Songklanakarin J. Sci. Technol.
43 (5), 1449-1455, Sep. - Oct. 2021

http://www.sjst.psu.ac.th

Original Article

Efficient method to compute search directions of infeasible primal-dual path-following interior-point method for large scale block diagonal quadratic programming

Duangpen Jetpipattanapong and Gun Srijuntongsiri*

School of Information, Computer, and Communication Technology, Sirindhorn International Institute of Technology, Thammasat University, Khlong Luang, Pathum Thani, 12120 Thailand

Received: 18 December 2019; Revised: 20 August 2020; Accepted: 30 November 2020

Abstract

Quadratic programming is an important optimization problem that has applications in many areas such as finance, control, and management. Quadratic programs arisen in practice are often large but sparse, and they usually cannot be solved efficiently without exploiting their structures. Since existing methods for quadratic programming deal with dense cases and do not take advantage of any specific sparsity patterns in the problems, we propose a method to efficiently compute the search directions for the primal-dual path-following interior-point method for the large-scale quadratic programs whose Hessian matrices have block diagonal structures and whose constraint matrices are dense by exploiting the special sparsity pattern in the problems to avoid unnecessary computations involving blocks of zeros. Examples of quadratic programs with such structure, to which our method can be applied, are linear model predictive control in automatic control and portfolio optimization where securities from different sectors are weakly correlated. The time complexity of our method is significantly smaller than that of using a sparse linear solver. Additionally, the computational results show that our method is faster.

Keywords: large scale quadratic programming, block diagonal, interior-point method, primal-dual path following, optimization

1. Introduction

Quadratic programming is a class of constrained optimization with quadratic objective function and linear constraints. It has applications in many areas such as prediction, control, modeling, finance, engineering, and management (Bartlett, Biegler, Backstrom, & Gopal, 2002; Kim & Rassias, 2007; Kouzoupis *et al.*, 2018; Liu, Wang, & Rong, 2009; Mitsuia & Tabata, 2008; Zhang, Xu, Di, & Thomson, 2002; Zhang, Zhong, Wu, & Liao, 2006;). Moreover, quadratic programming is used as a part of sequential quadratic programming (SQP) to solve nonlinear programming problems, which has many applications such as in the optimal power flow problem in DC grids (Montoya, Gil-Gonzáles, & Garces, 2019). The basic idea of SQP is to model nonlinear programs at a given approximate solution by a quadratic programming subproblem, and then use the solution of this subproblem to construct a better approximation in the next iteration (Boggs & Tolle, 1995).

Most algorithms developed for quadratic programming can be categorized into two classes: active set methods and interior-point methods. Active set methods begin by guessing the optimal active set of constraints, which are constraints that hold with equality at the current point. The methods repeatedly drop one index from the current active set and add a new one until the optimal set is detected (Hüeber, Mair, & Wohlmuth, 2005; Yu, Lin, & Hung, 2009). Interior-point methods were developed from Karmarkar's algorithm for linear programming (Karmarkar, 1984). They approach a solution by traversing the interior of the feasible region (Ternet & Biegler, 1999; Wright, 1992; Wang & Bai, 2009). We focus on interior-point methods as their worst-case time complexity is polynomial while that of active set methods is exponential. Additionally, it is much easier to exploit the sparsity pattern in interior-point methods than active set ones (Potra & Wright, 2000).

*Corresponding author
Email address: gun@siit.tu.ac.th

For large scale quadratic programs, the numbers of variables are so large that they cannot be solved straightforwardly in reasonable amount of time. Moreover, storing such large amount of data is impractical. For these reasons, many methods were proposed that exploit sparsity in the problems to reduce computational time. For example, Rosen and Pardalos (1986) proposed a method for large-scale constrained concave quadratic programming problems, which reduced a problem to an equivalent separable quadratic program and then solved a multiple-cost-row linear program with *2n* cost rows, where *n* was the dimension of the variable. If the solution was not a satisfactory approximation, a guaranteed ϵ-approximate solution was obtained by solving a single linear zero-one mixed integer programming problem. Gill *et al.* (U.S. Systems Optimization Lab., Stanford University, 1986) proposed a method based on the Schur complement. Their method is suitable for the problem with specialized factorization. Gould and Toint (Gould & Toint, 2002) proposed a method to solve large-scale nonconvex quadratic programming problems by using a working-set method. It is a two-level iterative method. The first level is to select the working set of constraints. The second level uses preconditioned conjugate gradient method to solve the problem with the selected working set. Hui-Min Li and Ke-Cun Zhang (Li & Zhang, 2006) decomposed the large-scale quadratic problem into a series of small problems and then solved these small problems serially to approximate the solution.

In this article, we study the quadratic programs whose Hessian matrix in the objective function is block diagonal with dense linearly inequality constraint matrices. For each iteration, an interior point method computes the search direction that improves the approximate solution. (Nocedal & Wright, 2006) However, computing the search direction for large scale problems takes prohibitively expensive computational time. To improve the efficiency when the Hessian matrix has the specific sparsity pattern, we propose an efficient method of compute the search direction for such special cases. We note that the computed search direction is the same one as in the traditional interior point method. By exploiting the known sparsity pattern of the problem, our method efficiently computes the search directions of an interior-point method for such quadratic programs without compromising the optimality of the method.

We begin by defining the block diagonal quadratic programs and reviewing primal-dual path-following interior-point methods in Section 2. Section 3 describes our method for the block diagonal quadratic problems. Section 4 shows the computational results. Section 5 shows the conclusion.

2. Block diagonal quadratic programs and primal-dual path-following interior-point method

Consider a block diagonal quadratic program with linear inequality constraints

$$\min_{x\in\mathbb{R}^n} f(x) = \frac{1}{2}x^T Hx + g^T x, \quad \text{subject to } Ax \geq b \tag{1}$$

where $x \in \mathbb{R}^n$, $H \in \mathbb{R}^{n\times n}$, $g \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$, and $b \in \mathbb{R}^m$. The Hessian matrix H is in the form

$$H = \begin{bmatrix} H_1 & 0 & \cdots & 0 \\ 0 & H_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & H_N \end{bmatrix},$$

where $H_i \in \mathbb{R}^{n_i\times n_i}$ (i = 1, 2,..., N) and N is the number of diagonal blocks in H. Note that H is symmetric positive semidefinite if and only if H_i's are symmetric positive semidefinite. Recall that $\sum_{i=1}^{N} n_i = n$. This Hessian structure is called block diagonal matrix.

Primal-dual path-following interior-point methods for quadratic programming use perturbed KKT conditions

$$f(x, y, \lambda; \sigma, \mu) = \begin{bmatrix} Hx - A^T\lambda + g \\ Ax - y - b \\ Y\Lambda e - \sigma\mu e \end{bmatrix} = 0 \tag{2}$$

where $\mu = y^T\lambda/m$, $Y = \mathrm{Diag}(y_1, y_2, \ldots, y_m)$, $\Lambda = \mathrm{Diag}(\lambda_1, \lambda_2, \ldots, \lambda_m)$, $e = [1,1,\ldots,1]^T$, $\sigma \in [0,1)$, and m is the number of constraints. Note that the variables y and λ are dual variables of (1).

Let (x^0, y^0, λ^0) be a starting point, not necessarily feasible, such that $(y^0, \lambda^0) > 0$. A primal-dual path-following interior-point method iterates by solving

$$\begin{bmatrix} H & 0 & -A^T \\ A & -I & 0 \\ 0 & \Lambda & Y \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} -w \\ -z \\ v \end{bmatrix} \tag{3}$$

for the search direction $(\Delta x, \Delta y, \Delta\lambda)$, where $w = Hx - A^T\lambda + g$, $z = Ax - y - b$, and $v = -\Lambda Ye + \sigma\mu e$, setting the next point to be

$$(x^{k+1}, y^{k+1}, \lambda^{k+1}) = (x^k, y^k, \lambda^k) + \alpha(\Delta x, \Delta y, \Delta\lambda), \tag{4}$$

where $\alpha \in (0,1)$ is the step length, and repeating until μ is close to 0 (Nocedal & Wright, 2006). The step length is typically chosen as the largest number to obtain $(y^{k+1}, \lambda^{k+1}) \geq 0$. Note that the "normal equations" form of (3) is

$$(H + A^T Y^{-1}\Lambda A)\Delta x = -w + A^T Y^{-1}\Lambda[-z - y + \sigma\mu\Lambda^{-1}e], \tag{5}$$

which can be solved by means of a modified Cholesky algorithm. Solving (5) for Δx is efficient if the term $A^T Y^{-1}\Lambda A$ is not too dense compared with H. In the case of

H being block diagonal, $H + A^T Y^{-1} \Lambda A$ is generally dense therefore we cannot take advantage of sparsity when solving (5).

3. Derivation of our Method

Traditionally, the search direction for each step is computed by solving (5) for Δx using the large Hessian matrix H and then substituting it to compute Δy and $\Delta\lambda$ from (3). Our method computes the same search direction but from the smaller nonzero diagonal blocks H_i. It divides the other variables corresponding to the size of each H_i. To take advantage of block diagonal Hessian, write x, g, A, and w as

$$\begin{aligned} x &= [x_1^T \quad x_2^T \quad \cdots \quad x_N^T]^T, \\ g &= [g_1^T \quad g_2^T \quad \cdots \quad g_N^T]^T, \\ A &= [A_1 \quad A_2 \quad \cdots \quad A_N], \\ w &= [w_1^T \quad w_2^T \quad \cdots \quad w_N^T]^T, \end{aligned}$$

where $x_i \in \mathbb{R}^{n_i}$, $g_i \in \mathbb{R}^{n_i}$, $A_i \in \mathbb{R}^{m \times n_i}$, and $w_i \in \mathbb{R}^{n_i}$. Our method computes the search direction using these smaller variables. Rewrite (3) as

$$H_i \Delta x_i - A_i^T \Delta\lambda = -w_i \ \ (i = 1, \ldots, N), \tag{6}$$

$$\sum_{i=1}^{m} A_i \Delta x_i - \Delta y = -z, \tag{7}$$

$$\Lambda \Delta y + Y \Delta\lambda = v \tag{8}$$

where

$$w_i = H_i x_i - A_i^T \lambda + g_i \quad (i = 1, \ldots, N), \tag{9}$$

Next, we rewrite (6) and (8) as

$$\Delta x_i = H_i^{-1}(A_i^T \Delta\lambda - w_i) \ (i = 1, \ldots, N), \tag{10}$$

$$\Delta y = \Lambda^{-1}(v - Y\Delta\lambda). \tag{11}$$

Finally, substituting (10) and (11) into (7) yields

$$\sum_{i=1}^{m} A_i H_i^{-1}(A_i^T \Delta\lambda - w_i) - \Lambda^{-1}(v - Y\Delta\lambda) = -z,$$

or, equivalently,

$$\left(\sum_{i=1}^{m} A_i H_i^{-1} A_i^T + \Lambda^{-1} Y\right)\Delta\lambda = -z + \Lambda^{-1} v + \sum_{i=1}^{m} A_i H_i^{-1} w_i. \tag{12}$$

Therefore, the search direction can be computed by solving (12) for $\Delta\lambda$ and obtain Δx_i and Δy from (10) and (11), respectively. Algorithm 1 below describes the path-following interior-point method that uses the proposed method to compute the search direction.

Algorithm 1

Set $S = 0$
Let (x_0, y_0, λ_0) be a point with $y_0, \lambda_0 \geq 0$
$\mu = y_0^T \lambda_0 / m$
for $i = 1, 2, 3, \ldots, N$ do
    $S = S + A_i H_i^{-1} A_i^T$
end for
for $k = 0, 1, 2, \ldots$ do
    Set $x, y, \lambda = x_k, y_k, \lambda_k$
    Compute $z = Ax - y - b$
    Compute $v = -\Lambda Y e + \sigma\mu e$
    Set $t = 0$
    for $i = 1, 2, 3, \ldots, N$ do
        Compute w_i from (9)
        $t = t + A_i H_i^{-1} w_i$
    end for
    Solve $(S + \Lambda^{-1} Y)\Delta\lambda = -z + \Lambda^{-1} v + t$ for $\Delta\lambda$
    $\Delta y = \Lambda^{-1}(v - Y\Delta\lambda)$
    for $i = 1, 2, 3, \ldots, N$ do
        Solve $H_i \Delta x_i = -w_i + A_i^T \Delta\lambda$ for Δx_i
    end for
    $\alpha_k^{pri} = \max\{\alpha \in (0,1] : y + \alpha\Delta y \geq (1 - \tau)y\}$
    $\alpha_k^{dual} = \max\{\alpha \in (0,1] : \lambda + \alpha\Delta\lambda \geq (1 - \tau)\lambda\}$
    Select $\alpha = \min(\alpha_k^{pri}, \alpha_k^{dual})$
    Set $x_{k+1}, y_{k+1}, \lambda_{k+1} = (x_k, y_k, \lambda_k) + \alpha(\Delta x, \Delta y, \Delta\lambda)$
    $\mu = (y_{k+1}^T \lambda_{k+1})/m$
end for

Remarks for the Algorithm 1

- We do not explicitly compute H_i^{-1}'s. Instead, we precompute the Cholesky factors of H_i's once and reuse them to compute S, t, and Δx_i.
- The direction Δy can be computed efficiently because Λ is diagonal.
- The parameter $\tau \in (0,1)$ controls how far we back off from the maximum step.
- Instead of computing w directly, we compute each w_i from)9(.

Algorithm 1 presents the interior point method using our proposed method for computing the search direction. Normally, for iteration k, the interior point method computes the search direction $(\Delta x, \Delta y, \Delta\lambda)$, determines the step length α, updates the solution $(x_{k+1}, y_{k+1}, \lambda_{k+1})$, and recomputes μ. The method repeats until μ converges to zero. Instead of computing the search direction by solving (5) for Δx and then substituting it to obtain Δy and $\Delta\lambda$, our method first computes $\sum_{i=1}^{m} A_i H_i^{-1} A_i^T$ once and use it in all iteration. For each iteration, we compute $\Delta\lambda$ from (12) and then Δy from (11). Finally, we compute each Δx_i (i = 1, 2,..., N) from (10). These computation are shown on line 16 to 20 of Algorithm 1. Recall that N is the number of diagonal blocks in H and m is the number of constraints.Our algorithm requires $O(m^2N + \sum_{i=1}^{N}(n_i^3 + mn_i^2))$ operations for the preprocessing and $O(m^3 + \sum_{i=1}^{N}(n_i^2 + mn_i))$ operations per iterate. As comparison, note that the conventional interior-point method that solves (5) for search directions requires $O(n^3)$ per iterate.

4. Computational Results

In this section we compare the computational time of the following three methods for computing search directions for block diagonal quadratic programs in MATLAB R2011a: (i) solving (5) for Δx and then substituting it to compute Δy and $\Delta\lambda$, (ii) solving (3) using MATLAB sparse linear solver, and (iii) our method as described in Section 3. Method (i) is the regular method to compute search direction without consider the sparsity of matrix. Method (ii) stores the variables in the sparse format and uses the sparse linear solver to compute the search direction. As these three methods compute the same solution but in different ways and the solution is too large to report in the tables, we show only the computational time in our experimental results. While there are publically available datasets for quadratic programming, to the best of our knowledge, there are none with block diagonal Hessian matrices with linear constraints. Moreover, the running time of our method does not depend on the values of the input but on the number of diagonal blocks and the sizes of each blocks. For this reason, we test the three methods on randomly generated data with varying sizes. The experiment was performed on different problem sizes varying from 100 to 2,500 variables on a computer with an Intel Pentium M 740 1.73 GHz processor and 2 GB RAM. For each problem size, we compare average computation time per iterate of problems with different numbers of equally-sized diagonal blocks. We also vary the number of constraints for 20, 50, and 80 percent of the number of variables. We use the same $\sigma = 0.5$ and $\tau = 0.9$ for the three methods in our experiment. Although the values of σ and τ can change the number of iterations needed to converge, they do not affect the required computation time in each iteration. Since the three methods always use the same number of iterations to converge, the chosen values do not affect the comparison between the methods in any case as they differ only in their computation time in each iteration. For each case, we test with ten different instances. The results are shown in Tables 1-7. We show average number of iterates ($iter.$) for each problems. Columns t_1, t_2, and t_3 show average time per iterate for methods (i), (ii), and (iii), respectively. Note that average time per iterate in our experiment also includes preprocessing time.

Table 1. Average number of iterates and average time per iterate of interior-point, interior-point with sparse matrix and our method for 100 variable problems with different number of constraints

N	n_i	m = 20				m = 50				m = 80			
		iter.	t_1 (ms)	t_2 (ms)	t_3 (ms)	*iter.*	t_1 (ms)	t_2 (ms)	t_3 (ms)	*iter.*	t_1 (ms)	t_2 (ms)	t_3 (ms)
2	50	19.8	0.983	3.783	0.457	20.2	1.595	11.021	0.705	21.5	2.195	12.746	1.130
5	20	19.7	1.016	2.710	0.619	20.7	1.521	6.555	0.874	21.8	2.135	11.115	1.288
10	10	19.7	0.983	2.501	0.990	20.7	1.523	5.335	1.240	21.8	2.075	9.718	1.684
20	5	19.5	0.988	2.077	1.707	21.0	1.570	5.094	1.969	21.6	2.140	9.472	2.396
50	2	19.5	0.984	2.087	3.917	20.6	1.527	4.820	4.205	21.7	2.054	9.459	4.693

Table 2. Average number of iterates and average time per iterate of interior-point, interior-point with sparse matrix and our method for 500 variable problems with different number of constraints

N	n_i	m = 100				m = 250				m = 400			
		iter.	t_1 (ms)	t_2 (ms)	t_3 (ms)	*iter.*	t_1 (ms)	t_2 (ms)	t_3 (ms)	*iter.*	t_1 (ms)	t_2 (ms)	t_3 (ms)
2	250	20.0	32.24	88.61	9.86	21.6	54.94	174.40	19.43	23.0	97.63	494.15	52.52
5	100	20.3	32.24	56.93	4.17	21.7	54.98	185.82	14.19	23.2	97.71	389.89	43.78
10	50	20.6	32.47	52.57	4.23	21.9	55.02	149.12	14.01	23.1	97.68	338.65	42.24
20	25	20.2	32.90	47.16	4.63	21.6	55.02	141.52	14.46	23.4	97.70	311.20	43.55
25	20	20.2	33.57	39.74	4.89	21.3	55.05	137.02	15.01	23.3	97.79	315.73	44.14
50	10	20.4	33.65	39.10	6.81	21.8	54.86	138.17	17.26	23.4	97.67	313.92	48.28
100	5	20.5	33.78	37.86	10.62	21.6	55.04	134.20	21.16	22.8	97.72	309.68	56.39
250	2	20.4	33.05	38.24	22.28	21.6	55.05	132.01	34.17	23.3	97.65	307.25	80.44

Table 3. Average number of iterates and average time per iterate of interior-point, interior-point with sparse matrix and our method for 1,000 variable problems with different number of constraints

N	n_i	m = 200				m = 500				m = 800			
		iter.	t_1 (ms)	t_2 (ms)	t_3 (ms)	*iter.*	t_1 (ms)	t_2 (ms)	t_3 (ms)	*iter.*	t_1 (ms)	t_2 (ms)	t_3 (ms)
2	500	20.5	0.183	0.408	0.067	21.8	0.339	0.804	0.140	23.3	0.647	2.592	0.310
5	200	20.6	0.183	0.262	0.019	22.1	0.340	0.879	0.095	23.7	0.652	1.946	0.263
10	100	20.8	0.184	0.242	0.015	22.0	0.338	0.745	0.082	24.0	0.653	1.723	0.256
20	50	21.0	0.183	0.219	0.014	22.0	0.337	0.689	0.081	23.8	0.652	1.624	0.242
25	40	20.9	0.183	0.188	0.014	22.0	0.337	0.680	0.081	23.9	0.651	1.572	0.243
40	25	21.0	0.183	0.185	0.016	22.0	0.338	0.682	0.084	23.7	0.652	1.562	0.249
50	20	20.5	0.182	0.180	0.016	21.9	0.337	0.660	0.086	24.0	0.652	1.581	0.254
100	10	20.9	0.183	0.176	0.020	22.2	0.339	0.706	0.099	24.3	0.653	1.606	0.283
200	5	20.5	0.183	0.177	0.029	22.0	0.341	0.638	0.123	23.8	0.652	1.569	0.342
500	2	20.6	0.183	0.179	0.054	22.4	0.347	0.649	0.193	24.0	0.652	1.579	0.508

Table 4. Average number of iterates and average time per iterate of interior-point, interior-point with sparse matrix and our method for 1,500 variable problems with different number of constraints

N	n_i	m = 300				m = 750				m = 1200			
		iter.	t_1 (ms)	t_2 (ms)	t_3 (ms)	*iter.*	t_1 (ms)	t_2 (ms)	t_3 (ms)	*iter.*	t_1 (ms)	t_2 (ms)	t_3 (ms)
2	750	20.5	0.502	0.983	0.157	22.1	0.996	1.974	0.361	23.8	2.010	7.208	0.887
5	300	20.8	0.503	0.632	0.082	22.2	1.001	2.264	0.281	24.4	2.005	5.406	0.819
10	150	20.8	0.503	0.585	0.037	22.5	1.002	1.818	0.243	24.0	2.013	4.740	0.784
20	75	20.9	0.503	0.516	0.039	22.2	1.001	1.696	0.224	24.2	2.015	4.308	0.782
25	60	21.0	0.503	0.446	0.035	22.3	1.002	1.724	0.217	24.3	2.015	4.257	0.783
30	50	21.0	0.502	0.438	0.036	22.4	1.001	1.684	0.219	24.1	2.016	4.295	0.745
50	30	20.9	0.503	0.432	0.038	22.1	1.003	1.656	0.226	24.3	2.017	4.264	0.763
60	25	20.9	0.503	0.428	0.039	22.4	1.003	1.671	0.231	24.1	2.016	4.253	0.777
75	20	21.1	0.504	0.423	0.040	22.5	1.003	1.644	0.238	24.4	2.016	4.154	0.792
150	10	20.9	0.503	0.416	0.050	22.5	1.002	1.644	0.278	24.1	2.016	4.117	0.880
300	5	20.8	0.503	0.418	0.069	22.2	1.003	1.619	0.362	24.2	2.015	4.188	1.061
750	2	20.7	0.503	0.423	0.124	22.5	1.001	1.599	0.601	24.2	2.001	4.227	1.587

Table 5. Average number of iterates and average time per iterate of interior-point, interior-point with sparse matrix and our method for 2,000 variable problems with different number of constraints

N	n_i	m = 400				m = 1000				m = 1600			
		iter.	t_1 (ms)	t_2 (ms)	t_3 (ms)	*iter.*	t_1 (ms)	t_2 (ms)	t_3 (ms)	*iter.*	t_1 (ms)	t_2 (ms)	t_3 (ms)
2	1000	20.6	1.107	1.835	0.318	22.0	2.724	3.865	0.779	24.1	5.804	17.512	1.939
5	400	21.0	1.110	1.173	0.162	22.7	2.724	4.522	0.626	24.4	5.805	11.573	1.786
10	200	20.9	1.112	1.080	0.096	22.5	2.734	3.597	0.559	24.5	5.805	10.088	1.733
20	100	21.1	1.110	0.976	0.080	22.9	2.737	3.401	0.544	24.4	5.808	9.469	1.725
25	80	21.0	1.111	0.842	0.073	22.5	2.734	3.376	0.542	24.3	5.809	9.157	1.725
40	50	21.0	1.108	0.814	0.067	22.5	2.740	3.326	0.505	24.5	5.815	8.901	1.737
50	40	21.1	1.109	0.794	0.068	22.4	2.736	3.239	0.512	24.3	5.813	9.067	1.696
80	25	21.2	1.110	0.801	0.073	22.5	2.737	3.333	0.536	24.2	5.813	8.923	1.751
100	20	20.9	1.105	0.779	0.077	22.4	2.734	3.686	0.553	24.6	5.816	9.291	1.788
200	10	21.0	1.110	0.779	0.094	22.7	2.734	3.195	0.646	24.5	5.817	9.113	1.983
400	5	21.0	1.109	0.780	0.129	22.5	2.734	3.089	0.830	24.7	5.810	9.086	2.375
1000	2	21.0	1.109	0.798	0.234	22.6	2.723	3.211	1.365	24.6	5.805	9.290	3.553

Table 6. Average number of iterates and average time per iterate of interior-point, interior-point with sparse matrix and our method for 2,500 variable problems with different number of constraints

N	n_i	m = 300				m = 750				m = 1200			
		iter.	t_1 (ms)	t_2 (ms)	t_3 (ms)	*iter.*	t_1 (ms)	t_2 (ms)	t_3 (ms)	*iter.*	t_1 (ms)	t_2 (ms)	t_3 (ms)
2	1250	21.0	1.948	3.002	0.486	22.4	4.743	6.394	1.249	24.5	11.202	34.712	3.539
5	500	21.1	1.982	1.903	0.275	22.7	4.655	7.647	1.034	24.4	11.403	21.441	3.326
10	250	21.0	2.014	1.777	0.177	22.3	4.693	6.078	0.941	24.6	11.384	19.382	3.242

Table 6. Continued.

N	n_i	m = 300				m = 750				m = 1200			
		iter.	t_1 (ms)	t_2 (ms)	t_3 (ms)	*iter.*	t_1 (ms)	t_2 (ms)	t_3 (ms)	*iter.*	t_1 (ms)	t_2 (ms)	t_3 (ms)
20	125	21.0	2.016	1.649	0.115	22.7	4.680	6.012	0.916	24.7	11.363	17.280	3.240
25	100	21.1	2.040	1.346	0.119	22.6	4.711	5.826	0.915	24.2	11.353	16.855	3.242
50	50	21.2	2.032	1.283	0.116	22.5	4.751	5.579	0.860	24.6	11.354	17.106	3.280
100	25	21.0	2.021	1.259	0.125	22.6	4.743	5.531	0.922	24.7	11.245	16.598	3.297
125	20	20.9	2.028	1.246	0.130	22.7	4.743	5.642	0.955	24.6	10.861	16.112	3.377
250	10	21.0	2.033	1.340	0.163	22.7	4.792	5.480	1.117	24.8	11.017	15.959	3.758
500	5	21.1	2.040	1.255	0.228	22.8	4.772	5.565	1.444	24.9	11.157	16.481	4.520
1250	2	21.1	2.021	1.276	0.432	22.6	4.790	5.398	2.458	24.8	11.289	16.293	6.773

Table 7. Average number of iterates and average time per iterate of interior-point, interior-point with sparse matrix and our method for 4,000 variable problems with different number of constraints

N	n_i	m = 300				m = 750				m = 1200			
		iter.	t_1 (ms)	t_2 (ms)	t_3 (ms)	*iter.*	t_1 (ms)	t_2 (ms)	t_3 (ms)	*iter.*	t_1 (ms)	t_2 (ms)	t_3 (ms)
2	2000	21.0	7.093	9.173	1.547	22.7	16.780	-	4.207	24.9	40.524	-	11.873
5	800	21.1	7.050	5.691	0.825	22.9	17.068	-	3.435	25.3	40.825	-	11.121
10	400	21.2	7.047	5.139	0.594	22.8	16.856	-	3.252	25.0	41.462	-	10.837
20	200	21.1	7.032	4.568	0.450	23.0	16.538	-	3.155	25.0	41.965	-	10.780
25	160	21.1	7.029	3.903	0.444	23.1	16.670	-	3.137	25.2	41.406	-	10.793
50	80	21.2	7.025	3.737	0.353	23.0	16.701	-	3.151	25.1	40.909	-	10.930
80	50	21.3	7.020	3.758	0.363	23.0	16.684	-	3.205	25.1	40.563	-	11.096
160	25	21.4	7.016	3.615	0.401	23.0	16.639	-	3.250	25.2	40.657	-	11.571
200	20	21.0	7.019	3.620	0.478	23.0	16.646	-	3.424	25.5	40.993	-	11.549
400	10	21.0	7.024	3.616	0.623	23.0	16.618	-	4.197	25.1	40.821	-	13.035
800	5	21.4	7.023	3.623	0.838	23.0	16.559	-	5.526	25.2	41.137	-	16.017
2000	2	21.3	7.021	3.758	1.610	23.0	16.453	-	9.285	25.1	40.840	-	24.824

The results of experiment show that, for problems with the same number of variables and constraints, average time per iterate of method (i) does not depend on the number of diagonal blocks, which is as expected since method (i) does not take advantage of any sparsity in the data. Method (ii), on the other hand, is more efficient for problems with many smaller diagonal blocks than for problems with few larger diagonal blocks. This is because a problem with few larger diagonal blocks has more nonzero elements compared to a problem with many smaller diagonal blocks. Finally, method (iii) performs best when the number of diagonal blocks is neither too large nor too small. In other words, it performs best when the number of groups is about the same as the number of variables in each group. This is because a larger number of (small) blocks implies the linear systems that the method has to solve are smaller. On the other hand, if the number of blocks is too high, the method has to solve many linear systems, which incur high overhead in MATLAB causing higher computation time.

In our experiment, the constraint matrices are dense. When the number of constraints is very high, method (ii) is the slowest among the three methods. The reason is that method (ii) stores the variables in the sparse format, which is not suitable for large and dense matrix computation. On large problems, such as those with 500 variables or more, method (iii) is the fastest among the three. We note that for a small number of constraints, the method (iii) is significantly faster than the other two methods.

Note that we do not have results of method (ii) for problems with 4,000 variables and 2,000 or more constraints. This is because the matrix in (3) that is used by method (ii) is too large and too dense to store in the main memory of our system. However, method (i) and method (iii) do not have this problem and can compute the search directions normally.

5. Conclusions

This article proposes an efficient method to compute search directions of primal-dual interior-point method for block diagonal quadratic programs. Our method separates variables according to the diagonal blocks of the Hessian matrix and uses the components to compute search directions. Our method has better time complexity, and we show experimentally that it uses less computational time than conventional methods for computing search directions. It is seen that our method is suitable for large scale problem with either small or large numbers of constraints. In any case, the advantage of our method is limited to the problems whose Hessian matrices are block diagonal; the method cannot be applied to quadratic programs with other sparsity patterns.

Acknowledgements

Authors gratefully acknowledge the financial support provided by Thammasat University Research Fund under the TU Research Scholar, Contract No. TP 2/24/2560.

References

Bartlett, R. A., Biegler, L. T., Backstrom, J., & Gopal, V. (2002). Quadratic programming algorithms for large-scale model predictive control. *Journal of Process Control, 13*(7), 775–795. doi:10.1016/S0959-1524(02)00002-1

Boggs, P. T., & Tolle, J. W. (1995). Sequential quadratic programming. *Acta Numerica, 4*, 1–51. doi:10.1017/S0962492900002518

Gould, N. I. M., & Toint, P. L. (2002). An iterative working-set method for large-scale nonconvex quadratic programming. *Applied Numerical Mathematics, 43* (1-2), 109–128. doi:10.1016/S0168-9274(02)00120-4

Hüeber, S., Mair, M., & Wohlmuth, B. (2005). A priori error estimates and an inexact primal-dual active set strategy for linear and quadratic finite elements applied to multibody contact problems. *Applied Numerical Mathematics, 54*, 555–576. doi:10.1016/j.apnum.2004.09.019

Karmarkar, N. (1984). A new polynomial-time algorithm for linear programming. *Combinatorica, 4*(4), 373–395. doi:10.1007/BF02579150

Kim, H., & Rassias, J. M. (2007). Generalization of Ulam stability problem for Euler–Lagrange quadratic mappings. *Journal of Mathematical Analysis and Applications, 336*(1), 277–296. doi:10.1016/j.jmaa.2007.02.075

Kouzoupis, D. (2018). Recent advances in quadratic programming algorithms for nonlinear model predictive control. *Vietnam Journal of Mathematics, 46*, 863–882. doi:10.1007/s10013-018-0311-1

Li, H., & Zhang, K. (2006). A decomposition algorithm for solving large-scale quadratic programming problems. *Applied Mathematics and Computation, 173*(1), 394–403. doi:10.1016/j.amc.2005.04.076

Liu, X., Wang, D., & Rong, J. (2009). Quadratic prediction and quadratic sufficiency in finite populations. *Journal of Mathematical Analysis and Applications, 100*(9), 1979–1988. doi:10.1016/j.jmva.2009.04.010

Mitsuia, K., & Tabata, Y. (2008). A stochastic linear–quadratic problem with Lévy processes and its application to finance. *Stochastic Processes and their Applications, 118*(1), 120–152. doi:10.1016/j.spa.2007.03.011

Montoya, O. D., Gil-Gonzáles, W., & Garces, A. (2019). Sequential quadratic programming models for solving the OPF problem in DC grids. *Electric Power Systems Research, 169*, 18–23. doi:10.1016/j.epsr.2018.12.008

Nocedal, J., & Wright, S. J. (2006). *Numerical optimization*. Berlin, Germany: Springer.

Potra, F. A., & Wright, S. J. (2000). Interior-point methods. *Journal of Computational and Applied Mathematics, 124*(1-2), 281–302. doi:10.1016/S0377-0427(00)00433-7

Rosen, J., & Pardalos, P. (1986). Global minimization of large-scale constrained concave quadratic problems by separable programming. *Applied Mathematics and Computation, 34*, 163–174. doi:10.1007/BF01580581

Ternet, D. J., & Biegler, L. T. (1999). Interior-point methods for reduced Hessian successive quadratic program ming. *Computers and Chemical Engineering, 23*, 859–873. doi:10.1016/S0098-1354(99)00013-7

U. S. Systems Optimization Lab., Stanford University. (1986). *A Schur-complement method for sparse quadratic programming.* (Report Number SOL-87-12) Retrieved from http://www.ccom.ucsd.edu/~peg/papers/schurQP.pdf

Wang, G., & Bai, Y. (2009). Primal-dual interior-point algorithm for convex quadratic semi-definite optimization. *Nonlinear Analysis, 71*, 3389–3402. doi:10.1016/j.na.2009.01.241

Wright, M. (1992). Interior methods for constrained optimization. *Acta Numerica, 1*, 341-407. doi:10.1017/S0962492900002300

Yu, M., Lin, T., & Hung, C. (2009). Active-set sequential quadratic programming method with compact neighbourhood algorithm for the multi-polygon mass production cutting-stock problem with rotatable polygons. *International Journal of Production Economics, 121*, 148–161. doi:10.1016/j.ijpe.2009.01.014

Zhang, H., Zhong, W., Wu, C., & Liao, A. (2006). Some advances and applications in quadratic programming method for numerical modeling of elastoplastic contact problems. *International Journal of Mechanical Sciences, 48*(2), 176–189. doi:10.1016/j.ijmecsci.2005.08.003

Zhang, H. W., Xu, W. L., Di, S. L., & Thomson, P. F. (2002). Quadratic programming method in numerical simulation of metal forming process. *Computer Methods in Applied Mechanics and Engineering, 191*(49-50), 5555–5578. doi:10.1016/S0045-7825(02)00462-0