1 Problem 1

Using the Lyapunov method, analyze the stability of the following system for $-\infty < k < \infty$. Then, using MATLAB, verify your stability results numerically for arbitrary values of k.

$$\dot{x} = \begin{bmatrix} 0 & k \\ -1 & -2 \end{bmatrix} x \tag{1}$$

1.1 Firstly, we review the Lyapunov equations and theorems

- Lyapunov equations
  - We assume $A \in \mathbb{R}^{n\times n}$, $P = P^T \in \mathbb{R}^{n\times n}$. It follows that $Q = Q^T \in \mathbb{R}^{n\times n}$.
  - For $\dot{x} = Ax$, $V(z) = z^T Pz$, we have $\dot{V}(z) = -z^T Qz$, where P, Q satisfy Lyapunov equation $A^T P + PA + Q = 0$.
- Lyapunov theorems
  - If $P > 0$, $Q > 0$, then system is (globally asymptotically) stable.
  - If $P > 0$, $Q \geq 0$, and (Q, A) observable, then system is (globally asymptotically) stable.
  - If $P > 0$, $Q \geq 0$, then all trajectories of the system are bounded
  - If $Q \geq 0$, then the sublevel sets $\{z|z^T Pz \leq a\}$ are invariant. (These are ellipsoids if $P > 0$.)
  - If $P \not\geq 0$ and $Q \geq 0$, then A is not stable.
- Converse theorems
  - If A is stable, there exists a quadratic Lyapunov function $V(z) = z^T Pz$ that proves it, i.e., there exists $P > 0$, $Q > 0$ that satisfies the (continuous- or discrete-time) Lyapunov equation.
  - If A is stable and $Q \geq 0$, then $P \geq 0$.
  - If A is stable, $Q \geq 0$, and (Q, A) observable, then $P > 0$.
- Lyapunov equation solvability conditions
  - If A is stable, Lyapunov equation has a unique solution P, for any $Q = Q^T$.

1.2 After this review, we will proceed as follows:

$$A^T P + PA = -Q \tag{2}$$

$$\begin{bmatrix} 0 & -1 \\ k & -2 \end{bmatrix} \begin{bmatrix} p_{11} & p_{12} \\ p_{12} & p_{22} \end{bmatrix} + \begin{bmatrix} p_{11} & p_{12} \\ p_{12} & p_{22} \end{bmatrix} \begin{bmatrix} 0 & k \\ -1 & -2 \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \tag{3}$$

$$-2p_{12} = -1 \tag{4}$$
$$kp_{11} - p_{22} - 2p_{12} = 0 \tag{5}$$
$$2kp_{12} - 4p_{22} = -1 \tag{6}$$

$$p_{11} = \frac{5+k}{4k}, \qquad p_{12} = \frac{1}{2}, \qquad p_{22} = \frac{1+k}{4} \tag{7}$$

$$P = \begin{bmatrix} \frac{5+k}{4k} & \frac{1}{2} \\ \frac{1}{2} & \frac{1+k}{4} \end{bmatrix} \tag{8}$$

Now if P is a positive definite matrix it's okay. One way to tell if a matrix is positive definite is to calculate all the eigenvalues and just check to see if they're all positive. The only problem with this is that calculating eigenvalues can be a real pain, especially for large matrices.

Another way we can test for if a matrix is positive definite is we can look at its n upper left determinants. All the pivots will be positive if and only if $det(A_k) > 0$ for all $1 \leqslant k \leqslant n$. So, if all upper left kxk determinants of a symmetric matrix are positive, the matrix is positive definite.

$$\frac{5+k}{4k} > 0 \longrightarrow k > 0 \tag{9}$$
$$\begin{vmatrix} \frac{5+k}{4k} & \frac{1}{2} \\ \frac{1}{2} & \frac{1+k}{4} \end{vmatrix} = \frac{k^2+2k+5}{16k} > 0 \longrightarrow k > 0$$

Therefore, for all $k > 0$ system is **globally asymptotically stable**.

Now, using MATLAB, we will confirm these results numerically. The following MATLAB code will help us verify the stability for a range of k values:

```
%% Lyapunov equation
Q = eye(2);

for k = -1:0.1:1
    A = [0, k; -1, -2];
    % Check the stability conditions
    try
        P = lyap(A', Q);             % Solve the Lyapunov equation for P
    catch
        fprintf('For k = %.2f, the Lyapunov equation is not solvable.\n', k);
        continue;
    end
    % Check the stability conditions
    if all(eig(P) > 0) && all(eig(Q) >= 0)
        fprintf('For k = %.2f, the system is globally asymptotically stable.\n
            ', k);
    else
        fprintf('For k = %.2f, the stability condition is not satisfied.\n', k
            );
    end
end
```

The results are printed as shown below:

```
For k = -1.00, the stability condition is not satisfied.
For k = -0.90, the stability condition is not satisfied.
For k = -0.80, the stability condition is not satisfied.
For k = -0.70, the stability condition is not satisfied.
For k = -0.60, the stability condition is not satisfied.
For k = -0.50, the stability condition is not satisfied.
For k = -0.40, the stability condition is not satisfied.
For k = -0.30, the stability condition is not satisfied.
For k = -0.20, the stability condition is not satisfied.
For k = -0.10, the stability condition is not satisfied.
For k = 0.00, the Lyapunov equation is not solvable.
For k = 0.10, the system is globally asymptotically stable.
For k = 0.20, the system is globally asymptotically stable.
For k = 0.30, the system is globally asymptotically stable.
For k = 0.40, the system is globally asymptotically stable.
For k = 0.50, the system is globally asymptotically stable.
For k = 0.60, the system is globally asymptotically stable.
For k = 0.70, the system is globally asymptotically stable.
For k = 0.80, the system is globally asymptotically stable.
For k = 0.90, the system is globally asymptotically stable.
For k = 1.00, the system is globally asymptotically stable.
>>
```

Now, to better understand the system's behavior with varying values of k, we will use MATLAB's ODE45 to simulate and plot the results for three different k values and the results are displayed in the figures 1.

```
%% Numerical simulation for a specific value of k
k_values = [-1, 0, 1];
x0 = [1; 1];
tspan = 0:0.01:10;
figure;

for i = 1:length(k_values)
    k = k_values(i);
    A = [0, k; -1, -2];
    [t, x] = ode45(@(t, x) A*x, tspan, x0);
    % Plot
    subplot(length(k_values),1, i);
    plot(t, x(:, 1), 'b', t, x(:, 2), 'r');
    xlabel('Time');
    ylabel('State Value');
    legend('x1', 'x2');
    title(['k = ', num2str(k)]);
end
```

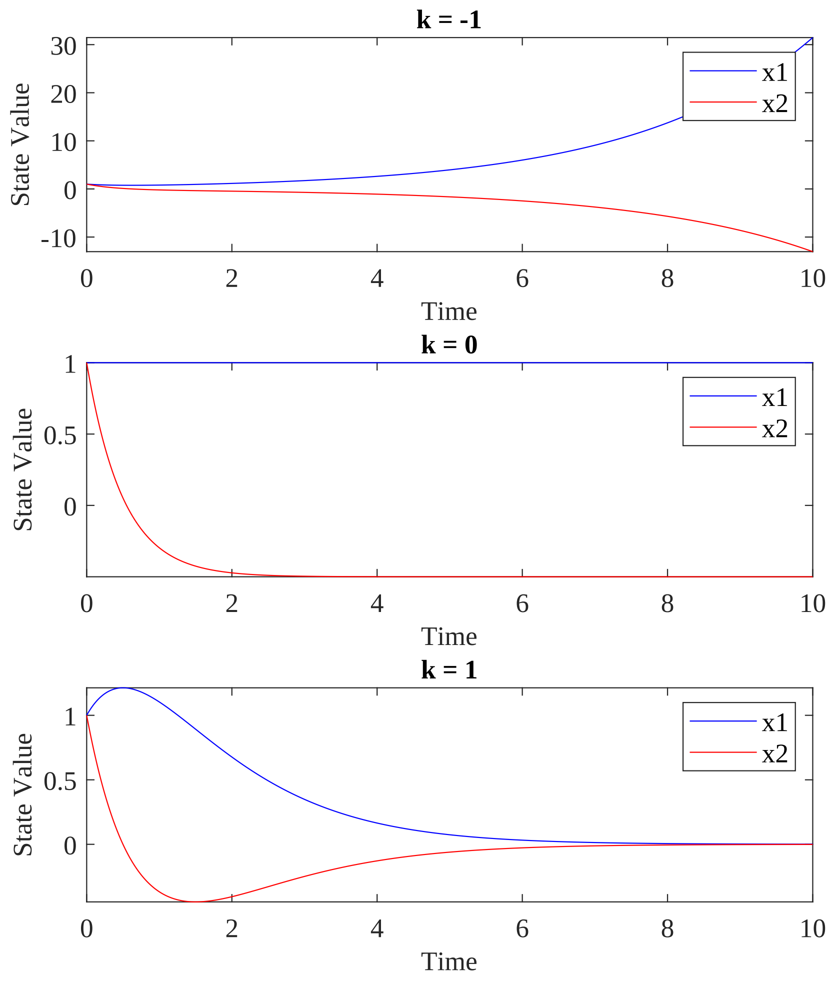

Figure 1: Numerical simulation for three different value of $k = -1, 0, 1$

Lastly, we analyze the eigenvalues based on varying values of k to assess the stability in another way:

```
%% check the eigenvalues based on k

for i = 1:length(-1:0.1:1)
    k = -1 + (i-1) * 0.1;
    A = [0, k; -1, -2];
    eigenvalues = eig(A);

    eigenvalues_1(i) = real(eigenvalues(1));
    eigenvalues_2(i) = real(eigenvalues(2));
end

% Plot the eigenvalues
figure;
plot(-1:0.1:1, [eigenvalues_1; eigenvalues_2], '-*', 'LineWidth', 1);
xlabel('k');
ylabel('Real Parts of Eigenvalues');
title('Eigenvalues vs. k');
line([-1 1], [0 0 ], 'Color', 'k', 'LineStyle', '--', 'LineWidth', 1.5);
legend('Eigenvalue 1', 'Eigenvalue 2','Stability line');
grid on
```

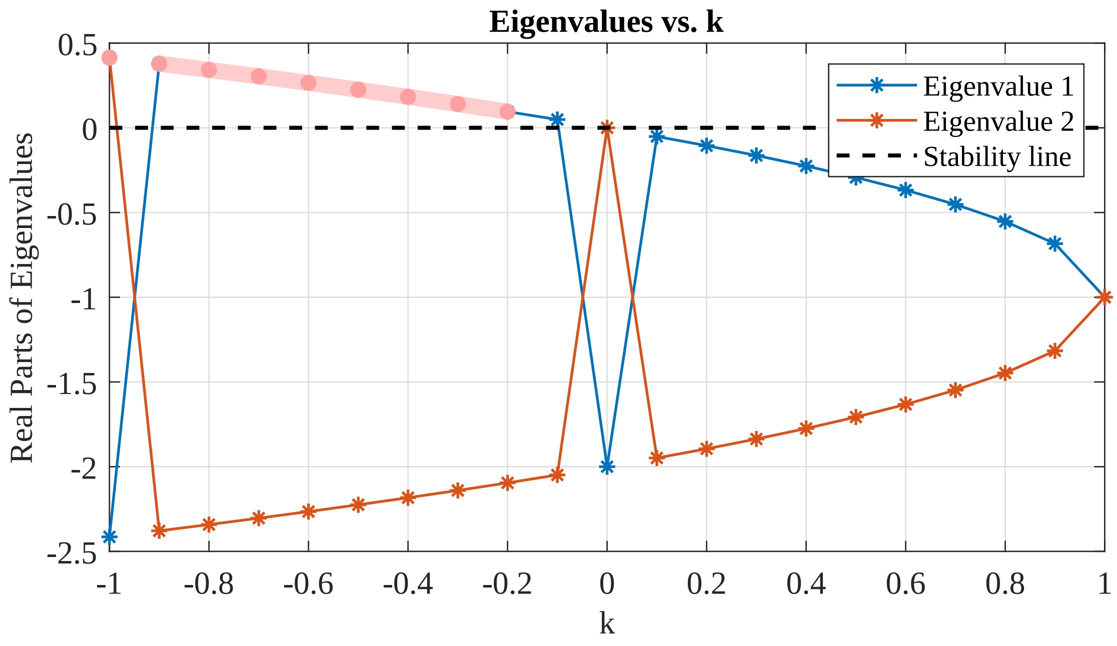

Figure 2: Illustration of the eigenvalues based on k

The eigenvalue plot is displayed in the Figure 2, showing the stability of the system for different values of k.

2 Problem 2

Consider a nonlinear system described by the following differential equations. Calculate the linearized state space equation of the system around the origin operating point.

$$\frac{d\theta_1(t)}{dt} = g_1(t) = 3\theta_1^3 - \theta_1(t)\sin(6\theta_2(t)) + u_1^2(t)$$
$$\frac{d\theta_2(t)}{dt} = g_2(t) = \theta_2(t) + u_1(t)u_2^3(t) - \theta_1(t)e^{-\theta_2(t)}$$

To calculate the linearized state space equation of the system around the origin operating point, we need to determine the matrices A, B, C, and D for the state space representation:

- Define the state variables:
  - State 1: $\theta_1(t)$
  - State 2: $\theta_2(t)$
- Write the state space equations in the following form:

$$\dot{x} = Ax + Bu$$
$$\dot{y} = Cx + Du$$

- Calculate the matrices A, B, C, and D:
  - A: The system matrix can be found by linearizing the nonlinear equations around the origin operating point. Compute the Jacobian matrix of the system:

$$A = \begin{bmatrix} \frac{\partial g_1}{\partial \theta_1} & \frac{\partial g_1}{\partial \theta_2} \\ \frac{\partial g_2}{\partial \theta_1} & \frac{\partial g_2}{\partial \theta_2} \end{bmatrix} = \begin{bmatrix} 9\theta_1^2 - \sin(6\theta_2) & -6\cos\theta_1(6\theta_2) \\ -e^{-\theta_2} & 1 + \theta_1 e^{-\theta_2} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ -1 & 1 \end{bmatrix}$$

  - B: The input matrix is defined as:

$$B = \begin{bmatrix} \frac{\partial g_1}{\partial u_1} & \frac{\partial g_1}{\partial u_2} \\ \frac{\partial g_2}{\partial u_1} & \frac{\partial g_2}{\partial u_2} \end{bmatrix} = \begin{bmatrix} 2u_1 & 0 \\ u_2^3 & 3u_1u_2^2 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

  - C: The output matrix can be defined as:

$$C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

  - D: This matrix, in this case, is typically zero since the inputs do not directly affect the outputs. Thus, $D = \mathbf{0}$.

3 Problem 3

Consider the following continuous-time open-loop transfer function. First, obtain the step response of the discretized system using Euler's approximation in MATLAB, and then analyze the impact of increasing and decreasing the value of dt on the response.

$$\frac{12}{s^2 + 3s}$$

In this section, we will tackle Problem 3, which involves a continuous-time open-loop transfer function. Our goal is to obtain the step response of the discretized system using Euler's approximation in MATLAB and analyze the impact of varying the value of dt on the system's response.

To begin, we consider the open-loop transfer function ($G = \frac{12}{s^2+3s}$). It is evident from Figure 3 that the step response of the continuous-time open-loop system is unstable. Therefore, our first task is to design a simple controller to stabilize the system.

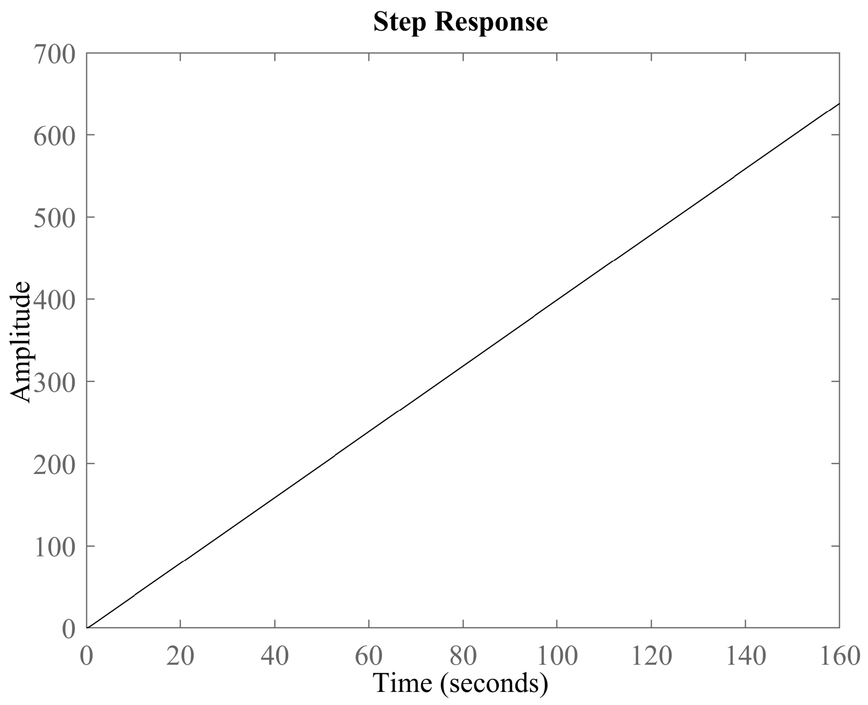

Figure 3: Continuous-time step response (open-loop system)

Figure 3 displays the continuous-time step response of the open-loop system, while Figure 4 presents the step response of the continuous-time closed-loop system, which is generated using the state-space representation of the system.

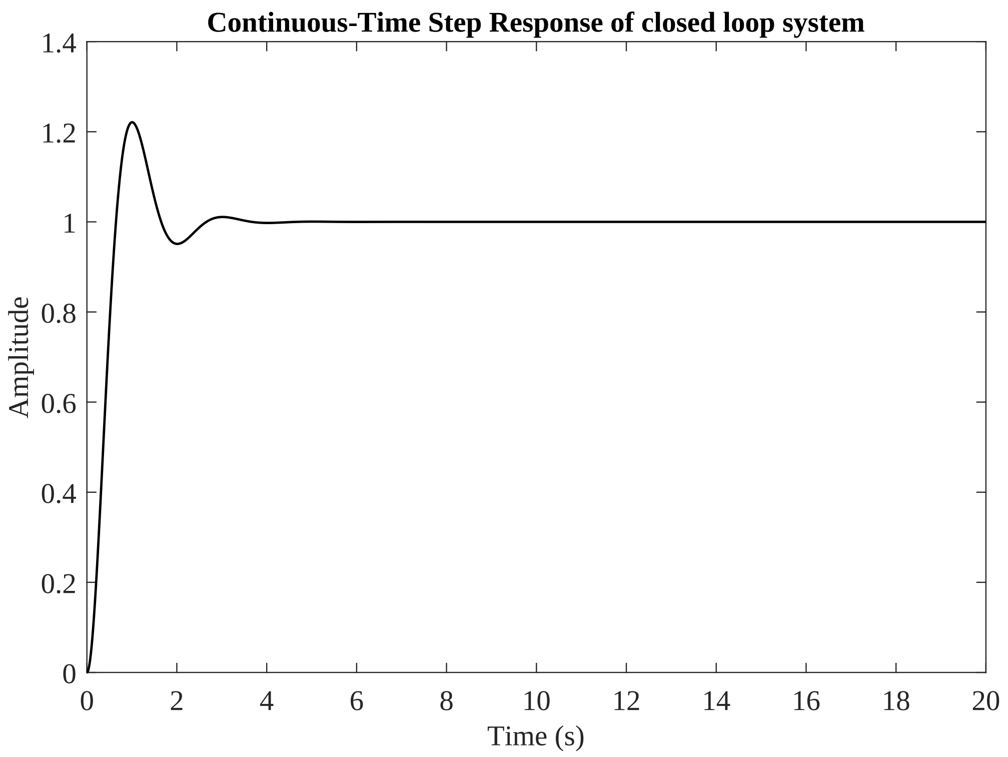

Figure 4: Continuous-time step response (closed-loop system)

For the numerical analysis, we employ MATLAB. The following code is employed to address this problem:

```
% Define the continuous-time open-loop transfer function
s = tf('s');
G = 12 / (s^2 + 3*s);

figure(1)
step(G, 'k')

% Create the closed-loop transfer function
T = feedback(G, 1);

% Convert the closed-loop system to state-space representation
state_space = ss(T);

% Define the simulation time and sampling times
Tf = 20;                    % Total simulation time
Ts_values = [0.2, 0.1, 0.05, 0.02, 0.01, 0.001]; % Different sampling times

% Create time vector for continuous plotting
t_continuous = 0:0.001:Tf;

% Step response of the continuous system
[y_continuous, t_continuous] = step(state_space, t_continuous);

figure(2)
```

```
plot(t_continuous, y_continuous, 'k' , 'LineWidth', 1)
title('Continuous-Time Step Response of closed loop system')
xlabel('Time (s)')
ylabel('Amplitude')


for i = 1:length(Ts_values)
    Ts = Ts_values(i);
    N = round(Tf / Ts);
    t = 0:Ts:Tf;
    x = [0; 0];
    Y_output = zeros(N+1, 1);

    % Euler approximation
    for j = 1:N+1
        x = Ts * (state_space.A * x + state_space.B * 1) + x;
        Y_output(j) = state_space.C * x;
    end
    figure(3);
    subplot(2, 3, i);
    plot(t, Y_output, 'r--');
    title(['Ts = ', num2str(Ts)]);
    xlabel('Time (s)');
    ylabel('Amplitude');
    hold on
    plot(t_continuous, y_continuous,'k', 'LineWidth', 1);

    figure(4)
    plot(t, Y_output,'--')
    hold on

end

title('Euler Approximation Step Response')
xlabel('Time (s)')
ylabel('Amplitude')
legend(cellstr(num2str(Ts_values', 'Ts = %.2f')))
plot(t_continuous, y_continuous, 'k' , 'LineWidth', 1)
legend('Euler (Ts = 0.2)', 'Euler (Ts = 0.1)', 'Euler (Ts = 0.05)', 'Euler (Ts
    = 0.02)','Euler (Ts = 0.01)','Euler (Ts = 0.001)', 'Continuous')
hold off
```

Figure 5 visually represents the step responses of the discretized system for various sampling times (Ts). Each subplot corresponds to a different Ts value, and the red dashed lines depict the Euler approximations. The black line represents the continuous-time response, and the legend provides information on the Ts values used for each subplot.

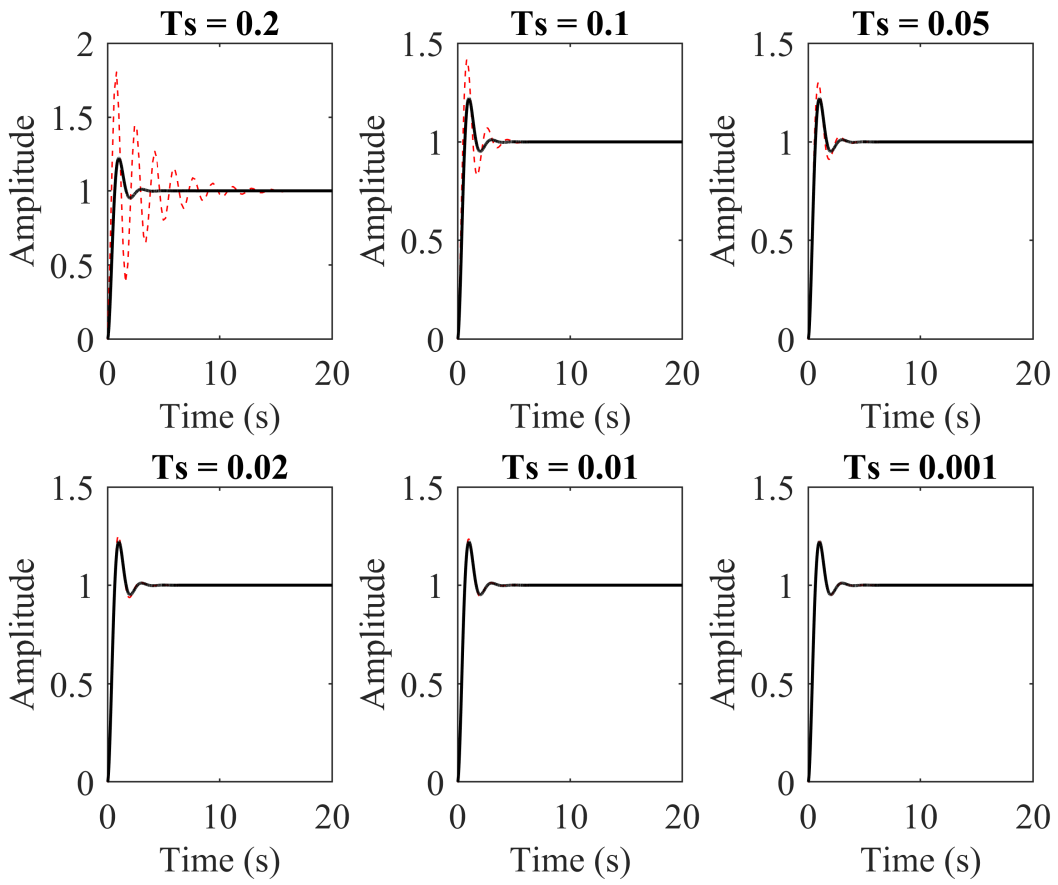

Figure 5: Euler Approximation Step Responses with defferent sampleing time

Finally, Figure 6 offers a comprehensive comparison of the Euler approximations for all Ts values, along with the continuous-time response. The dashed lines represent the discretized responses, and the solid black line represents the continuous-time response. The legend provides details about the Ts values used for each Euler approximation.

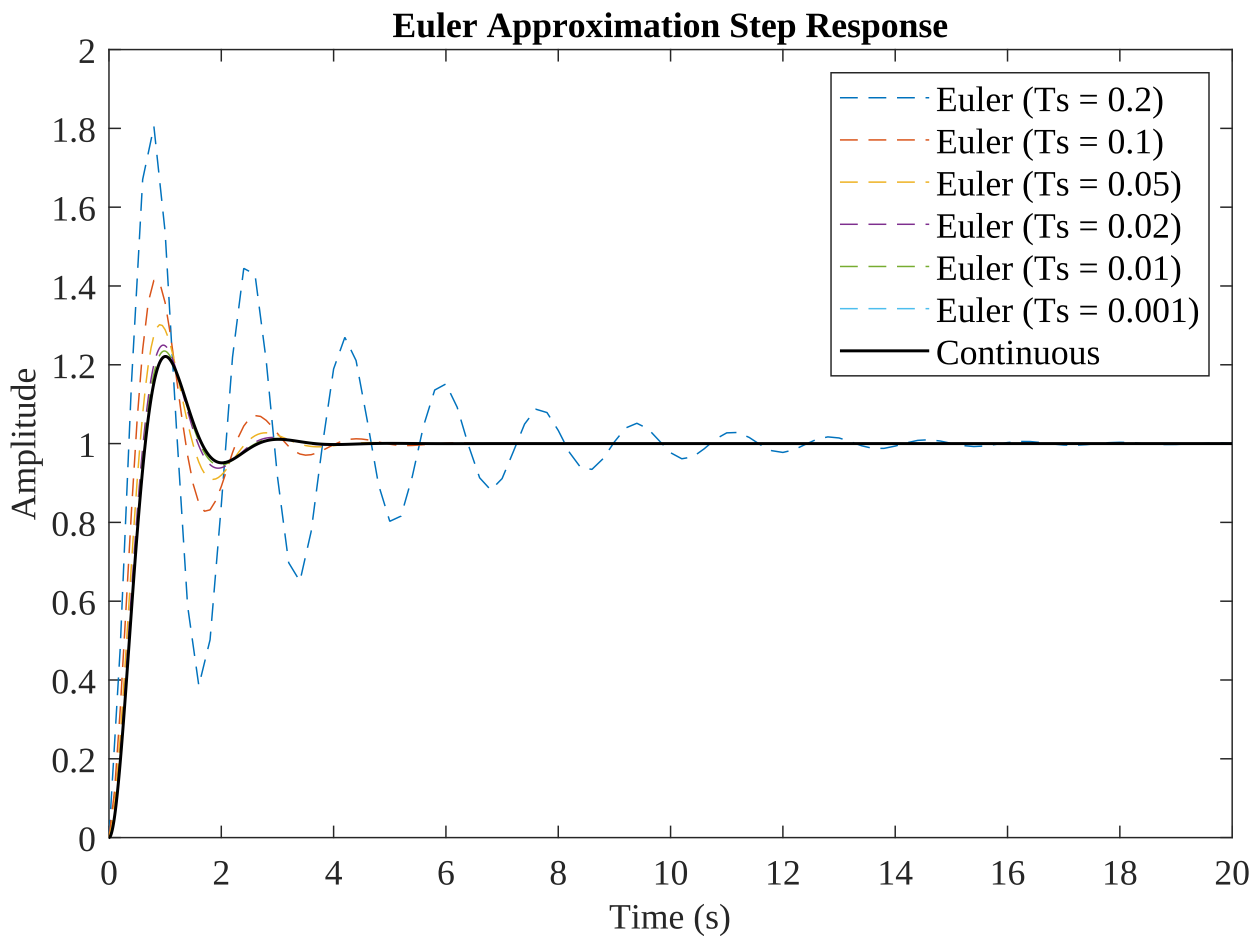

Figure 6: Comparison of Euler approximations with different dt and continuouse response

This analysis allows us to understand the impact of different sampling times on the discretized system. Furthermore, as we decrease the sampling times, we can observe how the discretized system tends to approximate the behavior of the original continuous-time system.

4 Problem 4

Consider a convex, differentiable, and continuous function as follows:

$$f(x) = (4x_1 - 7)^2 + (0.6x_2 - 2)^2 + 3(x_3 + 4)^2 + 12$$

a) Using MATLAB software, obtain the response and the optimal value using the gradient descent method.

b) Using MATLAB software, calculate the response and the optimal value using Newton's method.

c) Compare the results of parts a and b.

4.1 Gradient Descent Method

With the following code, the optimal solution using the gradient descent method is calculated:

```
% Initialization and stopping criterion
% x = [1 1 1];
x = rand(1,3);
maxIterations = 1000;
tolerance = 0.001;

% Define the objective function
objective = @(x) (4*x(1) - 7)^2 + (0.6*x(2) - 2)^2 + 3*(x(3) + 4)^2 + 12;

% Gradient Descent
objective_values = zeros(1, maxIterations);  % To store objective values at
    each iteration

x1_values = zeros(1, maxIterations);
x2_values = zeros(1, maxIterations);
x3_values = zeros(1, maxIterations);

for iter = 1:maxIterations
    % Evaluate function and gradient
    f_value = objective(x);
    gradient = computeGradient(x);

    % Compute step direction
    step_direction = -gradient;

    % Backtracking line search
    alpha = 0.3;  % Step size parameters
    beta = 0.8;
    t = 1;

    while objective(x + t * step_direction) > f_value + alpha * t * gradient'
        * step_direction
         t = beta * t;
    end
    % Update the current solution
    x = x + t * step_direction;

    % Store the objective value at this iteration
    objective_values(iter) = f_value;

    x1_values(iter) = x(1);
    x2_values(iter) = x(2);
    x3_values(iter) = x(3);

    % Print the optimization process in the command window
    fprintf('Iteration %d: Objective Value = %f\n', iter, f_value);

    % Check the stopping criterion
    if norm(gradient, 2) < tolerance
        break;
    end
end
```

The results of the Gradient Descent method for this optimization problem are as follows:

```
---------------------- Results ----------------------
Number of Iterations with Gradient Descent Method: 176
Optimal Solution: (1.750, 3.332, -4.000)
Optimal Value: 12.000000
```

For better understanding, the trajectory of x_1, x_2, x_3 and the optimal value based on iterations are visualized in Figures 7 and 8:

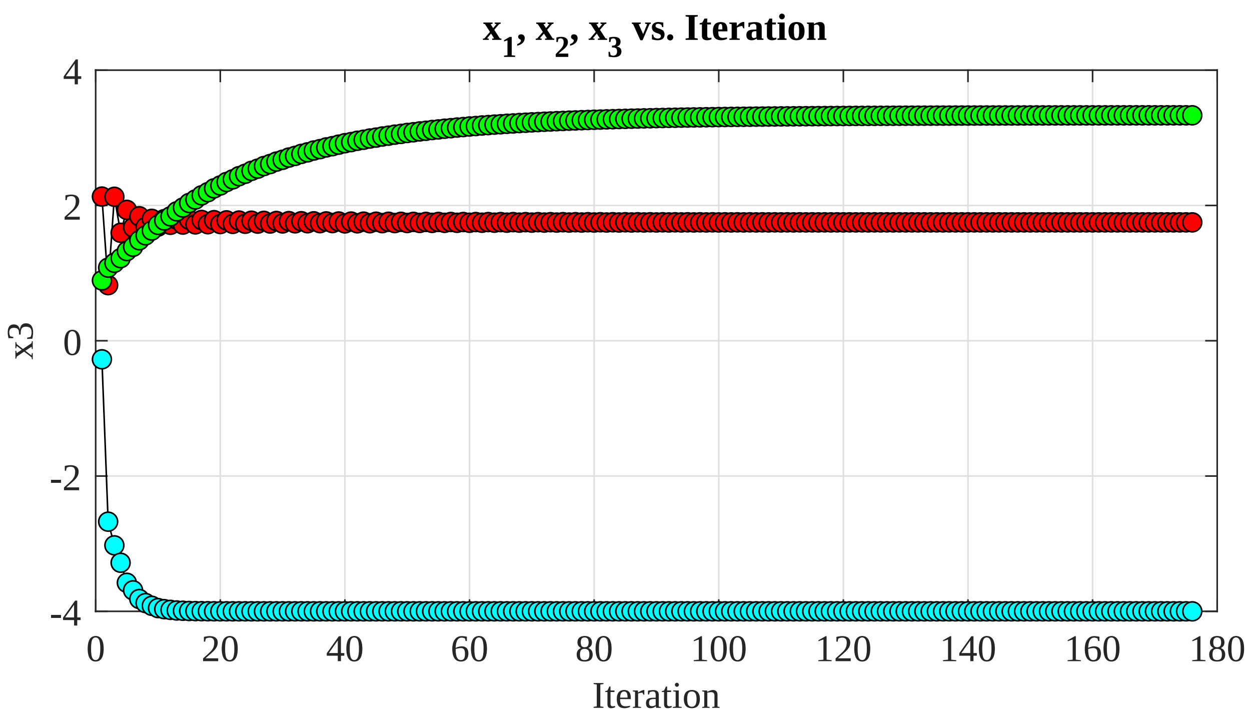

Figure 7: Values of x_1, x_2, x_3 in each iteration in gradient descent method

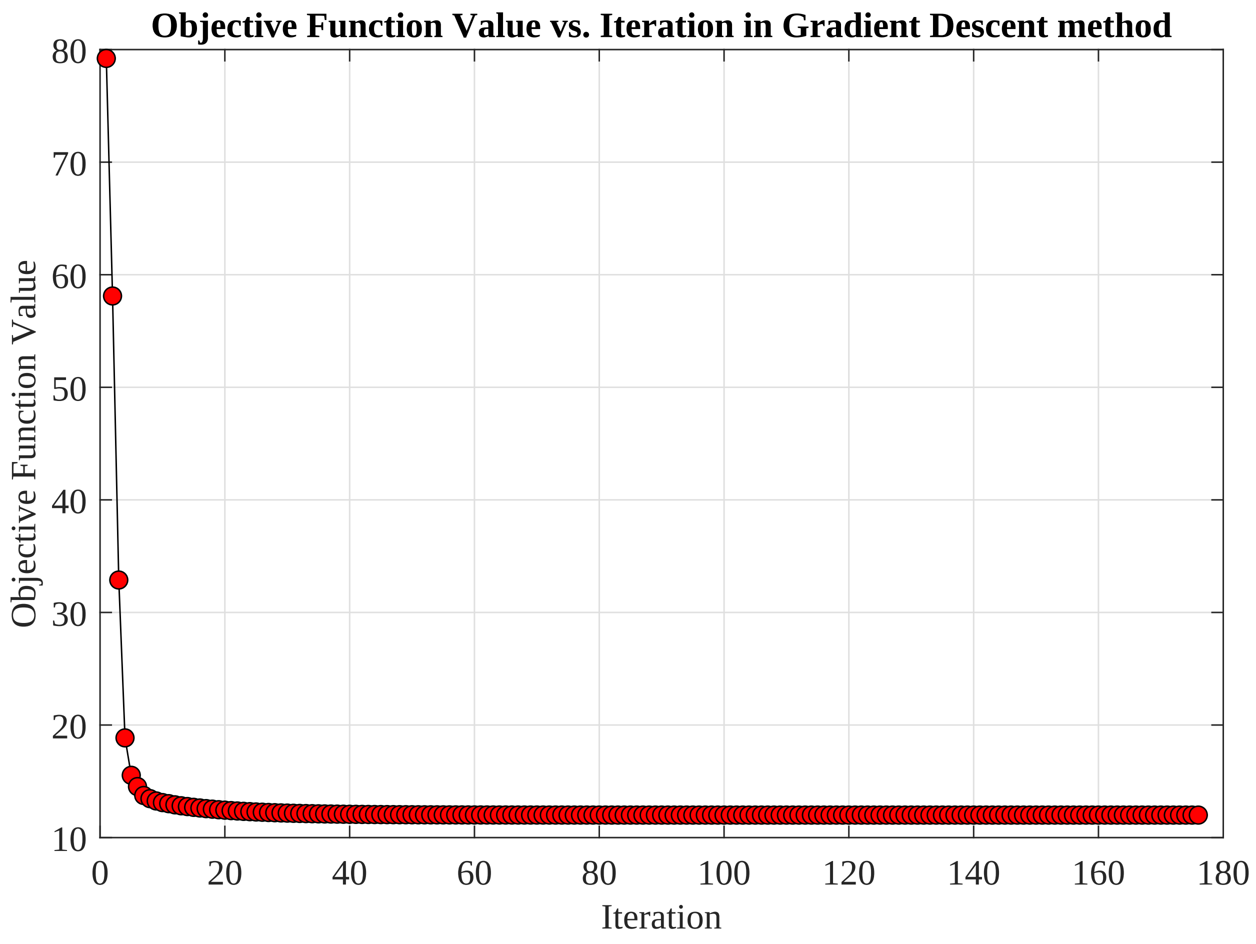

Figure 8: Objective function value in gradient descent method

4.2 Newton's Method

With the following code, the optimal solution using the gradient descent method is calculated:

```
x = rand(1, 3);
maxIterations = 1000;
tolerance = 0.001;

% Define the objective function
objective = @(x) (4*x(1) - 7)^2 + (0.6*x(2) - 2)^2 + 3*(x(3) + 4)^2 + 12;

% Newton's Method
objective_values = zeros(1, maxIterations);  % To store objective values at
    each iteration

x1_values = zeros(1, maxIterations);
x2_values = zeros(1, maxIterations);
x3_values = zeros(1, maxIterations);

for iter = 1:maxIterations
    % Evaluate function, gradient, and Hessian
    f_value = objective(x);
    gradient = computeGradient(x);
    hessian = computeHessian(x);

    % Compute step direction
    step_direction = -hessian \ gradient;

    % Backtracking line search
    alpha = 0.3;  % Step size parameters
    beta = 0.8;
    t = 1;

    while objective(x + t * step_direction) > f_value + alpha * t * gradient'
        * step_direction
        t = beta * t;
    end

    % Update the current solution
    x = x + t * step_direction;

    % Store the objective value at this iteration
    objective_values(iter) = f_value;

    x1_values(iter) = x(1);
    x2_values(iter) = x(2);
    x3_values(iter) = x(3);

    % Print the optimization process in the command window
    fprintf('Iteration %d: Objective Value = %f\n', iter, f_value);

    % Check the stopping criterion
    if norm(gradient, 2) < tolerance
        break;
    end
end
```

```
---------------------- Results ----------------------
Number of Iterations with Newton Method: 42
Optimal Solution: (1.750, 3.333, -4.000)
Optimal Value: 12.000000
```

For better understanding, the trajectory of x_1, x_2, x_3 and the optimal value based on iterations are visualized in Figures 9 and 10:

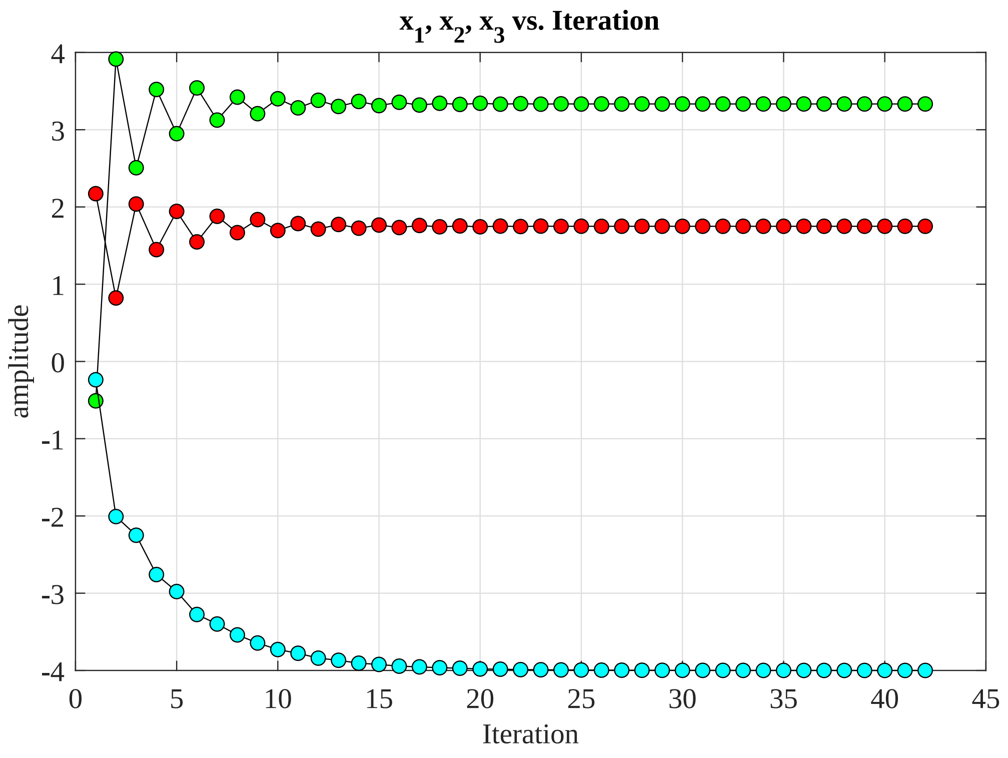

Figure 9: Values of x_1, x_2, x_3 in each iteration in Newton's method

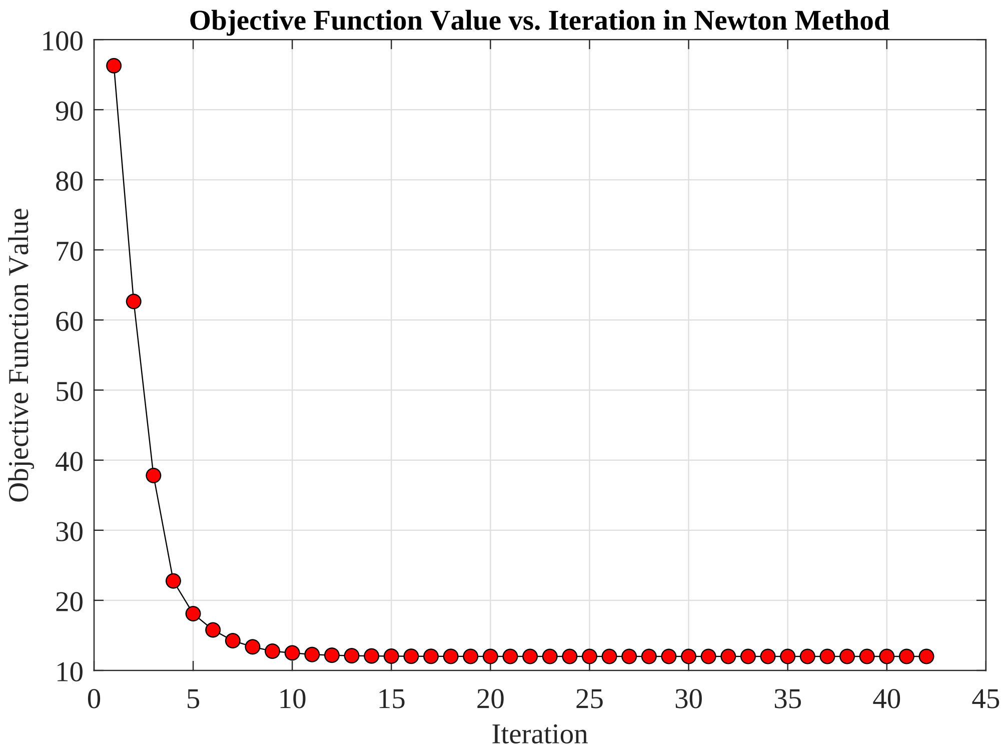

Figure 10: Objective function value in Newton's method

4.3 Comparison: Gradient Descent Method and Newton's Method

I used both the Gradient Descent method and Newton's Method to optimize the given convex, differentiable, and continuous function. Let's compare the results of these two optimization methods:

Table 1: Comparison of Gradient Descent and Newton's Method

Method	**Iterations**	**Optimal Solution**	**Optimal Value**	**Elapsed Time**
Gradient Descent	176	1.750, 3.332, -4.000	12.000000	0.038276 s
Newton's Method	42	1.750, 3.333, -4.000	12.000000	0.006796 s

Comparison:

- **Convergence Speed:** In this specific problem, Newton's Method converged significantly faster than Gradient Descent. It took 176 iterations for Gradient Descent to achieve the optimal solution, while Newton's Method only required 42 iterations.
- **Optimal Solution:** Both methods found the same optimal solution, which is (1.750, 3.333, -4.000).
- Optimal Value: Both methods achieved the same optimal value of 12.000000.
- **Efficiency:** Newton's Method is more efficient in terms of convergence speed for this problem, thanks to its second-order information utilization (Hessian matrix) compared to Gradient Descent, which relies on first-order information (gradient).
- **Computation Complexity:** In each iteration, Newton's Method involves the additional computation of the Hessian matrix and its inverse. This computational overhead can make it less efficient when compared to the simpler Gradient Descent, which only computes the gradient.

5 Problem 5

Obtain the optimal value of the following problem with the CVX toolbox.

$$\begin{aligned} \text{minimize} \quad & (x_1+5)^2 + (x_2-2)^2 \\ \text{subject to} \quad & 1 \leqslant x_1 \leqslant 3, \\ & -6 \leqslant x_2 \leqslant -2, \qquad x \in R^2 \end{aligned} \tag{10}$$

5.1 solve

The optimization problem is modeled using CVX, a MATLAB-based modeling system for convex optimization. The CVX code defines the optimization variables x_1 and x_2, the objective function to minimize, and the constraints.

```
% Define the optimization variables
cvx_begin
variables x1 x2

% Define the objective function to minimize
minimize((x1 + 5)^2 + (x2 − 2)^2)

% Define the constraints
subject to
1 <= x1 <= 3
−6 <= x2 <= −2
cvx_end
```

The optimization problem is solved using the SDPT3 solver. The solver iteratively refines the solution to converge to an optimal point. In this case, the solver converges to an optimal solution in 8 iterations.

The optimal values for x_1 and x_2 are $x_1 = 1$ and $x_2 = -2$, respectively, with a minimum objective function value of 52.

```
Calling SDPT3 4.0: 10 variables, 4 equality constraints
For improved efficiency, SDPT3 is solving the dual problem.
------------------------------------------------------------


 num. of constraints =  4
 dim. of sdp    var  =  4,   num. of sdp  blk  =  2
 dim. of linear var  =  4
*******************************************************************
   SDPT3: Infeasible path-following algorithms
*******************************************************************
 version  predcorr  gam  expon  scale_data
   HKM      1      0.000   1        0
it pstep dstep pinfeas dinfeas  gap      prim-obj      dual-obj    cputime
-------------------------------------------------------------------
 0|0.000|0.000|5.3e+00|2.0e+00|8.5e+02| 8.000000e+01  0.000000e+00| 0:0:00| chol  1  1
 1|0.867|0.874|7.0e-01|2.6e-01|9.8e+01|-2.573385e+01 -1.804677e+01| 0:0:00| chol  1  1
 2|0.687|0.696|2.2e-01|8.0e-02|5.1e+01|-3.991315e+01 -4.106970e+01| 0:0:00| chol  1  1
```

```
  3|1.000|1.000|9.4e-07|1.8e-04|1.8e+01|-4.165789e+01 -5.958894e+01| 0:0:00| chol  1  1
  4|0.976|0.971|2.6e-07|2.3e-05|5.0e-01|-5.180349e+01 -5.228966e+01| 0:0:00| chol  1  1
  5|0.986|0.983|5.4e-08|2.2e-06|7.7e-03|-5.199729e+01 -5.200411e+01| 0:0:00| chol  1  1
  6|0.979|0.985|3.2e-09|4.3e-08|1.4e-04|-5.199994e+01 -5.200006e+01| 0:0:00| chol  1  1
  7|0.957|0.979|1.5e-10|1.5e-09|4.7e-06|-5.200000e+01 -5.200000e+01| 0:0:00| chol  1  1
  8|1.000|1.000|1.3e-13|3.0e-11|7.0e-07|-5.200000e+01 -5.200000e+01| 0:0:00|
   stop: max(relative gap, infeasibilities) < 1.49e-08
-------------------------------------------------------------------
 number of iterations   =  8
 primal objective value = -5.19999997e+01
 dual   objective value = -5.20000003e+01
 gap := trace(XZ)       = 7.02e-07
 relative gap           = 6.69e-09
 actual relative gap    = 6.57e-09
 rel. primal infeas (scaled problem)   = 1.34e-13
 rel. dual     "        "       "      = 2.97e-11
 rel. primal infeas (unscaled problem) = 0.00e+00
 rel. dual     "        "       "      = 0.00e+00
 norm(X), norm(y), norm(Z) = 4.3e+01, 4.0e+01, 4.1e+01
 norm(A), norm(b), norm(C) = 4.2e+00, 2.4e+00, 1.6e+01
 Total CPU time (secs)  = 0.40
 CPU time per iteration = 0.05
 termination code       =  0
 DIMACS: 1.6e-13  0.0e+00  5.2e-11  0.0e+00  6.6e-09  6.7e-09
-------------------------------------------------------------------
Status: Solved
Optimal value (cvx_optval): +52
```

by using the following code, results printed:

```
% Display the results
disp('Optimal values:')
disp(['x1 = ', num2str(x1)])
disp(['x2 = ', num2str(x2)])
disp(['Minimum value: ', num2str((x1 + 5)^2 + (x2 - 2)^2)])
```

```
Optimal values: x1 = 1,  x2 = -2
Minimum value: 52
```

```
% Plot the objective function and constraints
x1_values = linspace(1, 3, 100);
x2_values = linspace(-6, -2, 100);
[X1, X2] = meshgrid(x1_values, x2_values);
f_values = (X1 + 5).^2 + (X2 - 2).^2;

figure;
contour(X1, X2, f_values, 20); % Contour plot of the objective function
hold on;
plot([1, 1, 3, 3, 1], [-6, -2, -2, -6, -6], 'r--'); % Constraint plot
scatter(x1, x2, 100, 'ro', 'filled'); % Optimal point
xlabel('x1');
ylabel('x2');
title('Objective Function and Constraints');
```

```
legend('Objective Function', 'Constraints', 'Optimal Point');
grid on;
axis([0 4 -7 -1])
```

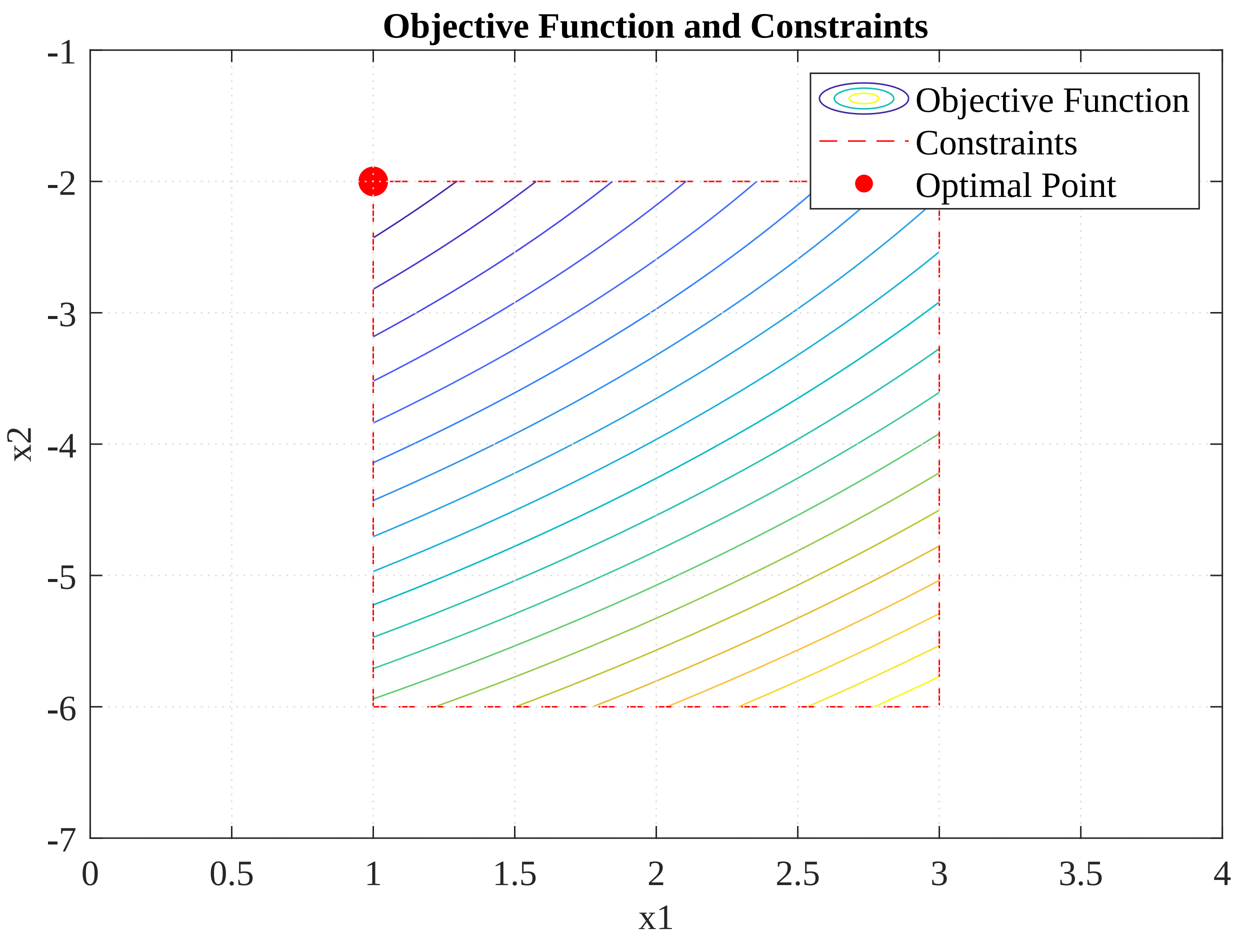

Figure 11: Objective function, constraints, and optimal point of problem 5

This resulting plot is depicted in Figure 11, showcasing the contours of the objective function, the constraint boundaries, and the marked optimal point.