPs vs n
subplot(2, 2, 1);
loglog(n_vals, total_flops_vec, 'o-', 'LineWidth', 2);
xlabel('Matrix size n');
ylabel('Total FLOPs');
title('Log-Log: Total FLOPs vs. n');
grid on;
```

```
        hold on;

        log_n = log(n_vals);
        log_flops = log(total_flops_vec);
        coeffs_flops = polyfit(log_n, log_flops, 1);
        b = coeffs_flops(1);
        a = exp(coeffs_flops(2));
        fit_flops = a * n_vals.^b;
        loglog(n_vals, fit_flops, '--', 'LineWidth', 1.5);
        text(n_vals(round(end * 0.3)), fit_flops(round(end * 0.3)),

        % Plot 2: Relative Error vs Iteration
        subplot(2, 2, 2);
        hold on;
        for i = 1:length(n_vals)
            rel_errors = rel_error_mat{i};
            k = 0:length(rel_errors)-1;
            semilogy(k, rel_errors, 'DisplayName', sprintf('n = %d',
                n_vals(i)));
        end
        xlabel('Iteration');
        ylabel('Relative Error (log scale)');
        title('Semi-Log: Relative Error vs. Iteration');
        legend('show', 'Location', 'northeastoutside');
        grid on;

        % Plot 3: |rho| vs. n
        subplot(2, 2, 3);
        loglog(n_vals, abs(rho_vec), 'o-', 'LineWidth', 2);
        xlabel('Matrix size n');
        ylabel('|Convergence Rate|');
        title('Log-Log: | rho | vs. n');
        grid on;

        % Plot 4: Fit |rho| vs. n
        subplot(2, 2, 4);
        loglog(n_vals, abs(rho_vec), 'o-', 'LineWidth', 2);
        xlabel('Matrix size n');
        ylabel('|Convergence Rate|');
        title('Log-Log: Fitted | rho| vs. n');
        grid on;
        hold on;

        log_rho = log(abs(rho_vec));
        coeffs_rho = polyfit(log_n, log_rho, 1);
        slope = coeffs_rho(1);
        a_rho = exp(coeffs_rho(2));
```

```
        fit_rho = a_rho * n_vals.^slope;
        loglog(n_vals, fit_rho, '--', 'LineWidth', 1.5);
        text(n_vals(round(end * 0.3)), fit_rho(round(end * 0.3)),
            'FontSize', 10, 'BackgroundColor', 'white');

        % User defined function
        function [num_iter, total_flops, rel_errors] = grad_descent(n, tolerance)
            A = generate_tridiagonal_matrix(n);
            iterations = 0;
            x_star = (1:n)';
            x_current = zeros(n,1);
            eigenvalues = eig(A);
            s = 2 / (min(eigenvalues) + max(eigenvalues));
            rel_errors = [];
            norm_x_star = norm(x_star);
            rel_errors(end+1) = norm(x_current - x_star) / norm_x_star;
            tic;
            while true
                iterations = iterations + 1;
                grad = A * (x_current - x_star);
                x_next = x_current - s * grad;

                % Track relative error
                rel_error = norm(x_next - x_star) / norm_x_star;
                rel_errors(end+1) = rel_error;

                if rel_error < tolerance
                    break;
                end
                x_current = x_next;
            end
            flops_per_iteration = n^2;
            num_iter = iterations;
            total_flops = flops_per_iteration * num_iter;
        end
```

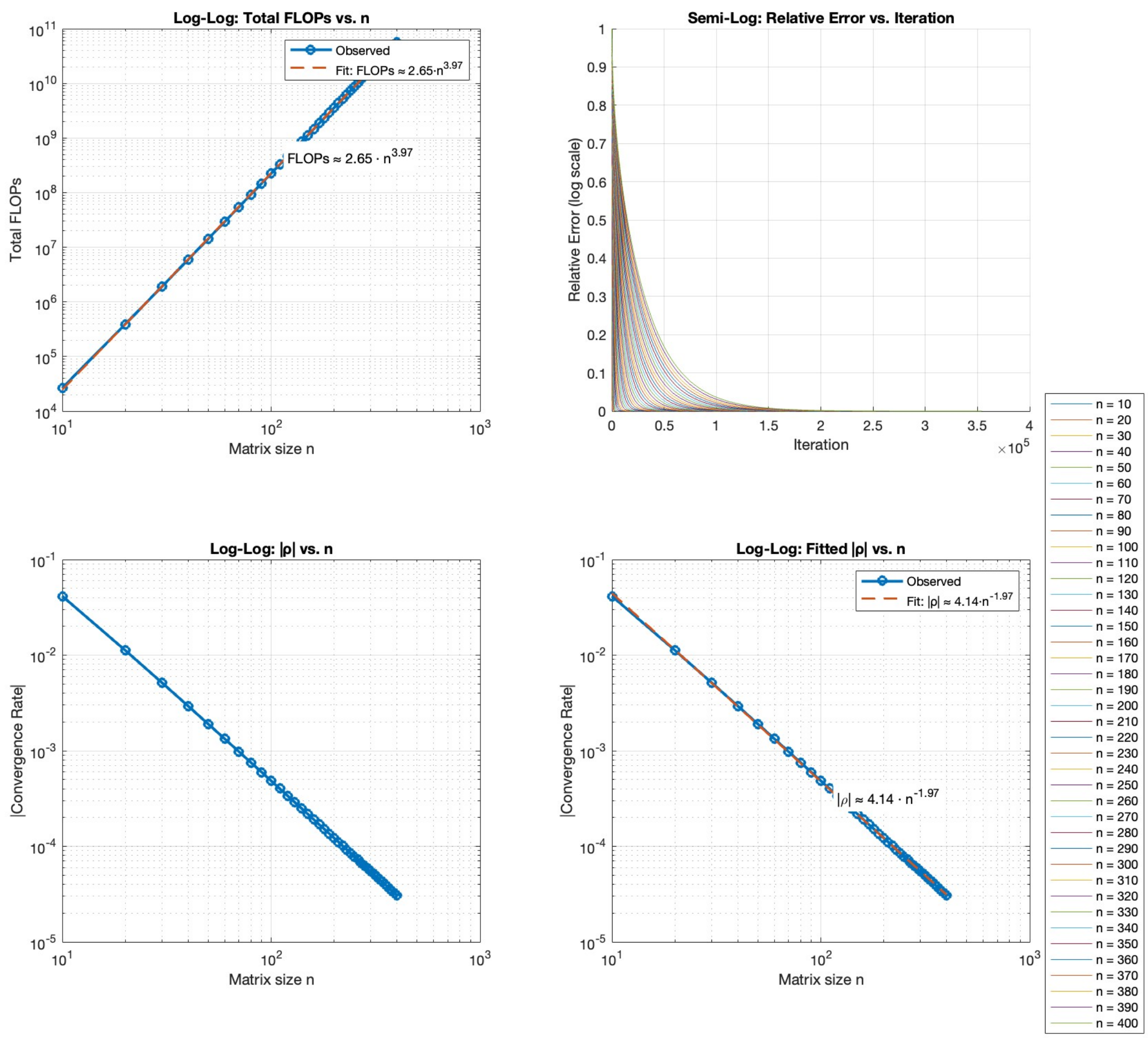

Figure 1: Gradient descent graphs

We can see that the computational complexity of this algorithm is $\mathcal{O}(n^4)$ from the top right graph. This implies that the rate of iteration increasing is $\mathcal{O}(n^2)$ as within the actual iteration, we do a matrix vector operation, $\mathcal{O}(n^2)$. Thus, the complexity for each step is $\mathcal{O}(n^2)$ the bottom right graph, we know that convergence rate is $\mathcal{O}(n^2)$ as we can extacrt the rate of convergence from the negative of the exponent. This is proved in the following, and let e_k denote the relative error:

$$\rho \;=\; \max_i \bigl|1 - s\,\lambda_i(A)\bigr| \;=\; \frac{\lambda_{\max}(A) - \lambda_{\min}(A)}{\lambda_{\max}(A) + \lambda_{\min}(A)} \;=\; \frac{\kappa - 1}{\kappa + 1} \;=\; 1 - \Theta\Bigl(\frac{1}{n^2}\Bigr).$$

Hence the error after k steps satisfies

$$\|x^{(k)} - x^*\| \;\le\; \rho^k \,\|x^{(0)} - x^*\|.$$

To drive $\|x^{(k)} - x^*\| \leq \epsilon$, solve

$$\rho^k \leq \frac{\epsilon}{\|x^{(0)} - x^*\|} \implies k \geq \frac{\ln\big(\epsilon / \|x^{(0)} - x^*\|\big)}{\ln(\rho)} = O\big(n^2 \ln \tfrac{1}{\epsilon}\big).$$

since we are using a fixed ϵ, it suffices to conclude that our algorithm converges in $\mathcal{O}\left(n^2\right)$ time. $\square$

Problem 3. Repeat Problem 2 by implementing a conjugate gradient method.

Solution:

```
n_vals = 10:10:2000;
tolerance = 1.5e-5;

iters_vec = zeros(length(n_vals), 1);
total_flops_vec = zeros(length(n_vals), 1);
rho_vec = zeros(length(n_vals), 1);
rel_error_mat = cell(length(n_vals), 1);

for i = 1:length(n_vals)
    disp("conj");
    disp(i)
    n = n_vals(i);
    [num_iter, total_flops, rel_errors] = conj_grad_descent(
       n, tolerance);

    iters_vec(i) = num_iter;
    total_flops_vec(i) = total_flops;
    rel_error_mat{i} = rel_errors;

    % Estimate convergence rate from slope of log(error) vs.
       iteration
    k = 0:length(rel_errors)-1;
    log_err = log(rel_errors(:));
    coeffs = polyfit(k', log_err, 1);
    rho_vec(i) = coeffs(1);  % Negative convergence slope
end

% ---- Start Combined Plot ----
figure;

% ---- Plot 1: Log-Log Total FLOPs vs. n ----
subplot(2,2,1);
loglog(n_vals, total_flops_vec, 'o-', 'LineWidth', 2);
xlabel('Matrix size n');
ylabel('Total FLOPs');
title('Log-Log: Total FLOPs vs. n');
grid on;
hold on;

log_n = log(n_vals);
log_flops = log(total_flops_vec);
coeffs_flops = polyfit(log_n, log_flops, 1);
b = coeffs_flops(1);
a = exp(coeffs_flops(2));
```

```
        fit_flops = a * n_vals.^b;
        loglog(n_vals, fit_flops, '--', 'LineWidth', 1.5);
        text(n_vals(round(end*0.3)), fit_flops(round(end*0.3)), ...


        % ---- Plot 2: Semi-log Relative Error vs. Iteration ----
        subplot(2,2,2);
        hold on;
        for i = 1:3:length(n_vals)  % Plot fewer curves for
           readability
            rel_errors = rel_error_mat{i};
            k = 0:length(rel_errors)-1;
            semilogy(k, rel_errors, 'DisplayName', sprintf('n = %d',
                n_vals(i)));
        end
        xlabel('Iteration');
        ylabel('Relative Error (log scale)');
        title('Semi-Log: Relative Error vs. Iteration');
        legend('show', 'Location', 'northeastoutside');
        grid on;

        % ---- Plot 3: Log-Log |rho| vs. n ----
        subplot(2,2,3);
        loglog(n_vals, abs(rho_vec), 'o-', 'LineWidth', 2);
        xlabel('Matrix size n');
        ylabel('|Convergence Rate|');
        title('Log-Log: |rho| vs. n');
        grid on;

        % ---- Plot 4: Fit |rho| approx a n^b ----
        subplot(2,2,4);
        loglog(n_vals, abs(rho_vec), 'o-', 'LineWidth', 2);
        xlabel('Matrix size n');
        ylabel('|Convergence Rate|');
        title('Log-Log: Fitted |rho| vs. n');
        grid on;
        hold on;

        log_rho = log(abs(rho_vec));
        coeffs_rho = polyfit(log_n, log_rho, 1);
        slope = coeffs_rho(1);
        a_rho = exp(coeffs_rho(2));
        fit_rho = a_rho * n_vals.^slope;
        loglog(n_vals, fit_rho, '--', 'LineWidth', 1.5);
        text(n_vals(round(end*0.3)), fit_rho(round(end*0.3)), ...
            'FontSize', 10, 'BackgroundColor', 'white');

```

```
    sgtitle('Conjugate Gradient Convergence Analysis (Tridiagonal A)');


    % User defined function

    function A = generate_tridiagonal_matrix(n)
        % Generates an n x n symmetric tridiagonal matrix with:
        %  2 on the diagonal and -1 on the sub- and super-diagonals

        e = ones(n,1);
        A = 2*diag(e) - diag(e(1:end-1),1) - diag(e(1:end-1),-1);
    end

    function [num_iter, total_flops, rel_errors] = conj_grad_descent(m, tolerance)
        A = generate_tridiagonal_matrix(m);

        initial_guess = zeros(m, 1);
        x_star = (1:m)';
        b = A * x_star;

        residual_old = b - A * initial_guess;
        direction = residual_old;
        rs_old = residual_old' * residual_old;

        iter = 0;
        rel_errors = [];
        norm_x_star = norm(x_star);

        while sqrt(rs_old) > tolerance
            Ap = A * direction;
            alpha = rs_old / (direction' * Ap);

            initial_guess = initial_guess + alpha * direction;
            residual_old = residual_old - alpha * Ap;
            rel_error_k = norm(initial_guess - x_star) / norm_x_star;
            rel_errors = [rel_errors, rel_error_k];

            rs_new = residual_old' * residual_old;
            beta = rs_new / rs_old;

            direction = residual_old + beta * direction;
            rs_old = rs_new;
```

```

                iter = iter + 1;
            end

            num_iter = iter;
            total_flops = num_iter * m^2;  % dense matrix assumption; for tridiagonal use ~5m
        end
```

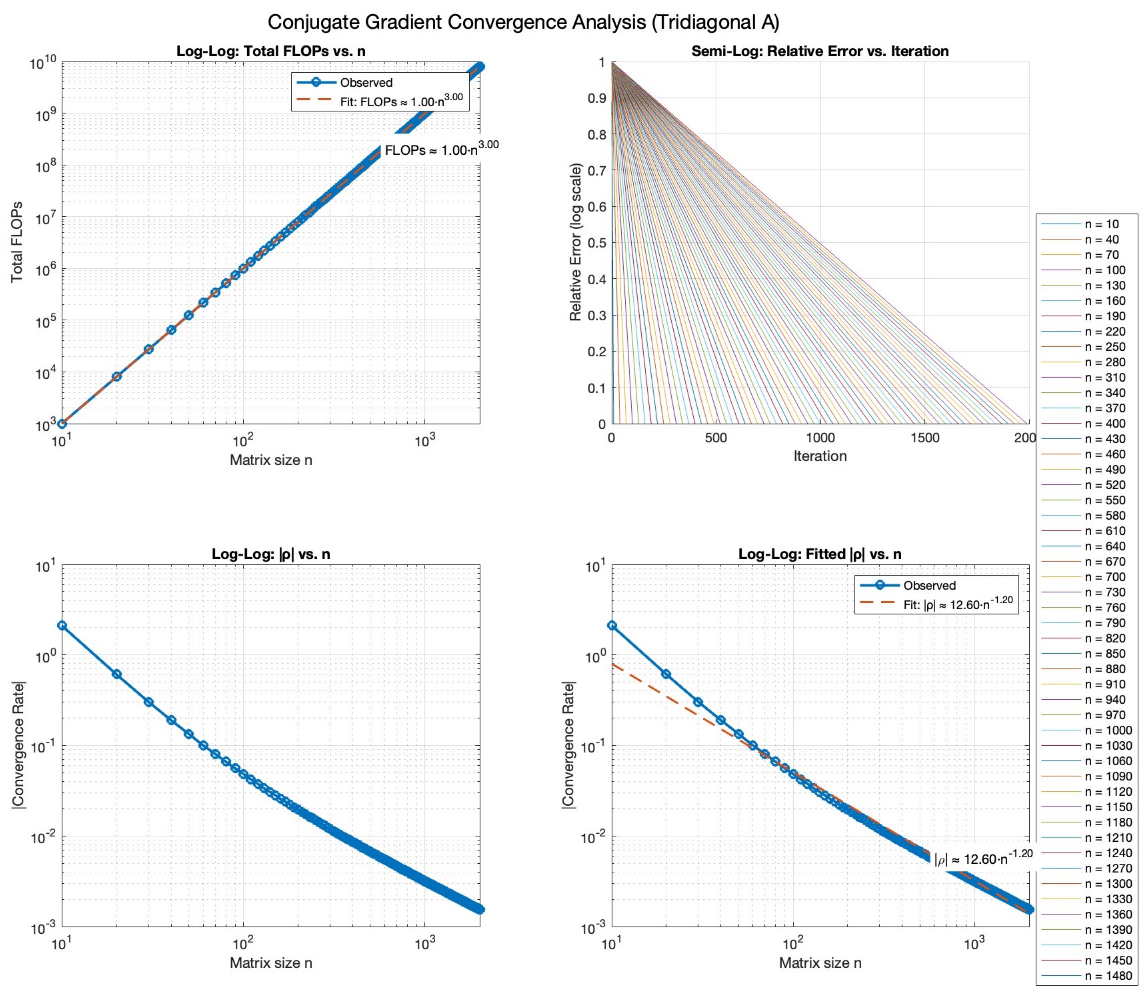

Figure 2: Conjugate Gradient descent method

We can see from the top right graph, the complexity for the whole algorithm is $\mathcal{O}(n^3)$. Since we know that in conjugate gradient, we eliminate the n components from the residual vector, the per step complexity must be $\mathcal{O}(n^2)$, which matches theory as we have a matrix vector product. Following a similar logic above, we know that convergence rate is approximately $\mathcal{O}(n)$. $\square$

Problem 4. Suppose in a gradient descent scheme, an error ϵ_k occurs:

$$x^{(k+1)} = x^{(k)} - s_k \nabla f(x^{(k)}) + \epsilon_k$$

where $\|\epsilon_k\| \leq \epsilon$, and $f = \frac{1}{2}(x - x_*)^T A(x - x_*)$ with A positive definite. Show the scheme can converge with a suitable choice of s_k. Make assumptions on ϵ if ncessary.

Solution: Note that $\nabla f = A(x - x^*)$, we begin with the following.

$$\begin{aligned}
\|x^{k+1} - x^*\|^2 &= \|x^k - s_k(A(x^k - x^*)) - x^* + \epsilon_k\| \\
&= \|(I - s_k A)(x^k - x^*) + \epsilon_k\| \\
&\leq (\|(I - s_k A)(x^k - x^*)\| + \|\epsilon_k\|)^2 \\
&\leq (\|(I - s_k A)(x^k - x^*)\|)^2 + 2\|\epsilon\|\|(I - s_k A)(x^k - x^*)\| + \epsilon^2
\end{aligned}$$

As proven in class, we know that if $s_k \in (0, \frac{2}{\lambda_k})$, by definition of the spectral norm, we can prove that

$$\|I - s_k A\| = \max_i |\lambda_i|$$

thus, we see that $\|I - s_k A\| < 1$. For notational sake, let $\rho = \|I - s_k A\|$ and let $r_k = \|x^k - x^*\|$, thus, note that

$$(\|(I - s_k A)(x^k - x^*)\|)^2 + 2\|\epsilon\|\|(I - s_k A)(x^k - x^*)\| + \epsilon^2 \leq r_k^2 \rho^2 + 2\epsilon\rho r_k + \epsilon^2$$

we can see that we are left with bound:

$$r_{k+1}^2 \leq r_k^2 \rho^2 + 2\epsilon\rho r_k + \epsilon^2$$

We aim to find a bound $R \geq r_i, \forall i \in \{0, 1, 2, \ldots, k\}$. Thus, it suffices to find a value such that:

$$\xi^2 = \xi^2 \rho^2 + 2\epsilon\rho\xi + \epsilon^2 \iff (1 - \xi^2)\rho^2 + 2\epsilon\rho\xi + \epsilon^2 = 0$$

Using the quadratic formula and simplyfing:

$$\xi = \frac{\epsilon(\rho + 1)}{1 - \rho^2} = \frac{\epsilon}{1 - \rho}$$

Note that this above term is a constant dependent on the maximal error term. Thus, we can see that given a $s \in (0, \frac{2}{\lambda_{max}})$, the algorithm converges for a sufficiently small epsilon. □