A LARGE-SCALE QUADRATIC PROGRAMMING SOLVER BASED ON BLOCK-LU UPDATES OF THE KKT SYSTEM

A DISSERTATION
SUBMITTED TO THE PROGRAM IN
SCIENTIFIC COMPUTING AND COMPUTATIONAL MATHEMATICS
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Hanh M. Huynh
September 2008

© Copyright by Hanh M. Huynh 2008
All Rights Reserved

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Michael Saunders) Principal Advisor

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Philip Gill)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Walter Murray)

Approved for the University Committee on Graduate Studies.

Abstract

Quadratic programming (QP) problems arise naturally in a variety of applications. In many cases, a good estimate of the solution may be available. It is desirable to be able to utilize such information in order to reduce the computational cost of finding the solution. Active-set methods for solving QP problems differ from interior-point methods in being able to take full advantage of such warm-start situations.

QPBLU is a new Fortran 95 package for minimizing a convex quadratic function with linear constraints and bounds. QPBLU is an active-set method that uses block-LU updates of an initial KKT system to handle active-set changes as well as low-rank Hessian updates. It is intended for convex QP problems in which the linear constraint matrix is sparse and many degrees of freedom are expected at the solution. Warm start capabilities allow the solver to take advantage of good estimates of the optimal active set or solution. A key feature of the method is the ability to utilize a variety of sparse linear system packages to solve the KKT systems.

QPBLU has been tested on QP problems derived from linear programming problems from the University of Florida Sparse Matrix Collection using each of the sparse direct solvers LUSOL, MA57, PARDISO, SuperLU, and UMFPACK. We emphasize the desirability of such solvers to permit separate access to the factors they compute in order to improve the sparsity of the updates. Further comparisons are made between QPBLU and SQOPT on problems with many degrees of freedom at the solution.

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Michael Saunders, for his insights and guidance. This work would not have been possible without his amazing depth of knowledge and tireless enthusiasm. I am very fortunate to have had the opportunity to work with him.

Special thanks go to Professor Philip Gill for his very helpful notes on quadratic programming and for pointing me towards SCCM in the first place. Many thanks go to Professor Walter Murray for his good cheer during my time here and for his many helpful insights.

I would like to thank Iain Duff for providing me with a license for MA57, and I would also like to thank Sherry Li for providing the separate L and U solve routines for SuperLU.

For their financial support, I would like to thank the Stanford Graduate Fellowship Program in Science and Engineering, COMSOL, the Army High-Performance Research Center, and the Institute for Computational and Mathematical Engineering.

Finally, I would like to thank my parents for their patience and support in all that I do.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Statement of the problem

The topic of interest of this paper is the quadratic programming (QP) problem of minimizing a quadratic objective function subject to linear constraints and bounds. Quadratic programming problems may be stated in many equivalent forms. We define the standard QP problem to be

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \quad & c^T x + \tfrac{1}{2} x^T H x \\ \text{subject to} \quad & Ax = b \\ & l \leq x \leq u, \end{aligned} \tag{1.1}$$

where the $n \times n$ Hessian matrix H is assumed to be symmetric, A is an $m \times n$ sparse matrix, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $l \in \mathbb{R}^n$, and $u \in \mathbb{R}^n$. Elements l_j and u_j may be taken as $-\infty$ or $+\infty$ if no bounds are present for these variables, and $l_j = u_j$ if fixed variables are present.

We assume that A is of full rank. In practice, it may be necessary to apply a preprocessing phase to A (such as those discussed in [54]) in order to ensure this property. For instance, rows of A may be eliminated to remove redundancies, or slack variables may be introduced.

Quadratic programming problems are typically classified as being convex or non-convex, depending on whether H is positive semidefinite or indefinite. In addition, we shall say that a quadratic programming problem is strictly convex if H is positive definite.

1.2 Optimality conditions

A triple (x, λ, z) is said to satisfy the first-order Karush-Kuhn-Tucker (KKT) conditions for (1.1) if

$$\begin{aligned} Ax &= b && (1.2)\\ z &= g(x) - A^T\lambda && (1.3)\\ \min(x - l,\ z) &= 0 && (1.4)\\ \min(u - x, -z) &= 0, && (1.5) \end{aligned}$$

where $g(x) = c + Hx$ is the gradient of the objective and z is the vector of reduced gradients. Conditions (1.4) and (1.5) enforce the bounds $l \leq x \leq u$ as well as the complementarity conditions

$$\begin{aligned} (x - l)^T z &= 0\\ (u - x)^T z &= 0. \end{aligned}$$

1.3 The dual problem

Duality provides an alternative formulation of a problem that may be more convenient computationally or that may be of theoretical interest. The dual formulation of the QP in standard form is

$$\begin{aligned} \underset{\lambda,x,v,w}{\text{maximize}} \quad & b^T\lambda + l^T v - u^T w - \tfrac{1}{2}x^T H x\\ \text{subject to} \quad & A^T\lambda + v - w = Hx + c \qquad (1.6)\\ & v, w \geq 0. \end{aligned}$$

It is important to note that if the primal problem is not convex then it might not be possible to derive the solution of the primal problem from the solution of the dual. See [31], for instance, for a more detailed discussion of the dual transformation.

1.4 Applications

Quadratic programming problems arise naturally in a variety of applications, such as portfolio optimization [60], structural analysis [4], and optimal control [9]. In many applications of quadratic programs, such as the classification of support vector machines [30], a good estimate of the final solution may be known in advance.

Quadratic programming problems are important in their own right, but they also play an important role in methods for nonlinear optimization. A survey by Gould and Toint [52] lists over 1000 published and unpublished works on quadratic programming. Not included in this list are the many hundreds of citations on sequential, successive, or recursive quadratic programming methods for nonlinear programming, in which generally constrained optimization problems are solved using a sequence of quadratic programming problems.

1.5 Overview

As larger and more challenging problems are considered, it becomes important to develop efficient algorithms for solving these QP problems. In this thesis, we consider efficient ways of solving quadratic programming problems in which the Hessian matrix H and linear constraint matrix A are large and sparse. By incorporating third-party "black box" linear system solvers, we are able to take advantage of the wealth of research and development in sparse linear algebra. In addition, this allows us to develop a QP solver that is easily adapted to parallel environments.

We give an overview of active-set methods for quadratic programming problems and a survey of some active-set solvers in Chapter 2. This motivates the development of the block-LU method described in Chapter 3, which is implemented using the inertia-controlling algorithm of Chapter 4. In Chapter 5 we consider a method for recovering from singular or ill-conditioned KKT systems. QPBLU, the Fortran 95 implementation of our active-set block-LU solver, is introduced in Chapter 6. Chapter 7 provides results and comparisons using different "black box" solvers within QPBLU and additional comparisons with the QP solver SQOPT [36]. Finally, insights and additional issues arising from the development of this block-LU method for quadratic programming are described in Chapter 8.

Chapter 2

Active-set methods

Active-set methods use an iterative process to find a solution of the QP problem. At the current (non-optimal) point x, a search direction p and nonnegative step length α are computed in order to define the next iterate $x + \alpha p$. An initial feasibility phase is used to find an initial point that satisfies the constraints in (1.1), and subsequent iterates are constructed to remain feasible by using a suitable search direction and step length.

The active set of constraints is defined to be the set of all constraints whose bounds hold with equality. For a problem in standard form (1.1), the active set contains the general linear constraints plus the set of variables that are temporarily fixed at their current value (usually on one of their bounds). The remaining variables are free to move. At each iteration, the quadratic programming algorithm defines a working set, which is a list of the constraints that are predicted to be active at the solution. The working set is a subset of the active constraints.

2.1 Properties of the standard form

Let the subscript "$_{FR}$" denote the indices of the columns of A that are not in the working set (the "free" variables), and let the subscript "$_{FX}$" denote the indices of the columns of A that are in the working set (the variables that are "fixed" at their current value). Let n_{FR} denote the number of free variables, so that the matrix A_{FR} denotes the $m \times n_{FR}$ submatrix of A whose columns correspond to the free variables. Similarly, let n_{FX} denote the number of fixed variables.

For a problem in standard form (1.1), the working-set matrix is composed of the matrix

A and rows of the identity corresponding to variables currently fixed at their current value. Given a permutation P such that $AP = \begin{pmatrix} A_{FR} & A_{FX} \end{pmatrix}$, the working-set matrix W can be divided into its "free" and "fixed" components

$$WP = \begin{pmatrix} A_{FR} & A_{FX} \\ 0 & I_{FX} \end{pmatrix},$$

where I_{FX} is the identity matrix with n_{FX} columns and whose rows correspond to the fixed variables.

Since the equality constraints $Ax = b$ will always be in the working-set matrix, working set changes correspond to adding or deleting rows of the identity from the working-set matrix.

2.2 Background

In a generic active-set method, if W is the working set for the current point x, the search direction p is obtained at each iteration by solving the equality constrained problem (EQP)

$$\begin{aligned} \underset{p \in \mathbb{R}^n}{\text{minimize}} \quad & g^T p + \tfrac{1}{2} p^T H p \\ \text{subject to} \quad & Wp = 0, \end{aligned} \tag{2.1}$$

where $g = c + Hx$. The constraints $Wp = 0$ ensure that the constraints in the working set remain active after $x + \alpha p$ for any α. The first-order necessary conditions for a point x to be a solution of (2.1) state that there is a vector of Lagrange multipliers λ such that x and λ satisfy the KKT system

$$\begin{pmatrix} H & W^T \\ W & 0 \end{pmatrix} \begin{pmatrix} p \\ -\lambda \end{pmatrix} = -\begin{pmatrix} g \\ 0 \end{pmatrix}. \tag{2.2}$$

If $p = 0$ is the solution to (2.1), then λ may be used to determine if the objective may be decreased by removing a row from $Wp = 0$. In the standard form, this means removing a variable from the set of fixed variables and allowing the variable to move from its current value.

If $p \neq 0$, then the objective function may be decreased by taking a step along p. We compute the largest possible step α, $0 \leq \alpha \leq 1$, that does not violate any bounds. If

$\alpha < 1$ then a new constraint becomes active, and the index of this constraint is added to the working set. If $\alpha = 1$, then $x + p$ solves the EQP. Optimality for the original QP problem may be checked using the Lagrange multipliers λ. If there is a multiplier with the "wrong" sign, then the corresponding constraint may be removed from the working set. Steps of length zero (degenerate steps) are possible when a constraint is active but not in the working set.

The working set at the next iteration differs from that of the current iteration by a single index. Unlike linear programming, neither the iterates nor the solution of the QP need be at vertices.

Active-set methods for quadratic programming can typically be classified as full-space, range-space, or null-space methods, depending on the manner in which the search direction is computed from the KKT system (2.2). In addition, these methods may be applied to either the primal (1.1) or dual (1.6) formulation of the QP problem and may employ either direct or iterative methods to solve the required linear systems.

Full-space methods work directly with the KKT matrix

$$K = \begin{pmatrix} H & W^T \\ W & 0 \end{pmatrix}$$

using, for instance, a triangular factorization. Since the KKT matrix is indefinite, the Cholesky factorization in general cannot be used. While Gaussian elimination with partial pivoting could be used, this approach does not take advantage of the symmetry of the system. A symmetric indefinite factorization of the form $K = LDL^T$ is an efficient strategy. Iterative methods for general linear systems or for symmetric indefinite systems are an alternative to approaches using a direct factorization, though they are not often used in active-set methods.

For a problem in standard form, the search direction can be computed from the smaller linear system

$$\begin{pmatrix} H_{FR} & A_{FR} \\ A_{FR} & \end{pmatrix} \begin{pmatrix} p_{FR} \\ -\lambda \end{pmatrix} = - \begin{pmatrix} g_{FR} \\ 0 \end{pmatrix},$$

where H_{FR} is the $n_{FR} \times n_{FR}$ submatrix of H corresponding to the free variables. The search direction is equal to zero for variables that are fixed. That is, $p_{FX} = 0$.

Range-space methods require H to be nonsingular in order to compute p by eliminating

the first block of (2.2):

$$\begin{aligned} WH^{-1}W^T\lambda &= WH^{-1}g \\ Hp &= W^T\lambda - g. \end{aligned}$$

This approach requires solves with H and forming and factorizing the matrix $WH^{-1}W^T$. Range-space methods can be used when H is positive definite, well-conditioned, and easy to invert (diagonal or block-diagonal, for example), when H^{-1} is known explicitly, or when the number of equality constraints is small, so that $WH^{-1}W^T$ is inexpensive to compute.

Null-space methods utilize an $n \times (n_{FR} - m)$ matrix Z that forms a basis for the null space of W. Like the range-space method, it uses the block structure of the KKT system to decouple (2.2) into two smaller systems. The search direction p is obtained from the system

$$Z^THZ = -Z^Tg.$$

The matrix Z^THZ is known as the reduced Hessian for the current working set, and Z^Tg is the corresponding reduced gradient. For large, sparse problems the columns of A_{FR} may be partitioned as

$$A_{FR} = \begin{pmatrix} B & S \end{pmatrix},$$

where B is an $m \times m$ nonsingular basis matrix and S is the $m \times (n_{FR} - m)$ matrix of the superbasic columns of A. The value $n_S = n_{FR} - m$ is known as the number of degrees of freedom. When A_{FR} has this form, a basis for the null space of A_{FR} may be defined by the columns of the matrix Z_{FR}, where

$$Z_{FR} = \begin{pmatrix} -B^{-1}S \\ I \end{pmatrix}, \qquad Z = \begin{pmatrix} Z_{FR} \\ 0 \end{pmatrix}$$

and

$$Z^THZ = Z_{FR}^T H_{FR} Z_{FR},$$

where H_{FR} is the $n_{FR} \times n_{FR}$ submatrix of H corresponding to the free variables. Note that the reduced Hessian and reduced gradient can be formed (using solves with B and B^T) without Z_{FR} being formed explicitly.

2.3 Properties of active-set methods

In some applications, a good estimate of the solution or the optimal active set will be available in advance. It is desirable, especially when solving a sequence of related problems, to be able to exploit this estimate of the final active set. Such "warm starts" can greatly reduce the computational effort required to find a solution.

At each iteration active-set methods maintain a prediction of the set of constraints that are active at the solution, and warm starts can be incorporated naturally. This is not the case with interior-point methods, although some advances in this area have been made [44, 74, 10]. Active-set methods have an advantage over interior-point methods in easily permitting such warm starts.

For this reason, active-set methods are generally the method of choice in such applications as model predictive control and in the solution of subproblems arising as part of sequential quadratic programming (SQP) algorithms. Problems in model predictive control are not usually encountered in isolation, but rather as a sequence of related problems. The data vary only slightly from one problem to the next. A good guess can be made for the initial starting values or for the set of active constraints by using the solution to the previous problem. SQP methods solve a sequence of related problems whose optimal active sets may be nearly identical. As the major iterations of an SQP converge, the QP subproblems require fewer changes to the current working set.

Warm start capabilities are particularly important with active-set methods, since the choice of an initial active set will influence the number of iterations needed to reach an optimal point. In a typical active-set method, at most one constraint is added to or deleted from the working set at each iteration. Even when there is more than one blocking constraint, only one constraint is added to the working set. Likewise, at most one constraint is deleted at each iteration. This places a natural lower bound on the number of iterations required to reach an optimal point. For example, if an initial set of ℓ active constraints is inactive at the solution, then at least ℓ iterations are needed to reach the solution. More iterations are required if a constraint is added to the working set at one iteration but is later removed.

This lower bound is unlike the situation for interior point methods, in which the number of iterations required typically scales very moderately with the number of variables. Interior-point methods typically require fewer, though more expensive, steps to reach an optimal point, while active-set methods generally require many more relatively inexpensive steps.

Name	Method	Large-scale	QP type	Authors
QPA	Primal, null-space	Yes	General	N. I. M. Gould, D. Orban, Ph. L. Toint
QPKWIK	Dual	No	Strictly convex	C. Schmid, L. T. Biegler
QPOPT	Primal, null-space	No	General	P. E. Gill, W. Murray, M. A. Saunders
QPSchur	Dual, full-space	Yes	Strictly convex	R. Bartlett, L. Biegler
SQOPT	Primal, null-space	Yes	Convex	P. E. Gill, W. Murray, M. A. Saunders

Table 2.1: A summary of some existing active-set QP solvers.

2.4 Some existing solvers

This section presents a survey of available active-set QP solvers. A summary of the solvers discussed in the following sections is provided in Table 2.1.

2.4.1 QPA

QPA is part of GALAHAD [49], a library of Fortran 90 packages for nonlinear optimization. It is primarily used within GALAHAD to solve quadratic programs in which a good estimate of the final active set is known and relatively few iterations are needed. This method uses a null-space approach based on a projected preconditioned conjugate gradient method to compute the search direction at each iteration. Preconditioning of the conjugate-gradient method requires solutions of systems of the form

$$\begin{pmatrix} M^{(k)} & W^{(k)T} \\ W^{(k)} & 0 \end{pmatrix} \begin{pmatrix} p \\ u \end{pmatrix} = \begin{pmatrix} g \\ 0 \end{pmatrix}, \tag{2.3}$$

where $M^{(k)}$ is an approximation to the Hessian and $W^{(k)}$ is the current working-set matrix. Rather than refactorizing this matrix at every iteration, this method factorizes an initial matrix

$$K_0 = \begin{pmatrix} M^{(l)} & W^{(l)T} \\ W^{(l)} & 0 \end{pmatrix}$$

is factorized at a given iteration l. Solutions to (2.3) at subsequent iterations use this factorization of K_0 and a factors of a certain small matrix.

2.4.2 QPKWIK

QPKWIK [70] is a convex quadratic programming solver based on the dual-space algorithm of Goldfarb and Idnani [42]. A feature of this method is that it requires only the inverse Cholesky factor of the positive definite Hessian matrix to be supplied. Since it does not use a sparse Cholesky, this solver is not suitable for large-scale problems. QPKWIK is used primarily to solve the highly structured convex quadratic programming problems that arise in nonlinear model predictive control. It was used as the engine for local steady-state optimization within the Aspen Technology's Aspen Target product [3] for model predictive control.

2.4.3 QPOPT

QPOPT [34] is an active-set method for general quadratic programming based on an inertia controlling method [41]. It uses a null-space approach and maintains a dense LQ factorization of the working-set matrix and a dense Cholesky factorization of the reduced Hessian. Because all matrices are treated as dense, this solver is not intended for sparse problems, although there is no fixed limit on the problem size. The method is most efficient when there are many constraints or bounds active at the solution or when applied to a sequence of related problems such as in SQP methods.

2.4.4 QPSchur

QPSchur [7] is an active-set, dual-feasible Schur-complement method for strictly convex quadratic programming problems. The method, based on the dual algorithm of Goldfarb and Idnani [42], computes a sequence of primal and dual vectors satisfying the KKT conditions except for primal feasibility. A disadvantage of the dual-space algorithm is that the Hessian matrix is required to be positive definite, so the method as a whole lacks generality.

Like other active-set methods, QPSchur can also take advantage of a good initial estimate of the working set at a solution. However, since the initial guess of the optimal working set may not be dual feasible, additional procedures must be used to adjust the initial working set when implementing warm starts. QPSchur can exploit structured Hessian and constraint

matrices (by calling specialized BLAS and LAPACK routines), making it suitable for large, sparse and structured systems.

2.4.5 SQOPT

SQOPT [36] is a package for solving convex quadratic programming problems. It uses a null-space method related to that used in the QPOPT package, except that Z is generated from a sparse basis factorization of the form (2.3). SQOPT maintains a dense Cholesky factorization of the reduced Hessian when the number of superbasic variables $n_S \leq 2000$ and uses a conjugate-gradient method when $n_S > 2000$. It is designed to solve large linear and quadratic problems in which the constraint matrices are sparse. Like QPOPT, SQOPT is most efficient on problems in which the solution has few degrees of freedom compared to the total number of variables. It is used as part of the SNOPT [35] package for large-scale nonlinearly constrained optimization.

2.4.6 Comparisons to other methods

While active-set methods may not be the method of choice for solving general quadratic programs, there are many situations in which an active-set method is to be preferred, especially when using warm starts.

A comparison between QPA and QPB [49], an interior-point trust-region method for quadratic programming, was performed by Gould and Toint [53] using the CUTE [15] QP test set. It was found that the two methods were comparable for modest-sized problems when started from random points. QPA required many more iterations than QPB as the problem dimensions increased. When a good estimate of the optimal active set was used to initialize the solvers, it was found that QPA generally outperformed QPB, except for problems that were degenerate, nearly degenerate, or ill-conditioned.

A comparison of active-set and interior-point methods for problems in Nonlinear Model Predictive Control (NMPC) was made by Bartlett et al. [8]. In this study, the active-set QP solvers QPSchur, QPOPT, QPKWIK, and an interior-point method using a Mehrotra predictor-corrector strategy were applied to a problem in NMPC as part of an SQP method.

It was found that although the active-set solvers generally required more iterations than the interior-point method as the problem sized increased, CPU times of the active-set methods were competitive with the interior-point method.

Chapter 3

A block-LU method

3.1 Motivation

As we have seen, many excellent active-set quadratic programming solvers are available. However, no solver or approach is ideal for all problems.

Methods applied to the dual formulation of the quadratic programing problem can only be used when the Hessian matrix is positive semidefinite. Range-space methods for quadratic programming are also limited to certain classes of problems. While the null-space method has a wider range of applicability than the range-space method, it may become inefficient when the number of degrees of freedom is large. A problem will have few degrees of freedom if, for instance, the number of active constraints is nearly as large as the number of variables. For problems with many degrees of freedom, the reduced Hessian $Z^T H Z$ can be expensive to form.

No matter what type of method is used, the solution of linear systems of equations forms the foundation of any method to solve quadratic programming problems. Care must be taken to ensure that these systems are solved efficiently. For instance, a drawback of the current implementation of QPSchur is in the handling of the Schur complement matrix. Depending on the inertia of the Schur complement matrix, this method uses either a Cholesky or Bunch-Kaufman [43] factorization of the Schur complement. A Cholesky factorization is updated when the Schur complement matrix is positive or negative definite, but if the matrix is found to be indefinite, a Bunch-Kaufman factorization is recomputed from scratch for every change to the Schur complement.

Linear algebraic computations comprise a substantial amount of the time spent solving

quadratic programs, even for problems of moderate size. The dominant cost per iteration in an active-set method is in the solution of the current KKT system, each of which is a rank-one modification of its predecessor. It is vital that QP methods be able to take advantage of advances in linear system solvers and of the structure and sparsity of the problem itself in order to compute a solution efficiently.

The particular concern of this thesis is in the use of active-set methods to find the solution of medium- to large-scale QP problems in which the linear constraint matrix is sparse and many degrees of freedom are expected at the solution. The block-LU method is presented as a technique that allows a variety of sparse linear system solvers to be utilized effectively on the sequence of systems that arise as the working set changes.

3.2 Background on the block-LU method

The block-LU method is a variation of the Schur-complement method, which was introduced for linear programming by Bisschop and Meeraus [14] and subsequently studied for quadratic programming by Gill et al. [40] and Betts and Frank [12]. The block-LU update was first discussed for linear programming in Gill et al. [37] and a variant of this method was also proposed by Wright [72] for quadratic programming. A block-LU method for linear programming has been implemented by Eldersveld and Saunders [29], which demonstrates not only the practicality of such a method but also its potential to take advantage of machine architecture and advances in linear algebra. Both the Schur-complement and block-LU methods are full-space methods that work directly with a factorization of an initial KKT matrix and take advantage of the fact that the working set changes by at most a single constraint at any iteration.

For a quadratic programming problem in standard form, an initial KKT system can be written as

$$K_0 \begin{pmatrix} p_{FR} \\ -\lambda \end{pmatrix} = -\begin{pmatrix} g_{FR} \\ 0 \end{pmatrix}, \qquad K_0 = \begin{pmatrix} H_0 & A_0^T \\ A_0 & 0 \end{pmatrix}. \tag{3.1}$$

The KKT matrix K_0 at the next iteration differs from the previous by the addition or deletion of only one row and column. After a sequence of iterations the solution of the

current KKT system may be found by way of an augmented system of the form

$$\begin{pmatrix} K_0 & V \\ V^T & D \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} = \begin{pmatrix} f \\ w \end{pmatrix}, \tag{3.2}$$

where the matrices V and D arise from changes in the working set due fixing initially free variables or freeing initially fixed variables.

In the Schur-complement method, the solution to the current KKT system is obtained using the factors of the initial KKT matrix K_0 and a dense Schur complement matrix $C = D - V^T K_0^{-1} V$. The following equations are solved in turn to obtain y, z:

$$\begin{aligned} K_0 t &= f \\ Cz &= w - V^T t \\ K_0 y &= f - Vz. \end{aligned}$$

Using this method, the work to compute the search direction in one QP iteration is dominated by two solves with K_0 and one solve with C. Solves with K_0 may be handled using an initial factorization $K_0 = L_0 U_0$.

The block-LU method maintains a block factorization of the augmented matrix in (3.2):

$$\begin{pmatrix} K_0 & V \\ V^T & D \end{pmatrix} = \begin{pmatrix} L_0 & \\ Z^T & I \end{pmatrix} \begin{pmatrix} U_0 & Y \\ & C \end{pmatrix}.$$

The block factors L_0, U_0, C, Y, and Z are defined by

$$\begin{aligned} K_0 &= L_0 U_0 \\ L_0 Y &= V \\ U_0^T Z &= V \\ C &= D - Z^T Y \\ &= D - V^T K_0^{-1} V, \end{aligned}$$

where the matrix C is the Schur complement of K_0 in the augmented matrix. The solution

to system (3.2) may then by found by solving

$$\begin{aligned} L_0 t &= f \\ Cz &= w - Z^T t \\ U_0 y &= t - Yz. \end{aligned}$$

This method requires only one solve with K_0 and one with C at each iteration in order to compute the search direction and λ.

3.3 Comparisons to the Schur-complement method

The block-LU method shares many of the same properties of the Schur-complement method. Both methods make use of an initial KKT matrix K_0 and the Schur complement of K_0 in an augmented system to represent the current KKT matrix. The primary differences result from storing the block factors of the augmented matrix.

In either method, the initial KKT matrix K_0 may be treated as a "black box" and factorized by any available method. This factorization is then used repeatedly to solve the current KKT system. After a maximum number of iterations, the current KKT matrix is formed, factorized, and is used as the next initial KKT matrix. Refactorization is generally used to limit the storage required for large-scale problems and also provides an opportunity to recompute an accurate solution after an accumulation of rounding errors. The elements of K_0 need not be stored or accessed once the factorization has been completed, unless iterative refinement is desired.

The dimensions of the Schur complement matrix C will not be larger and could be less than the refactorization frequency of K_0. Since the refactorization frequency will be relatively small for large-scale problems, it is efficient to treat C as a dense matrix in both methods. The matrix C itself is usually not formed and updated explicitly. Instead, QR or LU factors of C are updated for each change to the working set. Additional solves with K_0 may be necessary to update C. Provided that K_0 is reasonably well-conditioned, updates to the block factorization and Schur complement are stable. The Schur complement matrix C will tend to reflect the condition of the current KKT matrix. Ill-conditioning of C need not persist, however, since rows and columns of C may be deleted or added as the working set changes.

While more storage is required for the block-LU method than for the Schur-complement method, computing time may be saved by storing the block factors Y and Z rather than performing an additional solve with K_0. The block-LU method implemented by Eldersveld and Saunders [29] was used to update the basis matrix B_0 in a simplex method for linear programming problems. Their original intent was to investigate the performance of the Schur-complement method to update B_0. However, the Schur-complement method was found to be excessively expensive when compared to the Bartels-Golub update [6], due to the additional solves with B_0 and B_0^T required by the Schur-complement method. The block-LU method was (successfully) implemented as an alternative to the Schur-complement method, trading additional workspace for less computation time. This implementation was additionally able to take advantage of the vector hardware of a Cray-MP in the storage and handling of the block factors.

3.4 The augmented KKT matrix

The manner in which the block factorization of the current KKT matrix is updated depends on whether the column is entering or leaving the working set and whether that column formed part of the initial KKT matrix

$$K_0 = \begin{pmatrix} H_0 & A_0^T \\ A_0 & 0 \end{pmatrix},$$

where H_0 is the $n_0 \times n_0$ initial Hessian matrix, A_0 is the $m \times n_0$ matrix whose columns form the initial basis, and n_0 is the initial number of free variables. Changes to the working set result in an augmented KKT system of the form (3.2). At each iteration, this system must be updated to reflect the changes in the working set. These changes can be represented by bordering the initial KKT matrix by a row and column.

3.4.1 An example

There are four possible ways that a column of A may enter or leave the working set. To illustrate how to augment the initial KKT matrix as the working set changes, we consider an example with four variables and a single general constraint. Variables 1 and 3 are initially free to move and variables 2 and 4 are fixed on their bounds. The initial KKT system (3.1) is

$$\begin{pmatrix} h_{11} & h_{13} & a_{11} \\ h_{13} & h_{33} & a_{13} \\ a_{11} & a_{13} & 0 \end{pmatrix} \begin{pmatrix} p_1 \\ p_3 \\ -\lambda_1 \end{pmatrix} = \begin{pmatrix} g_1 \\ g_3 \\ 0 \end{pmatrix}. \tag{3.3}$$

Note that elements of the search direction p corresponding to the fixed variable are equal to zero. That is, $p_2 = p_4 = 0$. Since H is symmetric, it is written in terms of its upper triangular part.

Case 1. To free a variable r from its bound when r is not a member of the initial set of free variables, the KKT system is bordered with the free elements of a row and column of H and the r^{th} column of A. Here, we take $r = 2$:

$$\left(\begin{array}{ccc|c} h_{11} & h_{13} & a_{11} & h_{12} \\ h_{13} & h_{33} & a_{13} & h_{23} \\ a_{11} & a_{13} & 0 & a_{12} \\ \hline h_{12} & h_{23} & a_{12} & h_{22} \end{array}\right) \left(\begin{array}{c} p_1 \\ p_3 \\ -\lambda_1 \\ \hline p_2 \end{array}\right) = - \left(\begin{array}{c} g_1 \\ g_3 \\ 0 \\ \hline g_2 \end{array}\right).$$

Case 2. To fix a variable s at its current value when s is a member of the initial set of free variables, the KKT system is bordered with a row and column of the identity. Here, we take $s = 3$:

$$\left(\begin{array}{ccc|cc} h_{11} & h_{13} & a_{11} & h_{12} & 0 \\ h_{13} & h_{33} & a_{13} & h_{23} & 1 \\ a_{11} & a_{13} & 0 & a_{12} & 0 \\ \hline h_{12} & h_{23} & a_{12} & h_{22} & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right) \left(\begin{array}{c} p_1 \\ p_3 \\ -\lambda_1 \\ \hline p_2 \\ z_1 \end{array}\right) = - \left(\begin{array}{c} g_1 \\ g_3 \\ 0 \\ \hline g_2 \\ 0 \end{array}\right).$$

This additional row has the effect of setting $p_3 = 0$, adding $s = 3$ to the working set.

Case 3. To be able to fix a variable s on its bound when it is not a member of the initial set of free variables, s must have been added to the set of free variables and the KKT system updated as in *Case 1.* To fix this variable on its bound, the corresponding column

of V and row and column of D are deleted from the augmented system:

$$\left(\begin{array}{ccc|c} h_{11} & h_{13} & a_{11} & 0 \\ h_{13} & h_{33} & a_{13} & 1 \\ a_{11} & a_{13} & 0 & 0 \\ \hline 0 & 1 & 0 & 0 \end{array}\right) \left(\begin{array}{c} p_1 \\ p_3 \\ -\lambda_1 \\ \hline z_1 \end{array}\right) = - \left(\begin{array}{c} g_1 \\ g_3 \\ 0 \\ \hline 0 \end{array}\right)$$

Case 4. To be able to free a variable r from it bound when it was a member of the initial set of free variables, r must have been fixed on a bound and the KKT system updated as in *Case 2*. To free this variable from its bound, again the corresponding column of V and row and column of D are deleted from the augmented system, resulting in the original system (3.3).

The process described above can be repeated over a sequence of iterations to form the augmented system (3.2). In a block-LU method, this augmented system will usually increase in dimension by one for every change to the working set.

3.5 Updating the block factorization

Let K_0 be a nonsingular matrix with LU factorization $K_0 = L_0 U_0$. Consider the following block-LU factorization of the augmented matrix:

$$\begin{pmatrix} K_0 & V \\ V^T & D \end{pmatrix} = \begin{pmatrix} L_0 & \\ Z^T & I \end{pmatrix} \begin{pmatrix} U_0 & Y \\ & C \end{pmatrix}.$$

The matrices C, Y, Z are updated as rows or columns of $\begin{pmatrix} V \\ D \end{pmatrix}$ are added or removed from the augmented matrix. If a column $\begin{pmatrix} v \\ d \end{pmatrix}$ is appended, then C and Y are updated by setting

$$\begin{aligned} Y &= \begin{pmatrix} Y & y \end{pmatrix}, \text{ where } L_0 y = v \\ C &= \begin{pmatrix} C & c \end{pmatrix}, \text{ where } c = d - Z^T y. \end{aligned}$$

If a row $\begin{pmatrix} v^T & d^T \end{pmatrix}$ is appended, then C and Z are updated by setting

$$\begin{aligned} Z &= \begin{pmatrix} Z & z \end{pmatrix}, \text{ where } U_0^T z = v, \\ C &= \begin{pmatrix} C \\ c^T \end{pmatrix}, \quad \text{where } c^T = d^T - z^T Y. \end{aligned}$$

If a column of $\begin{pmatrix} V \\ D \end{pmatrix}$ is deleted, then the corresponding columns of C and Y are deleted. If a row of $\begin{pmatrix} V^T & D \end{pmatrix}$ is deleted, then the corresponding row of C and column of Z are deleted.

Instead of explicitly updating the matrix C or its inverse, we choose to update a factorization of C.

3.5.1 Hessian updates

The block factorization method can also be used within an SQP method in which updates to the Hessian matrix are of the form $H_1 = (I + vu^T)H_0(I + uv^T)$. When computed explicitly, this update may cause the current Hessian matrix to be very dense. The block-LU method can accommodate such changes to H while preserving sparsity in the current KKT system. The solution to a system of the form

$$\begin{pmatrix} H_1 & W^T \\ W & \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \end{pmatrix} \tag{3.4}$$

may be obtained by solving the augmented system

$$\left(\begin{array}{cc|cc} H_0 & W^T & \bar{u} & v \\ W & & & \\ \hline \bar{u}^T & & \gamma & -1 \\ v^T & & -1 & \end{array}\right) \left(\begin{array}{c} y_1 \\ y_2 \\ \hline r \\ s \end{array}\right) = \left(\begin{array}{c} d_1 \\ d_2 \\ \hline 0 \\ 0 \end{array}\right),$$

where $\bar{u} = H_0 u$ and $\gamma = u^T H_0 u$:

$$\begin{aligned} H_0 y_1 + W^T y_2 + r\bar{u} + sv &= d_1 \\ W y_1 &= d_2 \\ \bar{u}^T y_1 + r\gamma - s &= 0 \\ \bar{v}^T y_1 - r &= 0. \end{aligned}$$

Eliminating the terms $s = \bar{u}^T y_1 + r\gamma$ and $r = \bar{v}^T y_1$, we obtain

$$\begin{aligned} H_0 y_1 + (v^T y_1)\bar{u} + (\bar{u}^T y_1)v + \gamma(v^T y_1)v + W^T y_2 &= d_1 \\ W y_1 &= d_2. \end{aligned} \tag{3.5}$$

Expanding the updated Hessian

$$\begin{aligned} H_1 &= (I + vu^T)H_0(I + uv^T) \\ &= H_0 + H_0 uv^T + vu^T H_0 + vu^T H_0 uv^T \\ &= H_0 + \bar{u}v^T + v\bar{u}^T + \gamma vv^T, \end{aligned}$$

we find that (3.5) is equivalent to

$$\begin{aligned} H_1 y_1 + W^T y_2 &= d_1 \\ W y_1 &= d_2 \end{aligned}$$

and therefore solves system (3.4).

By induction, after k low-rank updates to H_0 of the form

$$\begin{aligned} H_k &= (I + v_k u_k^T)H_{k-1}(I + u_k v_k^T) \\ &= (I + v_k u_k^T)\cdots(I + v_1 u_1^T)H_0(I + u_1 v_1^T)\cdots(I + u_k v_k^T), \end{aligned}$$

the KKT system for the current Hessian H_{k+1}

$$\begin{pmatrix} H_{k+1} & W^T \\ W & \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \end{pmatrix}$$

is equivalent to an augmented KKT system involving the initial Hessian H_0:

$$\left(\begin{array}{cc|ccccc} H_0 & W^T & \bar{u}_1 & \bar{v}_1 & \cdots & \bar{u}_k & \bar{v}_k \\ W & & & & & & \\ \hline \bar{u}_1^T & & \gamma_1 & -1 & & & \\ v_1^T & & -1 & & & & \\ \vdots & & & & \ddots & & \\ \bar{u}_k^T & & & & & \gamma_k & -1 \\ v_k^T & & & & & -1 & \end{array}\right)\left(\begin{array}{c} y_1 \\ y_2 \\ \hline r_1 \\ s_1 \\ \vdots \\ r_k \\ s_k \end{array}\right) = \left(\begin{array}{c} d_1 \\ d_2 \\ \hline 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{array}\right),$$

where $\bar{u}_i = H_{i-1}u_i$ and $\gamma_i = u_i^T H_{i-1} u_i$ for $i = 1, \ldots, k$. This type of Hessian update may also be interleaved with working set updates to form an augmented system involving the initial H_0.

3.6 Sparsity of Y and Z

The accumulation of nonzeros elements in the block factors Y and Z depends on many things, such as the sparsity of the columns that form V, the sparsity of the initial matrix K_0, and the details of the factorization used to form $K_0 = L_0 U_0$. A simple and effective, though not always readily available, technique for maintaining sparsity in both Y and Z is to utilize separate solves with the matrix factors L_0 and U_0. Whenever possible, it is important to be able to treat L_0 and U_0 as two "black boxes" for solving linear systems. Doing so helps to maintain the highest degree of sparsity in both Y and Z, reducing the computation time for operations involving these matrices and preventing the premature refactorization of K_0 due to insufficient storage for the nonzero elements of Y or Z.

For solvers in which separate solves are available, Y and Z may be computed as

$$L_0 Y = V, \qquad U_0^T Z = V,$$

where the columns of V augment K_0. For symmetric factorizations of the form $K_0 = L_0 D_0 L_0^T$ we can define $U_0 = D_0 L_0^T$. Using separate solves with L_0 and U_0, we may expect approximately the same degree of sparsity in both Y and Z. For solvers in which separate solves are not available, we must take "L_0" $= I$ and "U_0" $= L_0 U_0$. That is, Y and Z are

defined by

$$Y = V, \qquad U_0^T L_0^T Z = V.$$

Since V is expected to be sparse, this method of updating Y and Z preserves the sparsity of Y at the expense of that of Z.

Chapter 4

QP algorithm

The algorithm for the quadratic program solver is based on the inertia-controlling method of Gill et al. [41]. This is a primal-feasible method that does not require that the objective function be strictly convex. Inertia-controlling methods are a class of algorithms for indefinite QP that never allow the reduced Hessian to have more than one nonpositive eigenvalue. In this section we introduce the inertia-controlling algorithm and present results from [41]. Proofs of the lemmas stated in this chapter may be found in [41] and [33].

4.1 The inertia-controlling method

The inertia-controlling method in [41] was derived for general quadratic programming problems of the form

$$\begin{array}{ll} \underset{x\in\mathbb{R}^n}{\text{minimize}} & c^Tx + \frac{1}{2}x^THx \\ \text{subject to} & Ax \geq b. \end{array}$$

The results of this section pertain to this form of the problem, with the assumption that H is positive semidefinite.

The working-set matrix W for this form of the QP typically consists of the rows of A that are active. Given W of full row rank, we say that a point x such that $g(x) = W^T\lambda$ for some vector λ is a subspace stationary point (with respect to W). Given a null-space basis Z for W, a subspace stationary point such that Z^THZ is positive definite is said to be a subspace minimizer (with respect to W).

The vector x is said to be a non-optimal subspace minimizer with respect to the working-set matrix W if there exists an index s in the working set such that

$$g = W^T\lambda \qquad \text{and} \qquad \lambda_s < 0.$$

Since $\lambda_s < 0$, it is possible to find a search direction p that reduces the value of the objective while retaining feasibility. One possibility for p is the solution to the equality constrained problem (2.1). In the strictly convex case, H is positive definite and the KKT system associated with the EQP is nonsingular. In the general convex case, the reduced Hessian Z^THZ (and therefore the KKT system) may be singular.

Inertia-controlling methods choose the working set in order to control the inertia of the reduced Hessian. In particular, the reduced Hessian is never permitted to have more than one zero eigenvalue. At any initial iterate x_0, it is possible to find enough real or temporary constraints to define a working-set matrix W_0 such that $Z_0^THZ_0$ is nonsingular. The reduced Hessian can only become singular when a constraint is deleted from the working set. When the reduced Hessian is singular at a non-optimal point, it is used to define a search direction p such that

$$Z^THZp_Z = 0 \qquad \text{and} \qquad g^TZp_Z < 0. \tag{4.1}$$

The vector $p = Zp_Z$ is said to be a direction of descent. A step in such a direction normally reaches a new constraint r (if the objective is bounded below within the feasible region). Constraint r is then added to the working set. Inertia-controlling methods differ from regular active-set methods in that no constraints are deleted from the working set until the reduced Hessian becomes nonsingular.

Lemma 4.1.1. *Let W_+ be a matrix of full row rank. Let W denote W_+ with its s^{th} row omitted, and note that W also has full row rank. Let the matrices Z_+ and Z denote the null-space bases for W_+ and W, and let $Z_+^THZ_+$ and Z^THZ be the corresponding reduced Hessians. Assume that $Z_+^THZ_+$ is positive definite and that x is a subspace minimizer such that $g = W_+^T\lambda$ and $\lambda_s < 0$. Consider the linear system*

$$\begin{pmatrix} H & W_+^T \\ W_+ & 0 \end{pmatrix} \begin{pmatrix} u \\ \mu \end{pmatrix} = \begin{pmatrix} 0 \\ e_s \end{pmatrix}. \tag{4.2}$$

Let μ_s denote the s^{th} component of μ. The following properties hold:

1. *If $\mu_s < 0$, then $Z^T H Z$ is positive definite.*

2. *If $\mu_s = 0$, then $Z^T H Z$ is singular.*

3. *If $Z^T H Z$ is positive definite, then $x + p$ with $p = (\lambda_s/\mu_s)u$ minimizes the equality constrained QP (2.1) with constraint matrix W.*

4. *If $Z^T H Z$ is singular, then $p = u$ satisfies both the descent condition $g^T p < 0$ and $p^T H p = 0$, in addition to maintaining feasibility of the constraints.*

The goal of this approach is to compute the search direction p using a system of the form

$$\begin{pmatrix} H & W_+^T \\ W_+ & 0 \end{pmatrix}, \tag{4.3}$$

that is guaranteed to be nonsingular regardless of whether the reduced Hessian $Z^T H Z$ is positive definite or not.

Lemma 4.1.1 depends on x being a subspace minimizer with respect to W_+. This is the case when a full-length step $\alpha = 1$ has just been taken, but it does not necessarily hold when $\alpha < 1$. This difficulty is remedied by retaining the index s of the most recently deleted constraint in the working set, even though this constraint may be inactive at the current point.

In this chapter we use the following notation. For a working set that includes the index s of the most recently deleted constraint, the corresponding working-set matrix is denoted by

$$W_+ = \begin{pmatrix} W \\ a_s^T \end{pmatrix}, \tag{4.4}$$

where W is formed from a subset of the active constraints. The subscript "$_+$" indicates a quantity corresponing to a working set in which the most recently deleted constraint has been retained. A bar over a quantity indicates its value at the next iteration. For example, $\bar{x} = x + \alpha p$ is the resulting iterate after a step along p has been taken. If a constraint has been added to the working set, and the working set does not include the most recently deleted constraint, then the working-set matrix at the next iteration is

$$\bar{W} = \begin{pmatrix} W \\ a_r^T \end{pmatrix}.$$

If constraint r has been added to a working set that retains the most recently deleted constraint s, then the working-set matrix at the next iteration is

$$\bar{W}_+ = \begin{pmatrix} \bar{W} \\ a_s^T \end{pmatrix} = \begin{pmatrix} W \\ a_r^T \\ a_s^T \end{pmatrix}.$$

Consider a non-optimal subspace minimizer x for W_+ at which all the constraints in W are active. It follows that

$$g = W^T\lambda_E + \lambda_s a_s, \qquad \text{with} \qquad \lambda_s < 0, \tag{4.5}$$

where λ_E is the subset of λ corresponding to W. The following lemma shows that $x + \alpha p$ is a subspace minimizer with respect to a set of shifted constraints with working-set matrix W_+.

Lemma 4.1.2. *Let g and W_+ be the gradient and working set at a non-optimal subspace minimizer x. Assume that W_+ can be written as in (4.4), with a_s satisfying (4.5). Let $\bar{x} = x + \alpha p$, where p has been defined as in Lemma 4.1.1. Assume that the constraint r is to be added to W_+ at $\bar{x}$ to give $\bar{W}_+$. Then*

1. *$\bar{g}$, the gradient at $\bar{x}$, is a linear combination of W^T and a_s.*
2. *There exists a vector $\bar{\lambda}_E$ and scalar $\bar{\lambda}_s$ such that*

$$\bar{g} = \bar{W}^T\bar{\lambda}_E + \bar{\lambda}_s a_s \qquad \textit{with} \qquad \bar{\lambda}_s < 0. \tag{4.6}$$

This lemma implies that $\bar{x}$ can be regarded as the solution of a problem in which the most recently deleted constraint is shifted to pass through the point $\bar{x}$. In a sequence of consecutive steps at which a constraint is added to the working set, each iterate is a subspace minimizer of an appropriately shifted problem. That is, $\bar{x}$ solves the equality constrained problem

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \quad & c^Tx + \tfrac{1}{2}x^THx \\ \text{subject to} \quad & Wx = b_W, \quad a_s^Tx = a_s^T\bar{x}. \end{aligned} \tag{4.7}$$

The optimal multipliers for this problem are $\bar{\lambda}_E$ and $\bar{\lambda}_s$ of Lemma 4.1.2. If the step $\alpha = 1$ is taken to reach the point $\bar{x}$, then this multiplier becomes zero. Therefore, $\bar{x}$ is a subspace minimizer with respect to W and W_+.

By induction, in a sequence of consecutive steps at which a constraint is added to the working set, each iterate is a subspace minimizer of an appropriately shifted problem. We call a point x an intermediate iterate if it is not a subspace stationary point with respect to W. These intermediate iterates occur after a constraint is deleted at a subspace minimizer but before the next subspace minimizer is reached, and each iterate is associated with the most recently deleted constraint s. The reduced Hessian at each of these iterates is positive definite, because adding a constraint in an inertia-controlling method cannot produce a stationary point where the reduced Hessian is not positive definite. If a constraint is added, then it must hold that $\alpha < 1$. The sequence of intermediate iterates ends when a step $\alpha = 1$ is taken, and a new sequence of intermediate iterates begins. Note that the maximum number of consecutive intermediate iterates is n, the number of steps from an unconstrained point to a vertex minimizer.

Some useful results concerning a sequence of intermediate iterates are given below.

Lemma 4.1.3. *In a sequence of intermediate iterates, g, λ_E, and λ_s satisfy (4.5) .*

The following lemma ensures that adding a constraint in an inertia-controlling method cannot produce a stationary point where the reduced Hessian is not positive definite.

Lemma 4.1.4. *Let x be an iterate in a sequence of consecutive intermediate iterates. Let $\bar{x} = x + \alpha p$ be the next iterate, where the step $\alpha < 1$ has just been taken to reach the constraint with normal a_r, which is added to W to form $\bar{W}$. Then*

1. *If $\bar{x}$ is a stationary point with respect to $\bar{W}$, then a_r is linearly dependent on W_+ and the reduced Hessian $\bar{Z}^T H \bar{Z}$ is positive definite.*

2. *If a_r^T is linearly dependent on W_+, then $\bar{x}$ is a minimizer with respect to $\bar{W}$.*

4.2 Updating the required vectors

The inertia-controlling algorithm requires the solution of three types of KKT-based systems in order to compute and update p and λ:

System 0

$$\begin{pmatrix} H & W_0^T \\ W_0 & 0 \end{pmatrix} \begin{pmatrix} p_0 \\ -\lambda_0 \end{pmatrix} = - \begin{pmatrix} g_0 \\ 0 \end{pmatrix} \tag{4.8}$$

System 1

$$\begin{pmatrix} H & W_+^T \\ W_+ & 0 \end{pmatrix} \begin{pmatrix} u \\ \mu \end{pmatrix} = \begin{pmatrix} 0 \\ e_s \end{pmatrix} \tag{4.9}$$

System 2

$$\begin{pmatrix} H & W_+^T \\ W_+ & 0 \end{pmatrix} \begin{pmatrix} z \\ \eta \end{pmatrix} = \begin{pmatrix} a_r \\ 0 \end{pmatrix}, \tag{4.10}$$

where H is the $n \times n$ Hessian matrix, W_0 is the initial working-set matrix, g_0 is the gradient at the initial point, e_s is the s^{th} row of the $m \times m$ identity matrix, W_+ is the working-set matrix (and includes the normal of the most recently deleted constraint), and a_r is the normal of the constraint to be added.

System 0 is solved only once at the initial point x_0. System 1 is solved at a subspace minimizer in order to compute the vectors u and μ that are used to obtain p and λ. System 2 is solved when a variable r is added to the working set. The vectors z and η are used to update u, μ, and λ to reflect the addition of a new row to the working-set matrix.

This section describes how u, μ, and λ are updated when r is added to the working set or s is deleted.

Lemma 4.2.1. *(Move to a new iterate) Suppose that x, p, u, μ are the current vectors of the inertia-controlling method. Let $\bar{x} = x + \alpha p$, where p has been defined according to Lemma 4.1.1. The solution $\bar{\lambda}_+ = \begin{pmatrix} \lambda_E \\ \lambda_s \end{pmatrix}$ of*

$$\bar{W}_+^T \bar{\lambda}_+ = \bar{g} = g(\bar{x}) = g + \alpha H p$$

is

$$\bar{\lambda}_E = \lambda_E - \alpha(a_s^T p)\mu_E, \tag{4.11}$$

where

$$\bar{\lambda}_s = \begin{cases} (1-\alpha)\lambda_s & \text{if } \mu_s < 0 \\ \lambda_s - \alpha\mu_s & \text{if } \mu_s = 0. \end{cases} \tag{4.12}$$

Lemma 4.2.2. *(Constraint addition, independent case) Let x, u, z, η be the current vectors of the inertia-controlling method. Assume that the constraint a_r is to be added to the working set at x, where W and a_r are linearly independent. Let $\rho = a_r^T u / a_r^T z$. Then the vectors $\bar{u}$ and $\bar{\mu}$ defined by*

$$\bar{u} = u - \rho z, \qquad \bar{\mu} = \begin{pmatrix} \mu - \rho\eta \\ \rho \end{pmatrix} \tag{4.13}$$

satisfy

$$\begin{pmatrix} H & \bar{W}_+^T \\ \bar{W}_+ & 0 \end{pmatrix} \begin{pmatrix} \bar{u} \\ \bar{\mu} \end{pmatrix} = \begin{pmatrix} 0 \\ e_s \end{pmatrix}, \tag{4.14}$$

where $\bar{W}_+$ is the working-set matrix that includes the new constraint a_r and the most recently deleted constraint a_s. That is, $\bar{u}$ and $\bar{\mu}$ satisfy System 1 (4.9) for $\bar{W}_+$.

If $Z^T H Z$ is positive definite and the working set contains the most recently deleted constraint s, then s can be deleted from the working set. The following lemma can be applied in two situations: when a constraint is deleted from the working set at a minimizer and the reduced Hessian remains positive definite after the deletion, and at an intermediate iterate after a constraint has been added that makes $Z^T H Z$ positive definite.

Lemma 4.2.3. *(Constraint deletion) Suppose that x is an iterate of an inertia-controlling method, the reduced Hessian at the current point $Z^T H Z$ is positive definite, and the working set does not contain the most recently deleted constraint s. Then the vector $\bar{\lambda}$ defined by*

$$\bar{\lambda} = \lambda_{_E} + \rho\mu_{_E}, \qquad \textit{where} \qquad \rho = -\lambda_s/\mu_s, \tag{4.15}$$

satisfies $g = \bar{W}^T\bar{\lambda}$.

Note that u and μ are no longer needed to define the search direction after s has been removed from the working set.

The last possibility occurs when the constraint a_r to be added to the working set is linearly dependent on W^T. In this case, we have $z = 0$ in System 2 (4.10). From Lemma (4.1.4), the point reached must be a minimizer with respect to the working set with constraint r, and constraint s is no longer needed. The following lemma describes the updates that simultaneously remove s and add r to the working set. After an application of these updates, the algorithm either terminates or deletes a constraint (that cannot be a_r).

Lemma 4.2.4. *(Constraint addition, dependent case) Suppose that x is an intermediate iterate. Assume that the constraint a_r to be added to the working set at x is linearly dependent on W, and let $\bar{W}$ be the working-set matrix with the additional row a_r. Define $\omega = \lambda_s/\eta_s$. The vector $\bar{\lambda} = \begin{pmatrix} \bar{\lambda}_E \\ \bar{\lambda}_r \end{pmatrix}$ with*

$$\bar{\lambda}_E = \lambda_E - \omega\eta_E \qquad \text{and} \qquad \bar{\lambda}_r = \omega \tag{4.16}$$

satisfies $\bar{W}^T\bar{\lambda} = g$.

The QP algorithm derived from the inertia-controlling method is given in Algorithm 1.

4.3 Linear systems for the standard form

In this section we derive the necessary linear systems (4.8)–(4.10) for a problem in standard form (1.1). Recall that because the working-set matrix will always include the linear constraints $Ax = b$, we can find a permutation P such that the working-set matrix can be written

$$WP = \begin{pmatrix} A_{FR} & A_{FX} \\ 0 & I_{FX} \end{pmatrix}.$$

Without loss of generality, we assume throughout this section that the permutation $P = I$. The subscript "$_E$" corresponds to the general constraints $Ax = b$, and the subscript "$_B$" corresponds to the active bounds in $l \le x \le u$. The index r is to be added to the working set, and the index s is to be removed.

For the standard form (1.1) we can solve Systems 0, 1, and 2 in terms of the initial reduced KKT matrix

$$\begin{pmatrix} H_{FR} & A_{FR} \\ A_{FR} & \end{pmatrix}.$$

System 0 can be written as

$$\begin{pmatrix} H_{FR} & H_{RX} & A_{FR}^T & 0 \\ H_{RX}^T & H_{FX} & A_{FX}^T & I_{FX} \\ A_{FR} & A_{FX} & 0 & 0 \\ 0 & I_{FX} & 0 & 0 \end{pmatrix} \begin{pmatrix} p_{FR} \\ p_{FX} \\ -\lambda_E \\ -\lambda_B \end{pmatrix} = - \begin{pmatrix} g_{FR} \\ g_{FX} \\ 0 \\ 0 \end{pmatrix},$$

where H_{RX} is the matrix of mixed off-diagonal terms of the Hessian, λ_E are the Lagrange

Algorithm 1 Convex QP algorithm

Solve System 0 for p_0 and λ_0
$x \leftarrow x_0 + p_0$, $\lambda \leftarrow \lambda_0$
subspace_minimizer $\leftarrow$ **true**
loop
  if *subspace_minimizer* **then**
    $\lambda_s \leftarrow \min(\lambda)$
    if *optimal* **then** STOP **end if**
    Solve System 1 for u and μ
    singular_H $\leftarrow (\mu_s = 0)$
  end if
  $p \leftarrow$ **if** *singular_H* **then** u **else** $(\lambda_s/\mu_s)u$ **end if**
  Compute step length α
  if $\alpha = \infty$ **then** *unbounded*, STOP **end if**
  $x \leftarrow x + \alpha p$
  $\lambda \leftarrow$ **if** *singular_H* **then** $\lambda - \alpha\mu$ **else** $\lambda - \alpha(\lambda_s/\mu_s)\mu$
  if hit constraint r **then**
    Solve System 2 for z and η
    if $z = 0$ **then**
      $\omega \leftarrow \lambda_s/\eta_s$, $\lambda \leftarrow \lambda - \omega\eta$, $\lambda_s \leftarrow \omega$
    else
      Add r to the working set
      $\rho \leftarrow u_r/z_r$, $u \leftarrow u - \rho z$, $\mu \leftarrow \mu - \rho\eta$, $\mu_s \leftarrow \rho$
    end if
  else
    Remove variable s from the working set
  end if
end loop

multipliers corresponding to the equality constraints and λ_B are the reduced gradients for the fixed variables (the Lagrange multipliers corresponding to the active bounds). This system reduces to

$$\begin{aligned}\begin{pmatrix} H_{FR} & A_{FR}^T \\ A_{FR} & 0 \end{pmatrix} \begin{pmatrix} p_{FR} \\ -\lambda_E \end{pmatrix} &= -\begin{pmatrix} g_{FR} \\ 0 \end{pmatrix} \\ p_{FX} &= 0 \\ \lambda_B &= g_{FX} + H_{RX}^T p_{FR} - A_{FX}^T \lambda_E.\end{aligned}$$

System 1 can be written

$$\begin{pmatrix} H_{FR} & H_{RX} & (h_s)_{FR} & A_{FR}^T & 0 & 0 \\ H_{RX}^T & H_{FX} & (h_s)_{FX} & A_{FR}^T & I_{FX} & 0 \\ (h_s)_{FR}^T & (h_s)_{FX} & h_{ss} & a_s^T & 0 & 1 \\ A_{FR} & A_{FX} & a_s & 0 & 0 & 0 \\ 0 & I_{FX} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} u_{FR} \\ u_{FX} \\ u_s \\ \mu_E \\ \mu_B \\ \mu_s \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \delta \end{pmatrix},$$

where

$$\delta = \begin{cases} 1 & \text{if } x \text{ moves off its lower bound} \\ -1 & \text{if } x \text{ moves off its upper bound.} \end{cases} \tag{4.17}$$

System 1 reduces to

$$\begin{aligned}\begin{pmatrix} H_{FR} & A_{FR}^T \\ A_{FR} & 0 \end{pmatrix} \begin{pmatrix} u_{FR} \\ \mu_E \end{pmatrix} &= -\begin{pmatrix} (h_s)_{FR} \\ a_s \end{pmatrix} \\ \begin{pmatrix} u_{FX} \\ u_s \end{pmatrix} &= \begin{pmatrix} 0 \\ \delta \end{pmatrix} \\ \begin{pmatrix} \mu_B \\ \mu_s \end{pmatrix} &= -\begin{pmatrix} H_{RX}^T \\ (h_s)_{FX}^T \end{pmatrix} u_{FR} - \begin{pmatrix} (h_s)_{FX} \\ h_s s \end{pmatrix} - \begin{pmatrix} A_{FX}^T \\ a_s^T \end{pmatrix} \mu_E.\end{aligned}$$

System 2 can be written

$$\begin{pmatrix} H_{FR} & H_{RX} & (h_s)_{FR} & A_{FR}^T & 0 & 0 \\ H_{RX}^T & H_{FX} & (h_s)_{FX} & A_{FX}^T & I_{FX} & 0 \\ (h_s)_{FR}^T & (h_s)_{FX} & h_{ss} & a_s^T & 0 & 1 \\ A_{FR} & A_{FX} & a_s & 0 & 0 & 0 \\ 0 & I_{FX} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} z_{FR} \\ z_{FX} \\ z_s \\ \eta_E \\ \eta_B \\ \eta_s \end{pmatrix} = \begin{pmatrix} \delta e_r \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix},$$

where δ is defined in (4.17). This reduces to

$$\begin{aligned} \begin{pmatrix} H_{FR} & A_{FR}^T \\ A_{FR} & 0 \end{pmatrix} \begin{pmatrix} z_{FR} \\ \eta_E \end{pmatrix} &= -\begin{pmatrix} \delta e_r \\ 0 \end{pmatrix} \\ \begin{pmatrix} z_{FX} \\ z_s \end{pmatrix} &= \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\ \begin{pmatrix} \eta_B \\ \eta_s \end{pmatrix} &= -\begin{pmatrix} H_{RX}^T \\ (h_s)_{FR}^T \end{pmatrix} z_{FR} - \begin{pmatrix} A_{FX}^T \\ a_s^T \end{pmatrix} \eta_E. \end{aligned}$$

4.4 Finding an initial feasible point

The active-set method assumes a starting point that satisfies the linear constraints $Ax = b$, and it maintains $Ax = b$ throughout. A point x is said to be infeasible if the bounds $l \le x \le u$ are not satisfied to within some tolerance. The constant δ is defined to be the feasibility tolerance with respect to the bound constraints. The constraint $l_j \le x_j \le u_j$ is considered to be

$$\begin{aligned} \text{satisfied} \quad &\text{if} \quad l_j - \delta \le x_j \le u_j + \delta, \\ \text{violated} \quad &\text{if} \quad x_j \le l_j - \delta \\ &\text{or} \quad u_j + \delta \le x_j. \end{aligned}$$

One approach to obtaining feasibility with respect to the bound constraints is to use a two-phase approach. The first phase (the feasibility phase) finds a feasible point by minimizing the sum of the infeasibilities. The second phase (the QP phase) minimizes the quadratic objective function in the feasible region.

A single-phase approach can also be used, which simultaneously tries to minimize the objective function while improving feasibility. In this procedure, a composite objective function is used, in which penalty terms are introduced into the original objective for variables that violate their bounds:

$$\phi(x, \rho) = c^T x + \tfrac{1}{2} x^T H x + \omega \sum_{j=1}^{n} \max(l_j - x_j, 0) + \omega \sum_{j=1}^{n} \max(x_j - u_j, 0),$$

where the weight ω on the infeasibilities should be large enough to reduce the penalty term. A strategy is used to determine whether a feasible point exists for the original problem, or whether ω has become sufficiently large. The value of ω is increased by some factor if an infeasible minimizer of the composite objective function is found and ω is not already at some large limiting value.

4.5 Step length

The step length α is chosen such that the point $x + \alpha p$ reaches one of the bounds on x. If x is currently infeasible, the number of infeasibilities is not permitted to increase. If the number of infeasibilities remains the same, then the sum of the infeasibilities will decrease. If the number of infeasibilities decreases, then the sum of the infeasibilities will usually also decrease.

4.6 Anti-cycling

As long as the step length α is nonzero, progress is being made. Zero-length steps are possible if the constraints are degenerate; that is, if there are constraints that are active but not in the working set. Theoretically, a sequence of zero-length steps could continue indefinitely.

The EXPAND [39] anti-cycling procedure is used, in which a working feasibility tolerance is maintained and increases slightly and consistently over a sequence of iterations. When free variables become fixed, they are permitted to violate their bounds slightly in order to maintain $Ax = b$. Few (if any) such variables are expected to be infeasible at a final solution. Infeasible fixed variables are reset on their bounds after a fixed number of iterations or at a potential optimal point, and the EXPAND procedure restarts. This procedure has been

used successfully within solvers such as MINOS [64] and SQOPT [36].

4.7 Optimality conditions

An optimality tolerance $\sigma > 0$ is used to determine whether x is a constrained stationary point. For x_j on its lower bound, the reduced gradient z_j of the objective (1.3) is considered optimal if $z_j \leq \sigma$. For x_j on its upper bound, z_j is optimal if $z_j \geq -\sigma$. If z_j is non-optimal, then the objective function can be reduced by removing x_j from the working set.

If optimal multipliers occur when the sum of the infeasibilities is nonzero and the weight on the infeasibilities ω has reached a maximum value, then the problem is declared to be infeasible.

Chapter 5

Recovering from singular KKT systems

"Basis repair" [35] is an important practical matter for implementations of the simplex method and reduced-gradient method. The analogous concept for QPBLU is KKT Repair.

In both cases, although linear constraints $Ax = b$, $\ell \leq x \leq u$ tend to be written in that form for convenience, in practice they are typically implemented in the more general form

$$Ax - s = 0, \qquad \ell \leq \begin{pmatrix} x \\ s \end{pmatrix} \leq u,$$

where s is a vector of slack variables whose bounds allow two-sided constraints $b_1 \leq Ax \leq b_2$ to be accommodated easily. An important benefit of including a full set of slacks is that the resulting constraint matrix $(A \ -I)$ has full row rank. Even if a slack variable s_i has bounds $\ell_{n+i} = u_{n+i} = 0$ (so that it could apparently be eliminated from the problem), the unit vector associate with such a "variable" remains vital for both Basis Repair and KKT Repair.

For convenience we revert to the form $Ax = b$, $\ell \leq x \leq u$ while bearing in mind that a full set of unit vectors exists among the columns of "A".

5.1 Basis repair and rank-revealing LU

The simplex method and the reduced-gradient method are two classic active-set methods based on the *null-space approach* for linearly constrained optimization. They maintain a nonsingular basis matrix B obtained from the columns of the constraint matrix A (really $(A \ -I)$ as just explained). Implementations such as in CPLEX [56] MINOS [64], and SQOPT [36] depend critically on the current B being reasonably well-conditioned relative to the floating-point precision in use (typically 15 digits).

Sparse LU factors of B are needed to implement each simplex or reduced-gradient iteration. Whenever the factors are computed directly (rather than updated), they are the key tool for judging whether B is nearly singular, and for determining which columns (if any) should be replaced by columns of $-I$ to obtain a better-conditioned basis before iterations proceed.

5.1.1 Threshold Partial Pivoting (TPP)

A conventional sparse LU factorization based on threshold partial pivoting (TPP) sometimes gives warning of near-singularity. In LUSOL [38], MA48 [27], and SuperLU [21] it takes the form

$$PBQ = LU, \qquad L_{jj} = 1, \qquad |L_{ij}| \leq \tau, \tag{5.1}$$

where P and Q are permutations chosen to promote sparsity in L and U while maintaining a bound on the subdiagonals of L. The *factor tolerance* τ is a specified parameter in the range $1 \leq \tau \leq 10$, say. It is chosen to balance sparsity and stability, with values nearer 1 giving improved stability. Except for pathological cases, this TPP strategy ensures that L is reasonably well-conditioned. It follows that the other factor U must reflect the condition of B. We believe that all sparse LU factorization packages should keep either L or U well-conditioned, and the documentation should tell us which case it is. The other factor can then be examined for signs of singularity, such as small diagonals and/or large off-diagonals. For example, if

$$U = \begin{pmatrix} 2 & & 1 & \\ & 1 & & 11 \\ & & 10^{-10} & \\ & & & 3 \end{pmatrix},$$

it seems clear that rank$(U) \approx 3$ and the third column of B should be replaced by the third column of $-I$ (assuming $P = Q = I$). If the "11" entry were 10^{11}, say, the last column of B would also need to be replaced.

On the other hand, for the example

$$B = \begin{pmatrix} \delta & 1 & & \\ & \delta & 1 & 1 \\ & & \delta & 1 \\ & & & \delta \end{pmatrix}, \qquad \delta = 10^{-10}, \qquad P = Q = I, \qquad L = I, \qquad U = B,$$

the "small diagonals" heuristic would regard U as having four singularities (rank$(U) \approx 0$!) and replace all columns of B, when in fact rank$(U) \approx 3$ and it would suffice to replace just the first column of B, using the last column of $-I$.

5.1.2 Threshold Rook Pivoting (TRP)

To guard against the preceding situation, LUSOL incorporates a relatively recent option known as threshold rook pivoting (TRP). Conceptually, the diagonals of U are factored out and we regard the sparse LU factors as

$$PBQ = LDU, \qquad L_{jj} = U_{jj} = 1, \qquad |L_{ij}| \leq \tau, \qquad |U_{ij}| \leq \tau. \tag{5.2}$$

Now, both L and U are likely to well-conditioned if τ is in the range $1 \leq \tau \leq 10$, say. It follows that D should reflect the condition of B.

We see that the TRP strategy would not allow $L = I$, $D = \delta I$, $DU = B$ in the above example because the off-diagonal elements of such a U would be very large. Instead, the first column of B will be interchanged with the last column (Q permutes columns 1 and 4), P will be the identity, and the factors become

$$PBQ = \begin{pmatrix} 1 & & & \delta \\ \delta & 1 & 1 & \\ & \delta & 1 & \\ & & \delta & \end{pmatrix} = LDU, \qquad L = \begin{pmatrix} 1 & & & \\ \delta & 1 & & \\ & \delta & 1 & \\ & & \delta & 1 \end{pmatrix}, \qquad DU = \begin{pmatrix} 1 & & & \delta \\ & 1 & -\delta & -\delta^2 \\ & & 1 & \delta^3 \\ & & & -\delta^4 \end{pmatrix}.$$

Even for δ as large as 10^{-3}, LUSOL will correctly identify rank$(U) \approx 3$. This ability has significantly improved the performance of SNOPT problems; see [35]. Experience suggests

that TRP requires $1 \leq \tau \leq 4$ and sometimes $\tau \leq 2$ to achieve reliable rank-revealing properties. The resulting LU factors are generally more dense and expensive to compute than with the conventional TPP strategy, but they remain practical even with $\tau = 1.1$.

The success of this LUSOL option led to the realization that the symmetric solvers MA27 [26] and MA57 [24] are (already) doing the symmetric equivalent of threshold rook pivoting, and also led to a new TRP option in the unsymmetric solver MA48.

5.2 Singular KKT systems

From the preceding discussion of rook pivoting, it is clear that MA57's symmetric indefinite $\mathrm{LDL^T}$ factorization should have rank-revealing properties for KKT systems if the factor tolerance is in the range $2 \leq \tau \leq 4$. (For reasons connected with the 1×1 and 2×2 block structure of the D factor, MA57 cannot allow $\tau < 2$.)

We examine the particular structure that QPBLU encounters within the context of SNOPT. When the current KKT system is factorized directly, it is always of the form

$$K = \begin{pmatrix} D_1 & & A_1^T \\ & & A_2^T \\ A_1 & A_2 & \end{pmatrix}, \tag{5.3}$$

where $D_1 \succ 0$ is part of a diagonal quasi-Newton approximation to the Hessian of the Lagrangian for the nonlinear optimization problem being solved. SNOPT enables a user to separate linear and nonlinear variables. In (5.3), A_1 and D_1 are associated with free nonlinear variables, and A_2 belongs to free linear variables (including slacks). In other words, $A_{FR}P = (A_1 \ A_2)$ for some permutation P.

Theorem 5.2.1. *The matrix K in (5.3) with $D_1 \succ 0$ is nonsingular if and only if the following conditions hold: $(A_1 \ A_2)$ has full row rank, and A_2 has full column rank.*

Proof. First assume that $(A_1 \ A_2)$ has full row rank and A_2 has full column rank. Since D is nonsingular, it is clear from the definition of K that K has full row rank. Because K is square, this implies that it is also nonsingular.

Next assume that K is nonsingular. Then

$$Kx = \begin{pmatrix} D_1 & & A_1^T \\ & & A_2^T \\ A_1 & A_2 & \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

implies $x = 0$. We have

$$\begin{aligned} D_1 x_1 + A_1^T x_3 &= 0 \\ A_1 x_1 + A_2 x_2 &= 0 \\ A_2^T x_3 &= 0. \end{aligned}$$

Using the nonsingularity of D_1, we derive the following expressions for x_1, x_2, x_3:

$$x_1 = -D_1^{-1} A_1^T x_3 \tag{5.4}$$

$$A_2 x_2 = A_1 D_1^{-1} A_1^T x_3 \tag{5.5}$$

$$A_2^T x_3 = 0. \tag{5.6}$$

In particular, using (5.4) and (5.6) we see that if

$$\begin{pmatrix} A_1^T \\ A_2^T \end{pmatrix} x_3 = 0 \qquad \text{with} \qquad x_3 \neq 0,$$

then we can find $x \neq 0$ such that $Kx = 0$. Therefore, in order for K to be nonsingular $\begin{pmatrix} A_1^T \\ A_2^T \end{pmatrix}$ must have full column rank, and thus $(A_1 \ A_2)$ must have full row rank.

Since we require $x_3 = 0$, equation (5.5) simplifies to

$$A_2 x_2 = 0.$$

Again, if A_2 does not have full column rank, then we can find $x \neq 0$ such that $Kx = 0$, a contradiction. Therefore, in order for K to be nonsingular it must hold that $(A_1 \ A_2)$ has full row rank and A_2 has full column rank.

□

As mentioned, H may not involve all variables. In particular, the QP subproblems in

SNOPT are quadratic in only the first n_H variables.

Theorem 5.2.2. *If the Hessian H in (1.1) involves $n_H < n$ variables, then the dimensions of the KKT matrix in (5.3) for any nonsingular working set cannot exceed $2m + n_H$.*

5.2.1 KKT repair with MA57

If MA57 declares that K is singular with τ reasonably close to 2, its rank-revealing properties may be trusted. Theorem 5.2.1 implies that only two repair actions are necessary. Singularities associated with the columns of A_2 can be resolved by deleting those columns of A_2. Other singularities must be associated with the rows of $(A_1\ A_2)$. The associated slack variable should be made free to render the row independent of the other rows (thus generating a new row and column of K).

5.2.2 KKT repair with LUSOL

If MA57 is not available, or its $\mathrm{LDL}^{\mathrm{T}}$ factors become too dense when τ is small enough to give rank-revealing properties, we may apply LUSOL with TRP to the much smaller matrices $(A_1\ A_2)$ and A_2 in turn. TRP factors of $(A_1\ A_2)$ should point to dependent rows whose slack variable should be made free. Then TRP factors of A_2 may reveal dependent columns that should be deleted.

5.3 Regularization

Rather few sparse factorization packages have rank-revealing properties (we know of LUSOL, MA48 [27], and MA57), and for reliability they must be used with a stricter-than-normal control on the size of the nonzeros in the factors. An alternative that eliminates the need for rank-revealing properties is to regularize the original problem.

For convex QP problems, it is reasonable to add a small quadratic term $\frac{1}{2}\delta x^T x$ to the objective to ensure that H is strictly positive definite (e.g., $\delta = 10^{-6}$ or 10^{-8} for well-scaled data). This has the desirable effect of making the optimal x unique.

To achieve uniqueness of the dual variables, we may work with a further perturbation to the problem. For some $\delta > 0$ and $\mu > 0$, the QP problem with primal and dual regularization

is

$$\begin{array}{ll} \underset{x,y}{\text{minimize}} & \phi(x,y) = c^T x + \frac{1}{2} x^T H x + \frac{1}{2}\delta\|x\|^2 + \frac{1}{2}\mu\|y\|^2 \\ \text{subject to} & Ax + \mu y = b, \\ & \ell \le x \le u, \end{array}$$

where y is a new primal variable that is *unconstrained*. It happens that y is also the dual variable for the constraints $Ax+\mu y = b$. The gradient and Hessian of the quadratic objective ϕ are

$$g_\phi = \begin{pmatrix} c + (H + \delta I)x \\ \mu y \end{pmatrix} \qquad \text{and} \qquad H_\phi = \begin{pmatrix} H + \delta I & \\ & \mu I \end{pmatrix}.$$

As before, the active-set strategy partitions x into free and fixed variables: $Ax = A_{FR}x_{FR} + A_{FX}x_{FX}$. To improve the current values of (x, y), a search direction $p = (\Delta x, \Delta y)$ can be computed from the quadratic program

$$\begin{array}{ll} \underset{p}{\text{minimize}} & g_\phi^T p + \frac{1}{2} p^T H_\phi p \\ \text{subject to} & \begin{pmatrix} A & \mu I \end{pmatrix} p = 0, \end{array}$$

where the active-set strategy uses $\Delta x_{FX} = 0$. This becomes

$$\begin{array}{ll} \underset{\Delta x,\, \Delta y}{\text{minimize}} & g_{FR}^T \Delta x_{FR} + \mu y^T \Delta y + \frac{1}{2}\delta \Delta x_{FR}^T (H_{FR} + \delta I)\Delta x_{FR} + \frac{1}{2}\mu\|\Delta y\|^2 \\ \text{subject to} & A_{FR}\Delta x_{FR} + \mu \Delta y = 0, \end{array}$$

where $g_{FR} = c_{FR} + (H_{FR} + \delta I)x_{FR}$. The solution is given by

$$\begin{pmatrix} H_{FR} + \delta I & A_{FR}^T \\ A_{FR} & -\mu I \end{pmatrix} \begin{pmatrix} \Delta x_{FR} \\ -\Delta y \end{pmatrix} = - \begin{pmatrix} g_{FR} - A_{FR}^T y \\ 0 \end{pmatrix}. \tag{5.7}$$

With H positive semidefinite and δ and μ both positive, we see that this KKT-like system is nonsingular.

Note that $\mu > 0$ ensures that the perturbed QP problem is always feasible. In reality the formulation is equivalent to a classical quadratic penalty method for the constraints $Ax = b$. However, treating x and y as primal variables may be somewhat unconventional, and we see that little change should be needed to the block-LU implementation. The approach was

not used for the numerical results in Chapter 7, but we expect that it would eliminate most of the difficulties with singularity.

Chapter 6

Fortran implementation

QPBLU is a new Fortran 95 package for minimizing a quadratic function with linear equality and bound constraints. QPBLU implements an active-set method with an ℓ_1 penalty function to achieve feasibility. It uses block-LU updates of an initial KKT system to represent active-set changes in addition to low-rank Hessian updates. It is intended for convex quadratic programming problems in which the linear constraint matrix is sparse, a good estimate of the optimal active set or solution is available in advance, and many degrees of freedom are expected at the solution. A key feature of QPBLU is its ability to incorporate a variety of third-party sparse linear system solvers to handle the KKT systems.

6.1 Overview

The major components of QPBLU are grouped into modules, which are collections of data, type definitions, and procedure definitions. Abstract data types are defined for the representations of the Hessian, the KKT matrix, and the components that define the KKT matrix. These data types and all operations that act upon them are organized into separate modules, allowing the components that make up the solver to be more easily understood, shared, maintained, modified, or extended.

Since QPBLU has been designed with modularity in mind, most of its components can be reused in other applications. It is also possible to modify the implementation of a data type or module routine without altering its calling sequence. That is, while the implementation of a routine may change, the routines calling it need not. In particular, this program structure makes it possible to take advantage of new developments in linear algebra software,

especially for parallel environments, without having to re-write substantial amounts of the code.

The modules that comprise the QPBLU package are as follows:

- `qpbluSolver` is the top-level module that defines the QP solver and provides the user-callable routines.
- `qpbluKKT` defines the data type representing the current KKT matrix. Routines are provided to update the current KKT matrix as columns enter and leave the working set and as the Hessian is updated.
- `blockmod` defines the data type that stores representations of the block factors C, Y, and Z of an augmented matrix K. Routines are provided to update these block factors as additional rows and columns are appended to K.
- `lumod` defines the data type representing the factors of a dense matrix C. Routines are provided to update the dense factorization when rows and columns are added to or deleted from C, and additional routines are provided to solve systems involving C or C^T.
- `spMatmod` defines the sparse matrix type to store the block factors Y, Z. Routines are provided to multiply with each matrix and its transpose and to update it as new columns are added or existing ones deleted.
- `qpbluHessian` defines the data type representing an initial (sparse) Hessian and any rank-one updates to that initial matrix.
- `qpbluLinAlg` is a collection of linear algebra routines.
- `qpbluConstants` defines constants used throughout the QPBLU package.
- `qpbluOptions` defines the data type to store the QPBLU solver options. Routines are provided to read the options data from a file, to set the options to their default values, and to display the options data.
- `luInterface` defines the data type to store the factors of a symmetric matrix K. Routines are provided to factor and solve with K using a variety of third-party linear system solvers. This module depends on additional modules to interface with each third-party package.
- `qpbluPrint` defines routines to display or write solver output.

The following auxiliary modules are also available:

- `qpbluScale` provides routines to scale and unscale the problem data.
- `qpbluSetup` provides routines to read a matrix in Rutherford-Boeing format or vectors

in Matrix Market format [20]. Routines are also provided to obtain an initial point that satisfies the linear constraints $Ax = b$.

The dependency structure of the modules that make up QPBLU is depicted in Figure 6.1. Detailed descriptions of some of the QPBLU modules are provided in the sections below; in particular, we provide details of the modules that may be reused in other applications.

6.2 QPBLU

QPBLU is callable by way of three routines:

1. `qpbluBegin` initializes the solver options and print streams.

2. `qpbluSolver` calls the quadratic programming solver. Storage for the representation of the Hessian and KKT matrix and for any additional workspace is allocated in this routine.

3. `qpbluEnd` cleans up workspace used by the solver. In particular, this deallocates the storage of the current Hessian and KKT matrix.

Three options are available in QPBLU to initialize the iterations: cold starts, warm starts, or hot starts. The starting type is determined by the user based on amount of information known in advance about the solution or active set of the quadratic program.

Cold starts are intended for problems in which there is no advance knowledge of an initial set of free or fixed variables or initial point. QPBLU attempts to find an initial set of free variables by using LUSOL with Threshold Partial Pivoting with tolerance 1.1 to compute an LU factorization of A^T. If this procedure is able to identify a set of m linearly independent columns of A (corresponding to m linearly independent rows of A^T), then these columns are defined to be the initial set of free variables; if not, then QPBLU is unable to proceed. Once an initial set of free variables is identified, it is used to compute the initial starting point x_0. Let the subscript "$_{FR}$" denote the indices of the columns of A corresponding to the initial free variables, and "$_{FX}$" the indices corresponding to the variables initially held fixed. We begin by setting

$$x_{FX} = \min\{\max(l_{FX}, 0), \min(u_{FX}, 0)\}$$

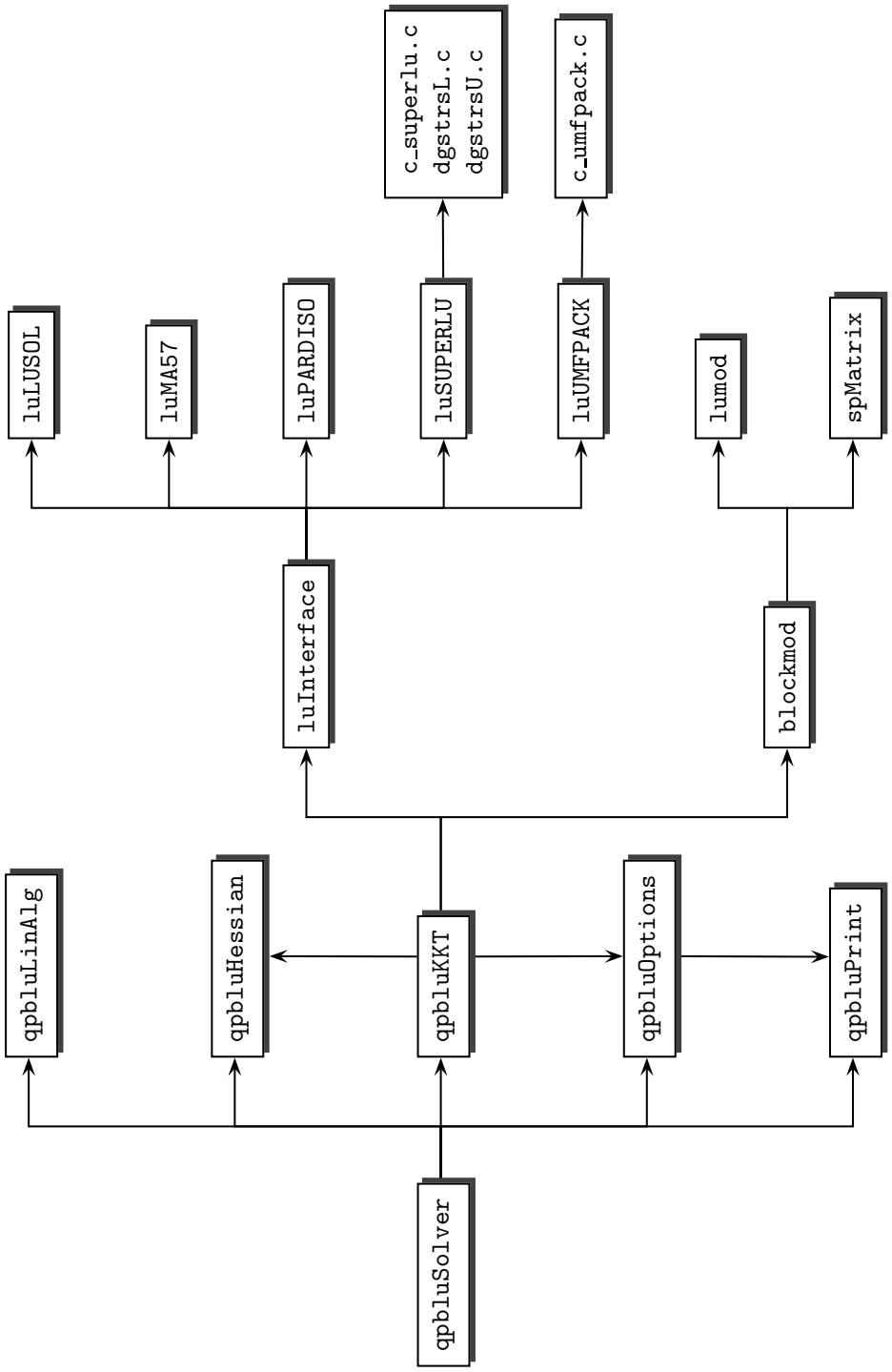

Figure 6.1: Organization of modules and routines for QPBLU. All modules depend on the module qpbluConstants (not shown) to define real and integer precision. The additional routines needed for solvers written in C/C++ are included.

and then solving

$$A_{FR}x_{FR} = b - A_{FX}x_{FX}.$$

Given the permutation P such that $AP = \begin{pmatrix} A_{FR} & A_{FX} \end{pmatrix}$, we define the initial starting point to be

$$x_0 = P \begin{pmatrix} x_{FR} \\ x_{FX} \end{pmatrix}.$$

This point satisfies the linear equality constraints, though some bounds on x_{FR} may be violated.

Warm starts are intended for problems in which there is a good estimate of the active set and initial point. In particular, it is ideal for the case where a linearly independent set of m free variables and a point x_0 satisfying the linear constraints is known.

Hot starts differ from warm starts in that additional information regarding the current KKT matrix is known. Hot starts are intended to be called following either a successful (optimal) exit from QPBLU or an exit in which the maximum number of iterations has been reached. Enough information about the current working set, KKT matrix, and Hessian is retained after an exit from `qpbluSolver` to make it possible to re-start the iterations from the last computed point. Hot starts are also intended for the case in which the Hessian has been modified using a rank-one update of the form

$$H_{k+1} = (I + v_{k+1}u_{k+1}^T)H_k(I + u_{k+1}v_{k+1}^T) \tag{6.1}$$

in between successive calls to the QP solver.

6.3 Constants

The module `qpbluConstants` defines global constants used throughout the package. In particular, this module defines the real and integer precision used throughout the program units. The Fortran intrinsics `selected_real_kind` and `selected_int_kind` are used to provide a portable method specifying the actual precision or exponent range needed. The default precision for real values is double (`dp=selected_real_kind(15)`), but this can easily be altered throughout the QPBLU package by changing the value of `dp` in this module.

6.4 Hessian

The module `qpbluHessian` defines the representation of the Hessian matrix in QPBLU.

The Hessian is assumed to be symmetric and sparse. The user may input it in one of four forms:

1. Diagonal, in which only the diagonal elements (including any zeros) of the Hessian are input. This format is intended for cases where the Hessian is a diagonal matrix.

2. Upper triangle in compressed sparse column format with ascending row indices. Equivalently, the lower triangle in compressed sparse row format with ascending column indices.

3. Upper triangle in compressed sparse row format with ascending column indices. Equivalently, the lower triangle in compressed sparse column format with ascending row indices.

4. Upper triangle in coordinate format.

For convenience, the Hessian is converted and stored internally in either diagonal or compressed column format if it is not already in one of these two formats. This strategy may require additional storage for the Hessian, but it facilitates formation of the KKT matrix.

This Hessian data type is also designed to handle rank-one updates of the form (6.1). The rank-one update vectors u_k and v_k are assumed to be dense. Additional storage is allocated and deallocated by `qpbluHessian` for the internal representation of these rank-one updates.

`qpbluHessian` also provides routines to multiply with the current Hessian, to extract a column of the initial Hessian, and to obtain the vectors needed to perform a rank-one update of H within the block-LU factorization.

6.5 KKT factorization

The module `qpbluKKT` defines the representation of the KKT matrix in QPBLU and defines routines that update and solve the current KKT system. The current KKT matrix is represented in terms of the block factorization of an augmented initial KKT matrix K_0.

Routines are provided to update the KKT matrix when a column of A enters or leaves the set of free variables, incorporate a rank-one update to the Hessian, and to solve the current KKT system. Details of the implementation of the block factors follow.

6.5.1 Overview

The Fortran module `blockMod` contains routines for maintaining the block factors Y, Z, and C of a block factorization of an unsymmetric augmented matrix

$$\begin{pmatrix} K_0 & V \\ W^T & D \end{pmatrix} = \begin{pmatrix} L_0 & \\ Z^T & I \end{pmatrix} \begin{pmatrix} U_0 & Y \\ & C \end{pmatrix},$$

where $K_0 = L_0 U_0$. Solutions to systems involving L_0 and U_0 are obtained from an external source, such as an existing factorization, and are used as input to update the block factors. A routine for solving the system

$$\begin{pmatrix} K_0 & V \\ W^T & D \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

is not included, because operations with the block factors L_0 and U_0 are not assumed to be known to this module. However, routines for multiplying with Y and Z and for solving with C are provided to facilitate such solves from the calling routine.

Routines are available in this module to update the block factorization in the following cases:

1. A column $\begin{pmatrix} v \\ d \end{pmatrix}$ is appended to K_0 or the augmented matrix.
2. A row $\begin{pmatrix} w^T & d^T \end{pmatrix}$ is appended to K_0 or the augmented matrix.
3. The j^{th} column of $\begin{pmatrix} V \\ D \end{pmatrix}$ is deleted from the augmented matrix.
4. The j^{th} row of $\begin{pmatrix} W^T & D \end{pmatrix}$ is deleted from the augmented matrix.

This module is used to update the factorization of the augmented KKT matrix.

6.5.2 Block factors L_0, U_0

The factorization of the initial KKT matrix $K_0 = L_0 U_0$ is obtained using a third-party linear system solver such as MA57. The module `luInterface` provides the top-level interface to these third-party solvers. Details of this module and the currently incorporated third-party solvers are provided in section 6.6.

6.5.3 Block factors Y, Z

The module `spMatmod` defines the sparse matrix type `spMat` used to represent the block factors Y and Z. Routines are provided to multiply with these matrices and to update them as new columns are added or existing ones deleted. `spMat` uses three arrays to store the matrices in compressed sparse column format. An additional indexing array is also used to indicate which of the stored columns of the matrix are currently in use.

New matrix columns are added by appending elements to the existing matrix representation and updating the list of columns in use. Columns are "deleted" by removing the indices of these columns from the list of columns in use. Only a subset of the total columns stored may actually be in use. This method for column deletion requires more storage overall than a method that shifts array elements whenever a column is deleted, but computational time is saved by not these copying array values.

6.5.4 Block factor C

Because the block factor C is expected to be relatively small and dense, it is efficient to maintain a dense LU factorization of C. The module `lumod` uses routines based on Gaussian elimination to maintain a factorization of a dense square matrix of the form $LC = U$ as C gains or loses rows and columns. The matrix factor L is well-conditioned and is the product of stabilized elementary transformations in

$$\begin{pmatrix} 1 & \\ \mu & 1 \end{pmatrix} \quad \text{or} \quad \begin{pmatrix} & 1 \\ 1 & \mu \end{pmatrix},$$

where $|\mu| \leq 1$ (see [71] for details). Both the square matrix L and upper triangular matrix U are stored row-wise by `lumod` in 1D arrays.

Routines are available in this module to update C in the following cases:

1. A row is appended to the $n \times n$ or $(n-1) \times n$ matrix C.

2. A column is appended to the $n \times n$ or $n \times (n-1)$ matrix C.

3. The j^{th} row of the $n \times n$ or $(n+1) \times n$ matrix C is deleted.

4. The j^{th} column of the $n \times n$ or $n \times (n+1)$ matrix C is deleted.

Row and column additions (as well as row and column deletions) are assumed to come in pairs, so that the matrix C remains square. New rows and columns are assumed to be appended to C. After a column deletion, the remaining columns of C are shifted left. After a row deletion, rows are shifted up after a row deletion. Row or column replacement may be achieved by deleting the row or column to be replaced and appending a new row or column to the matrix.

Column deletions in C result in a deletion of the corresponding column in U, and the remaining columns are shifted to the left. Row deletions in C result in a deletion of the corresponding row in U, and the remaining rows are shifted up. On average, half the elements of U will be moved, but the copy operation is cheap because U is relatively small. Within the QPBLU context, typically less than half the block-LU updates require deletion.

The matrix C is allowed to be a 0×0 matrix. The factorization of an existing $n \times n$ matrix C is formed by starting with a 0×0 matrix and then adding rows and columns from C to build up the factorization. The LU factors are well-defined even if C is singular, in which case U is also singular.

6.5.5 Refactorization

As the dimensions of the block factors grow, the work needed to solve the current KKT system also increases. It is necessary to refactorize the KKT matrix from scratch. The KKT matrix is refactorized if one of the following situations is reached:

1. The maximum number of iterations with the factorization of the initial KKT matrix has been reached.

2. The maximum number of columns that the block factors Y and Z are able to store has been reached.

3. The maximum number of nonzero elements that the block factors Y and Z are able to store has been reached.

4. The maximum dimension of the Schur complement matrix C has been reached.

5. The Schur complement may be ill-conditioned. Because the matrix L is always well-conditioned, the condition of C is estimated using the absolute value of the ratio of the largest and smallest diagonal elements of the matrix U.

For large systems, forming the factorization of the initial KKT matrix is much more expensive than updating or solving with the block factors. It is preferable that the KKT matrix be refactorized upon reaching the maximum number of iterations, rather than because of a resource limitation. These parameters determining whether the KKT matrix is refactorized may be redefined by the user to suit the needs of a particular problem or machine. For instance, for very large problems it may be worthwhile to increase the upper limit of the dimensions of the Schur complement from its default value of 50 and to increase the upper limit of iterations with the KKT matrix.

6.6 Interface to third-party solvers

Direct methods for solving linear systems of equations are ideal for the block-LU method because they provide an efficient way of solving multiple KKT systems with the same initial matrix and different right-hand side vectors. Once the factorization of an initial KKT system is performed, this factorization can be reused in subsequent iterations to solve the current KKT system as part of the block-LU method.

It is highly desirable that a variety of sparse linear system solvers be available within QPBLU to compute the factorization $K_0 = L_0 U_0$. For instance, specialized LU solvers may be used for structured problems or for novel machine architectures. Much progress has been made in the development of efficient and robust sparse linear system solvers for distributed or shared memory parallel machines [58, 22, 1, 2, 55]. Also, some solvers are only available commercially [16], or to academic researchers [24, 67], or as prototypes, since many of these solvers are still under active development.

Ideally, to take advantage of symmetry of the KKT matrix, K_0 should be represented by a symmetric indefinite factorization of the form $K_0 = LDL^T$, where L is lower triangular and D is block diagonal, with 1×1 or 2×2 blocks. General unsymmetric LU factorization routines for sparse matrices may also be used. Although these solvers do not take advantage of symmetry, they are generally well-developed, robust, and readily available.

The module `luInterface` provides the interface to third-party linear solvers and defines the data type to store data from each factorization, such as the matrix factors, options,

permutations, and the matrix K_0 itself if iterative refinement is specified.

`luInterface` depends on additional Fortran modules, each specific to a particular third-party solver. These solver-specific modules are based on a pre-defined template. This generic interface model facilitates the incorporation of additional solvers. New methods may be incorporated by creating a new module for the linear solver based on the standard template and editing the module `luInterface`.

Access to all options or parameters of the linear solvers may not be available using `luInterface`. Most parameters are set to each solver's default values, usually with a call to the solver's own initialization routine. QPBLU defines its own default values for some parameters, such as numerical thresholds, which may differ from the default values used by the linear solver itself. The ability to modify some common solver options is also available to the user, though no error checking is done within the QPBLU modules to determine validity of the values.

An important characteristic of black-box LU solvers is the ability to solve systems of equations with each factor L and U. For solvers where a separate solve with the factors is unavailable, we take "L_0" $= I$ and "U_0" $= L_0 U_0$. Care must be taken in the selection of a particular solver and in its parameters so that the factors of K_0 do not fill in significantly. The sparsity of L_0 and U_0 is reflected in the block factors Y and Z.

The matrix K_0 to be factored is assumed to be symmetric, though not necessarily of the KKT form, with the upper triangle of the matrix stored in compressed sparse row format. Columns are to be provided in ascending order with all (possibly zero) diagonals explicitly represented. If an unsymmetric solver is used, the representation of the whole matrix is required. In this case, the matrix is converted to an appropriate unsymmetric format (where both upper and lower triangular parts are expressed) within the Fortran module specific to that unsymmetric solver.

Note that since the matrix to be factorized is symmetric, the compressed sparse row representation of the upper triangle is equivalent to the compressed sparse column representation of its lower triangle. Likewise, the compressed sparse row representation of the full matrix is equivalent to its compressed sparse column representation.

The use of this particular storage format for `luInterface` facilitates conversion to other matrix storage formats. Routines are available to convert a matrix from symmetric compressed sparse row format to unsymmetric compressed sparse row format, and to convert a matrix from symmetric compressed sparse row format into symmetric and unsymmetric

coordinate forms. For third-party solvers that require matrices in other formats, software packages such as SPARSKIT [66] can be used to convert between many different sparse matrix formats.

6.6.1 Third-party solvers

A great number of solvers for the solution of large sparse linear systems of equations have been developed [16, 24, 1, 67, 2, 21, 22, 58, 19, 55]. Summaries of most of the available direct solvers for sparse linear systems are maintained by Davis [18] and by Li [57]. The choice of a particular solver depends upon several factors, including availability, performance, and machine hardware.

A comparison of sparse symmetric solvers in the HSL mathematical software library was performed by Gould and Scott [51, 50] in 2003. Four of these solvers (MA27 [26], MA47 [28], MA57, MA67 [25]) were capable of handling symmetric indefinite systems, and two of them (MA47, MA67) were designed for solving symmetric indefinite systems of the form

$$\begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix}. \tag{6.2}$$

They found that MA57 was usually the fastest and most reliable HSL package, even outperforming MA47 and MA67 on augmented systems of the form (6.2). Robustness of the solvers for large indefinite systems was still a concern, because a few of the test problems were not solved by any of the solvers.

In an extended study, Gould et al. [47, 46] made a comparison of several sparse direct solvers for symmetric systems. This study considered serial solvers or serial versions of parallel solvers run under their default settings on a single computing platform. Only three of the solvers studied had a success rate of 90% or better on indefinite problems: MA57, PARDISO, and UMFPACK. Of these three, only MA57 and PARDISO are sparse symmetric indefinite solvers. It is important to note that this study found that reliably solving large, sparse indefinite problems remains a challenge.

While symmetric indefinite solvers are able to take advantage of symmetry, their lack of robustness is a matter of concern. Although iterative refinement is often available as part of a solver's solution phase, it could add considerably to the time required in the block-LU method because the initial factorization is used repeatedly. In addition to requiring

Package	Version	Authors	Reference
LUSOL	Mar 2006	P. E. Gill, W. Murray, M. A. Saunders, M. H. Wright	[38]
MA57	2.2.1	I. S. Duff, HSL	[24]
PARDISO	3.2	O. Schenk, K. Gärtner	[69, 67, 68]
SuperLU	3.0	J. W. Demmel, J. R. Gilbert, X. S. Li	[21]
UMFPACK	5.1	T. A. Davis	[19]

Table 6.1: Summary of sparse linear system solvers used in this study.

Package	Availability	Website
LUSOL	FOS	`http://www.stanford.edu/group/SOL/software.html`
MA57	C-FA	`http://www.cse.scitech.ac.uk/nag/hsl/`
PARDISO	C-FA	`http://www.pardiso-project.org/`
SuperLU	FL	`http://crd.lbl.gov/~xiaoye/SuperLU/`
UMFPACK	FL	`http://www.cise.ufl.edu/research/sparse/umfpack/`

Table 6.2: Obtaining the sparse linear system solvers used in this study. FOS = Free, Open Source; C-FA = Commercial, Free to Academics; FL = Free, Licensed.

additional storage for the initial KKT matrix, iterative refinement prevents the use of the matrix factors separately when solving linear systems. This may lead to an increased number of total nonzero elements stored in the block factors in the block-LU algorithm (see section 7.3). While general-purpose LU solvers are not able to exploit symmetry fully, they typically tend to be very robust, and for this reason are included in this study.

Currently, three unsymmetric solvers (LUSOL, SuperLU, UMFPACK) and two symmetric solvers (MA57, PARDISO) may used within QPBLU to factorize and solve with the KKT matrix. A summary of these solvers is given in Table 6.1, and their availability is described in Table 6.2. All of the third-party solvers currently incorporated into QPBLU are freely available to academic researchers. Features of each solver are listed in Table 6.3, and a brief description of each software package is provided in the subsections below.

LUSOL

LUSOL [38] is a package for solving sparse linear systems $Ax = b$. It allows A to be square or rectangular, and it can update its LU factors when rows or columns of A are added, deleted, or replaced. For its direct LU factorization, a Markowitz strategy is used for

Package	Method	Type	Parallel	Separate L & U
LUSOL	Markowitz	Unsym	No	Yes
MA57	Multifrontal	Sym	No	Yes
PARDISO	Left-right looking	Sym, Sym-pat	SM	No
SuperLU	Left-looking	Unsym	No	Yes
UMFPACK	Multifrontal	Unsym	No	Yes

Table 6.3: Summary of features of the sparse linear solvers used in this study. Sym = symmetric, Sym-pat = symmetric nonzero pattern with unsymmetric values, Unsym = unsymmetric, SM = shared memory.

LUSOL parameter	Value	Description
luparm(1)	6	Output stream
luparm(2)	-1	No output
luparm(6)	0	Threshold Partial Pivoting
parmlu(1)	2.0	max $\|L_{ij}\|$ during factor

Table 6.4: QPBLU default options for LUSOL.

suggesting sparse pivots, and a choice of threshold partial pivoting, threshold rook pivoting, or threshold complete pivoting is provided for balancing stability and sparsity. All pivoting options control the condition of L. A rank-revealing factorization may be obtained by using either the rook or complete pivoting strategies with a rather strict threshold pivoting parameter. This will produce a factorization in which the condition and rank of U reflect the condition and rank of A. LUSOL is used as the basis factorization package for such optimization software as lp_solve [11], MINOS, SNOPT, and SQOPT. LUSOL is available as open source FORTRAN 77 or ANSI C libraries.

QPBLU default options for LUSOL are listed in Table 6.4.

MA57

MA57 [24] is a package for the solution of symmetric indefinite systems and is part of the HSL mathematical software library. This package supersedes the earlier HSL package MA27 and incorporates Level 2 and Level 3 BLAS. MA57 uses a multifrontal approach to compute a factorization of the form $A = LDL^T$, where the matrix L is unit lower triangular and D is a block diagonal matrix with blocks of order 1 or 2. To help preserve sparsity during the symbolic factorization phase, either the Approximate Minimum Degree (AMD) algorithm or

MA57 parameter	Value	Description
cntl(1)	0.25	Threshold pivoting tolerance
icntl(1-3)	6	Print streams
icntl(5)	0	No output
icntl(6)	5	Automatic choice of MeTiS or AMD
icntl(15)	1	Scaling on

Table 6.5: QPBLU default options for MA57.

the nested dissection algorithm using MeTiS are used to choose a row and column ordering.

In the 2005 study by Gould et al., MA57 was found to be one of the leading solvers for symmetric indefinite systems. While MA57 was more cautious in the factorization phase than PARDISO, this paid off in a faster solution phase.

MA57 has been incorporated into such optimization software as OOQP [32] and IPOPT [73], and is now part of MATLAB (version 7.5) [61] for sparse indefinite linear systems. MA57 is available commercially and may be licensed free to academic researchers.

QPBLU default options for MA57 are listed in Table 6.5.

PARDISO

PARDISO [69, 67, 68] is a package for the solution of large sparse symmetric and unsymmetric linear systems on shared memory multiprocessors using a combination of left- and right-looking Level 3 BLAS supernode techniques. The default ordering is a modified version of MeTiS, although a minimum degree ordering is also available. PARDISO employs a static pivoting technique that does not alter the ordering suggested by its analyze phase during the numerical factorization. In order to be able to use such a potentially unstable ordering, some pivots may be perturbed during the factorization, and iterative refinement is then required to solve the linear system.

In the study 2005 study by Gould et al., PARDISO was found to be one of the leading solvers for symmetric indefinite systems. This study also found that if pivots were perturbed during the factorization phase, then a solve using PARDISO could be two or three times slower than the comparable MA57 solve because of the refinement steps needed. The static pivoting strategy used by PARDISO results in a faster, less cautious factorization phase, but as a consequence results in a slower solution phase if iterative refinement steps are necessary. On a few problems the faster, less cautious factorization strategy also led to

PARDISO parameter	Value	Description
iparm(2)	2	Pivot ordering – nested dissection from MeTiS
iparm(3)	1	Number of processors
iparm(10)	10^{-8}	Threshold for perturbed pivots
iparm(11) & iparm(13)	0	Scalings and matchings off
MSGLVL	0	No output

Table 6.6: QPBLU default options for PARDISO.

inaccurate solutions that did not converge with iterative refinement.

PARDISO is available commercially and may be licensed free by academic researchers. QPBLU default options for PARDISO are listed in Table 6.6.

SuperLU

SuperLU [21] is a library for the solution of large, sparse general systems of linear equations on high performance machines. It is available in a sequential version for conventional machines (SuperLU) and in parallel versions for shared memory multiprocessors (SuperLU_MT [22]) or distributed memory parallel processors (SuperLU_DIST [58]). All three libraries use variations of Gaussian elimination. The sequential version implements Gaussian elimination with partial pivoting using a left-looking supernodal technique and Level 2 BLAS to optimize performance. SuperLU uses a preordering phase for sparsity that is completely separate from the numerical factorization, allowing the matrix columns to be preordered before the factorization using either the supplied routines or a user-supplied ordering routine.

Separate solves with the matrix factors L and U are not available in the standard SuperLU distribution. Two additional routines `dgstrsL` and `dgstrsU` have been provided by SuperLU developer Xiaoye Li to supplement the SuperLU package. These new routines allow for the solution of systems using L, L^T, U, or U^T.

SuperLU is widely used in research and commercial applications. It has been integrated into such solvers as PETSc [5] and is used commercially in Mathematica [65] and COMSOL Multiphysics [17]. SuperLU is available as open source C code with some licensing restrictions. QPBLU default options for SuperLU are listed in Table 6.7.

SuperLU parameter	Value	Description
options.ColPerm	COLAMD	Column ordering approximate minimum degree ordering
options.DiagPivotThresh	0.5	Threshold pivoting tolerance
options.Equil	YES	Scale rows and columns
options.PrintStat	NO	No output

Table 6.7: QPBLU default options for SuperLU.

UMFPACK parameter	Value	Description
Control [UMFPACK_PIVOT_TOLERANCE]	0.50	Threshold pivot tolerance

Table 6.8: QPBLU default options for UMFPACK.

UMFPACK

UMFPACK [19] is a package for sparse LU factorization of an unsymmetric matrix A, where A can be square, rectangular, singular, nonsingular, real, or complex. Only square systems can be used to solve $Ax = b$ and related systems. UMFPACK utilizes a column pre-ordering technique with right-looking unsymmetric-pattern mulitfrontal numerical factorization.

Though not a symmetric solver, UMFPACK offers a symmetric pivoting strategy for (nearly) symmetrically structured matrices. UMFPACK uses Level 3 BLAS to obtain high performance on a wide range of machines. UMFPACK has been incorporated into MATLAB (versions 6.5+) to factor and solve sparse unsymmetric (or symmetric) systems of equations.

UMFPACK is available as open source ANSI/ISO C code under the GNU Lesser General Public License (versions 3.2-5.1) or the GNU General Public License (versions 5.2+). QPBLU default options for UMFPACK are listed in Table 6.8.

Chapter 7

Computational results

7.1 Problem data

Our QP test problems are derived from linear programming test problems available from the University of Florida Sparse Matrix Collection [18]. These LP problems have been converted to the standard form

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & l \leq x \leq u. \end{aligned}$$

These test problems include the LPnetlib data and the linear programming test problems of Hans Mittelmann [63] and Csaba Mészáros [62].

The LPnetlib data consists of 109 feasible and 29 infeasible LP test problems. Of the feasible problems, 34 have been found to have numerical rank $r < m$, where m is the row dimension of A. Estimation of numerical rank was performed by factorizing the matrices A using LUSOL with a factor tolerance of 1.1 using threshold partial pivoting. A summary of the LPnetlib test problems used in this thesis is given in Table 7.3.

Quadratic programming test problems have been generated from these LP test problems by including a quadratic term $\frac{1}{2}x^T H x$ in the objective. In particular, we use $H = \Delta I$ for some positive scalar Δ. In general, increasing Δ leads to more degrees of freedom at the solution.

7.1.1 Scaling

The Fortran module `qpbluScale` contains routines that may be used if row and column scalings of the linear constraint matrix A are desired. The scaling routine `scaleData` makes several passes through the columns and rows of A, computing the geometric mean of the nonzeros in each (scaled) column or row and using that as the new scale factor. These scalings are applied to A and the rest of the problem data. Optionally, the objective function may also be scaled by $\max(1, \|c\|_\infty)$.

For all test problems used in this chapter, the problem data and objective function have been scaled using `qpbluScale`.

7.2 Computing platform

Numerical results were obtained using a 3.00 GHz Dell Precision 470 workstation with an Intel Xeon processor and 2 GB of RAM. Fortran 77 and Fortran 95 codes were compiled using gfortran 4.1.2 with full optimization. Serial versions of parallel solvers were used, and wherever possible, optimized BLAS routines were provided by GotoBLAS [45]. All CPU times are in seconds and do not include the time required to load the problem data.

7.3 Growth of nonzeros in Y, Z

To emphasize the importance of separate solves with the LU factors of K_0 in maintaining sparsity in both block factors Y and Z, we begin by examining the accumulation of nonzero elements in these matrices when separate solves are available and when they are not. The majority of the linear solvers currently incorporated into QPBLU provide the ability to solve separately with each LU factor of K_0 (see Table 6.3). As mentioned, separate solves with L and U are not available as standard routines within the SuperLU package, but have been derived for QPBLU by modifying the SuperLU solve routine `dgstrs`. To demonstrate the effect that these separate solves have, we consider QPBLU using SuperLU as the linear system solver, both with and without separate L and U solves. For these examples, we solve

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & c^T x + \tfrac{1}{2} x^T H x \\ \text{subject to} \quad & Ax = b \\ & l \le x \le u \end{aligned}$$

using data from the LPnetlib collection with $H = \Delta I$ for nonnegative values of the scalar Δ.

Recall that columns of Y and Z are not explicitly deleted in this implementation of QPBLU. Instead, columns are marked as having been "deleted," so we are examining total number of nonzero elements generated in Y and Z since the last refactorization. Note that the number of nonzero elements stored in the block factors drops to zero upon a refactorization of the KKT matrix.

Figure 7.1 shows the accumulation of nonzero elements using $\Delta = 1$ on the `scsd6` dataset. The constraint matrix A of this test problem has 147 rows, 1350 columns, and $4,316$ nonzero elements. Separate solves with L and U maintain a similar degree of growth in the number of elements in both Y and Z, while non-separate solves maintain sparsity primarily in the block factor Y. The total number of nonzeros stored in both Y and Z is significantly less in the case where separate solves are available.

Figure 7.2 shows a more extreme difference in the total number of nonzeros stored. This example uses the `stocfor2` dataset and $\Delta = 1$. The constraint matrix A of this test problem has $2,157$ rows, 3045 columns, and $9,357$ nonzero elements. In this case it is clear that separate solves with L_0 and U_0 maintain a greater degree of sparsity in both Y and Z, and requires less storage overall.

Figures 7.3 and 7.4 use the `fit2d` dataset with $\Delta = 100$ to demonstrate the growth of nonzero elements in Y and Z as the number of free variables increases at each iteration. The constraint matrix A of this test problem has 25 rows, $10,524$ columns, and $129,042$ nonzero elements. In this case, the initial KKT matrix is bordered at each iteration by rows and columns of the Hessian and constraint matrix. Each factorized KKT matrix is larger in dimension than the one preceding it. As a result, the block factors Y and Z always gain a column, and these additional columns contain an increasing number of nonzero elements when either separate or non-separate solves are used. It is clear in Figure 7.4 that despite the increasing number of nonzeros elements, the total number of nonzeros stored is still less when using separate solves with L and U.

While each third-party linear system solver may generate different LU factors, these results with SuperLU are typical of what happens when separate L and U solves are and are not available. The ability to solve separately with these factors of K_0 is an important feature for linear system solvers used within QPBLU. Whenever possible, separate solves should be utilized. These separate solves help to maintain sparsity in both Y and Z and also help to

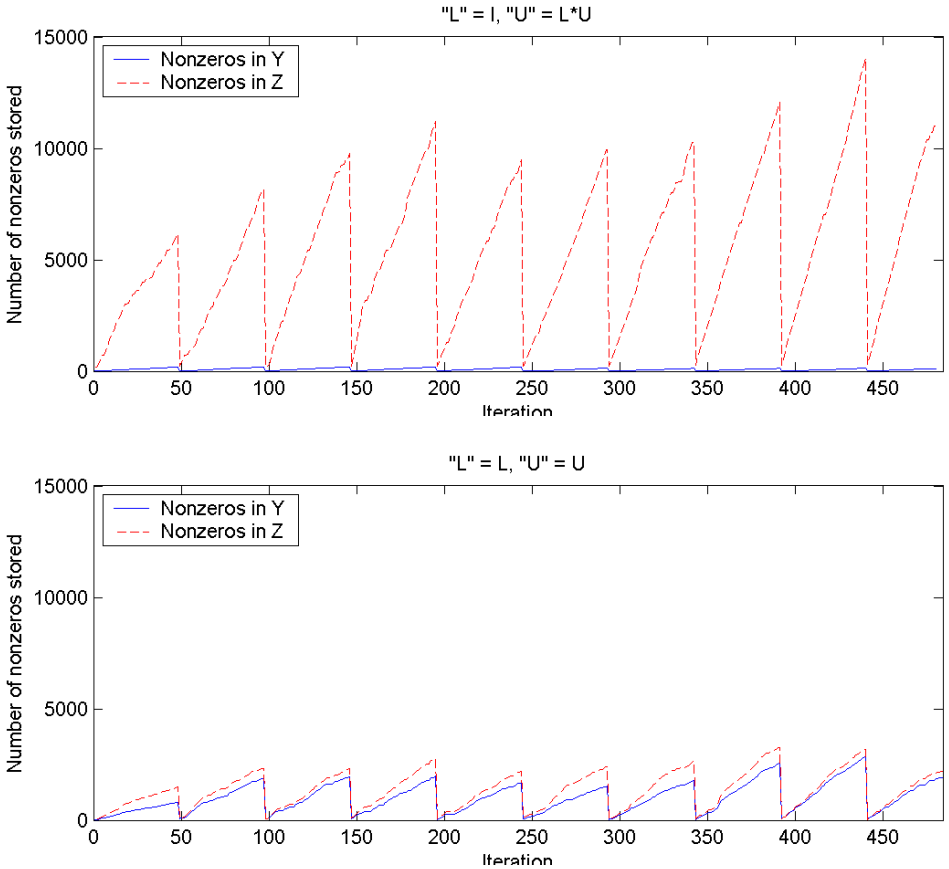

Figure 7.1: Example of the accumulation of nonzero elements stored in the block factors Y and Z when SuperLU is used within QPBLU on the `scsd6` test problem with $H = I$. In the top figure, separate solves with the LU factors of SuperLU are not used. In the bottom figure, separate solves with L and U are utilized.

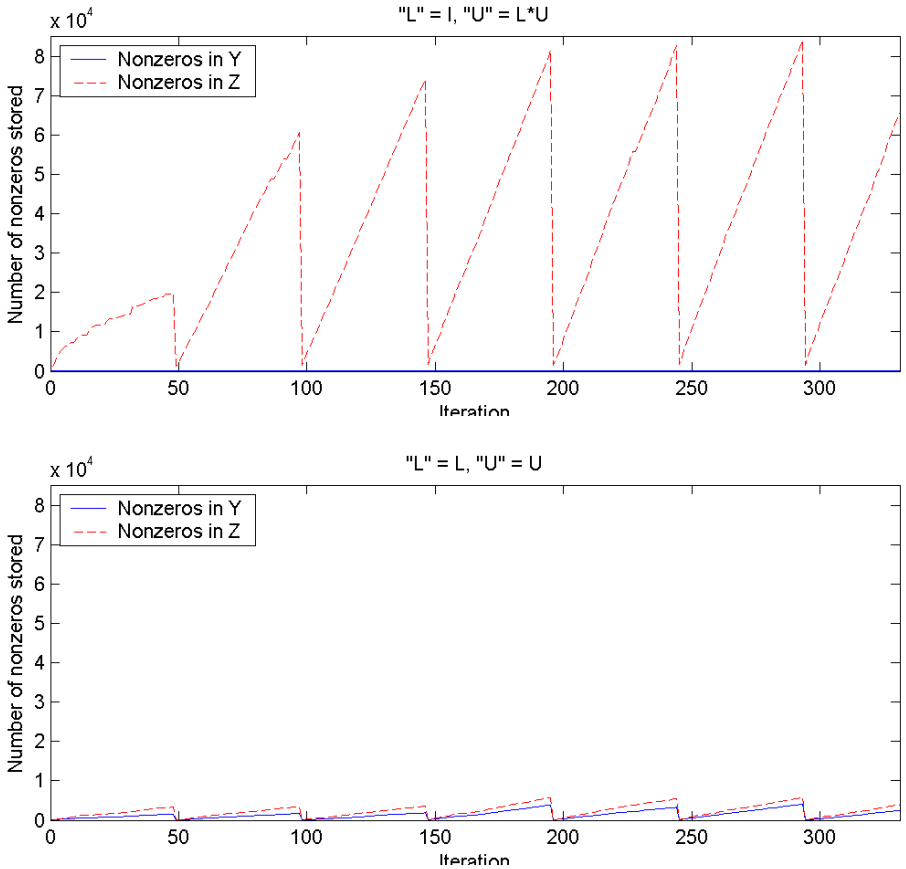

Figure 7.2: Example of the accumulation of nonzero elements stored in the block factors Y and Z when SuperLU is used within QPBLU on the `stocfor2` test problem with $H = I$. In the top figure, separate solves with the LU factors of SuperLU are not used. In the bottom figure, separate solves with L and U are utilized.

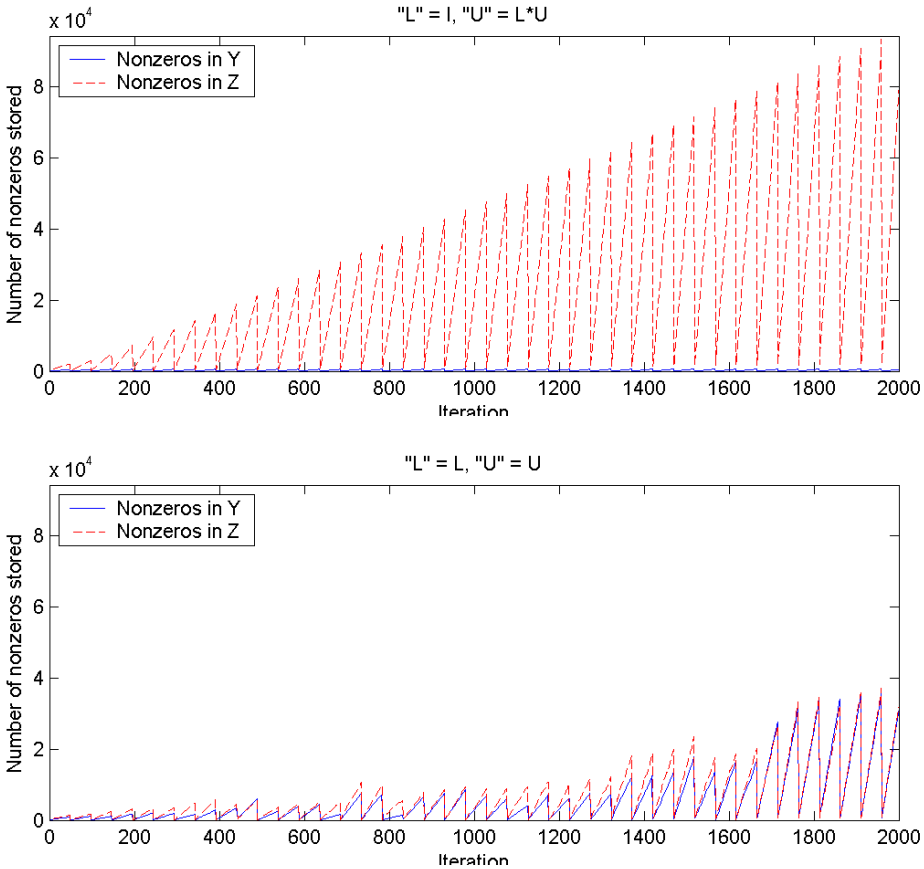

Figure 7.3: Example of the accumulation of nonzero elements stored in the block factors Y and Z when SuperLU is used within QPBLU on the `fit2d` test problem with $H = 100I$. In the top figure, separate solves with the LU factors of SuperLU are not used. In the bottom figure, separate solves with L and U are utilized.

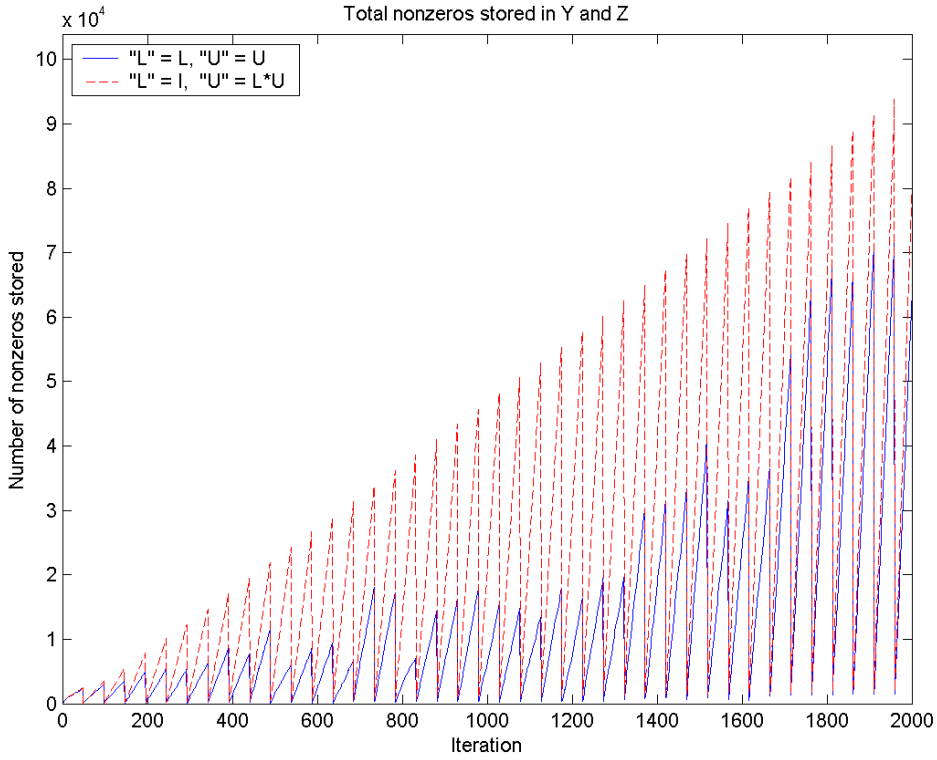

Figure 7.4: Example of the accumulation of the total number of nonzero elements stored in both block factors Y and Z with and without separate solves with the LU factors of SuperLU on the `fit2d` dataset with $H = 100I$.

decrease the amount of storage needed to store both of the block factors. Keeping the block factors as sparse as possible decreases the likelihood that K_0 will need to be refactorized because of a storage limitation.

7.4 Background on performance profiles

Performance profiles provide a means of comparing the performance of several solvers at once on a large set of problems while eliminating possible biases, such as the potential influence of a small number of problems on the benchmarking process. We use this technique to compare the performance of QPBLU using each of the linear solvers LUSOL, MA57, PARDISO, SuperLU, and UMFPACK, and to compare the performance of QPBLU to SNOPT.

This method of comparison was initially developed by Billups, Dirkse, and Ferris [13], who used ratios of runtimes to compare large-scale mixed complementarity problems. Their approach was later expanded by Dolan and Moré [23] to incorporate the use of a performance profile as a tool for evaluating and comparing the performance of optimization software. Their approach is outlined here.

We define n_s to be the number of solvers s and n_p to be the number of problems p. Using computing time as a measure of performance, results are obtained by running a solver s on a set $\mathcal{P}$ of problems. We define

$$t_{p,s} = \text{computing time needed to solve problem } p \text{ with solver } s.$$

The performance ratio is defined to be the ratio of the resource time of a given solver to the best time of all solvers:

$$\rho_{p,s} = \frac{t_{p,s}}{\min\{t_{p,s} : 1 \leq s \leq n_s\}}.$$

A parameter ρ_M is defined such that $\rho_M \geq \rho_{p,s}$ for all solvers s and problems p. The value of a performance ratio $\rho_{p,s} = \rho_M$ if and only if solver s does not solve problem p. This parameter eliminates the need to exclude any test problem from consideration, and gives credit to solvers that successfully solve a problem for which other solvers may fail.

To obtain an overall assessment of the performance of a solver, the cumulative distribution function of the performance ratio

$$P_s(\tau) = \frac{1}{n_p}\text{size}\{p \in \mathcal{P} : \rho_{p,s} \leq \tau\}$$

is considered. $P_s(\tau)$ is the probability that the performance ratio is within a factor of τ of the best possible ratio. That is, it is the fraction of problems that a solver can solve if allowed a maximum resource time of τ times the minimum resource time for each problem. The value $1 - P_s(\tau)$ is the fraction of problems that the solver cannot solve within a factor τ of the best solver.

In particular, $P_s(1)$ is the probability that solver s will win in comparison to the others. This value may be used to compare solvers if only the number of wins is to be considered. As τ becomes large, $P_s(\tau)$ reflects the overall probability that a solver will be successful. Thus, if we are interested in solvers with a high probability of success regardless of the amount of time required, the values of $P_s(\rho_M)$ should be examined.

A log scale of the performance profile may be used to show all activity with $\tau < \rho_M$ while preserving the behavior of τ near 1. That is, we plot

$$\frac{1}{n_p}\text{size}\{p \in \mathcal{P} : \log_2(\rho_{p,s}) \leq \tau\}$$

versus τ.

7.5 Pivot threshold for third-party solvers

Packages for sparse matrix factorizations strive not only for sparsity in the matrix factors, but also for numerical stability and efficiency. Often, a threshold pivoting parameter may be specified to strike the right balance between a sparse yet less stable factorization and a factorization that is more stable yet more costly.

The default pivot tolerance defined by a particular solver may not be ideal for a given type of problem or application. Since the block-LU method uses the factorization $K_0 = L_0U_0$ repeatedly in order to update the block factors and to compute the search direction at each iteration, obtaining a stable factorization is of vital importance. In an effort to evaluate the effect that the choice of pivot tolerance has on the block-LU method, we examine the performance of QPBLU using a given solver as the pivot tolerance varies. CPU time is used as the metric in our performance profiles.

As an example, we apply QPBLU to test problems generated from the LPnetlib collection using $H = I$ with MA57 as the linear system solver. The threshold pivoting value for MA57 is allowed to vary within its range in order to investigate its effect on the outcome of the QP solve. Performance profiles for this test are given in Figure 7.5 and the full set of results

is given in Table 7.4.

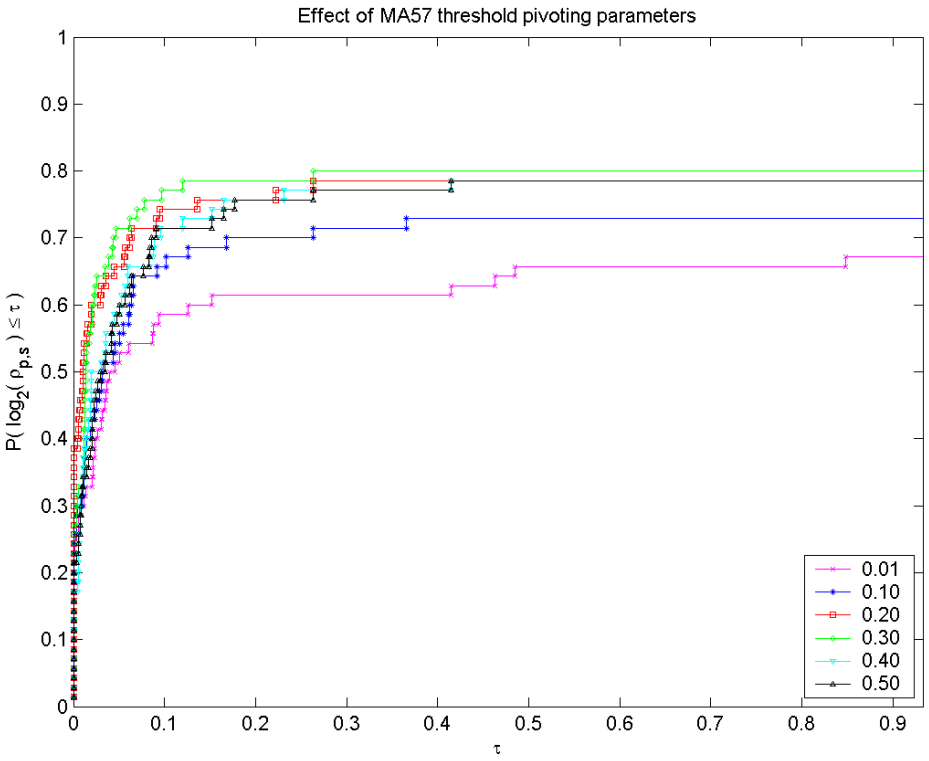

Figure 7.5: Performance profile for QPBLU using MA57 with various values of the threshold pivoting tolerance on the LPnetlib test data with $H = I$.

The MA57 default pivot tolerance of 0.01 yields the least efficient performance for this QP problem on this dataset, with the least overall likelihood of success and often longer computation time. This tolerance allows the elements of L to be as large as 100. Pivot tolerances of 0.20 or greater give the best performance in this example. The results for pivot tolerances of $0.20, 0.30, 0.40, 0.50$ are comparable in terms of overall success. Computation time for these pivot tolerances did not vary greatly, though pivot tolerances of 0.20 and 0.30 yielded somewhat faster solve times.

As we can see from this example, the choice of the pivot tolerance of the linear solver plays an important role in the success and performance of QPBLU. For this reason, QPBLU uses default values of the pivot tolerance that may differ from the defaults of the linear solver itself. QPBLU also provides the user a means of adjusting these default values.

7.6 Comparisons of third-party solvers

QPBLU currently provides an interface to five different third-party solvers for sparse linear systems, and it has been designed to facilitate the inclusion of many more. The user's choice of a particular solver depends on many things, including availability, machine hardware, and most importantly, performance. In this section, we use performance profiles to compare the performance of QPBLU using each of the available linear solvers.

We apply QPBLU with each of the solvers to QP problems generated from the LPnetlib collection with $H = I$. The QPBLU default options for each of the solvers is used (see Tables 6.4 to 6.8 on pages 57–60). The performance profile for this experiment, using CPU times as the metric, is given in Figure 7.6. Full results of this test are provided in Table 7.5.

We see that for this test set, QPBLU-LUSOL generally had the fastest CPU times and was among the most reliable of the solvers used. QPBLU-PARDISO fared the worst in this example, with the slowest computation time and lowest overall probability of success. The additional time required by QPBLU-PARDISO is probably because of the static pivoting approach used by PARDISO and the resulting iterative refinement steps that may have been required for a solve with the initial KKT matrix.

Note that QPBLU was not able to solve all of the test problems, with any of the solvers. For several problems (notably the `pilot` models), the KKT matrix was determined to be ill-conditioned or near-singular by the linear system solvers. In these cases, since KKT repair has not yet been implemented, QPBLU exits and the problem is not solved.

7.7 Comparisons to SQOPT

Using the results of section 7.6, we select the top performing linear system solver (LUSOL) to compare QPBLU with SQOPT, an active-set quadratic programming solver that uses a null-space method suitable for large-scale problems. SQOPT is most efficient when there are few degrees of freedom. When the number of superbasic variables n_S becomes very large ($n_S \geq 2000$, say), it uses a conjugate-gradient method to solve the required linear systems, so the performance may degrade significantly. The comparison of QPBLU-LUSOL is also of interest because SQOPT makes use of LUSOL to maintain the LU factors of its basis matrix.

Both solvers were warm-started from the same point with the same active set. The initial point was obtained by first finding a square, nonsingular basis matrix B from the

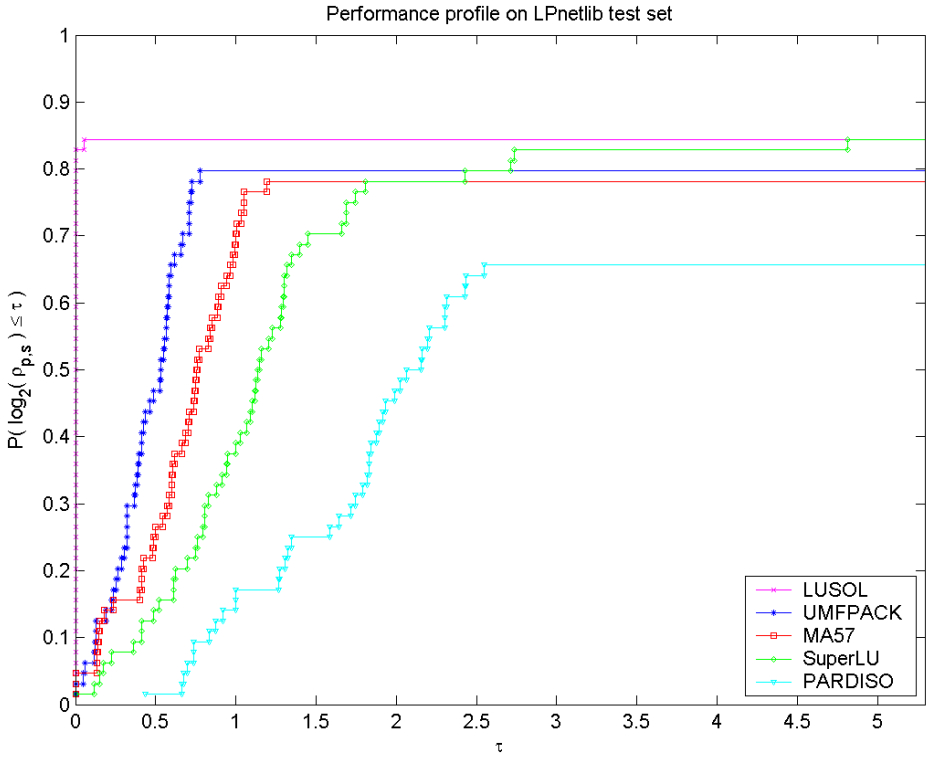

Figure 7.6: Performance profile for QPBLU on QP problems derived from the LPnetlib collection using the Hessian $H = I$. Each of the available linear system solvers LUSOL, MA57, SuperLU, PARDISO, and UMFPACK was used within QPBLU. CPU time is used as the performance metric in this example.

columns of A. Given a suitable permutation matrix P, we can write

$$\begin{aligned} AP &= \begin{pmatrix} B & N \end{pmatrix}, \\ x_0 &= P \begin{pmatrix} x_B \\ x_N \end{pmatrix}, \\ Ax &= Bx_B + Nx_N = b. \end{aligned}$$

The initial starting point was computed by setting x_N to values satisfying its bounds ($l_N \leq x_N \leq u_N$) and then solving $Bx_B = b - Nx_N$. The point x_0 then satisfies the equality constraints $Ax = b$, though some bounds x_B may be violated. The columns of A that form N correspond to the initial set of fixed variables, and the columns of A that form B are the initial set of free variables. LUSOL was used on A^T with a Threshold Partial Pivoting tolerance of 1.1 to generate the basis matrix B and to compute x_0.

Using the `rail507` data from the Mittelmann LP collection and varying the value of Δ, we are able to generate quadratic programs in which the number of superbasic variables at the solution varies from 0 to 6749. The constraint matrix A for this test problem has 507 rows, 63,516 columns, and 40,9856 nonzero elements. The results from this experiment are detailed in Table 7.1 and Figure 7.7.

In this example, SQOPT is much faster than QPBLU-LUSOL when the number of degrees of freedom is less than 2000. As n_S increases, the CPU time required by SQOPT increases significantly, while the increase in CPU time for QPBLU-LUSOL is much more gradual.

As an additional example, we compare SQOPT to QPBLU-LUSOL and QPBLU-MA57 on quadratic programs derived from the `deter` problem set from the Mészáros LP collection using $H = I$. This set consists of nine problems, with the number of degrees of freedom at the solution ranging from 2369 to 8998. The results of this experiment are detailed in Table 7.2 and Figure 7.8.

For this problem set, we see that both QPBLU-LUSOL and QPBLU-MA57 require much less CPU time than SQOPT when the number of superbasic variables is very large.

From this experiment and the previous using the `rail507` data, we can see that there is no predetermined point or value of n_S at which QPBLU is to be preferred over SQOPT. In general, however, SQOPT will become less efficient once $n_S > 2000$ because of its use of conjugate-gradient iterations.

Δ	n_S	SQOPT	QPBLU-LUSOL
0.0	0	98.5	669.4
0.1	634	32.3	268.7
0.2	885	26.3	216.4
0.3	1031	25.5	191.8
0.4	1199	24.1	182.7
0.5	1349	26.3	177.9
0.6	1498	27.3	171.6
0.7	1598	28.7	181.8
0.8	1690	32.3	174.3
0.9	1813	31.5	171.0
1.0	1903	35.3	171.2
2.0	2735	131.0	187.9
3.0	3377	211.7	207.1
4.0	3796	288.2	222.8
5.0	4125	330.8	227.1
6.0	4394	395.7	224.1
7.0	4617	459.3	224.9
8.0	4804	494.5	232.0
9.0	4977	525.7	239.0
10.0	5093	534.9	250.3
20.0	5913	715.7	279.0
30.0	6236	830.4	271.5
40.0	6386	866.6	277.4
50.0	6472	895.9	277.0
60.0	6521	951.0	289.3
70.0	6570	947.5	264.4
80.0	6601	908.1	289.8
90.0	6626	918.9	307.2
100.0	6648	932.3	294.1
110.0	6659	941.5	297.4
120.0	6679	955.0	298.8
130.0	6690	957.8	303.0
140.0	6696	972.9	313.4
150.0	6707	942.5	326.4
160.0	6719	993.7	329.2
170.0	6725	982.2	340.0
180.0	6736	977.1	329.8
190.0	6744	1021.0	333.0
200.0	6749	994.3	324.5

Table 7.1: CPU time in seconds required by QPBLU-LUSOL and by SQOPT for solving the quadratic programming problems generated from the `rail507` problem of the Mittelmann LP collection with $H = \Delta I$. The number of degrees of freedom at the solution is increased by increasing the value of Δ from 0 to 200.

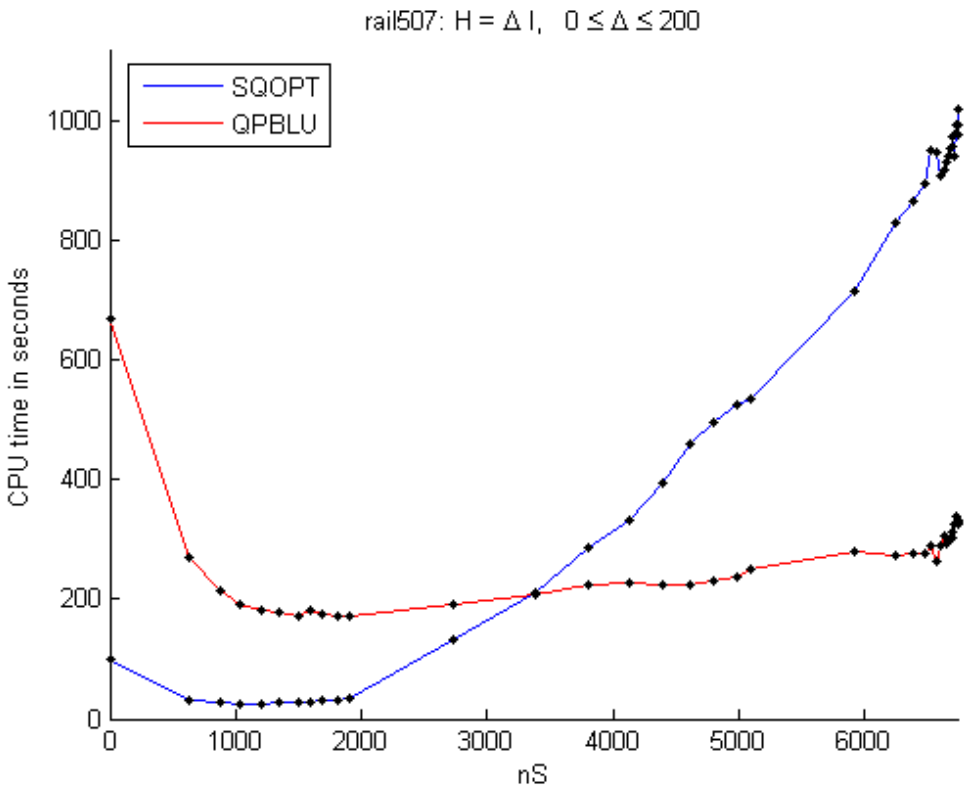

Figure 7.7: CPU time in seconds required by QPBLU-LUSOL and by SQOPT for solving the QP generated from the `rail507` problem of the Mittelmann LP collection with $H = \Delta I$. The number of degrees of freedom at the solution is increased by increasing the value of Δ from 0 to 200.

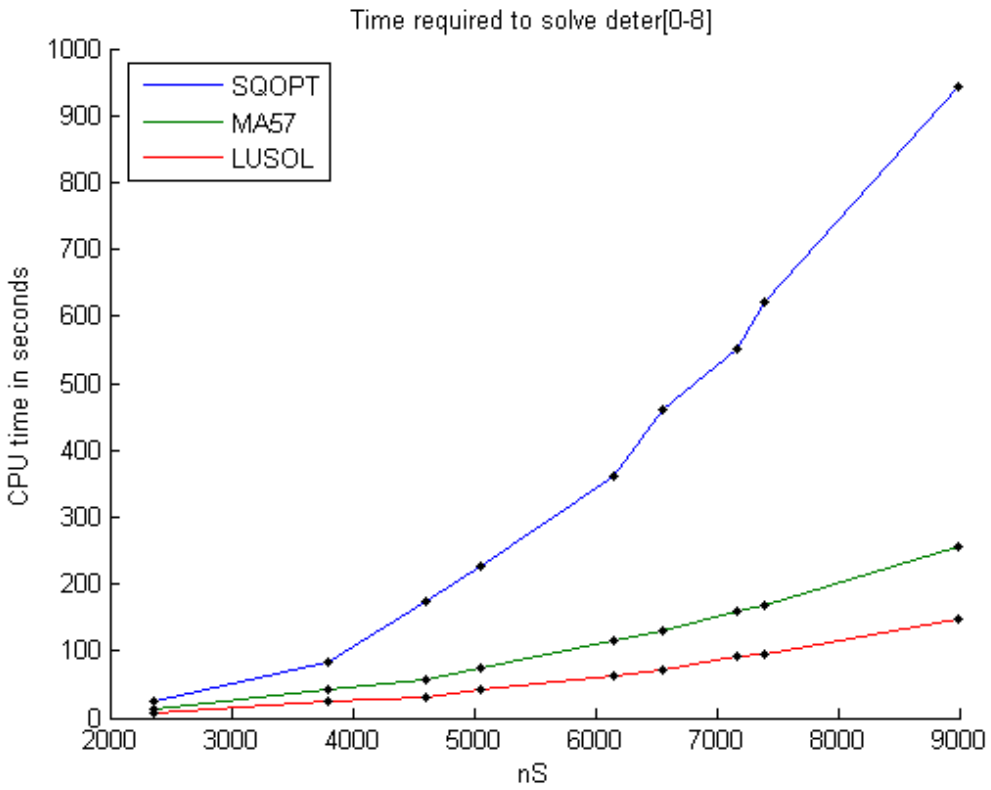

Figure 7.8: CPU time in seconds required by SQOPT, QPBLU-LUSOL, and QPBLU-MA57 on the `deter` dataset from the Mészáros LP collection using $H = I$.

problem	m	n	nnz	n_S	SQOPT	QPBLU-MA57	QPBLU-LUSOL
deter0	1923	5,468	11,173	2369	24.6	14.4	7.8
deter4	3235	9,133	19,231	3792	83.7	43.5	24.9
deter8	3831	10,905	22,299	4600	174.9	57.3	31.7
deter6	4255	12,113	24,771	5053	226.8	74.7	41.5
deter5	5103	14,529	29,715	6159	361.2	115.1	63.2
deter1	5527	15,737	32,187	6562	459.6	130.0	72.1
deter2	6095	17,313	35,731	7177	550.2	158.2	91.5
deter7	6375	18,153	37,131	7398	621.4	169.4	94.8
deter3	7647	21,777	44,547	8998	944.1	256.2	146.7

Table 7.2: CPU times in seconds required by QPBLU-LUSOL, QPBLU-MA57 and SQOPT on the `deter` problem set of the Mészáros LP collection. The data is ordered by the number of superbasic variables at the solution. The number of suberbasics is given by the value n_S. The values of m, n, nnz give the number of rows, number of columns, and number of nonzeros of the constraint matrix A.

Table 7.3:

Name	m	n	nnz
80bau3b	2262	12061	23264
adlittle	56	138	424
afiro	27	51	102
agg	488	615	2862
agg2	516	758	4740
agg3	516	758	4756
bandm	305	472	2494
beaconfd	173	295	3408
blend	74	114	522
bnl2	2324	4486	14996
capri	271	482	1896
czprob	929	3562	10708
d2q06c	2171	5831	33081
e226	223	472	2768
etamacro	400	816	2537
fffff800	524	1028	6401
finnis	497	1064	2760
fit1d	24	1049	13427
fit1p	627	1677	9868
fit2d	25	10524	129042
fit2p	3000	13525	50284
ganges	1309	1706	6937
gfrd_pnc	616	1160	2445
grow15	300	645	5620
grow22	440	946	8252
grow7	140	301	2612
israel	174	316	2443
kb2	43	68	313
lotfi	153	366	1136
maros	846	1966	10137
maros_r7	3136	9408	144848
osa_07	1118	25067	144812
osa_14	2337	54797	317097
osa_30	4350	104374	604488
osa_60	10280	243246	1408073
perold	625	1506	6148

Continued on the next page

Table 7.3: (continued)

Name	m	n	nnz
pilot	1441	4860	44375
pilot4	410	1123	5264
pilot87	2030	6680	74949
pilot_ja	940	2267	14977
pilot_we	722	2928	9265
pilotnov	975	2446	13331
recipe	91	204	687
sc105	105	163	340
sc205	205	317	665
sc50a	50	78	160
sc50b	50	78	148
scagr25	471	671	1725
scagr7	129	185	465
scfxm1	330	600	2732
scfxm2	660	1200	5469
scfxm3	990	1800	8206
scrs8	490	1275	3288
scsd1	77	760	2388
scsd6	147	1350	4316
scsd8	397	2750	8584
sctap1	300	660	1872
sctap2	1090	2500	7334
sctap3	1480	3340	9734
share1b	117	253	1179
share2b	96	162	777
stair	356	614	4003
standata	359	1274	3230
standmps	467	1274	3878
stocfor1	117	165	501
stocfor2	2157	3045	9357
stocfor3	16675	23541	76473
woodw	1098	8418	37487

Table 7.3: Summary of the LPnetlib problems used in this thesis. The number of rows, columns, and nonzeros of the constraint matrix A are given by m, n, and nnz.

Table 7.4:

Name	δ					
	0.01	0.10	0.20	0.30	0.40	0.50
80bau3b						
adlittle	3.30E−02	3.30E−02	3.30E−02	3.40E−02	3.70E−02	3.70E−02
afiro	3.00E−03	3.00E−03	3.00E−03	3.00E−03	4.00E−03	4.00E−03
agg	1.21E−01	1.20E−01	1.19E−01	1.20E−01	1.24E−01	1.23E−01
agg2		5.16E−01	5.15E−01	5.19E−01	5.28E−01	5.28E−01
agg3		5.01E−01	5.01E−01	5.05E−01	5.16E−01	5.13E−01
bandm	5.01E−01	5.00E−01	4.94E−01	4.92E−01	4.95E−01	4.95E−01
beaconfd	4.70E−02	4.70E−02	4.60E−02	4.60E−02	5.00E−02	4.80E−02
blend	2.30E−02	2.40E−02	2.40E−02	2.40E−02	2.70E−02	2.60E−02
bnl2						
capri		4.03E−01	4.06E−01	4.08E−01	4.18E−01	4.19E−01
czprob	2.28E+00	2.29E+00	2.27E+00	2.31E+00	2.30E+00	2.30E+00
d2q06c						
e226	3.38E−01	3.34E−01	3.33E−01	3.36E−01	3.47E−01	3.43E−01
etamacro		1.55E+00	1.47E+00	1.38E+00	1.39E+00	1.38E+00
fffff800	1.96E+00		2.00E+00	2.02E+00	2.03E+00	2.03E+00
finnis	1.08E+00	1.07E+00	1.07E+00	1.07E+00	1.08E+00	1.09E+00
fit1d	6.83E−01	6.98E−01	6.72E−01	6.78E−01	6.93E−01	7.13E−01
fit1p	3.14E+00	3.18E+00	3.14E+00	3.17E+00	3.18E+00	3.19E+00
fit2d		8.29E+01	8.11E+01	8.11E+01		
fit2p	2.87E+02	2.89E+02	2.89E+02	2.90E+02	2.91E+02	2.92E+02
ganges			1.79E+00	1.82E+00	1.79E+00	1.80E+00
gfrd_pnc	1.44E+00	1.44E+00	1.39E+00	1.41E+00	1.40E+00	1.41E+00
grow15	7.11E−01	7.33E−01			7.29E−01	7.32E−01
grow22	1.45E+00				1.47E+00	1.48E+00
grow7	1.61E−01	1.61E−01	1.56E−01	1.58E−01	1.56E−01	1.57E−01
israel	2.73E−01	2.69E−01	2.58E−01	2.59E−01	2.58E−01	2.57E−01
kb2	1.00E−02	9.00E−03	9.00E−03	9.00E−03	1.00E−02	1.00E−02
lotfi	1.87E−01	1.95E−01	1.88E−01	1.90E−01	1.89E−01	1.90E−01
maros						
maros_r7						
osa_07	2.10E+01	2.07E+01	2.08E+01	2.10E+01	2.08E+01	2.10E+01
osa_14	1.14E+02	1.14E+02	1.14E+02	1.15E+02	1.14E+02	1.14E+02
osa_30	4.23E+02	4.23E+02	4.25E+02	4.26E+02	4.24E+02	4.22E+02
osa_60	2.54E+03	2.55E+03	2.53E+03	2.54E+03	2.54E+03	2.53E+03

Continued on the next page

Table 7.4: (continued)

Name	δ					
	0.01	0.10	0.20	0.30	0.40	0.50
perold						
pilot						
pilot4		2.88E+00	2.89E+00	3.04E+00		
pilot87						
pilot_ja						
pilot_we						
pilotnov						
recipe	7.00E−03	5.00E−03	5.00E−03	5.00E−03	5.00E−03	5.00E−03
sc105	2.40E−02	2.40E−02	2.30E−02	2.20E−02	2.20E−02	2.30E−02
sc205	7.30E−02	7.20E−02	7.10E−02	7.10E−02	7.10E−02	7.20E−02
sc50a	9.00E−03	6.00E−03	6.00E−03	5.00E−03	5.00E−03	6.00E−03
sc50b	8.00E−03	6.00E−03	7.00E−03	6.00E−03	6.00E−03	6.00E−03
scagr25	8.67E−01	8.75E−01	8.74E−01	8.74E−01	8.77E−01	8.77E−01
scagr7	6.80E−02	6.70E−02	6.40E−02	6.40E−02	6.40E−02	6.40E−02
scfxm1	3.90E−01	3.75E−01	3.74E−01	3.75E−01	3.76E−01	3.75E−01
scfxm2	1.61E+00					
scfxm3				3.38E+00	3.23E+00	3.22E+00
scrs8		8.62E−01	8.74E−01	8.69E−01	9.21E−01	9.18E−01
scsd1	1.53E−01	1.43E−01	1.22E−01	1.11E−01	1.12E−01	1.11E−01
scsd6			4.08E−01	3.82E−01		
scsd8	2.55E+00		2.65E+00	2.77E+00	2.71E+00	2.70E+00
sctap1	3.17E−01	2.97E−01	3.00E−01	2.98E−01	3.01E−01	3.00E−01
sctap2	2.82E+00	2.86E+00	2.87E+00	2.81E+00	2.83E+00	2.83E+00
sctap3	5.11E+00	5.12E+00	5.07E+00	5.12E+00	5.09E+00	5.09E+00
share1b	1.38E−01	1.38E−01	1.39E−01	1.39E−01	1.41E−01	1.42E−01
share2b	4.20E−02	4.40E−02	4.10E−02	4.20E−02	4.20E−02	4.10E−02
stair	9.73E−01	9.71E−01	9.59E−01	9.58E−01	9.49E−01	9.49E−01
standata	2.01E−01	2.02E−01	2.03E−01	1.98E−01	1.99E−01	1.99E−01
standmps	7.48E−01	7.80E−01	7.42E−01	7.83E−01	7.35E−01	7.32E−01
stocfor1	5.00E−03	5.00E−03	5.00E−03	6.00E−03	5.00E−03	5.00E−03
stocfor2	1.23E+00	1.23E+00	1.24E+00	1.23E+00	1.23E+00	1.23E+00
stocfor3	8.20E+01	8.25E+01	8.19E+01	8.21E+01	8.23E+01	8.23E+01
woodw		1.15E+01	1.10E+01	1.13E+01	1.17E+01	1.16E+01

Table 7.4: CPU times for QPBLU-MA57 on the LPnetlib test problems using the Hessian $H = I$. Various values of the threshold pivoting tolerance δ were used within MA57.

Table 7.5:

Name	LUSOL	MA57	SuperLU	PARDISO	UMFPACK
80bau3b	2.760E+01		6.007E+01		
adlittle	2.400E−02	3.500E−02	3.699E−02	5.799E−02	3.200E−02
afiro	3.000E−03	3.999E−03	3.999E−03	4.999E−03	4.000E−03
agg	7.099E−02	1.200E−01	1.490E−01	2.890E−01	9.598E−02
agg2	3.260E−01	5.159E−01	6.649E−01	1.129E+00	
agg3	2.990E−01	5.019E−01	6.619E−01		
bandm	2.750E−01	4.929E−01	6.849E−01	1.022E+00	4.149E−01
beaconfd	2.600E−02	4.699E−02	4.999E−02	6.599E−02	4.099E−02
blend	1.700E−02	2.400E−02	2.600E−02	4.199E−02	2.200E−02
bnl2					
capri	1.980E−01	4.059E−01	4.879E−01		3.230E−01
czprob	1.637E+00	2.281E+00	2.772E+00	6.250E+00	1.789E+00
d2q06c			1.881E+01		
e226	1.960E−01	3.349E−01	3.919E−01	6.959E−01	2.930E−01
etamacro	6.059E−01	1.384E+00	1.912E+00	2.709E+00	1.038E+00
fffff800		1.977E+00			
finnis	5.829E−01	1.075E+00	1.242E+00	2.434E+00	8.049E−01
fit1d	6.149E−01	6.769E−01	6.929E−01	8.319E−01	6.729E−01
fit1p	2.140E+00	3.181E+00	1.404E+01	7.545E+00	2.641E+00
fit2d	8.003E+01				8.341E+01
fit2p	3.033E+02	2.918E+02	8.207E+03		3.334E+02
ganges	8.709E−01	1.801E+00	2.117E+00	4.684E+00	1.290E+00
grow15	5.539E−01		1.018E+00		6.929E−01
grow22			2.233E+00		1.558E+00
grow7	1.190E−01	1.570E−01	2.000E−01		1.490E−01
israel	1.530E−01	2.580E−01	3.889E−01	5.119E−01	2.240E−01
kb2	8.999E−03	9.999E−03	9.999E−03	3.300E−02	1.100E−02
lotfi	1.240E−01	1.890E−01	2.690E−01	4.069E−01	1.840E−01
maros					
maros_r7					
osa_07	1.840E+01	2.087E+01	4.538E+01	3.276E+01	2.003E+01
osa_14	1.032E+02	1.142E+02	3.326E+02	1.672E+02	1.117E+02
osa_30	3.892E+02	4.277E+02	2.094E+03	6.170E+02	4.029E+02
osa_60	2.306E+03	2.533E+03	1.537E+04	3.675E+03	2.747E+03
perold					
pilot					

Continued on the next page

Table 7.5: (continued)

Name	LUSOL	MA57	SuperLU	PARDISO	UMFPACK
pilot4					
pilot_ja					
pilot_we					
pilot87					
pilotnov					
recipe	3.999E−03	6.000E−03	6.999E−03	9.999E−03	5.999E−03
sc105	1.500E−02	2.300E−02	2.599E−02	4.499E−02	2.100E−02
sc205	4.399E−02	7.199E−02	7.699E−02	1.580E−01	5.899E−02
sc50a	4.999E−03	6.999E−03	6.999E−03	9.998E−03	5.999E−03
sc50b	5.999E−03	7.999E−03	7.998E−03	1.100E−02	5.999E−03
scagr25	4.399E−01	8.769E−01	1.012E+00	2.016E+00	6.369E−01
scagr7	3.999E−02	6.499E−02	7.099E−02	1.250E−01	5.999E−02
scfxm1	2.010E−01	3.769E−01	4.489E−01		
scfxm2	7.989E−01	1.562E+00		3.947E+00	1.320E+00
scfxm3	1.687E+00	3.380E+00	4.143E+00	7.526E+00	2.761E+00
scrs8	4.509E−01	8.919E−01	9.908E−01	2.083E+00	6.499E−01
scsd1	9.499E−02	1.120E−01	1.220E−01	1.800E−01	1.120E−01
scsd6	2.800E−01		3.030E−01		3.669E−01
scsd8	1.703E+00		3.287E+00		2.227E+00
sctap1	1.810E−01	3.020E−01	3.879E−01	6.839E−01	2.690E−01
sctap2	1.486E+00	2.861E+00	4.785E+00	7.328E+00	2.446E+00
sctap3	2.726E+00	5.063E+00	9.122E+00	1.357E+01	4.459E+00
share1b	9.498E−02	1.440E−01	1.540E−01		1.250E−01
share2b	3.199E−02	4.299E−02	4.899E−02	7.699E−02	3.999E−02
stair	5.979E−01	9.609E−01	1.630E+00		8.769E−01
standata	1.320E−01	2.000E−01	2.490E−01	4.699E−01	1.650E−01
standmps	4.429E−01	7.859E−01	1.168E+00	1.762E+00	5.709E−01
stocfor1	6.000E−03	5.999E−03	7.000E−03	9.999E−03	6.999E−03
stocfor2	6.139E−01	1.229E+00	1.487E+00	3.320E+00	9.759E−01
stocfor3	3.972E+01	8.217E+01	9.288E+01	2.322E+02	6.100E+01
woodw	6.582E+00	1.102E+01	2.304E+01	1.317E+01	9.508E+00

Table 7.5: CPU times for QPBLU on QP problems derived from the LPnetlib collection using the Hessian $H = I$. Each of the available linear system solvers LUSOL, MA57, SuperLU, PARDISO, and UMFPACK was used within QPBLU.

Chapter 8

Contributions, conclusions, and future work

QPBLU is a new active-set quadratic programming solver based on block-LU updates of the KKT system. The block-LU approach used by QPBLU complements null-space methods for quadratic programming problems, because it is relatively efficient for problems with many degrees of freedom.

The Fortran 95 modules that comprise the QPBLU package are portable and may be re-used in other applications. In particular, the module `luInterface` is extensible and provides a single interface to third-party solvers to factorize and solve with a sparse symmetric matrix. Currently, interfaces to five different linear system solvers have been incorporated into this module. The potential exists for many more solvers to be added, allowing QPBLU to take immediate advantage of any advances in methods for sparse linear algebra. The `luInterface` module enables the user to call many different linear system solvers with a single calling routine, facilitating the comparison of third-party solvers within the context of a QP solver.

The use of different third-party solvers within QPBLU has provided some insight into useful properties of linear system solvers, and we hope that this may contribute to the development of features within these solvers. As the example with SuperLU in Chapter 7 shows, separate solves with L and U are needed to help maintain sparsity in the block-LU factors. Ideally, such solves would be able to take full advantage of the sparse right-hand side vectors that arise in our QP method. It is also important to be able to obtain rank-revealing L or U factors in order to implement the KKT repair of Chapter 5.

LUSOL and MA57 already have rank-revealing properties that point to dependent rows and columns in a given KKT matrix K_0. Once singularity is indicated, currently only LUSOL (and recent versions of MA48) can pinpoint singular columns in A^T and A_2 of section 5.2. Ideally, future unsymmetric solvers would provide similar rank-revealing pivot options for rectangular matrices.

The block-LU approach used by QPBLU is an effective method for machines with advanced architecture. The formation and solution of the KKT system is the primary source of computational work at each iteration, and the use of a parallel linear system solver within QPBLU is an easy and effective way to take advantage of distributed or shared memory parallel machines. The sparse matrix multiplications with block-LU factors Y and Z are also parallelizable, and as demonstrated in [29], Y and Z can also be designed to take advantage of machine hardware.

QPBLU has a great deal of potential, but additional testing with larger and more varied problems is still needed. In addition, since the QP algorithm used by QPBLU is based on an inertia-controlling method, it is possible to extend the solver to more general (non-convex) QP problems. Results given in this thesis are obtained using a special form of the Hessian, and more varied test problems would give greater insight into its performance, especially when using different third-party solvers. An interface to the CUTEr [48] testing environment would provide access to many hundreds of test problems, such the Maros and Mészáros QP test set [59].

It will also be interesting to examine the performance of this solver when it is used within an SQP solver. In particular, we wish to examine its performance when used within the SQP solver SNOPT when the number of superbasic variables is very large and when a sequence of limited memory Hessians of the form $H_{k+1} = (I + v_{k+1}u_{k+1}^T)H_k(I + u_{k+1}v_{k+1}^T)$ is used. Efforts to integrate QPBLU into SNOPT are currently underway.

Much work could be done to improve the robustness of QPBLU, as it was unable to solve, with any third-party solver, several of the problems generated from the LPnetlib data. This was usually because the KKT matrix was determined to be ill-conditioned or near-singular by the third-party solver. Although the general reliability of solvers on large indefinite systems is still a matter of concern [46, 47, 50, 51], the robustness of QPBLU should be improved using KKT repair, as already mentioned. Alternatively, other techniques, such as the regularization approach of section 5.3 or proximal point methods, could be applied.

Bibliography

[1] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.*, 23(1):15–41, 2001.

[2] C. Ashcraft, R. G. Grimes, and J. G. Lewis. Accurate symmetric indefinite linear equation solvers. *SIAM J. Matrix Anal. Appl.*, 20:513–561, 1998.

[3] Aspen Technology, Inc. Aspen Target. `http://www.aspentech.com`.

[4] J. Atkociunas. Quadratic programming for degenerate shakedown problems of bar structures. *Mechanics Research Communications*, 23(2):195–203, 1996.

[5] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. Curfman McInnes, B. F. Smith, and H. Zhang. PETSc Web page. `http://www.mcs.anl.gov/petsc`, 2001.

[6] R. H. Bartels. A stabilization of the simplex method. *Numer. Math.*, 16:414–434, 1971.

[7] R. Bartlett and L. Biegler. QPSchur: A dual, active-set, Schur-complement method for large-scale and structured convex quadratic programming. *Optimization and Engineering*, 7(1):5–32, 2006.

[8] R. A. Bartlett, A. Wächter, and L. T. Biegler. Active set vs. interior point strategies for model predictive control. In *Proceedings of the American Control Conference*, pages 4229–4233, June 2000.

[9] G. Bashein and M. Enns. Computation of optimal controls by a method combining quasi-linearization and quadratic programming. *International Journal of Control*, 16(1):177–187, 1972.

[10] H. Y. Benson and D. F. Shanno. An exact primal-dual penalty method approach to warm-starting interior-point methods for linear programming. *Comput. Optim. Appl.*, 38(3):371–399, 2007.

[11] M. Berkelaar, K. Eikland, and P. Notebaert. lp_solve 5.5. `http://lpsolve.sourceforge.net/5.5/`.

[12] J. T. Betts and P. D. Frank. A sparse nonlinear optimization algorithm. *J. Optim. Theory and Applics.*, 82(3):519–541, 1994.

[13] S. C. Billups, S. P. Dirkse, and M. C. Ferris. A comparison of algorithms for large-scale mixed complementarity problems. *Comp. Optim. Appl.*, 7:3–25, 1997.

[14] J. Bisschop and A. Meeraus. Matrix augmentation and partitioning in the updating of the basis inverse. *Math. Program.*, 13(3):241–254, 1977.

[15] I. Bongartz, A. R. Conn, N. I. M. Gould, and Ph. L. Toint. CUTE: Constrained and unconstrained testing environment. *ACM Trans. Math. Software*, 21(1):123–160, 1995.

[16] The Boeing Company. BCSLIB-EXT. `http://www.boeing.com/phantom/bcslib-ext`.

[17] COMSOL. COMSOL Mulitphysics. `http://www.comsol.com`.

[18] T. A. Davis. The University of Florida sparse matrix collection. `http://www.cise.ufl.edu/research/sparse/matrices`. Submitted to SIAM J. Matrix Anal. Appl.

[19] T. A. Davis. Algorithm 832: UMFPACK v4.3, an unsymmetric-pattern multifrontal method with a column pre-ordering strategy. *ACM Trans. Math. Software*, 30(2), 2004.

[20] T. A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, Philadelphia, 2006.

[21] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Anal. Appl.*, 20(3):720–755, 1999.

[22] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM J. Matrix Anal. Appl.*, 20(4):915–952, 1999.

[23] E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Math. Program.*, 91(2), 2002.

[24] I. S. Duff. MA57: a Fortran code for the solution of sparse symmetric definite and indefinite systems. *ACM Trans. Math. Software*, 30(2):118–144, 2004.

[25] I. S. Duff, N. I. M. Gould, J. K. Reid, J. A. Scott, and K. Turner. The factorization of sparse symmetric indefinite matrices. *IMA J. Numer. Anal.*, 11:181–204, 1991.

[26] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Software*, 9(3):302–325, 1983.

[27] I. S. Duff and J. K. Reid. The design of MA48: a code for the direct solution of sparse unsymmetric linear systems of equations. *ACM Trans. Math. Software*, 22(2):187–226, 1996.

[28] I. S. Duff and J. K. Reid. Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Software*, 22(2):227–257, 1996.

[29] S. Eldersveld and M. A. Saunders. A block-LU update for large-scale linear programming. *SIAM J. Matrix Anal. Appl.*, 13(1):191–201, 1992.

[30] M. C. Ferris and T. S. Munson. Interior-point methods for massive support vector machines. *SIAM J. Optim.*, 13(3):783–804, 2002.

[31] R. Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, New York, 1987.

[32] E. M. Gertz and S. J. Wright. Object-oriented software for quadratic programming. *ACM Trans. Math. Software*, 29:58–81, 2003.

[33] P. E. Gill. Optimization course notes for Math 271. University of California, San Diego.

[34] P. E. Gill, W. Murray, and M. A. Saunders. User's guide for QPOPT (version 1.0): a Fortran package for quadratic programming. Technical Report NA 95–1, University of California, San Diego, 1995.

[35] P. E. Gill, W. Murray, and M. A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Review*, 47(1):99–131, 2005.

[36] P. E. Gill, W. Murray, and M. A. Saunders. User's guide for SQOPT 7: Software for large-scale linear and quadratic programming. Technical Report NA 05–1, University of California, San Diego, 2005.

[37] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. Sparse matrix methods in optimization. *SIAM J. Sci. and Statist. Comput.*, 5(3), 1984.

[38] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. Maintaining LU factors of a general sparse matrix. *Linear Algebra and its Applications*, 88/89:239–270, 1987.

[39] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. A practical anti-cycling procedure for linearly constrained optimization. *Math. Program.*, 45(3):437–474, 1989.

[40] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. A Schur-complement method for sparse quadratic programming. In M. G. Cox and S. J. Hammarling, editors, *Reliable Numerical Computation*, pages 113–138. Oxford University Press, 1990.

[41] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. Inertia-controlling methods for general quadratic programming. *SIAM Review*, 33(1):1–36, 1991.

[42] D. Goldfarb and A. Idnani. A numerically stable dual method for solving strictly quadratic programs. *Math. Program.*, 27:1–33, 1983.

[43] G. H. Golub and C. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[44] J. Gondzio. Warm start of the primal-dual method applied in the cutting plane scheme. *Math. Program.*, 83(1):125–143, 1998.

[45] K. Goto. GotoBLAS. `http://www.tacc.utexas.edu/resources/software`.

[46] N. I. M. Gould, Y. Hu, and J. A. Scott. Complete results from a numerical evaluation of sparse direct solvers for the solution of large, sparse, symmetric linear systems of equations. Numerical Analysis Internal Report 2005-1, Rutherford Appleton Laboratory, 2005.

[47] N. I. M. Gould, Y. Hu, and J. A. Scott. A numerical evaluation of sparse direct symmetric solvers for the solution of large sparse, symmetric linear systems of equations. Technical Report RAL-TR-2005-005, Rutherford Appleton Laboratory, 2005.

[48] N. I. M. Gould, D. Orban, and Ph. L. Toint. CUTEr and SifDec: A constrained and unconstrained testing environment, revisited. *ACM Trans. Math. Software*, 29(4):373–394, 2003.

[49] N. I. M. Gould, D. Orban, and Ph. L. Toint. GALAHAD, a library of thread-safe Fortran 90 packages for large-scale nonlinear optimization. *ACM Trans. Math. Software*, 29(4):353–372, December 2003.

[50] N. I. M. Gould and J. A. Scott. Complete results from a numerical evaluation of HSL packages for the direct solution of large sparse, symmetric linear systems of equations. Numerical Analysis Group Internal Report 2003-2, Rutherford Appleton Laboratory, 2003.

[51] N. I. M. Gould and J. A. Scott. A numerical evaluation of HSL packages for the direct solution of large sparse, symmetric linear systems of equations. *ACM Trans. Math. Software*, 30(3):300–325, 2004.

[52] N. I. M. Gould and Ph. L. Toint. A quadratic programming bibliography. Numerical Analysis Group Internal Report 2000-1, Rutherford Appleton Laboratory, 2000.

[53] N. I. M. Gould and Ph. L. Toint. Numerical methods for large-scale non-convex quadratic programming. In A. H. Siddiqi and M. Kočvara, editors, Trends in Industrial and Applied Mathematics, 2002.

[54] N. I. M. Gould and Ph. L. Toint. Preprocessing for quadratic programming. *Math. Program.*, 100(1):95–132, 2004.

[55] A. Gupta, M. Joshi, and V. Kumar. WSMP: A high-performance serial and parallel sparse linear solver. Technical Report RC 22038 (98932), IBM T. J. Watson Research Center, 2001.

[56] ILOG, Inc. ILOG CPLEX: High-performance software for mathematical programming and optimization. `http://www.ilog.com/products/cplex/`.

[57] X. S. Li. Direct solvers for sparse matrices. `http://crd.lbl.gov/~xiaoye/SuperLU/SparseDirectSurvey.pdf`, 2006.

[58] X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Software*, 29(2):110–140, 2003.

[59] I. Maros and C. Mészáros. A repository of convex quadratic programming problems. *Optim. Meth. Software*, 11&12:671–681, 1999.

[60] A. D. Martin. Mathematical programming of portfolio selections. *Management Science*, 1(2):152–166, 1955.

[61] The MathWorks. MATLAB. `http://www.mathworks.com`.

[62] C. Mészáros. Linear programming problems. `http://www.sztaki.hu/~meszaros/public_ftp/lptestset`.

[63] H. Mittelmann. Linear programming problems. `http://plato.asu.edu/ftp/lptestset`.

[64] B. A. Murtagh and M. A. Saunders. Large-scale linearly constrained optimization. *Math. Program.*, 14:41–72, 1978.

[65] Wolfram Research. Mathematica. `http://www.wolfram.com`.

[66] Y. Saad. SPARSKIT: A basic toolkit for sparse matrix computations, version 2. `http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html`, 1994.

[67] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems*, 20(3):475–487, 2004.

[68] O. Schenk and K. Gärtner. On fast factorization pivoting methods for symmetric indefinite sytems. *Elec. Trans. Numer. Anal.*, 23:158–179, 2006.

[69] O. Schenk, K. Gärtner, and W. Fichtner. Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors. *BIT*, 40(1):158–176, 2000.

[70] C. Schmid and L. T. Biegler. Quadratic programming methods for reduced Hessian SQP. *Comp. Chem. Eng.*, 18(9), 1994.

[71] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Clarendon Press, Oxford, 1965.

[72] S. J. Wright. Applying new optimization algorithms to model predictive control. In Chemical Process Control-V, volume 93 of AIChE Symposium Series, Number 316, pages 147–155. CACHE Publications, 1997.

[73] A. Wchter and L. T. Biegler. On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. *Math. Program.*, 106(1):25–57, 2006.

[74] E. A. Yildirim and S. J. Wright. Warm-start strategies in interior-point methods for linear programming. *SIAM J. Optim.*, 12(3):782–810, 2002.