EXPLOITING SPARSITY FOR LARGE-SCALE QUADRATIC PROGRAMMING

BY

DUANGPEN JETPIPATTANAPONG

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY (TECHNOLOGY)
SIRINDHORN INTERNATIONAL INSTITUTE OF TECHNOLOGY
THAMMASAT UNIVERSITY
ACADEMIC YEAR 2016

EXPLOITING SPARSITY FOR LARGE-SCALE QUADRATIC PROGRAMMING

BY

DUANGPEN JETPIPATTANAPONG

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY (TECHNOLOGY)
SIRINDHORN INTERNATIONAL INSTITUTE OF TECHNOLOGY
THAMMASAT UNIVERSITY
ACADEMIC YEAR 2016

EXPLOITING SPARSITY FOR LARGE-SCALE QUADRATIC PROGRAMMING

A Dissertation Presented

By

DUANGPEN JETPIPATTANAPONG

Submitted to
Sirindhorn International Institute of Technology
Thammasat University
In partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY (TECHNOLOGY)

Approved as to style and content by

Advisor and Chairperson of Thesis Committee	[illegible] (Asst. Prof. Gun Srijuntongsiri, Ph.D.)
Co-Advisor	[illegible] (Prof. Stanislav S. Makhanov, Ph.D.)
Committee Member and Chairperson of Examination Committee	[illegible] (Assoc. Prof. Cholwich Nattee, D.Eng.)
Committee Member	P. Aimmanee (Asst. Prof. Pakinee Aimmanee, Ph.D.)
Committee Member	[illegible] (Assoc. Prof. Weerachai Anotaipaiboon, Ph.D.)

External Examiner: Prof. Prabhas Chongstitvatana, Ph.D.

August 2016

Abstract

EXPLOITING SPARSITY FOR LARGE SCALE QUADRATIC PROGRAMMING

by

DUANGPEN JETPIPATTANAPONG

Bachelor of Engineering (Computer Engineering), King Mongkut's University of Technology Thonburi, 1998
Master of Engineering (Computer Engineering), Kasetsart University, 2002
Doctor of Philosophy (Technology), Sirindhorn International Institute of Technology Thammasat Univesity, 2017

Quadratic programming is a class of constrained optimization problems with quadratic objective function and linear constraints. It is an important optimization problem with applications in many areas and is also used to solve nonlinear optimization problems. Quadratic programs arisen in practice are often large, but sparse, and have a special Hessian structure which we can exploit. They usually cannot be solved efficiently without exploiting their structures.

This thesis proposes methods for solving several classes of quadratic programming with structure. We show a heuristic method for the large-scale quadratic programs with block diagonal Hessian matrices and dense constraint matrices. Our method separates the problem into smaller problems, computes optimal solutions for the smaller problems, and uses them to construct the solution to the original problem. Computational results show that our method is highly efficient at computing approximate solutions for large-scale problems. The other method is an efficient method to compute the search directions for the primal-dual path-following interior-point method for the similar class of Hessian matrix structure and dense constraint matrices. The time complexity of the method is significantly smaller than that of using a sparse linear solver. The computational results also show that the proposed method is faster.

This thesis also proposes a pivot selection algorithm for the factorization of the Karush-Kuhn-Tucker (KKT) matrix for the equality constrained quadratic programs whose constraint matrices are block diagonal. Such factorization should maintain both sparsity and numerical stability of the factors, both of which depend solely on the choices of the pivots. The proposed method maintains the sparsity and stability of the problem. The experiments show that the pivot selection algorithm appears to produce no fill-ins in the factorization of such matrices. In addition, we compare the method with MA57 and find that the factors produced by our method are sparser. Finally, we propose a pivot selection technique for symmetric indefinite factorization of sparse matrices. Our method is based on the minimum degree algorithm and also considers the stability of the factors at the same time. The experiments show that our method produces factors that are sparser than the factors computed by MA57 and are stable.

Keywords: Large-scale Quadratic programming, Block diagonal constraint, Heuristic Algorithm, Separable Quadratic Optimization, Interior-point method, Primal-dual path following, Symmetric indefinite factorization

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor, Asst. Prof. Dr. Gun Srijuntongsiri for the continuous support of my Ph.D study and research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank the rest of my co-advisor Prof. Dr. Stanislav S. Makhanov and thesis committee: Asst. Prof. Dr. Cholwich Nattee, Asst. Prof. Dr. Pakinee Aimmanee, and Asst. Prof. Dr. Weerachai Anotaipaiboon for their insightful comments and encouragement, but also for the hard question which incited me to widen my research from various perspectives.

Last but not the least, I would like to thank my parents, for giving birth to me in the first place and supporting me spiritually throughout my life.

Duangpen Jetpipattanapong

Table of Contents

List of Tables

Chapter 1
Introduction

1.1 Quadratic programming

Nonlinear programming problems arise in the mathematical modeling of the real world problem. Quadratic programming is an important class of nonlinear programming. This kind of problem has a quadratic objective function and linear constraints. The general form of the quadratic programming problem is as follows

$$\min_{x\in\mathbb{R}^n} \frac{1}{2}x^T Hx + c^T x,$$
$$\text{subject to } Ax \geq b$$

where $x \in \mathbb{R}^n$, $H \in \mathbb{R}^{n\times n}$, $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, and $m < n$.

The objective function can be either convex or non-convex depending on the Hessian matrix H. When the Hessian matrix is positive or semidefinite, the objective function is convex. For the convex case, the local optimum is also the global optimum. If H is not positive semidefinite, the objective function is non-convex. Non-convex objective functions may have many local optimal solutions. Solving the indefinite quadratic function is difficult and required global optimization methods.

1.2 Quadratic programming with applications

Quadratic programming arises in many areas such as prediction, control, modeling, finance, engineering, and management [1-10]. Moreover, quadratic programming is used as a part of Sequential Quadratic Programming (SQP) approaches to solve nonlinear programming problems. 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 [11,12].

1.3 Overview of methods for solving quadratic programming

There are many methods for solving quadratic programs; they can be classified as either direct or iterative methods. The direct methods attempt to find the solution by directly solving the resulting linear systems with a finite number of operations, usually by using matrix factorization depending on the type of the matrix. As an example, symmetric positive matrices are factorized by Cholesky factorization. Iterative methods, on the other hand, start with an initial guess and successively generate better approximate solutions at each iteration. The running time of an iterative method depends directly on the required accuracy of the solution. Iterative methods have two well-known classes of methods: active set and interior-point methods. The 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 new one until the optimal set is detected [13, 14]. The interior-point methods are developed from the Karmarkar's algorithm for linear programming [15]. They approach a solution by traversing the interior of the feasible region [16-18]. Unlike direct methods, the running time of iterative methods depends on the required accuracy of the solutions.

1.4 Pivot selection in direct solution methods

Direct solution methods for solving quadratic programming typically involves symmetric indefinite factorization of the Karush-Kuhn-Tucker (KKT) matrix [19]. Symmetric indefinite factorization (SIF) is not unique as the resulting factors depend on the choices of the pivots during the factorization. Pivots should be chosen such that the resulting factors are stable and do not have many fill-ins−the entries that are zeros in the original matrix but are nonzeros in the factors.

There are many heuristic techniques for selecting pivots to minimize the number of fill-ins for the related problem of Cholesky factorization, which is the most suitable factorization for symmetric positive definite matrices, in literature. We briefly discuss a few such well-known techniques here since some of their ideas are also applicable to SIF. These ordering algorithms can be classified into three classes: local, global, and hybrid approaches. Local approach such as the minimum degree and the minimum fill algorithms [20-24] selects the pivot that is expected to minimize the number of fill-ins at each factorization step in a greedy fashion. Global approach such as Cuthill-McKee and nested dissection methods [25-27] selects pivots by considering the overall structure of the matrix. Hybrid approach, on the other hand, combines the ideas from both local and global approaches.

The well-known minimum degree algorithm [20] chooses the column that has the minimum off-diagonal nonzero elements in the remaining matrix as the pivot for the current step. Different improvements of the minimum degree algorithm have been proposed [23] such as multiple minimum degree [28] and approximate minimum degree algorithms [29] and become the practical standard in the implementations.

Another famous pivot selection algorithm is the nested dissection [26]. By defining a graph whose vertices represent each column of the matrix and whose edges represent nonzero entries in the matrix, nested dissection recursively find a separator−a set of vertices that partitions the graph into two disconnected subgraphs−and ordering the pivots recursively with the two subgraphs first followed by the separator vertices. Cuthill-McKee [25] propose another pivot selection algorithm that aims to reduce the bandwidth of the matrix based on a breadth first search of the structure graph.

The main difference between Cholesky factorization and SIF is in the size of pivots. For SIF, each pivot can be either a scalar or a 2-by-2 matrix while pivots in Cholesky factorization are all scalars. Moreover, unlike Cholesky factorization, the choice of pivots in SIF also affects the stability of the resulting factors [30].

There are many pivot selection algorithms proposed specifically for SIF such as Bunch-Parlett [31], Bunch-Kaufman [32], and bounded Bunch Kaufman (BBK) [33] algorithms. Bunch-Parlett method searches the whole remaining submatrix at each stage for the largest-magnitude diagonal and the largest-magnitude off-diagonal. It chooses the largest-magnitude diagonal as the 1-by-1 pivot if the resulting growth rate is acceptable. Otherwise, it selects the largest-magnitude off-diagonal and its relative diagonal elements as the 2-by-2 pivot block. This method requires $\mathrm{O}(n^3)$ comparisons and yields a matrix L whose maximum element is bounded by 2.781. Bunch-Kaufman pivoting strategy searches for the largest-magnitude off-diagonal elements of at most two columns for each iteration. It requires $\mathrm{O}(n^2)$ comparisons, but the elements in L are unbounded. BBK combines the two above strategies. By monitoring the size of the elements in L, BBK uses the Bunch-Kaufman strategy when it yields modest element growth. Otherwise, it repeatedly searches for an acceptable pivot. In average cases, the total cost of BBK is the same as Bunch-Kaufman, but in the worst cases its cost can be the same as that of the Bunch-Parlett strategy.

Moreover, when the KKT matrix is sparse, the choice of pivots also affects the sparsity of the resulting factors, which in turn affects the time needed to solve the linear system. Hence, choosing suitable pivots that both maintain stability and preserve sparsity is not trivial.

Additionally, there are other types of techniques for solving sparse symmetric indefinite linear systems. Paige and Saunders [34] propose two algorithms, SYMMLQ and MINRES, for solving such systems. The algorithms apply orthogonal factorization together with the conjugate gradient method to solve the system. Duff et al. [35] propose a pivotal strategy for decomposing sparse symmetric indefinite matrices. They use relative pivot tolerance by limits the magnitude of the element in the factors for stability and generalization of the criterion of Markowitz [20] to consider a 2-by-2 pivot for the sparsity. Olaf and Klaus [36] propose Supernode-Bunch-Kaufman pivoting method, which applies the Bunch-Kaufman pivot selection algorithm for the sparse case, supplemented by pivot perturbation techniques. Duff and Reid [37] propose a multifrontal method to solve indefinite sparse symmetric linear systems based on minimum degree ordering. The multifrontal approach is widely used in many sparse direct solvers such as MA57 and MUMPS [38, 39].

1.5 Large-scale quadratic programming

There are many problems that have to optimize large-scale quadratic problem. For large-scale quadratic programs, the numbers of variables are so large that they cannot be solved straightforwardly in a reasonable amount of time. Moreover, storing such large amount of data is impractical. Fortunately the large-scale problems are always sparse and have special structure such as separable or block-diagonal. For these reasons, many methods are proposed that exploit sparsity in the problems to reduce the computational time. For example, Rosen and Pardalos [40] propose a method for large-scale constrained concave quadratic programming problems, which reduces a problem to an equivalent separable quadratic program. Then solves a multiple-cost-row linear program with $2n$ cost rows, where n is the dimension of the variable. If the solution is not a satisfactory approximation, a guaranteed ϵ-approximate solution is obtained by solving a single linear zero-one mixed integer programming problem. [41] decompose the large-scale quadratic problem into a series of small problems and then solve these small problems serially to approximate the solution. Gill et al. [42] propose a method based on the Schur complement. Their method is suitable for the problem with specialized factorization. Gould and Toint [43] propose a method to solve large-scale nonconvex quadratic programming problems by using 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 the preconditioned conjugate gradient method to solve the problem with the selected working set.

1.6 Our objectives

This thesis focuses on two classes of quadratic programming, quadratic programs whose Hessian matrix is block diagonal with dense linearly constraint matrices and equality-constrained quadratic programs whose constraint matrices are block diagonal.

For the quadratic programs whose Hessian matrix is block diagonal, we propose two subclasses for this kind of structure. First subclass is the quadratic programs whose Hessian is a block diagonal structure with dense nonnegative linear constraint and lower bounds. We separate the problem into many smaller problems. For the first subproblem, we use the lower bounds of the other subproblems to calculate its optimal sub-solution. For the other subproblems, we use the optimal sub-solution and the lower bounds of the other subproblems to find their optimal sub-solutions. We repeat this step until all of the subproblems are solved. Then, we construct the approximate solution with these optimal sub-solutions. Our experiment shows the comparison between our heuristic method and an interior-point method. The result shows that our method can efficiently approximate solutions when there are not too many numbers of subproblems. The second subclass is the quadratic programs whose Hessian matrix in the objective function is block diagonal with dense linear inequality constraint matrices. We propose a way to efficiently compute the search directions of an interior-point method for such quadratic programs without compromising the optimality of the method.

The second class of quadratic programs, equality-constrained quadratic programs whose constraint matrices are block diagonal, often arises in practice when different groups of variables are independent but variables in the same group must satisfy some constraints. Using a direct method, we propose a pivot selection algorithm for this type of quadratic programs. By exploiting the known structure of the quadratic program, the algorithm can efficiently identify the pivot candidates that can maintain the sparsity of the factors. This work uses the condition number of each pivot candidates as part of the information for pivot selection in order to also maintain stability.

Finally, we propose a new pivot selection algorithm for sparse SIF. Our algorithm applies the idea of minimum degree ordering to consider both 1-by-1 and 2-by-2 pivots while also considers the stability of the resulting factors. Our experiments show that our algorithm produces stable factors that are sparser than the factors produced by MA57.

Chapter 2
An Efficient Heuristic Method for Large-Scale Block Diagonal Quadratic Programs

2.1 Block diagonal quadratic with nonnegative linearly constraints problem

This chapter considers how to compute an approximate solution for large-scale block diagonal quadratic programs with nonnegative inequality linear constraints and lower bounds, which has the following form:

$$\begin{aligned} \min_{x\in\mathbb{R}^n} f(x) &= \tfrac{1}{2}x^T Hx + c^T x \\ \text{subject to } Ax &\geq b \\ l &\leq x \end{aligned} \tag{2.1}$$

where $x \in \mathbb{R}^n$, $H \in \mathbb{R}^{n\times n}$, $c \in \mathbb{R}^n$, $A \in \mathbb{R}_+^{m\times n}, b \in \mathbb{R}^m$ and $l \in \mathbb{R}^n$. Here, $\mathbb{R}_+^{m\times n}$ denotes the set of $m \times n$ matrices whose entries are nonnegative real numbers. The Hessian matrix *H* is in the following 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*). Note that $\sum_{i=1}^N n_i = n$. Recall H is positive semidefinite if and only if all H_i's are positive semidefinite.

2.2 Separable structure

With the block diagonal quadratic structure, we also write *x*, *g*, *A*, and *l* as

$$\begin{aligned} x &= [x_1^T \quad x_2^T \quad \cdots \quad x_N^T]^T, \\ c &= [c_1^T \quad c_2^T \quad \cdots \quad c_N^T]^T, \\ A &= [A_1^T \quad A_2^T \quad \cdots \quad A_N^T]^T, \\ l &= [l_1^T \quad l_2^T \quad \cdots \quad l_N^T]^T, \end{aligned}$$

where $x_i \in \mathbb{R}^{n_i}$, $c_i \in \mathbb{R}^{n_i}$, $A_i \in \mathbb{R}^{m\times n_i}$, and $l_i \in \mathbb{R}^{n_i}$. The problem can be separated to the summation of *N* quadratic subproblems as follows:

$$\begin{aligned} \min_{x\in\mathbb{R}^n} f(x) &= \textstyle\sum_{i=1}^N f_i(x_i) \\ \text{subject to } \textstyle\sum_{i=1}^N A_i x_i &\geq b \\ l &\leq x \end{aligned} \tag{2.2}$$

where

$$f_i(x_i) = \frac{1}{2} x_i^T H_i x_i + c_i^T x_i \tag{2.3}$$

2.3 The heuristic algorithm

Our algorithm finds the optimal solution to each subproblem $f_i(x_i)$, which we apply the new constraint as in (2.4)

$$\begin{aligned} \min_{x_i \in \mathbb{R}^{n_i}} f_i(x_i) &= \frac{1}{2} x_i^T H_i x_i + c_i^T x_i \\ \text{subject to } A_i x_i &\geq b - \sum_{\substack{j:(1 \leq j \leq N) \\ \text{and}(i \neq j)}} A_j z_j \\ l_i &\leq x_i \end{aligned} \tag{2.4}$$

where $z_j \in \mathbb{R}^{n_j}$. If the *j*th subproblem was already computed and its optimal solution is x_j^*, it becomes the variable z_j for the remaining subproblems (i.e., $z_j = x_j^*$). Otherwise, we set z_j to be the lower bound of x_j. In other words,

$$z_j = f(x) = \begin{cases} x_j^*; & \text{if there exists } \mathrm{x}_\mathrm{j}^*, \\ l_j\,; & \text{otherwise.} \end{cases} \tag{2.5}$$

After we optimize all of the subproblems, $x^* = \begin{bmatrix} x_1^{*T} & x_2^{*T} & x_3^{*T} & \dots & x_N^{*\,T} \end{bmatrix}^T$ is an approximate solution.

Our algorithm can be described as follows.

Algorithm 2.1

```
while all subproblems are not yet solved do
    Choose an unsolved subproblem ith
    for j = 1, 2, …, N do
        if $j \neq i$ then
            if the jth subproblem is unsolved then
                set $z_j = l_j$
            else
                set $z_j = x_j^*$
            end if
        end if
    end for
    Solve (2.4) for $x_i^*$
end while
Construct the solution $x^*$ from $x_i^*$
```

Note that the subproblems can be solved in any order. Also, any algorithm can be used to solve the subproblems.

2.4 Computational results

In our experiment, we compare our algorithm with interior-point method using MATLAB R2009b. The experiments were performed in 100, 400, and 900 variables with different numbers of equally-sized diagonal block and different number of constraint. The test problems are randomly generated in the following way: Let $\widehat{H_i} \in \mathbb{R}^{n_i \times n_i}$. Each element of $\widehat{H_i}$, c, b, and l, and each nonzero element of A is randomly generated between zero and one according to the uniform distribution. Then, let $H_i = \widehat{H}_i \widehat{H}_i^T$. For each problem, we experiment with ten different instances. In this experiment, we use an interior-point method to solve each subproblem. We show average computation time for interior-point method (t_1) and our heuristic method (t_2). We also show relative errors ($rel.$) between the two methods. The relative errors are calculated by

$$rel. = \left|\frac{val_1 - val_2}{val_1}\right| , \tag{2.6}$$

where val_1 is the optimal value found by the interior-point method and val_2 is the approximate value of our heuristic method. The results are shown in Tables 2.1-2.3.

The results show that our heuristic method is faster than the interior-point method especially in the large-scale problems. It is also more efficient for problems with smaller N and many constraints. Finally, when comparing relative errors, our heuristic method is very accurate when the number of variables for each subproblem is larger than the number of diagonal blocks.

Table 2.1 Average time of interior-point method and heuristic method and relative error for 100 variables problem with different number of constraints

N	n_i	m	t_1(s)	t_2(s)	$rel.$
2	50	20	0.26	0.10	0.00000015
2	50	50	0.41	0.17	0.00000020
2	50	80	1.32	0.28	0.00000017
10	10	20	0.17	0.08	0.00000083
10	10	50	0.48	0.19	0.00000056
10	10	80	1.12	0.37	0.00000093
50	2	20	0.23	0.26	0.18553561
50	2	50	0.78	0.65	0.28241298
50	2	80	1.71	1.26	0.13400407

Table 2.2 Average time of interior-point method and heuristic method and relative error for 400 variables problem with different number of constraints

N	n_i	m	t_1(s)	t_2(s)	$rel.$
2	200	80	9.67	4.17	0.00000003
2	200	200	23.02	10.86	0.00000004
2	200	320	220.72	22.28	0.00000004
20	20	80	4.89	1.34	0.00000028
20	20	200	63.62	9.67	0.00000027
20	20	320	169.59	43.69	0.00000038
200	2	80	8.53	8.49	0.30711597
200	2	200	91.76	48.14	0.85615394
200	2	320	222.32	212.76	0.83945318

Table 2.3 Average time of interior-point method and heuristic method and relative error for 900 variables problem with different number of constraints

N	n_i	m	t_1(s)	t_2(s)	$rel.$
2	450	180	143.25	58.14	0.00000002
2	450	450	283.71	155.44	0.00000002
2	450	720	4392.90	327.42	0.00000002
30	30	180	283.56	15.68	0.00000027
30	30	450	1496.65	217.44	0.00000021
30	30	720	3427.95	761.30	0.00000022
450	2	180	446.50	107.73	0.87042419
450	2	450	2135.86	1118.29	1.00916398
450	2	720	4701.08	3720.97	1.03955722

Chapter 3
An Efficient Method to Compute Search Directions of an Infeasible Primal-Dual Path-Following Interior-Point Method for Large-Scale Block Diagonal Quadratic Programming

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

Consider a block diagonal quadratic program with linear inequality constraints

$$\min_{x\in\mathbb{R}^n} \frac{1}{2}x^T Hx + c^T x, \quad \text{subject to } Ax \geq b \tag{3.1}$$

where $x \in \mathbb{R}^n$, $H \in \mathbb{R}^{n\times n}, c \in \mathbb{R}^n, A \in \mathbb{R}^{m\times n}$, and $b \in \mathbb{R}^m$, $m < n$. 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

$$f(x,y,\lambda;\sigma,\mu) = \begin{bmatrix} Hx - A^T\lambda + c \\ Ax - y - b \\ Y\Lambda e - \sigma\mu e \end{bmatrix} = 0, \tag{3.2}$$

Where $\mu = \frac{y^T\lambda}{m}$, $Y = \mathrm{Diag}(y_1, y_2, \dots, y_m)$, $\Lambda = \mathrm{Diag}(\lambda_1, \lambda_2, \dots, \lambda_m)$, $e = [1,1,\dots,1]^T$ and $\sigma \in [0,1)$. Note that the variables y and λ are dual variables of (3.1).

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

$$\begin{bmatrix} H & 0 & -A^T \\ A & -I & 0 \\ 0 & \Lambda & Y \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} -w \\ -z \\ v \end{bmatrix} \tag{3.3}$$

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

$$(x^{k+1}, y^{k+1}, \lambda^{k+1}) = (x^k, y^k, \lambda^k) + \alpha(\Delta x, \Delta y, \Delta\lambda), \tag{3.4}$$

where $\alpha \in (0,1)$ is the step length, and repeating until μ is close to 0 [44]. 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 (3.3) is

$$(H + A^T Y^{-1}\Lambda A)\Delta x = -w + A^T Y^{-1}\Lambda[-z - y + \sigma\mu\Lambda^{-1}e], \tag{3.5}$$

which can be solved by means of a modified Cholesky algorithm. Solving (3.5) 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 (3.5).

3.2 Derivation of our method

To take advantage of block diagonal Hessian, write x, c, A, and w as

$$\begin{aligned} x &= [x_1^T \quad x_2^T \quad \dots \quad x_N^T]^T, \\ c &= [c_1^T \quad c_2^T \quad \dots \quad c_N^T]^T, \\ A &= [A_1^T \quad A_2^T \quad \dots \quad A_N^T], \\ w &= [w_1^T \quad w_2^T \quad \dots \quad w_N^T]^T, \end{aligned}$$

where $x_i \in \mathbb{R}^{n_i}, c_i \in \mathbb{R}^{n_i}, A_i \in \mathbb{R}^{m\times n_i}$, and $w_i \in \mathbb{R}^{n_i}$. Rewrite (3.3) as

$$H_i\Delta x_i - A_i^T\Delta\lambda = -w_i (i = 1, \dots, n), \tag{3.6}$$
$$\textstyle\sum_{i=1}^m A_i\Delta x_i - \Delta y = -z, \tag{3.7}$$
$$\Lambda\Delta y + Y\Delta\lambda = v, \tag{3.8}$$

where

$$w_i = H_i x_i - A_i^T\lambda + c_i (i = 1, \dots, n). \tag{3.9}$$

Next, we rewrite (3.6) and (3.8) as

$$\Delta x_i = H_i^{-1}(A_i^T\Delta\lambda - w_i)(i = 1, \dots, n), \tag{3.10}$$
$$\Delta y = \Lambda^{-1}(v - Y\Delta\lambda). \tag{3.11}$$

Finally, substituting (3.10) and (3.11) into (3.7) yields

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

or, equivalently,

$$(\textstyle\sum_{i=1}^m A_i H_i^{-1} A_i^T + \Lambda^{-1}Y)\Delta\lambda = -z + \Lambda^{-1}v + \sum_{i=1}^m A_i H_i^{-1} w_i. \tag{3.12}$$

Therefore, the search direction can be computed by solving (3.12) for $\Delta\lambda$ and obtain Δx_i and Δy from (3.10) and (3.11), respectively. Algorithm 3.1 below describes the path-following interior-point method that uses the proposed method to compute the search direction.

Remarks for the above algorithm

- We do not explicitly compute H_i^{-1}'s. Instead, we precompute the Cholesky factors of H_i^{-1}'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 (3.9).

Our algorithm requires $\mathrm{O}(m^2N + \sum_{i=1}^{N}(n_i^3 + mn_i^2))$ operations for the preprocessing and $\mathrm{O}(m^3 + \sum_{i=1}^{N}(n_i^2 + mn_i))$ operations per iterate. As comparison, note that the conventional interior-point method that solves (3.5) for search directions requires $\mathrm{O}(n^3)$ per iterate.

Algorithm 3.1

```
Set $S = 0$
Let $(x_0, y_0, \lambda_0)$ be a point with $y_0, \lambda_0 \geq 0$
$\mu = \frac{y_0^T\lambda_0}{m}$
for $i$= 1,2,3, … ,$N$ do
    $S = S + A_iH_i^{-1}A_i^T$
end for
for $k$ = 0, 1, 2, . . . do
    Set $x, y, \lambda = x_k, y_k, \lambda_k$
    Compute $z = Ax - y - b$
    Compute $v = -\Lambda Ye + \sigma\mu e$
    Set $t$= 0
    for $i$= 1, 2, 3, …, $N$ do
        Compute $w_i$ from (3.9)
        $t = t + A_iH_i^{-1}w_i$
    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, …, $N$ do
        Solve $H_i\Delta x_i = -w_i + A_i^T\Delta\lambda$ for $\Delta x_i$
    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 $x_{k+1}, y_{k+1}, \lambda_{k+1} = (x_k, y_k\lambda_k) + \alpha(\Delta x, \Delta y, \Delta\lambda)$
    $\mu = \frac{y_{k+1}^T\lambda_{k+1}}{m}$
end for
```

3.3 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 (3.5) for Δx and then substituting it to compute Δy and $\Delta\lambda$, (ii) solving (3.3) using the sparse linear solver in MATLAB R2011a, and (iii) our method as described in Section 3.2. The experiment was performed on different problem sizes varying from 100 to 2500 variables. The test problems are randomly generated in the following way: Let $\widehat{H} \in \mathbb{R}^{n\times n}$. Each element of $\widehat{H}$, c, A, and b is randomly generated between zero and one according to the uniform distribution. Then, let $H = \widehat{H}\widehat{H}^T$. Now that H is dense, we zero out all of its entries outside the block diagonal to make it a block diagonal matrix.

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. For each case, we test with ten different instances. The results are shown in Tables 3.1-3.7. 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.

The results of experiment show that, for problems with the same number of variables, average time per iterate of method (i) does not depend on the number of diagonal blocks. 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.

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. For large problems, such as those with 500 variables or more, method (iii) is the fastest among the three.

Note that we do not have results of the method (ii) for problems with 4000 variables and 2,000 or more constraints. This is because the matrix in (3.3) that is used by the method (ii) is too large and too dense to store in the main memory of our system. However, the method (i) and method (iii) do not have this problem and can compute the search directions normally.

Table 3.1 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 3.2 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	174.4	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	134.2	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 3.3 Average number of iterates and average time per iterate of interior-point, interior-point with sparse matrix and our method for 1000 variable problems with different number of constraints

N	n_i	*m*=200				*m*=500				*m*=800			
		iter.	t_1 (s)	t_2 (s)	t_3 (s)	*iter.*	t_1 (s)	t_2 (s)	t_3 (s)	*iter.*	t_1 (s)	t_2 (s)	t_3 (s)
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 3.4 Average number of iterates and average time per iterate of interior-point, interior-point with sparse matrix and our method for 1500 variable problems with different number of constraints

N	n_i	m=300				m=750				m=1200			
		iter.	t_1 (s)	t_2 (s)	t_3 (s)	*iter.*	t_1 (s)	t_2 (s)	t_3 (s)	*iter.*	t_1 (s)	t_2 (s)	t_3 (s)
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 3.5 Average number of iterates and average time per iterate of interior-point, interior-point with sparse matrix and our method for 2000 variable problems with different number of constraints

N	n_i	m=400				m=1000				m=1600			
		iter.	t_1 (s)	t_2 (s)	t_3 (s)	*iter.*	t_1 (s)	t_2 (s)	t_3 (s)	*iter.*	t_1 (s)	t_2 (s)	t_3 (s)
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 3.6 Average number of iterates and average time per iterate of interior-point, interior-point with sparse matrix and our method for 2500 variable problems with different number of constraints

N	n_i	*m*=500				*m*=1250				*m*=2000			
		iter.	t_1 (s)	t_2 (s)	t_3 (s)	*iter.*	t_1 (s)	t_2 (s)	t_3 (s)	*iter.*	t_1 (s)	t_2 (s)	t_3 (s)
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
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 3.7 Average number of iterates and average time per iterate of interior-point, interior-point with sparse matrix and our method for 4000 variable problems with different number of constraints

N	n_i	*m*=800				*m*=2000				*m*=3200			
		iter.	t_1 (s)	t_2 (s)	t_3 (s)	*iter.*	t_1 (s)	t_2 (s)	t_3 (s)	*iter.*	t_1 (s)	t_2 (s)	t_3 (s)
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

Chapter 4
A New Pivot Selection Algorithm for Symmetric Indefinite Factorization Arising in Quadratic Programming with Block Constraint Matrices

4.1 Quadratic programs with block diagonal constraint matrices

We considers equality constraint quadratic programs whose constraint matrices are block diagonal. The problem is as follows:

$$\min_{x\in\mathbb{R}^n} \frac{1}{2} x^T H x + c^T x \qquad (4.1)$$
$$\text{subject to } Ax = e$$

where $x \in \mathbb{R}^n$, $H \in \mathbb{R}^{n\times n}$, $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$, $e \in \mathbb{R}^m$, $m < n$, and the constraint matrix A is of the form

$$A = \begin{bmatrix} A_1 & 0 & \cdots & 0 \\ 0 & A_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A_N \end{bmatrix},$$

where $A_i \in \mathbb{R}^{m_i\times n_i}$ (i = 1, 2,..., N), $m_i < n_i$, and N is the number of diagonal blocks in A. Note that $\sum_{i=1}^{N} m_i = m$ and $\sum_{i=1}^{N} n_i = n$. Assume that A has full row rank. Recall from the first-order necessary conditions that, for x^* to be a solution of (4.1), there must be x^* and λ^* satisfying

$$\begin{bmatrix} H & -A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} x^* \\ \lambda^* \end{bmatrix} = \begin{bmatrix} -c \\ e \end{bmatrix} \qquad (4.2)$$

[19]. The above system of equations can be rewritten to a more useful form of Karush-Kuhn-Tucker (KKT) system

$$\begin{bmatrix} H & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} -p \\ \lambda^* \end{bmatrix} = \begin{bmatrix} g \\ h \end{bmatrix}, \qquad (4.3)$$

where $= x^* - x,\ g = c + Hx$, and $h = Ax - e$. The matrix in (4.3) is known as the KKT matrix.

4.2 Symmetric indefinite factorization

To solve the KKT system in (4.3), note that since the KKT matrix is symmetric indefinite, we cannot use Cholesky factorization to factorize it. Instead, we can perform symmetric indefinite factorization [45]. Let K be the KKT matrix, a symmetric indefinite factorization of K is in the following form

$$P^T K P = L B L^T \,, \tag{4.4}$$

where L is a unit lower triangular matrix, B is a block diagonal matrix with block dimension equal to 1 or 2, and P is a permutation matrix. The permutation matrix P is chosen to maintain numerical stability of the computation. In case K is large and sparse, P is chosen to also maintain the sparsity in L in addition to maintaining the stability. After factorization, back and forward substitutions are used to compute the solution of (4.3) by the following steps:

(i) Solve $z : Lz = P^T \begin{bmatrix} g \\ h \end{bmatrix}$.

(ii) Solve $\hat{z} : B\hat{z} = z$.

(iii) Solve $\bar{z} : L^T \bar{z} = \hat{z}$.

(iv) Set : $\begin{bmatrix} -p \\ \lambda^* \end{bmatrix} = P\bar{z}$.

Recall that multiplication with a permutation matrix (P and P^T) is done by arranging the elements in the vector. Matrix B is 1 or 2 dimensional block diagonal, therefore computing $\hat{z}$ is inexpensive. Cost of triangular substitutions with L and L^T depends on the sparsity of L. Normally, the significant cost of solving the system comes from the cost of performing factorization and triangular substitution, the latter of which depends on the sparsity of L. Observe that B is a block diagonal matrix

$$B = \begin{bmatrix} B^{(1)} & 0 & \cdots & 0 \\ 0 & B^{(2)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & B^{(T)} \end{bmatrix},$$

where blocks $B^{(t)}$ are either 1-by-1 or 2-by-2 matrix and nonsingular. To perform symmetric indefinite factorization, let $K^{(t)}$ be the matrix that remains to be factorized in the *t*th iteration. The algorithm starts with $K^{(1)} = K$. For each iteration, we first identify a submatrix $B^{(t)}$ from elements of $K^{(t)}$ that are suitable to be used as a pivot block (There are many methods for selecting a suitable pivot $B^{(t)}$. Our method is described in Section 4.3). The submatrix $B^{(t)}$ is either a single diagonal element of $K^{(t)} \left(\begin{bmatrix} k_{ll}^{(t)} \end{bmatrix} \right)$ or a 2-by-2 block with two diagonal elements of $K^{(t)} \left(\begin{bmatrix} k_{ll}^{(t)} & k_{lr}^{(t)} \\ k_{rl}^{(t)} & k_{rr}^{(t)} \end{bmatrix} \right)$.

Next, we find the permutation matrix $P^{(t)}$ satisfying

$$\left(P^{(t)}\right)^T K^{(t)} P^{(t)} = \begin{bmatrix} B^{(t)} & \left(C^{(t)}\right)^T \\ C^{(t)} & Z^{(t)} \end{bmatrix}. \tag{4.5}$$

The right-hand side of (4.5) can be factorized as

$$\left(P^{(t)}\right)^T K^{(t)} P^{(t)} = \begin{bmatrix} I & 0 \\ C^{(t)}\left(B^{(t)}\right)^{-1} & I \end{bmatrix} \cdot \begin{bmatrix} B^{(t)} & 0 \\ 0 & Z^{(t)} - C^{(t)}\left(B^{(t)}\right)^{-1}\left(C^{(t)}\right)^T \end{bmatrix} \cdot \begin{bmatrix} I & \left(B^{(t)}\right)^{-1}\left(C^{(t)}\right)^T \\ 0 & I \end{bmatrix}. \quad (4.6)$$

Let $L^{(t)} = C^{(t)}\left(B^{(t)}\right)^{-1}$and $K^{(t+1)} = Z^{(t)} - C^{(t)}\left(B^{(t)}\right)^{-1}\left(C^{(t)}\right)^T$. The above can be rewritten as

$$\left(P^{(t)}\right)^T K^{(t)} P^{(t)} = \begin{bmatrix} I & 0 \\ L^{(t)} & I \end{bmatrix} \cdot \begin{bmatrix} B^{(t)} & 0 \\ 0 & K^{(t+1)} \end{bmatrix} \cdot \begin{bmatrix} I & \left(L^{(t)}\right)^T \\ 0 & I \end{bmatrix} \quad (4.7)$$

The same process can be repeated recursively on the matrix $K^{(t+1)}$. Note that the dimension of $K^{(t+1)}$ is less than the dimension of $K^{(t)}$ by either one or two depending on the dimension of $B^{(t)}$. Choosing pivot at each step should be inexpensive, lead to at most modest growth in the elements of the remaining matrix, and L should not be too much denser than the original matrix. There are various methods to identify pivot block $B^{(t)}$ for dense matrices. Bunch and Parlett searches the whole submatrix at each stage for the largest-magnitude diagonal $k_{qq}^{(t)}$ and the largest-magnitude off-diagonal $k_{rl}^{(t)}$. It identifies $k_{qq}^{(t)}$ as the 1-by-1 pivot block if the resulting growth rate is acceptable. Otherwise, it selects $\begin{bmatrix} k_{ll}^{(t)} & k_{lr}^{(t)} \\ k_{rl}^{(t)} & k_{rr}^{(t)} \end{bmatrix}$ as the 2-by-2 pivot block. This method requires $O(n^3)$ comparisons and yields a matrix L whose maximum element is bounded by 2.781. Bunch-Kaufman pivoting strategy searches for the largest-magnitude off-diagonal elements of at most two columns for each iteration. It requires $O(n^2)$ comparisons but the elements in L are unbounded. BBK combines the two above strategies and is widely used to select pivot blocks. By monitoring the size of the elements in L, BBK uses the Bunch-Kaufman strategy when it yields modest element growth. Otherwise, it repeatedly searches for an acceptable pivot [33]. BBK algorithm is shown in Algorithm 4.1 below. In average cases, the total cost of BBK is the same as Bunch-Kaufman, but in the worst case it can be the same as the cost of the Bunch-Parlett strategy.

Algorithm 4.1 The BBK algorithm

```
Set $\alpha = (1+\sqrt{17})/8$
Set $\gamma_1$ = maximum magnitude of any subdiagonal entry in column 1
**if** $|k_{11}| \geq \alpha\gamma_1$ **then**
    Use $k_{11}$ as a 1×1 pivot
**else**
    Set $l = 1; \gamma_l = \gamma_1$ ;
    **repeat**
        Set r = row index of first (subdiagonal) entry of maximum magnitude in
        column l
        Set $\gamma_r$ = maximum magnitude of any off-diagonal entry in column r
        **If** $|k_{rr}| \geq \alpha\gamma_r$ **then**
            Use $k_{rr}$ as a 1×1 pivot
        **else if** $\gamma_l = \gamma_r$ **then**
            Use $\begin{bmatrix} k_{ll} & k_{lr} \\ k_{rl} & k_{rr} \end{bmatrix}$ as 2×2 pivot
        **else**
            Set $l = r; \gamma_l = \gamma_r$
        **end if**
    **until** A pivot is chosen
**end if**
```

4.3 A new pivot selection for block constraint quadratic programming

This section describes our proposed pivot selection method for the quadratic programs with block diagonal constraint matrices. The goals of our method are to maintain sparsity and stability in the factors. First, we identify candidate pivots that can maintain sparsity of L. Second, we select among these candidates to maintain stability of the factors. The last subsection describes the overall algorithm for factoring the KKT matrix.

4.3.1 Candidate pivots identification

Consider the structure of the KKT matrix of our quadratic program with a block diagonal constraint matrix. Elements of the KKT matrix K can be classified into three types: the elements of the Hessian matrix h_{ij}, the nonzero elements of the constraint matrix a_{ij}, and the zero submatrices. For better readability, we use a 6-by-6 Hessian matrix with two blocks of constraints as an example in our explanation. The structure of a sample KKT matrix is as follows:

$$K = \begin{bmatrix} h_{11} & h_{12} & h_{13} & h_{14} & h_{15} & h_{16} & a_{11} & a_{21} & 0 & 0 \\ h_{21} & h_{22} & h_{23} & h_{24} & h_{25} & h_{26} & a_{12} & a_{22} & 0 & 0 \\ h_{31} & h_{32} & h_{33} & h_{34} & h_{35} & h_{36} & a_{13} & a_{23} & 0 & 0 \\ h_{41} & h_{42} & h_{43} & h_{44} & h_{45} & h_{46} & 0 & 0 & a_{34} & a_{44} \\ h_{51} & h_{52} & h_{53} & h_{54} & h_{55} & h_{56} & 0 & 0 & a_{35} & a_{45} \\ h_{61} & h_{62} & h_{63} & h_{64} & h_{65} & h_{66} & 0 & 0 & a_{36} & a_{46} \\ a_{11} & a_{12} & a_{13} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_{34} & a_{35} & a_{36} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_{44} & a_{45} & a_{46} & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{4.8}$$

Note that $h_{ij} = h_{ji}$ due to H being symmetric. For our KKT matrix, there are three possible cases for pivot $B^{(t)}$. The first case is a 1-by-1 matrix selected from one of the nonzero diagonal elements in matrix $K^{(t)}$ (i.e., $h_{ll}^{(t)}$). The second case is a 2-by-2 matrix where both diagonal elements are nonzero (i.e., $\begin{bmatrix} h_{ll}^{(t)} & h_{lr}^{(t)} \\ h_{rl}^{(t)} & h_{rr}^{(t)} \end{bmatrix}$). In this case, the off-diagonal elements are the elements of the Hessian matrix h_{ij}, where $i \neq j$. The last possible case is a 2-by-2 matrix where one diagonal element is zero and the other three elements are nonzero (i.e., $\begin{bmatrix} h_{ll}^{(t)} & a_{rl}^{(t)} \\ a_{rl}^{(t)} & 0 \end{bmatrix}$). In other words, the off-diagonal elements are the nonzero elements of the constraint matrix a_{ij}. Selecting a pivot in any other ways besides the three mentioned above is not possible as they all lead to singular $B^{(t)}$. Each form of pivot $B^{(t)}$ directly affects the sparsity of the factor $L^{(t)}$ and also the sparsity and the stability of the remaining matrix $K^{(t+1)}$. Note that the sparsity of $K^{(t+1)}$ affects the sparsity of $L^{(t+1)}$, too (Recall that $L^{(t)} = C^{(t)}\left(B^{(t)}\right)^{-1}$and $K^{(t+1)} = Z^{(t)} - C^{(t)}\left(B^{(t)}\right)^{-1}\left(C^{(t)}\right)^{T}$). Now we consider the three cases of pivot in more details, for the first case, where $B^{(t)}$ is a 1-by-1 matrix, the number of zeros in $L^{(t)}$ is equal to the number of zeros in $C^{(t)}$but many zeros in $K^{(t)}$ become nonzeros (fill-ins) in the remaining matrix $K^{(t+1)}$. Selecting pivot of this form generally cannot maintain the sparsity of the factors. For example, let $B^{(1)} = [h_{55}]$ be the pivot for the matrix in (4.8). After permutation, we have

$$\left(P^{(1)}\right)^{T} K^{(1)} P^{(1)} = \begin{bmatrix} h_{55} & h_{52} & h_{53} & h_{54} & h_{51} & h_{56} & 0 & 0 & a_{35} & a_{45} \\ h_{25} & h_{22} & h_{23} & h_{24} & h_{21} & h_{26} & a_{12} & a_{22} & 0 & 0 \\ h_{35} & h_{32} & h_{33} & h_{34} & h_{31} & h_{36} & a_{13} & a_{23} & 0 & 0 \\ h_{45} & h_{42} & h_{43} & h_{44} & h_{41} & h_{46} & 0 & 0 & a_{34} & a_{44} \\ h_{15} & h_{12} & h_{13} & h_{14} & h_{11} & h_{16} & a_{11} & a_{21} & 0 & 0 \\ h_{65} & h_{62} & h_{63} & h_{64} & h_{61} & h_{66} & 0 & 0 & a_{36} & a_{46} \\ 0 & a_{12} & a_{13} & 0 & a_{11} & 0 & 0 & 0 & 0 & 0 \\ 0 & a_{22} & a_{23} & 0 & a_{21} & 0 & 0 & 0 & 0 & 0 \\ a_{35} & 0 & 0 & a_{34} & 0 & a_{36} & 0 & 0 & 0 & 0 \\ a_{45} & 0 & 0 & a_{44} & 0 & a_{46} & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$C^{(1)} = \begin{bmatrix} h_{25} \\ h_{35} \\ h_{45} \\ h_{15} \\ h_{65} \\ 0 \\ 0 \\ a_{35} \\ a_{45} \end{bmatrix}, L^{(1)} = \begin{bmatrix} \times \\ \times \\ \times \\ \times \\ \times \\ 0 \\ 0 \\ \times \\ \times \end{bmatrix}, \text{ and } K^{(2)} = \begin{bmatrix} \times & \times & \times & \times & \times & \times & \times & \bullet & \bullet \\ \times & \times & \times & \times & \times & \times & \times & \bullet & \bullet \\ \times & \times & \times & \times & \times & 0 & 0 & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \bullet & \bullet \\ \times & \times & \times & \times & \times & 0 & 0 & \times & \times \\ \times & \times & 0 & \times & 0 & 0 & 0 & \bullet & \bullet \\ \times & \times & 0 & \times & 0 & 0 & 0 & \bullet & \bullet \\ \bullet & \bullet & \times & \bullet & \times & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \times & \bullet & \times & \bullet & \bullet & \bullet & \bullet \end{bmatrix}.$$

Note that $\times$ denotes the nonzero elements in $L^{(1)}$ and $K^{(2)}$in which the elements of matrix $C^{(1)}$and $Z^{(1)}$(in the same positions) are also nonzero and $\bullet$ denotes the fill-in elements compared to $C^{(1)}$and $Z^{(1)}$, respectively. For the second case where the pivot is a 2-by-2 matrix with no zero elements, the number of nonzeros in $L^{(t)}$ may be equal to or greater than that of $C^{(t)}$. It also results in a large number of fill-ins in the remaining matrix $K^{(t+1)}$. For example, suppose $B^{(1)} = \begin{bmatrix} h_{55} & h_{56} \\ h_{65} & h_{66} \end{bmatrix}$. We have

$$\left(P^{(1)}\right)^T K^{(1)} P^{(1)} = \begin{bmatrix} h_{55} & h_{56} & h_{53} & h_{54} & h_{51} & h_{52} & 0 & 0 & a_{35} & a_{45} \\ h_{65} & h_{66} & h_{63} & h_{64} & h_{61} & h_{62} & 0 & 0 & a_{36} & a_{46} \\ h_{35} & h_{36} & h_{33} & h_{34} & h_{31} & h_{32} & a_{13} & a_{23} & 0 & 0 \\ h_{45} & h_{46} & h_{43} & h_{44} & h_{41} & h_{42} & 0 & 0 & a_{34} & a_{44} \\ h_{15} & h_{16} & h_{13} & h_{14} & h_{11} & h_{12} & a_{11} & a_{21} & 0 & 0 \\ h_{25} & h_{26} & h_{23} & h_{24} & h_{21} & h_{22} & a_{12} & a_{22} & 0 & 0 \\ 0 & 0 & a_{13} & 0 & a_{11} & a_{12} & 0 & 0 & 0 & 0 \\ 0 & 0 & a_{23} & 0 & a_{21} & a_{22} & 0 & 0 & 0 & 0 \\ a_{35} & a_{36} & 0 & a_{34} & 0 & 0 & 0 & 0 & 0 & 0 \\ a_{45} & a_{46} & 0 & a_{44} & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$C^{(1)} = \begin{bmatrix} h_{35} & h_{36} \\ h_{45} & h_{46} \\ h_{15} & h_{16} \\ h_{25} & h_{26} \\ 0 & 0 \\ 0 & 0 \\ a_{35} & a_{36} \\ a_{45} & a_{46} \end{bmatrix}, L^{(1)} = \begin{bmatrix} \times & \times \\ \times & \times \\ \times & \times \\ \times & \times \\ 0 & 0 \\ 0 & 0 \\ \times & \times \\ \times & \times \end{bmatrix}, \text{ and } K^{(2)} = \begin{bmatrix} \times & \times & \times & \times & \times & \times & \bullet & \bullet \\ \times & \times & \times & \times & 0 & 0 & \times & \times \\ \times & \times & \times & \times & \times & \times & \bullet & \bullet \\ \times & \times & \times & \times & \times & \times & \bullet & \bullet \\ \times & 0 & \times & \times & 0 & 0 & 0 & 0 \\ \times & 0 & \times & \times & 0 & 0 & 0 & 0 \\ \bullet & \times & \bullet & \bullet & 0 & 0 & \bullet & \bullet \\ \bullet & \times & \bullet & \bullet & 0 & 0 & \bullet & \bullet \end{bmatrix}.$$

Note that $K^{(\mathrm{t}+1)}$ can be denser than in the above example for some other pivots such as $B^{(1)} = \begin{bmatrix} h_{33} & h_{36} \\ h_{63} & h_{66} \end{bmatrix}$. Lastly, consider the case where the pivot is a 2-by-2 matrix with one diagonal element being zero. In this case, the number of zeros in $L^{(t)}$ is equal to the number of zeros in $C^{(\mathrm{t})}$ and there is no fill-in in the remaining matrix $K^{(t+1)}$. For example, suppose $B^{(1)} = \begin{bmatrix} h_{55} & a_{35} \\ a_{35} & 0 \end{bmatrix}$. We have

$$\left(P^{(1)}\right)^T K^{(1)} P^{(1)} = \begin{bmatrix} h_{55} & a_{35} & h_{53} & h_{54} & h_{51} & h_{65} & 0 & 0 & h_{52} & a_{45} \\ a_{35} & 0 & 0 & a_{34} & 0 & a_{36} & 0 & 0 & 0 & 0 \\ h_{53} & 0 & h_{33} & h_{43} & h_{31} & h_{63} & a_{13} & a_{23} & h_{32} & 0 \\ h_{54} & a_{34} & h_{43} & h_{44} & h_{41} & h_{64} & 0 & 0 & h_{42} & a_{44} \\ h_{51} & 0 & h_{31} & h_{41} & h_{11} & h_{61} & a_{11} & a_{21} & h_{21} & 0 \\ h_{65} & a_{36} & h_{63} & h_{64} & h_{61} & h_{66} & 0 & 0 & h_{62} & a_{46} \\ 0 & 0 & a_{13} & 0 & a_{11} & 0 & 0 & 0 & a_{12} & 0 \\ 0 & 0 & a_{23} & 0 & a_{21} & 0 & 0 & 0 & a_{22} & 0 \\ h_{52} & 0 & h_{32} & h_{42} & h_{21} & h_{62} & a_{12} & a_{22} & h_{22} & 0 \\ a_{45} & 0 & 0 & a_{44} & 0 & a_{46} & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$C^{(1)} = \begin{bmatrix} h_{53} & 0 \\ h_{54} & a_{34} \\ h_{51} & 0 \\ h_{65} & a_{36} \\ 0 & 0 \\ 0 & 0 \\ h_{52} & 0 \\ a_{45} & 0 \end{bmatrix}, L^{(1)} = \begin{bmatrix} 0 & \times \\ \times & \times \\ 0 & \times \\ \times & \times \\ 0 & 0 \\ 0 & 0 \\ 0 & \times \\ 0 & \times \end{bmatrix}, \text{ and } K^{(2)} = \begin{bmatrix} \times & \times & \times & \times & \times & \times & \times & 0 \\ \times & \times & \times & \times & 0 & 0 & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & 0 \\ \times & \times & \times & \times & 0 & 0 & \times & \times \\ \times & 0 & \times & 0 & 0 & 0 & \times & 0 \\ \times & 0 & \times & 0 & 0 & 0 & \times & 0 \\ \times & \times & \times & \times & \times & \times & \times & 0 \\ 0 & \times & 0 & \times & 0 & 0 & 0 & 0 \end{bmatrix}.$$

We see that the first and second types of pivots generate fill-in in remaining matrix $K^{(t+1)}$. The third type yields sparser $L^{(t)}$ than the other two and generally produces no fill-ins in the remaining matrix $K^{(t+1)}$. Therefore, our algorithm first identifies all candidate pivots that are of the form $\begin{bmatrix} h_{ii} & a_{ji} \\ a_{ji} & 0 \end{bmatrix}$. By choosing this form of pivots, the factor L is as sparse as possible and there are no fill-ins in the remaining matrix $K^{(t+1)}$.

4.3.2 Pivot selection

There are generally many pivot candidates of the form that we are interested in. We compare the condition numbers of these candidate pivots and then choose the one with the smallest condition number. Recall that the condition number of a 2-by-2 matrix is defined as

$$\operatorname{cond}(B) = \|B\| \cdot \|B^{-1}\|. \tag{4.9}$$

When the candidate pivot B is of the form $\begin{bmatrix} b_{ii} & b_{ij} \\ b_{ij} & 0 \end{bmatrix}$, B^{-1} becomes $\left(-\frac{1}{b_{ij}^2} \cdot \begin{bmatrix} 0 & -b_{ij} \\ -b_{ij} & b_{ii} \end{bmatrix}\right)$. Using infinity norm, we see that

$$\begin{aligned} \|B\|_\infty &= \max\{|b_{ii}| + |b_{ij}|, |b_{ij}|\} \\ &= |b_{ii}| + |b_{ij}| \\ \|B^{-1}\|_\infty &= \left(\frac{1}{b_{ij}^2}\right) \max\{|b_{ij}|, |b_{ij}| + |b_{ii}|\} \\ &= \frac{|b_{ij}| + |b_{ii}|}{b_{ij}^2} \end{aligned}$$

$$\begin{aligned}\operatorname{cond}_\infty(B) &= \frac{(|b_{ii}|+|b_{ij}|)\cdot(|b_{ij}|+|b_{ii}|)}{b_{ij}^2}\\ &= \left(1+\frac{|b_{ii}|}{|b_{ij}|}\right)^2 \qquad (4.10)\end{aligned}$$

Therefore, we need only to compare $|b_{ii}|/|b_{ij}|$ to find the pivot candidate with the minimum condition number. We do so and select the candidate with the smallest condition number as the pivot. Note that, when a_{ij} is zero, the condition numbers of the candidates containing this a_{ij} are infinity. In this case, such candidates are not chosen by our algorithm.

4.3.3 The algorithm

This subsection gives the complete picture of our algorithm. First, we choose the pivot that maintains the sparsity of the factors. We select a 2-by-2 pivot matrix with one of the diagonal elements being zero as described previously. These pivots yield no fill-ins and we can choose this type of pivots for the first m iterations, where m is the number of constraints in the quadratic program. Afterward, $K^{(m+1)}$ is completely dense therefore we switch to use a general (non-sparse) symmetric indefinite factorization at this point.

Our method keeps track of the current and original positions of elements a_{ij} (as the elements may change in the permutation step). These positions are used to efficiently produce the pivot candidates of the third form. We consider only the candidates with the off-diagonal entries from the same block A_i, where i is chosen arbitrarily. (The pivot candidates are identified from the elements from each block by block.) Among them, we select the candidate with the smallest condition number. Note that, according to (4.10), we need to compute only the condition numbers of the candidates with the largest $|b_{ij}|$ for each column j and selecting the one with the smallest condition number. If pivot $\begin{bmatrix} k_{ll}^{(t)} & k_{lr}^{(t)} \\ k_{rl}^{(t)} & 0 \end{bmatrix}$ is selected as the pivot, we remove row r and column l from the lists of available row and column. If row r is the last row in a block, we remove the block containing row r from the available block list, too. Then we continue to the pivot candidates from the next block. Note that our pivot selection method requires $O(\sum_{i=1}^{N} m_i^2 n_i)$ operations. Our method is shown in Algorithm 4.2.

Algorithm 4.2 Symmetric indefinite factorization for QP block constraint KKT matrix

```
Set $K$ = KKT matrix of QP with block constraints
Set $N$ = number of blocks in constraints, $n$ = number of all constraints
Set $m$ = number of variables, $s = n + m$ // size of matrix $K$
Set $L$ = $s$-by-$s$ identity matrix, $B$ = $s$-by-$s$ zero matrix
Set $aB$ = {1, 2, ...,  } // list of available block
Set $aC$ = {1, 2, ..., $n$} // list of available column
Set $aR$ = {$n$ + 1, $n$ + 2, ..., $n + m$} // list of available row
Set $P = [1\ 2 \dots s]$// list of columns (1 to $n$) and rows ($n$ + 1 to $s$) position in matrix
Set $sR = [sR_1, sR_2, \dots, sR_N]$ // $sR_i$is the first row of $A_i$
Set $eR = [eR_1, eR_2, \dots, eR_N]$ // $eR_i$is the last row of $A_i$
Set $sC = [sC_1, sC_2, \dots, sC_N]$ // $sC_i$is the first column of $A_i$
Set $eC = [eC_1, eC_2, \dots, eC_N]$// $eC_i$is the last column of $A_i$
Set $p = 1$
**while** $p < m \times 2$ **do**
    Set $mincond = \infty, removeBl = 0$
    Randomly select $t$ from $aB$
    Set $avaiRowInBl = \{x : x \in aR; sR_t \le x \le eR_t\}$
    Set $avaiColInBl = \{x : x \in aC; sC_t \le x \le eC_t\}$
    Set $posRowInBl = \{x : x = P_i; i \in avaiRowInBl\}$
    Set $posColInBl = \{x : x = P_i; i \in avaiColInBl\}$
    Set $mincond = \min\left\{\left(\left|\frac{K_{jj}}{K_{ij}}\right|\right) : i \in posRowInBl, j \in posColInBl\right\}$
    Set $l$ = column of $mincond$
    Set $r$ = row of $mincond$
    **if** $|avaiRowInBl| = 1$ **then**
        Remove $t$ from $aB$
    **end if**
    Use $\begin{bmatrix} k_{ll} & k_{lr} \\ k_{lr} & k_{rr} \end{bmatrix}$ as the 2-by-2 pivot
    Remove elements $\{x : x = P_r$or $x = P_l\}$ from $aR$ and $aC$
    Swap $P_l$ and $P_r$ to $P_p$ and $P_p + 1$, respectively
    $p = p + 2$
**end while**
Factorize the remaining matrix with a general (non-sparse) symmetric indefinite factorization method
```

4.4 Experiment and results

In this section, we compare the efficiency between the following two methods: (i) MA57 and (ii) our method. The experiment was performed in Matlab 2012a on problems with 500, 1000, and 1500 variables. For each problem size, we test with 10, 50, and 100 blocks in the constraint matrix, where each block is of equal size. The numbers of constraints are 40 and 80 percent of the number of variables. The test problems are randomly generated in the following way: Let $\widehat{H} \in \mathbb{R}^{n \times n}$. Each element of $\widehat{H}$, c, and e, and each nonzero element of A is randomly generated between zero and one according to the uniform distribution. Then, let $H = \widehat{H}\widehat{H}^T / \left(\max_{i,j} \hat{h}_{i,j}\right)$. For each problem, we experiment with 10 different instances. We compare the average numbers of nonzeros in factor L. The results of this experiment, which are shown in Table 4.1, show that our method yields sparser L when computing the symmetric indefinite factorization than the MA57 algorithm. After the factorization, we use the factors from the two methods to compute the solution of the quadratic program following Steps (i) - (iv) in Section 4.2. Table 4.1 also shows the solving time and the accuracy of our algorithm. The results show that using the factors L, B, and P from our method reduces the time needed to solve the KKT system compares to using the factors from MA57. Our method also yields accurate solutions with small residuals.

Normally, the Hessian matrix may not be dense. We therefore also experiment on the problems with sparse Hessian matrices having 30, 50, and 70 percent of their entries being nonzeros. We test with the constraint matrices having 10, 50, and 100 blocks, where each block is of equal size. The numbers of constraints are 40 and 80 percent of the number of variables. For each problem, we experiment with 10 different instances. We compare the average numbers of nonzeros in the factor L. The results of this experiment are shown in Table 4.2. We see that even when the Hessian matrix is sparse, our method still maintains more sparsity in L than MA57 can.

Finally, we compare both methods on problems where each blocks in the constraint matrices are of different sizes. The results are shown in Table 4.3. For these problems, our method also produces sparser factors and requires less solving time than MA57.

Table 4.1 Average numbers of nonzeros in factor L, average solving time, and average residual of MA57 and our algorithm for problems with 500, 1000, and 1500 variables and constraint matrices with equal-sized blocks.

n	$N \times (n_i \times m_i)$	Ave. num. of nonzeros in L		Ave. solving time (ms)		Ave. residual $(\times 10^{-10})$	
		MA57	Our Method	MA57	Our Method	MA57	Our Method
500	10×(50×10)	162210.0	130250.0	3.81	2.39	0.046	0.028
500	10×(50×40)	333090.0	145250.0	6.58	4.37	0.190	0.033
500	20×(25×5)	158829.4	127750.0	3.43	2.33	0.042	0.032
500	20×(25×20)	319564.8	135250.0	6.70	4.31	0.208	0.033
500	50×(10×2)	156691.3	126250.0	3.65	2.32	0.037	0.030
500	50×(10×8)	311022.7	129250.0	6.41	4.35	0.302	0.032
1000	10×(100×20)	648420.0	520500.0	15.15	7.81	0.169	0.113
1000	10×(100×80)	1332180.0	580500.0	32.12	15.89	0.620	0.138
1000	20×(50×10)	634909.3	510500.0	15.64	7.84	0.146	0.113
1000	20×(50×40)	1278127.7	540500.0	31.38	15.78	0.560	0.132
1000	50×(20×4)	626351.3	504500.0	15.16	7.80	0.128	0.109
1000	50×(20×16)	1243965.5	516500.0	30.59	16.00	0.478	0.131
1500	10×(150×30)	1458630.0	1170750.0	37.23	16.65	0.336	0.260
1500	10×(150×120)	2997261.7	1305750.0	83.14	35.26	1.104	0.295
1500	20×(75×15)	1428238.8	1148250.0	36.45	16.54	0.300	0.227
1500	20×(75×60)	2875679.8	1215750.0	80.21	34.67	1.019	0.276
1500	50×(30×6)	1408997.6	1134750.0	35.58	16.43	0.286	0.231
1500	50×(30×24)	2798788.2	1161750.0	75.85	34.52	1.105	0.268

Column n is the number of variables. Column $N \times (n_i \times m_i)$ indicates the dimensions of each diagonal block in the constraint matrix. Columns Ave. num. of nonzeros in L show the number of nonzeros in L from MA57 algorithm and our method, respectively. Columns Ave. solving time shows the solving time of the two methods. Columns Ave. residual represent the residuals of the results of both methods. The residual is $\|Kx - v\|_2$, where K is our KKT matrix, x is the computed solution, and v is the vector $\begin{bmatrix} g \\ h \end{bmatrix}$ in (4.3).

Table 4.2 Average numbers of nonzeros in L of MA57 algorithm and our algorithm for problems with 1000 variables with 30, 50, and 70% of nonzeros in Hessian matrix and constraint matrices with equal-sized blocks.

n	$N \times (n_i \times m_i)$	Ave. num. of nonzeros in L					
		30% nonzeros in H		50% nonzeros in H		70% nonzeros in H	
		MA57	Our Method	MA57	Our Method	MA57	Our Method
1000	10×(100×20)	631400	507800	643900	514800	640900	518000
1000	10×(100×80)	1509800	567900	1524700	575000	1441600	578100
1000	20×(50×10)	612900	486700	622400	499900	632300	505800
1000	20×(50×40)	1295700	516300	1296600	530100	1273900	536000
1000	50×(20×4)	593800	455200	614300	479900	620900	493500
1000	50×(20×16)	1320100	458100	1301300	491300	1279600	505600

Column n is the number of variables. $N(n_i \times m_i)$ indicates the dimensions of each diagonal blocks in the constraint matrix. Columns Ave. num. of nonzeros in L show the number of nonzeros in L from MA57 algorithm and our method, respectively.

Table 4.3 Average numbers of nonzeros in factor L, average solving time, and average residual of MA57 and our algorithm for problems with 500, 1000, and 1500 variables constraint matrices with unequal-sized blocks.

n	$N \times (n_i \times m_i)$	Ave. num. of nonzeros in L		Ave. solving time (ms)		Ave. residual ($\times 10^{-10}$)	
		MA57	Our Method	MA57	Our Method	MA57	Our Method
500	5× (50×10, 75×15,100×20, 125×25,150×30)	160937.0	136500.0	3.55	2.34	0.053	0.033
500	5× (50×40,75×60, 100×80,125×100, 150×120)	328058.0	170250.0	6.59	4.74	0.221	0.035
1000	5×(100×20, 150×30,200×40, 250×50,300×60)	643154.0	545500.0	14.98	7.85	0.186	0.142
1000	5×(100×80, 150×120,200×160, 250×200,300×240)	1311324.0	680500.0	31.48	16.05	0.824	0.155
1500	5×(150×30, 225×45,300×60, 375×75,450×90)	1446635.9	1227000.0	37.45	16.52	0.468	0.420
1500	5×(150×120, 225×180,300×240, 375×300,450×360)	2949798.0	1530750.0	81.45	35.07	1.283	0.610

Column n is the number of variables. Column $N \times (n_i \times m_i)$ indicates the dimensions of each diagonal blocks in the constraint matrix. Columns Ave. num. of nonzeros in L show the number of nonzeros in L from MA57 algorithm and our method, respectively. Columns Ave. solving time shows the solving time of the two methods. Columns Ave. residual represent the residuals of the results of both methods. The residual is $\|Kx - v\|_2$, where K is our KKTmatrix, x is the computed solution, and v is the vector $\begin{bmatrix} g \\ h \end{bmatrix}$ in (4.3).

Chapter 5
New Pivot Selection for Sparse Symmetric Indefinite Factorization

This chapter proposes a pivot selection method for solving linear system

$$Ax = b,$$

where $A \in \mathbb{R}^{n\times n}$ is sparse symmetric indefinite without any known sparsity pattern. Solving a symmetric indefinite linear system is generally done by first obtaining the symmetric indefinite factorization as shown in Section 4.2. The computational time for solving the linear system depends solely on the factorization and back and forward substitutions, which in turn depend on the sparsity of factor L. The pivot selection during the factorization directly affects the sparsity and stability of factors.

5.1 Pivot selection with minimum degree

Finding the optimal ordering that minimizes fill-in is NP-hard [24] therefore a heuristic is often used for pivot selection. Choosing pivot at each step should be inexpensive, lead to at most modest growth in the elements of the remaining matrix, and not cause L to be too much denser than the original matrix. One of the well-known and efficient pivot selection techniques is the minimum degree algorithm [20-22]. The algorithm considers the pivot based on the following graph model. Define an undirected graph $G = (V, E)$, where $V = \{1, \dots, n\}$ and $E = \Big\{\{i, j\}: i \neq j \text{ and } a_{ij} \neq 0\Big\}$. Observe that the degree of v ($\deg(v)$), where $v \in V$, is the number of nonzero off-diagonal elements on the vth row. The vertex v with minimum $\deg(v)$ is chosen as the pivot.

Define the elimination graph $G_v = (V \setminus \{v\}, E')$, where $E' = E \cup \Big\{\{i, j\}: \{i, v\} \in E \text{ and } \{v, j\} \in E\Big\} \setminus \{\{v, i\}: i = 1, 2, \dots, n\}$. Graph G_v is used to choose the next pivot, and so on. That is, the minimum degree algorithm is as follows.

```
Algorithm 5.1 Minimum Degree Algorithm

Define $G$ as described above.
while $G \neq \emptyset$ do
    $v$ = the vertex with minimum deg($v$)
    $G = G_v$
end while
```

Note that the minimum degree algorithm identifies the pivot at each step without any numerical calculation. For this reason, it can be used as the ordering step before factorizing the matrix. Many improvements of the minimum degree algorithm and its implementation have been proposed [23] such as decreasing the computation time for the degree update by considering the indistinguishable nodes [46] or minimum degree independent nodes [28], reducing the computation cost by using an approximate minimum degree [29], and saving space by using the quotient graph model [47].

5.2 Our pivot selection algorithm

Unlike in Cholesky factorization, pivots in symmetric indefinite factorization can be either a scalar or a 2-by-2 matrix therefore the minimum degree algorithm cannot be used as is in this case.

The stability condition that our algorithm uses is proposed by Duff et al. [48] and also used as a thresholding test for 1-by-1 and 2-by-2 pivots in MA57 [39]. We consider a 1-by-1 pivot a_{ii} to be *acceptably stable* if

$$|a_{ii}| \geq \alpha \max_{r \neq i} |a_{ri}|. \tag{5.1}$$

Similarly, a 2-by-2 pivot $\begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix}$ is considered to be *acceptably stable* if

$$\left| \begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix}^{-1} \right| \cdot \begin{bmatrix} \max\limits_{r \neq i, r \neq j} |a_{ri}| \\ \max\limits_{r \neq i, r \neq j} |a_{rj}| \end{bmatrix} \leq \begin{bmatrix} \alpha^{-1} \\ \alpha^{-1} \end{bmatrix}. \tag{5.2}$$

Conditions (5.1) and (5.2) limit the magnitudes of the entries of L to $1/\alpha$ at most. The appropriate value of α is $0 < \alpha \leq 0.5$. The default value of α in MA57 is 0.01 [39].

Let us call the column with the fewest number of off-diagonal nonzeros the *minimum degree column*. Let i be the minimum degree column of the matrix A. We accept a_{ii} as the 1-by-1 pivot ($B^{(k)}$) if a_{ii} satisfies (5.1). Otherwise, we proceed to search for a suitable 2-by-2 pivot $\begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix}$ that satisfies (5.2) as follows. Let

$$Z_i = \{z | a_{iz} \neq 0 \text{ and } z \neq i\}. \tag{5.3}$$

Consider all submatrices $\begin{bmatrix} a_{ii} & a_{iz} \\ a_{zi} & a_{zz} \end{bmatrix}$, where $z \in Z_i$, as the candidates for a 2-by-2 pivot. The degree of each candidate $\deg(i,z)$ is the number of rows l where $l \neq i$, z and at least one of a_{li} and a_{lz} is nonzero. To compute $\deg(i,z)$, define

$$d(i,z,l) = \begin{cases} 0, & \text{if } a_{li} = 0 \text{ and } a_{lz} = 0, \\ 1, & \text{otherwise.} \end{cases} \tag{5.4}$$

Hence,

$$\deg(i,z) = \sum_{l \neq i,z} d(i,z,l). \tag{5.5}$$

Our algorithm then considers all of the candidates with the minimum out-degree. Specifically, $\begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix}$ is qualified if

$$\deg(i,j) = \min_{z \in Z_i} \deg(i,z). \tag{5.6}$$

If a qualified candidate also satisfies (5.2), it is chosen as a pivot. Otherwise, we remove j from the Z_i and repeat the process of selecting a 2-by-2 pivot until we either find a qualified candidate that also satisfies (5.3) or Z_i becomes empty. In the latter case, we set i to be the next minimum degree column and repeat the process from the beginning (from testing whether a_{ii} is a suitable 1-by-1 pivot). The algorithm is as shown in Algorithm 5.2 below. Lastly, when the remaining matrix is fully dense, we continue with a conventional pivot selection algorithm such as BBK instead.

Algorithm 5.2 Our Pivot Selection Algorithm

```
// $A$ is a $n$-by-$n$ symmetric indefinite matrix
Let $M = \{1, 2, \ldots, n\}$
while a suitable pivot is not yet found and $M$ is not empty do
    Let $i$ be the minimum degree column among all column indices in $M$
    if $a_{ii}$ is accepted then
        Use $a_{ii}$ as the 1-by-1 pivot
    else
        Let $Z_i = \{z | a_{iz} \neq 0 \text{ and } z \neq i\}$
        while a suitable pivot is not yet found and $Z_i$ is not empty do
            Let $j$ be such that $\begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix}$ has the minimum out-degree and $j \in Z_i$
            if $\begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix}$ satisfies (5.7) then
                Use $\begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix}$ as the 2-by-2 pivot
            else
                Remove $j$ from $Z_i$
            end if
        end while
        Remove $i$ from $M$
    end if
end while
```

5.3 Experiments and results

This section compares the efficiency of our algorithm with MA57, which is based on the multifrontal method. The experiments are performed in Matlab 2011a on matrices of varying dimensions from 100 to 5000. For each dimension, we vary the percentage of nonzeros in the matrices from 5 to 30 percent. The test problems are randomly generated in the following way: Let $\hat{A} \in \mathbb{R}^{n\times n}$. Each element of $\hat{A}$ and b is randomly generated between zero and one according to the uniform distribution. Then, let $A = \hat{A}\hat{A}^T/\left(\max_{i,j}\hat{a}_{i,j}\right)$. We then randomly zero out some of its entries in the lower triangular part and its corresponding entries in the upper triangular part until we reach the desired sparsity of A while retaining its symmetry. We test with 20 different instances for problems with 100, 300, and 500 dimensions and 10 different instances for problems with 1000, 3000, and 5000 dimensions. We show the percentage of nonzeros in the factor L of the two methods in Table 5.1, which shows that our method produces sparser factors than MA57 in all cases. Note that the small percentage improvement for large matrices are not insignificant as small decrease in nonzeros does lead to significantly faster factorization time. Finally, Table 5.2 shows the residuals $||P^TAP - LBL^T||$ of the results of both methods. The result shows that our method produces more accurate factors than MA57.

Table 5.1 Average percentage of nonzeros in the factor L produced by MA57 and our algorithm for problems with 100, 300, 500, 1000, 3000, and 5000 dimensions and 30, 20, 10, and 5 percent of nonzeros in the matrix. The percentage of nonzeros in L is computed by dividing the number of nonzeros in L by n^2 and then multiplying the result by 100.

n	Percentage of nonzeros in L							
	30		20		10		5	
	MA57	Our method	MA57	Our method	MA57	Our method	MA57	Our method
100	46.20	45.54	40.90	39.24	22.68	18.73	11.02	6.60
300	46.07	45.39	43.03	41.89	35.76	33.15	25.17	21.23
500	47.37	46.98	45.26	44.52	39.98	38.17	17.42	12.04
1000	48.53	48.35	47.39	47.00	44.01	43.02	38.46	36.36
3000	49.46	49.37	49.00	48.84	47.56	47.21	45.11	44.19
5000	49.64	49.61	49.36	49.26	48.47	48.22	46.86	46.23

Table 5.2 Average residuals of the factorization produced by MA57 and our algorithm for problems with 300, 500, 1000, and 2000 dimensions and 30, 20,10 and 5 percent of nonzeros in the matrix.

n	Residual ($\times 10^{-10}$)							
	30		20		10		5	
	MA57	Our method	MA57	Our method	MA57	Our method	MA57	Our method
100	0.00339	0.00018	0.00410	0.00022	0.00190	0.00016	0.00045	0.00006
300	0.03072	0.00077	0.02634	0.00083	0.02378	0.00083	0.01039	0.00059
500	0.08489	0.00128	0.06665	0.00161	0.04665	0.00169	0.02199	0.00076
1000	0.20679	0.00342	0.21691	0.00374	0.17399	0.00333	0.10509	0.00355
3000	1.63656	0.01312	1.80491	0.01281	1.32961	0.02150	1.13003	0.02160
5000	4.45974	0.02488	3.49949	0.02361	2.51524	0.03264	2.20916	0.03152

Chapter 6

Conclusions and Recommendations

Quadratic programming comes in many structures, which can be exploited to solve quadratic programs more efficiently. This thesis investigates a few efficient methods for solving block constraint quadratic programs and block diagonal quadratic programs.

In the first problem structure, we propose two methods for two kinds of the problem. First, we propose a heuristic method to find approximate solution for block diagonal quadratic programs. We focus on only dense and nonnegative constraints with lower bounds. Our method separates the original problem to subproblems and optimizes each subproblem. Then, we use the optimal solution of each subproblem to construct the approximate solution for the original problem. The results of the experiment show that our heuristic method is highly efficient for large scale problems especially when the problem does not have too many diagonal blocks. The second method is to solve quadratic programs with block diagonal Hessian and dense linear inequality constraint matrices. Our method is based on a direct method using symmetric indefinite factorization. This method exploits the known structure of the quadratic problem to efficiently compute the factors that are stable and retain the sparsity of the problem. The results of the experiments show that the proposed method is better at maintaining sparsity of the factors than the MA57 algorithm. Consequently, using the factors from this method to solve the KKT system is faster than using the factors from MA57 while yielding the solution that is as accurate. Note that, the steps in this pivot selection algorithm are easily parallelizable and therefore can be made more efficient with parallel computing.

To solve block constraint quadratic programs using a primal-dual interior-point method, the search directions must be computed. During search direction computation, the variables are separated according to the diagonal blocks of the Hessian matrix. The result of the experiment shows that the proposed method has better time complexity and uses less computational time than conventional methods for computing search directions.

Finally, we propose a new pivot selection algorithm for sparse symmetric indefinite factorization. Our method is based on the minimum degree algorithm but is able to select both 1-by-1 and 2-by-2 pivots that are stable. Our experimental results show that our algorithm produces factors that are stable and also sparser than MA57.

References

1. Zhang, H. W., Xu, W. L., Di, S. L., and 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.

2. Aboudolas, K., Papageorgiou, M., Kouvelas, A., and Kosmatopoulos, E. (2010). A rolling-horizon quadratic-programming approach to the signal control problem in large-scale congested urban road networks. *Transportation Research Part C: Emerging Technologies*, 18(5), 680–694.

3. Best, M. J. and Hlouskova, J. (2008). Quadratic programming with transaction costs. *Computers & Operations Research*, 35(1), 18–33.

4. Meng, C., Tuqan, J., and Ding, Z. (2009). A quadratic programming approach to blind equalization and signal separation. *IEEE Transactions on Signal Processing*, 57 (6), 2232–2244.

5. Nordebo S., Claesson, I., and Nordholm, S. (1994). Weighted Chebyshev approximation for the design of broadband beamformers using quadraticprogramming. *IEEE Signal Processing Letters*, 1(7), 103–105.

6. Bartlett, R. A., Biegler, L. T., Backstrom, J., and Gopal, V. (2002). Quadratic programming algorithms for large-scale model predictive control. *Journal of Process Control*, 13 (7), 775–795.

7. Zhang, H., Zhong, W., Wu, C., and 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.

8. Mitsui, K. and Tabata, Y. (2008). A stochastic linear–quadratic problem with L´evy processes and its application to finance. *Stochastic Processes and their Applications*, 118 (1), 120–152.

9. Kim, H. and Rassias, J. M. (2007). Generalization of Ulam stability problem for Euler-Lagrange quadratic mappings. *Journal of Mathematical Analysis and Applications*, 336(1), 277–296.

10. Liu, X., Wang, D., and Rong, J. (2009). Quadratic prediction and quadratic sufficiency in finite populations. *Journal of Mathematical Analysis and Applications*, 100 (9), 1979–1988.

11. Boyd, S. and Vandenberghe, L. (2004). *Convex optimization*. UK: Cambridge University Press.

12. Boggs, P. T. and Tolle, J. W. (1995). Sequential Quadratic Programming. *Acta Numerica*, 4, 1–51.

13. Hüeber, S., Mair, M., and Wohlmuth, B. I. (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(3-4), 555–576.

14. Yu, M. T., Lin, T. Y., and 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(1), 148–161.

15. Karmarkar, N. (1984). A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4), 373–395.

16. Ternet, D. J. and Biegler, L.T. (1999). Interior-point methods for reduced hessian successive quadratic programming. *Computers and Chemical Engineering*, 23(7), 859–873.

17. Wang, G. Q. and Bai, Y. Q. (2009). Primal-dual interior-point algorithm for convex quadratic semi-definite optimization. *Nonlinear Analysis: Theory, Methods & Applications*, 71(7–8), 3389–3402.

18. Wright, M. H. (1992). Interior methods for constrained optimization. *Acta Numerica*, 1, 341–407.

19. Nocedal, J. and Wright, S. J. (2006). *Numerical optimization.2nd Edn*. New York: Springer.

20. Markowitz, H. M. (1957). The elimination form of the inverse and its application to linear programming. *Management Science, 3*(3),255–269.

21. Tinney, W. F. and Walker, J. W. (1967). Direct Solution of Sparse Network Equations by Optimally Ordered Triangular Factorization. *Proceedings of the IEEE,* 55(11), 1801–1809.

22. Rose, D. J. (1972). A graph−theoretic study of the numerical solution of sparse positive definite systems of linear equations. In Reed, R. C. (Ed.). *Graph Theory and Computing*. New York: Academic Press.

23. George A. and Liu J. W. H. (1989). The evolution of the minimum degree ordering algorithm. *SIAM Rev.*, 31 (1), 1–19.

24. Yannakakis, M. (1981). Computing the Minimum Fill-In is NP-Complete. *SIAM Journal on Algebraic Discrete Methods,* 2(1), 77-79.

25. Cuthill, E. and McKee, J. (1969). Reducing the Bandwidth of Sparse Symmetric Matrices. *ACM '69 Proceedings of the 1969 24th national conference*, 157-172.

26. George, A. (1973). Nested Dissection of a Regular Finite Element Mesh. *SIAM Journal on Numerical Analysis*, 10(2), 345-363.

27. Lipton, R, J., Rose, D. J., and Tarjan, R. E. (1979). Generalized nested dissection. *SIAM Journal on Numerial Analysis,* 16(2), 346-358.

28. Liu, J. W. H. (1985). Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11(2), 141-153.

29. Amestoy, P. R., Davis, T. A., and Duff, I. S. (1996). An approximate minimum degree ordering algorithm. *SIAM Journal Matrix Analysis and Applications*, 17(4), 886-905.

30. Gill, P. E., Murray, W., and Wright, M. H.(1991). *Numerical Linear Algebra and Optimization; Vol. 1.* CA: Addison-Wesley.

31. Bunch, J. R. and Parlett, B. N. (1971). Direct Methods for Solving Symmetric Indefinite Systems of Linear Equations. SIAM Journal on Numerical Analysis, 8(4), 639-655.

32. Bunch, J. R. and Kaufman, L. (1977). Some Stable Methods for Calculating Inertia and Solving Symmetric Linear Systems. *Mathematics of Computation*, 31 (137), 163-179.

33. Ashcraft C., Grimes R. G., and Lewis J. G. (1995). Accurate Symmetric Indefinite Linear Equation Solvers. *SIAM J. Matrix Anal. Appl.*, 20 (2), 513–561.

34. Paige, C. C. and Saunders, M. A. (1974). Solution of Sparse Indefinite Systems of Linear Equations. *SIAM Journal on Numerial Analysis*, 12(4), 617-629.

35. Duff I. S., Reid J. K., Munksgaard, N., and Nielsen, H. B. (1979). Direct Solution of Sets of Linear Equations whose Matrix is Sparse, Symmetric and Indefinite. *IMA Journal of Applied Mathematics*, 23(2), 235–250.

36. Schenk, O. and Gärtner, K. (2006), On fast factorization pivoting methods for sparse symmetric indefinite systems. *Electronic Transactions on Numerical Analysis*, 23, 158–179.

37. Duff I. S. and Reid J. K. (1983). The Multifrontal Solution of Indefinite Sparse Symmetric Linear Equations. *ACM Trans. Math.Softw*, 9(3), 302-325.

38. Gould, N. I. M., Scott, J. A., and Hu, Y., A (2007). numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. *ACM Transactions on Mathematical Software*, 33(2) 118–144.

39. Duff I. S. (2004). MA57 – a code for the solution of sparse symmetric definite and indefinite systems. *ACM Transactions on Mathematical Software,* 30 (2), 118-144.

40. Rosen, J. B. and Pardalos, P. M. (1986). Global minimization of large-scale constrained concave quadratic problems by separable programming. Applied Mathematics and Computation, 34(2), 163–174.

41. Li, H. and Zhang, K. (2006). A decomposition algorithm for solving large-scale quadratic programming problems. *Applied Mathematics and Computation*, 173(1), 394–403.

42. Gill, P., Murray, W., Saunders, M., and Wright, M. (1987, October). A Schur-complement method for sparse quadratic programming. (Report No. SOL 87-12). Retrieved December 25, 2015, from
http://www.ccom.ucsd.edu/~peg/papers/schurQP.pdf

43. Gould, N. I. M. and Toint, P. L. (2002). An iterative working-set method for large-scale nonconvex quadratic programming. *Applied Numerical Mathematics*, 43(1-2), 109–128.

44. Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization*. USA: Springer.

45. Golub, G. H. and Loan, C. F. V. (2012). *Matrix computations, 4th Edn*. USA: The Johns Hopkins University Press.

46. George, A. and McIntyre, D. R. (1978). On the application of the minimum degree algorithm to finite element systems. *SIAM Journal on Numerical Analysis*, 15(1), 90-112.

47. George, A. and Liu, J. W. H. (1980). A Fast Implementation of the Minimum Degree Algorithm Using Quotient Graphs. *ACM Transactions on Mathematical Software,* 6(3), 337-358.

48. Duff, I. S., Gould, N. I. M., Reid, J. K., and Scott, J.A. (1991). The factorization of sparse symmetric indefinite matrices. *IMA Journal of Numerical Analysis*, 11(2), 181-204.