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

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

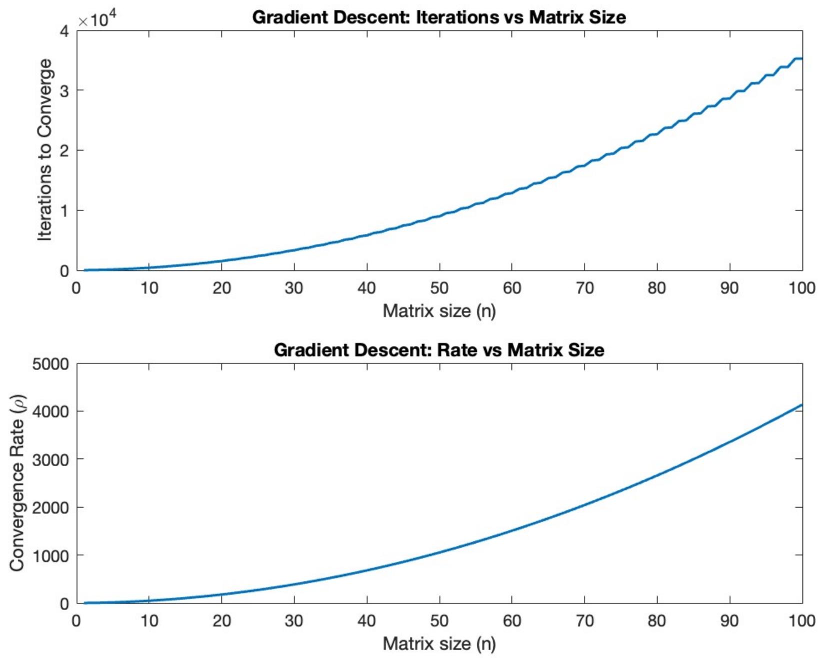
where $x_* = [1, 2, \ldots, n]^T$. What is the complexity per step of gradient descent? (Show the complexity as n grows using a plot). Furthermore, by repeating the experiment for different n, extract the rate of convergence and show its dependency on n using big-O notation.

Solution:

```
n = 100;
error = 0.000000015;
rate_vec = zeros(n, 1);
iterations_vec = zeros(n, 1);
for iteration = 1:n
    disp(iteration);
    [x, rate, iteration_num] = grad_descent_2(iteration, error);
    rate_vec(iteration) = rate;
    iterations_vec(iteration) = iteration_num;
end
figure;

subplot(2,1,1);
plot(1:n, iterations_vec, 'LineWidth', 1.5);
xlabel('Matrix size (n)');
ylabel('Iterations to Converge');
title('Gradient Descent: Iterations vs Matrix Size');

subplot(2,1,2);
plot(1:n, rate_vec, 'LineWidth', 1.5);
xlabel('Matrix size (n)');
ylabel('Convergence Rate (\rho)');
title('Gradient Descent: Rate vs Matrix Size');

function [x, rate, steps] = grad_descent_2(m, error)
    A = generate_tridiagonal_matrix(m);
    iterations = 0;
    converge = true;
    x_star = 1:m;
    x_intial = zeros(m, 1);
    x_next = zeros(m,1);
    eigenvalues = eig(A);
    s = 2 / (min(eigenvalues) + max(eigenvalues));

    while converge
        iterations = iterations + 1;
        x_next = x_intial - s * (A * (x_intial - x_star));
        if norm(x_next - x_intial) < error
```

```
            converge = false;    % stop loop
        else
            x_intial = x_next;  % continue updating
        end
    end
    x = x_next;
    rate = max(eigenvalues) / min(eigenvalues);
    steps = iterations;
end
```

$\square$



Using the graphs above, we see that the graphs are $\mathcal{O}\left(n^2\right)$

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

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.

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

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

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

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

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. $\square$