Programming Exercise Report

1 Problem Definition

Solve the following quadratic programming problem by using the interior point method:

$$\begin{aligned} \min_{x\in\mathbb{R}^4} \quad & \frac{1}{2}x^TQx + c^Tx, \\ \text{s.t.} \quad & a_1^Tx \le b_1, \\ & a_2^Tx \le b_2, \end{aligned} \tag{1}$$

Where:

$$\begin{aligned} Q &= \begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 8 \end{pmatrix}, \qquad c = \begin{pmatrix} -8 \\ -6 \\ -4 \\ -2 \end{pmatrix}, \\ a_1 &= \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix},\ b_1 = 3, \qquad a_2 = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix},\ b_2 = 4. \end{aligned} \tag{2}$$

2 Solution

Using the interior method, there are several steps.

2.1 Constructing $Q'(x, r)$ and Differentiation

The function $Q'(x, r)$ is defined as:

$$Q'(x,r) = \frac{1}{2}x^TQx + c^Tx - r\left(\ln(b_1 - a_1^Tx) + \ln(b_2 - a_2^Tx)\right) \tag{3}$$

Differentiating $Q'(x, r)$ with respect to x gives us the gradient $\nabla_x Q'(x, r)$(Note that Q here is a symmetric matrix):

$$\frac{\partial Q'}{\partial x} = Qx + c - r\left(\frac{-a_1}{b_1 - a_1^Tx} + \frac{-a_2}{b_2 - a_2^Tx}\right) \tag{4}$$

The partial derivative $\frac{\partial Q'}{\partial x}$ is the gradient vector of the function $Q'(x, r)$.

The code of this part is at here:

```
Grad_Q_prime = @(x) (Q * x + c) - ...
    r_val * ( (-a1 / (b1 - a1' * x)) + (-a2 / (b2 - a2' * x)) );

Hessian_Q_prime = @(x) Q + r_val * ( ...
    (a1 * a1') / (b1 - a1' * x)^2 + ...
    (a2 * a2') / (b2 - a2' * x)^2 );
```

2.2 Solving the equation

Let $\frac{\partial Q'}{\partial x} = 0$, we can get the solution. If we want to solve the equation by computer, we should use the Newton Method. Firstly, define the Hessian matrix

$$\mathbf{H} = Q + r\left[\frac{a_1 a_1^T}{(b_1 - a_1^T x)^2} + \frac{a_2 a_2^T}{(b_2 - a_2^T x)^2}\right] \tag{5}$$

Then at the kth iteration process, we can solve the equation repeatedly like this:

Calculate the Newton step:

$$\mathbf{H}_k \mathbf{p}_k = -\frac{\partial Q'}{\partial x_k} \tag{6}$$

Update x:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \left[\nabla_x^2 Q'(\mathbf{x}_k)\right]^{-1} \nabla_x Q'(\mathbf{x}_k) \tag{7}$$

Check the stop condition:

$$\|\mathbf{g}_k\|_2 < \varepsilon \tag{8}$$

The code is at here:

```
for k = 1:MAX_ITER
    g_k = Grad_Q_prime(x_k);
    H_k = Hessian_Q_prime(x_k);

    if norm(g_k) < TOLERANCE
        x_star = x_k;
        return;
    end

    delta_x = -H_k \ g_k;
    t = 1.0;
    x_k = x_k + t * delta_x;

    if (b1 - a1' * x_k) <= 0 || (b2 - a2' * x_k) <= 0
```

```
            warning('Newton step leads to infeasible point.');
            x_star = x_k - t * delta_x;
            return;
        end
    end
    warning('Newton''s Method failed to converge within MAX_ITER.');
    x_star = x_k;
```

In that progress, r is a const number, we should decrease it to a very small value to find the optimal feasible solution.

```
    for path_k = 1:MAX_PATH_ITER

        Grad_Q_prime = @(x) (Q * x + c) - ...
            r_val * ( (-a1 / (b1 - a1' * x)) + (-a2 / (b2 - a2' * x)) );
        Hessian_Q_prime = @(x) Q + r_val * ( ...
            (a1 * a1') / (b1 - a1' * x)^2 + ...
            (a2 * a2') / (b2 - a2' * x)^2 );

        x_center = newtonSolver(Grad_Q_prime, Hessian_Q_prime, x_k);

        obj_val = 0.5 * x_center' * Q * x_center + c' * x_center;
        fprintf('%9d | %7.4e | %20.6f \n', path_k, r_val, obj_val);

        if r_val < R_TOLERANCE
            x_opt = x_center;
            r_final = r_val;
            return;
        end

        r_val = mu * r_val;

        x_k = x_center;
    end
    x_opt = x_center;
    r_final = r_val;
```

The final answer is:

$$x = \begin{pmatrix} 1.333 \\ 1.667 \\ 0.667 \\ 0.25 \end{pmatrix} \tag{9}$$

Now we solve the quadratic programming problem by using the function “quadprog” in MATLAB, the code is here.

```
clear all;
Q = [4, 0, 0, 0;
    0, 2, 0, 0;
    0, 0, 6, 0;
    0, 0, 0, 8];
c = [-8; -6; -4; -2];
a1 = [1; 1; 0; 0];
b1 = 3;
a2 = [0; 1; 1; 1];
b2 = 4;
A = [a1',
    a2'];
b = [b1;
    b2];
LB = [];
UB = [];
[x_quadprog, fval_quadprog, exitflag, output] = ...
quadprog(Q, c, A, b, [], [], LB, UB);
disp(x_quadprog);
```

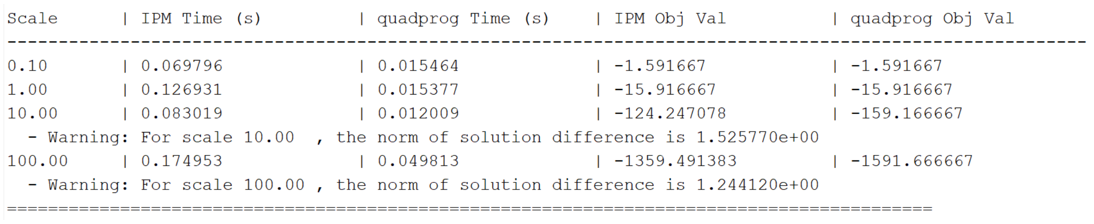
3 Comparison

Firstly, we can change the scale of value in the original problem and compare the correctness and run time of the interior point method and the matlab quadprog function separately. The code:

```
scale_factors = [0.1, 1, 10, 100];

num_scales = length(scale_factors);
all_results = cell(num_scales, 5); % scale, ipm_time, qp_time, ipm_obj, qp_obj
```

```
solution_diffs = zeros(num_scales, 1);

for i = 1:num_scales
    scale = scale_factors(i);

    Q_scaled = scale * Q;
    c_scaled = scale * c;

    tic;
    [x_ipm, ~] = ...
    pathFollowingQP(Q_scaled, c_scaled, a1, b1, a2, b2, x0_start, true, scale);
    time_ipm = toc;
    obj_val_ipm = 0.5 * x_ipm' * Q_scaled * x_ipm + c_scaled' * x_ipm;

    A = [a1'; a2'];
    b = [b1; b2];
    options = optimoptions('quadprog', 'Display', 'none');

    tic;
    [x_qp, obj_val_qp] = ...
    quadprog(Q_scaled, c_scaled, A, b, [], [], [], [], [], options);
    time_quadprog = toc;

    all_results{i, 1} = scale;
    all_results{i, 2} = time_ipm;
    all_results{i, 3} = time_quadprog;
    all_results{i, 4} = obj_val_ipm;
    all_results{i, 5} = obj_val_qp;

    solution_diffs(i) = norm(x_ipm - x_qp);
end
```

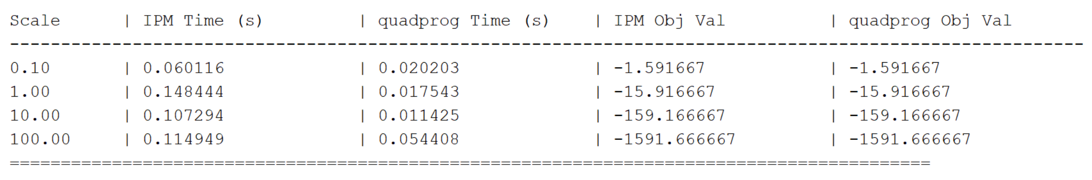
Run the code, the result:



```
Scale       | IPM Time (s)          | quadprog Time (s)     | IPM Obj Val           | quadprog Obj Val
-----------------------------------------------------------------------------------------------------------
0.10        | 0.069796              | 0.015464              | -1.591667             | -1.591667
1.00        | 0.126931              | 0.015377              | -15.916667            | -15.916667
10.00       | 0.083019              | 0.012009              | -124.247078           | -159.166667
  - Warning: For scale 10.00  , the norm of solution difference is 1.525770e+00
100.00      | 0.174953              | 0.049813              | -1359.491383          | -1591.666667
  - Warning: For scale 100.00 , the norm of solution difference is 1.244120e+00
===========================================================================================
```

Figure 1: The Original Result

As can be seen, when the magnitude increases to 10, the results obtained using the interior point method become incorrect. This is because we initially set the penalty coefficient r to a fixed value when solving the problem. When the scale of the problem coefficients increases, the initial value of r cannot guarantee obtaining a feasible solution, leading to incorrect iteration directions and erroneous answers. After analysis, we found that by synchronously multiplying r by the scale scaling coefficient, we can obtain feasible and stable solutions.



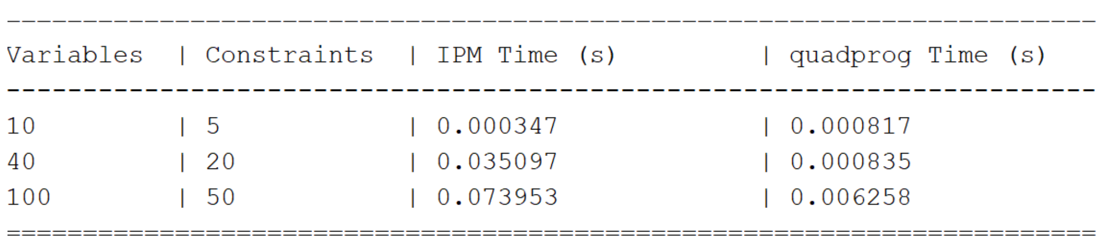
```
Scale      | IPM Time (s)          | quadprog Time (s)     | IPM Obj Val           | quadprog Obj Val
---------------------------------------------------------------------------------------------------------
0.10       | 0.060116              | 0.020203              | -1.591667             | -1.591667
1.00       | 0.148444              | 0.017543              | -15.916667            | -15.916667
10.00      | 0.107294              | 0.011425              | -159.166667           | -159.166667
100.00     | 0.114949              | 0.054408              | -1591.666667          | -1591.666667
==========================================================================================
```

Figure 2: The Corrected Result

The results indicate that our manually defined interior-point method exhibits relatively slow convergence speeds. While there is a tendency for convergence speed to decrease as the number of parameters increases, the overall change is not significant.

Secondly, we can change the scale of the problem (change the number of variables) in the original problem and compare the correctness and run time of the interior point method and the matlab quadprog function separately. The code:

```
problem_sizes = [10, 40, 100];
num_constraints_factor = 0.5; % n_constraints = n_vars * factor

fprintf('Starting QP Solver Comparison for Different Problem Sizes...\n\n');
results = cell(length(problem_sizes), 5);

for i = 1:length(problem_sizes)
    n = problem_sizes(i);
    m = floor(n * num_constraints_factor);

    problem = generateQP(n, m);

    f_ipm = @() pathFollowingQP(problem.Q, problem.c, ...
    problem.A, problem.b, problem.x0, true);
    time_ipm = timeit(f_ipm);

    options = optimoptions('quadprog', 'Display', 'none',...
    'Algorithm', 'interior-point-convex');
    f_qp = @() quadprog(problem.Q, problem.c, problem.A, problem.b, ...
    [], [], [], [], [], options);
    time_quadprog = timeit(f_qp);

    [x_ipm, ~] = pathFollowingQP(problem.Q, problem.c, problem.A, problem.b, ...
    problem.x0, true);
    [x_qp, ~] = quadprog(problem.Q, problem.c, problem.A, problem.b,...
    [], [], [], [], [], options);

    results{i, 1} = n;
    results{i, 2} = m;
    results{i, 3} = time_ipm;
    results{i, 4} = time_quadprog;

    norm_x_qp = norm(x_qp);
    if norm_x_qp > 0
        results{i, 5} = norm(x_ipm - x_qp) / norm_x_qp;
    else
        results{i, 5} = norm(x_ipm - x_qp);
    end
end
```



```
------------------------------------------------------------------------
Variables   | Constraints   | IPM Time (s)           | quadprog Time (s)
------------------------------------------------------------------------
10          | 5             | 0.000347               | 0.000817
40          | 20            | 0.035097               | 0.000835
100         | 50            | 0.073953               | 0.006258
========================================================================
```

Figure 3: Results after Adjusting the Problem Scale

As shown in the figure, when the number of variables is small, the speed of solving using the interior-point method decreases significantly. As the problem size continues to increase, the solving speed stabilizes. This demonstrates that the interior-point method is efficient for solving large-scale problems.