Preprint ANL/MCS-P891-0701
Mathematics and Computer Science Division
Argonne National Laboratory

Michael Gertz · Stephen Wright

Object-Oriented Software for Quadratic Programming[*]

October 2001

Abstract. We describe the object-oriented software package OOQP for solving convex quadratic programming problems (QP). The primal-dual interior point algorithms supplied by OOQP are implemented in a way that is largely independent of the problem structure. Users may exploit problem structure by supplying linear algebra, problem data, and variable classes that are customized to their particular applications. The OOQP distribution contains default implementations that solve several important QP problem types, including general sparse and dense QPs, bound-constrained QPs, and QPs arising from support vector machines and Huber regression. The implementations supplied with the OOQP distribution are based on such well known linear algebra packages as MA27/57, LAPACK, and PETSc.

Key words. Quadratic Programming, Object-Oriented Software, Interior-Point Methods

1. Introduction

Convex quadratic programming problems (QPs) are optimization problems in which the objective function is a convex quadratic and the constraints are linear. They have the general form

$$\min_x \tfrac{1}{2} x^T Q x + c^T x \ \text{ s.t. } \ Ax = b, \ Cx \geq d, \tag{1}$$

where Q is a symmetric positive semidefinite $n \times n$ matrix, $x \in \mathsf{R}^n$ is a vector of unknowns, A and C are (possibly null) matrices, and b and d are vectors of appropriate dimensions. The constraints $Ax = b$ are referred to as equality constraints while $Cx \geq d$ are known as inequality constraints.

QPs arise directly in such applications as least-squares regression with bounds or linear constraints, robust data fitting, Markowitz portfolio optimization, data

Michael Gertz: Electrical and Computer Engineering Department, Northwestern University, Evanston, IL 60208, `gertz@ece.nwu.edu`, and Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439

Stephen Wright: Computer Sciences Department, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, WI 53706; `swright@cs.wisc.edu`

Mathematics Subject Classification (1991): 68N19, 65-04, 65K05

[*] Research supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; and by the National Science Foundation Grants CDA-9726385 and ACI-0082065.

mining, support vector machines, and tribology. They also arise as subproblems in optimization algorithms for nonlinear programming (in sequential quadratic programming algorithms and augmented Lagrangian algorithms) and in stochastic optimization (regularized decomposition algorithms). The data objects that define these applications exhibit a vast range of properties and structures, and it is desirable—often essential—to exploit the structure when solving the problem computationally. The wide variety of structures makes it difficult to provide a single piece of software that functions efficiently on any given application. In this paper, we describe the next-best thing: an object-oriented software package called OOQP that includes the following features:

- Interior-point algorithms that are implemented in a structure-independent way, permitting reuse of the optimization-related sections of OOQP across the whole application space.
- Isolation of structure-dependent operations and storage schemes into classes that can be customized by the user to fit particular applications.
- A linear algebra layer that can be used to assemble solvers for specific problem structures.
- Implementations of solvers for general large, sparse QPs and several other generic problem types.
- Implementations of solvers for several special problem types, including Huber regression and support vector machines, to demonstrate customization of the package to specific applications.
- A variety of interfaces to the bundled implementations that allow problem definition and data entry via ASCII files, MPS format, the AMPL modeling language, and MATLAB.

In this introduction, we first outline the basic design rationale of OOQP, then discuss related efforts in object-oriented numerical codes, particularly codes related to optimization.

1.1. OOQP Design Rationale

The algorithms implemented in OOQP are of the primal-dual interior-point type. These methods are well suited for structured problems, mainly because the linear systems that must be solved to compute the step at each iteration retain the same dimension and structure throughout the computation. When this linear system is sparse, it may not be necessary to perform storage allocation and ordering for a direct factorization anew at each iteration, but possibly just once at the initial solve. The coding effort involved in setting up and solving the linear system efficiently is typically much less than for the rival active-set approach, in which the matrix to be factored grows and shrinks as the computation progresses.

Interior-point algorithms are well suited to object-oriented implementation because the best heuristics, devices, and parameter settings used in these algorithms are largely independent of the underlying problem structure. Mehrotra's heuristics (see [23]) for choosing the centering parameter, step length, and corrector terms give significant improvements over standard path-following algorithms regardless of whether we are solving a linear program or a sparse structured QP. Gondzio's multiple correctors [17] also yield improvements across a wide range of problem types. Object-oriented design allows the classes that implement the interior-point algorithms to be written in a way that is independent of the problem structure. Users who wish to implement a customized version of OOQP for their problem type need not concern themselves with the interior-point sections of the code at all, but rather can focus on constructing classes to store data and variables and to perform the various linear algebra operations required by the interior-point algorithm. The code that implements the core of the algorithm, including all its sophisticated heuristics, can be reused across the entire space of problem structures and applications.

Codes that simply target general QP formulations (of the form (1), for instance) may not be able to solve all QPs efficiently, even if they exploit sparsity in the objective Hessian and constraint matrices. A dramatic example of a situation in which a generic solver would perform poorly is described by Ferris and Munson [12], who solve a QP arising from support-vector machine computations in which the Hessian has the form

$$Q = D + VV^T, \tag{2}$$

where D is a diagonal matrix with positive diagonal elements and V is a dense $n \times m$ matrix, where $n \gg m$. This Q is completely dense, and a generic dense implementation would solve an $n \times n$ dense matrix at each interior-point iteration to find the step. Such an approach is doomed to failure when n is large (of the order of 10^6, for example). OOQP includes an implementation specifically tailored to his problem structure, in which we store V rather than Q and use specialized factorization routines based on judicious block elimination to perform the linear algebra efficiently. A similar approach is described by Ferris and Munson [12].

As well as being useful for people who want to develop efficient solvers for structured problems, the OOQP distribution contains *shrink-wrapped solvers* for general QPs and for certain structured problems. We provide an implementation for solving sparse general QPs that can be invoked by procedure calls from C or C++ code; as an executable with an input file that defines the problem in MPS format extended appropriately for quadratic programming (Maros and Mészáros [22]); or via invocations from the higher-level languages AMPL and MATLAB. The distribution also includes an implementation of a solver for QPs arising from support vector machines and from Huber regression. Both these implementations accept input either from an ascii file or through a MATLAB interface.

The code is also useful for optimization specialists who wish to perform *algorithm development*, experimenting with variants of the heuristics in the interior-point algorithm, different choices of search direction and step length, and so on. Such researchers can work with the C++ class that implements the algorithm, without concerning themselves with the details associated with specific problem types and applications.

In addition, encapsulation of the linear algebra operations allows users of the code to *incorporate alternative linear algebra packages* as they become available. In OOQP's implementation of the solver for sparse general QPs, the MA27 code from the HSL Library [11,18] for sparse symmetric indefinite systems is used as the engine for solving the linear systems that arise at each interior-point iteration. We have implemented solvers based on other codes, including Oblio [10], HSL's MA57, and SuperLU [7]. These solvers differ from the distributed version only in the methods and classes specific to the linear algebra. The classes that define the interior-point algorithm, calculate the residuals, define the data, store and operate on the variables, and read the problem data from an input file are unaffected by the use of different linear solvers.

We chose to write OOQP in the C++ programming language. The object-oriented features of this language make it possible to express the design of the code in a natural way. Moreover, C++ is a well-known language for which stable, efficient compilers are available on a wide range of hardware platforms.

1.2. Related Work

Several other groups have been working on object-oriented numerical software in a variety of contexts in optimization, linear algebra, and differential equations. We mention some of these efforts here.

The Hilbert Class Library (HCL) (Gockenbach and Symes [16]) is a collection of C++ classes representing vectors, linear and nonlinear operators, and functions, together with a collection of methods for optimization and linear algebra that are implemented in terms of these abstract classes. Particular characteristics of HCL include an ability to handle large data sets and linear operators that are not defined explicitly in terms of matrices. The philosophy of OOQP is similar to that of HCL, though our more specific focus on structured quadratic programs distinguishes our effort. The `rSQP++` package (Bartlett [2]) is a C++ package that currently implements reduced-space SQP methods for nonlinear programming. Basic components of the algorithm are abstracted, such as computation of the null space and the quasi-Newton update. In structuring the package, particular attention is paid to the linear algebra layer and interfaces to it. The COOOL package (Deng, Gouveia, and Scales [8]) is another collection of C++ classes and includes implementations of a wide variety of algorithms and algorithm components.

The PETSc project (Balay et al. [1]) focuses on the development of software components for large-scale linear algebra, allowing data-structure-independent implementation of solvers for partial differential equations and nonlinear equations, on serial and parallel architectures. Although PETSc is implemented chiefly in C, its follows object-oriented design principles. PETSc solvers and design conventions are used in the TAO package (Benson, Curfman McInnes, and Moré [3]), which currently implements solvers for large-scale unconstrained and bound-constrained optimization problems on parallel platforms. An object-oriented direct solver for sparse linear algebra problems is discussed by Dobrian, Kumfert, and Pothen [9]; we have used their Oblio package [10] in implementations of OOQP for solving general sparse QPs.

Other nonoptimization object-oriented efforts that of Chow and Heroux [5], who focus on preconditioning of iterative solvers for linear systems and describe a C++ package for that allows implementation of block preconditioners in a way that is independent of the storage scheme for the submatrix blocks. The Diffpack code of Bruaset and Langtangen [4] is a C++ object-oriented implementation of iterative solvers for sparse linear systems.

1.3. Outline of This Paper

Section 2 of this paper describes the primal-dual interior-point algorithms that are the basis of OOQP. The layered structure of the code and its major classes are outlined in Section 3, where we also illustrate each class by discussing its implementation for the particular formulation (1). Section 3.6 outlines the linear algebra layer of OOQP, showing how abstractions of the important linear algebra objects and operations can be used in the higher layers of the package, while existing software packages can be used to implement these objects and operations. Other significant classes in OOQP are discussed in Section 4. Section 5 further illustrates the usefulness of the object-oriented approach by describing three QPs with highly specialized structure and outlining how each is implemented efficiently in the OOQP framework. In Section 6, we outline the contents of the OOQP distribution file.

Further information on OOQP can be found in the *OOQP User Guide* [15], which is included in the distribution and can also be obtained from the OOQP Web site, `www.cs.wisc.edu/~swright/ooqp`.

2. Primal-Dual Interior-Point Algorithms

In this section we describe briefly the interior-point algorithms implemented in OOQP. For concreteness, we focus our discussion on the formulation (1).

2.1. Optimality Conditions

The optimality conditions for (1) are that there exist Lagrange multiplier vectors y and z and a slack vector s such that the following relations hold:

$$Qx - A^T y - C^T z = -c, \tag{3a}$$

$$Ax = b, \tag{3b}$$

$$Cx - s = d, \tag{3c}$$

$$z \geq 0 \perp s \geq 0. \tag{3d}$$

The last row indicates that we require z and s to be complementary nonnegative variables, that is, we require $z^T s = 0$ in addition to $z \geq 0$, $s \geq 0$. We assume that A and C have m_A and m_C rows, respectively, so that $y \in \mathsf{R}^{m_A}$ and $z \in \mathsf{R}^{m_C}$.

Primal-dual interior-point algorithms generate iterates (x, y, z, s) that are strictly feasible with respect to the inequality constraints, that is, $(z, s) > 0$. The complementarity measure μ defined by

$$\mu = z^T s / m_C \tag{4}$$

is important in measuring the progress of the algorithm, since it measures violation of the complementarity condition $z^T s = 0$. In general, each iterate will also be infeasible with respect to the equality constraints (3a), (3b), and (3c), so our optimality measure also takes into account violation of these constraints.

2.2. Mehrotra Predictor-Corrector Algorithm

We implement two algorithms: Mehrotra's predictor-corrector method [23] and Gondzio's higher-order corrector method [17]. (See also [26, Chapter 10] for a detailed discussion of both methods.) These algorithms have proved to be the most effective methods for linear programming problems and in our experience are just as effective for QP. Mehrotra's algorithm is outlined below.

```
**Algorithm MPC (Mehrotra Predictor-Corrector)**
Given starting point $(x, y, z, s)$ with $(z, s) > 0$, parameter $\tau \in [2, 4]$;
repeat
    Set $\mu = z^T s / m_C$;
    Solve for $(\Delta x^{\mathrm{aff}}, \Delta y^{\mathrm{aff}}, \Delta z^{\mathrm{aff}}, \Delta s^{\mathrm{aff}})$:
```

$$\begin{bmatrix} Q & -A^T & -C^T & 0 \\ A & 0 & 0 & 0 \\ C & 0 & 0 & -I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x^{\mathrm{aff}} \\ \Delta y^{\mathrm{aff}} \\ \Delta z^{\mathrm{aff}} \\ \Delta s^{\mathrm{aff}} \end{bmatrix} = - \begin{bmatrix} r_Q \\ r_A \\ r_C \\ ZSe \end{bmatrix}, \tag{5}$$

where

$$\begin{aligned} S &= \mathrm{diag}(s_1, s_2, \ldots, s_{m_C}), & (6a) \\ Z &= \mathrm{diag}(z_1, z_2, \ldots, z_{m_C}), & (6b) \\ r_Q &= Qx + c - A^T y - C^T z, & (6c) \\ r_A &= Ax - b, & (6d) \\ r_C &= Cx - s - d. & (6e) \end{aligned}$$

```
    Compute $\alpha_{\mathrm{aff}}$ to be the largest value in $(0, 1]$ such that
```

$$(z, s) + \alpha(\Delta z^{\mathrm{aff}}, \Delta s^{\mathrm{aff}}) \geq 0;$$

```
    Set $\mu_{\mathrm{aff}} = (z + \alpha_{\mathrm{aff}} \Delta z^{\mathrm{aff}})^T (s + \alpha_{\mathrm{aff}} \Delta s^{\mathrm{aff}}) / m_C$;
    Set $\sigma = (\mu_{\mathrm{aff}} / \mu)^\tau$;
    Solve for $(\Delta x, \Delta y, \Delta z, \Delta s)$:
```

```
$$\begin{bmatrix} Q & -A^T & -C^T & 0 \\ A & 0 & 0 & 0 \\ C & 0 & 0 & -I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = - \begin{bmatrix} r_Q \\ r_A \\ r_C \\ ZSe - \sigma\mu e + \Delta Z^{\text{aff}} \Delta S^{\text{aff}} e \end{bmatrix}, \qquad (7)$$
            where $\Delta Z^{\text{aff}}$ and $\Delta S^{\text{aff}}$ are defined in an obvious way;
        Compute $\alpha_{\max}$ to be the largest value in $(0,1]$ such that
$$(s,z) + \alpha(\Delta s, \Delta z) \geq 0;$$
        Choose $\alpha \in (0, \alpha_{\max})$ according to Mehrotra's heuristic;
        Set
$$(x,y,z,s) \leftarrow (x,y,z,s) + \alpha(\Delta x, \Delta y, \Delta z, \Delta s);$$
    until convergence or infeasibility test satisfied.
```

The direction obtained from (7) can be viewed as an approximate second-order step toward a point (x^+, y^+, z^+, s^+) at which the conditions (3a), (3b), and (3c) are satisfied and, in addition, the pairwise products $z_i^+ s_i^+$ are all equal to $\sigma\mu$. The heuristic for σ yields a value in the range $(0,1)$, so the step usually produces a reduction in the average value of the pairwise products from their current average of μ.

Gondzio's approach [17] follows the Mehrotra algorithm in its computation of directions from (5) and (7). It may then go on to enhance the search direction further by solving additional systems similar to (7), with variations in the last m_C components of the right-hand side. Successive corrections are performed so long as (i) the length of the step $\alpha_{\max}$ that can be taken along the corrected direction is lengthened appreciably; and (ii) the pairwise products $s_i z_i$ whose values are either much larger than or much smaller than the average are brought into closer correspondence with the average. The maximum number of corrections is dictated by the ratio of the time taken to factor the coefficient matrix in (7) to the time taken to solve the system using these factors for a given right-hand side. When the cost of the solve is small relative to the cost of factorization, we allow more correctors to be calculated, up to a limit of 5.

The algorithm uses the steplength heuristic described in Mehrotra [23, Section 6], modified slightly to ensure that the same step lengths are used for both primal and dual variables.

2.3. Convergence Conditions

We use convergence criteria similar to those of PCx [6]. To specify these, we use (x^k, y^k, z^k, s^k) to denote the primal-dual variables at iteration k, and $\mu_k \overset{\text{def}}{=} (z^k)^T s^k / m_C$ to denote the corresponding value of μ. Let r_Q^k, r_A^k, and r_C^k be the values of the residuals at iteration k, and let gap_k be the duality gap at iteration k, which is defined by

$$\text{gap}_k \overset{\text{def}}{=} (x^k)^T Q x^k - b^T y^k + c^T x^k - d^T z^k. \tag{8}$$

(It can be shown that $\text{gap}_k = m_C \mu_k$ when (x^k, y^k, z^k, s^k) is feasible with respect to the conditions (3a), (3b), (3c), and (3d).) We define the quantity ϕ_k as

$$\phi_k \overset{\text{def}}{=} \frac{\|(r_Q^k, r_A^k, r_C^k)\|_\infty + \text{gap}_k}{\|(Q, A, C, c, b, d)\|_\infty},$$

where the denominator is simply the element of largest magnitude in all the data quantities that define the problem (1). Note that $\phi_k = 0$ if and only if (x^k, y^k, z^k, s^k) is optimal.

Given parameters $\texttt{tol}_\mu$ and $\texttt{tol}_r$ (both of which have default value 10^{-8}), we declare successful termination when

$$\mu_k \le \texttt{tol}_\mu, \quad \|(r_Q^k, r_A^k, r_C^k)\|_\infty \le \texttt{tol}_r \|(Q, A, C, c, b, d)\|_\infty. \tag{9}$$

We declare the problem to be probably infeasible if

$$\phi_k > 10^{-8} \text{ and } \phi_k \ge 10^4 \min_{0 \le i \le k} \phi_i. \tag{10}$$

We terminate with status "unknown" if the algorithm appears to be making slow progress, that is,

$$k \ge 30 \text{ and } \min_{0 \le i \le k} \phi_i \ge \frac{1}{2} \min_{1 \le i \le k-30} \phi_i, \tag{11}$$

or if the ratio of infeasibility to the value of μ appears to be blowing up, that is,

$$\|(r_Q^k, r_A^k, r_C^k)\|_\infty > \texttt{tol}_r \|(Q, A, C, c, b, d)\|_\infty \tag{12a}$$

$$\text{and } \|(r_Q^k, r_A^k, r_C^k)\|_\infty / \mu_k \ge 10^8 \|(r_Q^0, r_A^0, r_C^0)\|_\infty / \mu_0. \tag{12b}$$

We also terminate when the number of iterations exceeds a specified maximum.

2.4. Major Arithmetic Operations

We can now identify the key arithmetic operations to be performed at each iteration of the interior-point algorithm. Computation of the residuals r_Q, r_A, and r_C from the formulae (6c), (6d), and (6e) is performed once per iteration. Solution of the systems such as (5) and (7), which have the same coefficient matrix but different right-hand sides, is performed between two and six times per iteration. Inner products are needed in the computation of μ and μ_{aff}. Componentwise vector operations are needed to determine $\alpha_{\max}$, and "saxpy" operations are needed to take the step. The implementation of all these operations depends heavily on the storage scheme used for the the problem data and variables, on the specific structure of the problem data, and on the choice of algorithm for solving the linear systems. The interior-point algorithm does not need to know about these details, however, so it can be implemented in a way that is independent of these considerations. This observation is the basis of our design of OOQP.

3. Layered Design of OOQP and Major Classes

OOQP derives much of its flexibility from a layered design in which each layer is built from abstract operations defined by the layer below it. Those who wish to create a specialized solver for a certain type of QP may customize one of the three layers. In this section, we outline the layer structure and describe briefly the major classes within these layers.

The top layer is the *QP solver layer*, which consists of the interior-point algorithms and heuristics for solving QPs. The OOQP distribution contains two implementations of the `Solver` class in this layer, one for Mehrotra's predictor-corrector algorithm and one for Gondzio's variant.

Immediately below the solver layer is the *problem formulation layer*, which defines classes with behavior of immediate interest to interior-point QP solvers. Included are classes with methods to store and manipulate the problem data (Q, A, C, c, b, d), the current iterate (x, y, z, s), and the residuals (r_Q, r_A, r_C), as well as classes with methods for solving linear systems such as (5) and (7). The major classes in this layer—`Data`, `Variables`, `Residuals`, and `LinearSystem`—are discussed below. We indicate briefly how these classes would be implemented for the particular case of the formulation (1) in which Q, A, and C are dense matrices. (This formulation appears in the OOQP distribution in the directory `src/QpExample`.)

The lowest layer of OOQP is the *linear algebra layer*. This layer contains code for manipulating linear algebra objects, such as vectors and matrices, that provides behavior useful across a variety of QP formulations.

3.1. `Solver` *Class*

The `Solver` class contains methods for monitoring and checking the convergence status of the algorithm, methods to determine the step length along a given direction, methods to define the starting point, and the `solve` method that implements the interior-point algorithm. The `solve` method for the two derived classes `MehrotraSolver` and `GondzioSolver` implements the algorithms described in Section 2 and stores the various parameters used by these algorithms. For instance, the parameter τ in Algorithm MPC is fixed to a default value in the constructor routines for `MehrotraSolver`, along with a tolerance parameter to be used in termination tests, a parameter indicating maximum number of iterations allowed, and so on. Even though some fairly sophisticated heuristics are included directly in the `solve` code (such as Gondzio's rules for additional corrector steps), the code implementing `solve` contains fewer than 150 lines of C++ in both cases. Key operations—residual computations, saxpy operations, linear system solves—are implemented by calls to abstract classes in the problem formulation layer, making our implementation structure independent.

Apart from `solve`, the other important methods in *Solver* include the following.

`start`: Implements a default starting-point heuristic. While interior-point theory places fairly loose restrictions on the choice of starting point, the choice of heuristic can significantly affect the robustness and efficiency of the algorithm. The heuristic implemented in the OOQP distribution is described further in Section 3.7.

`finalStepLength`: Implements a version of Mehrotra's starting point heuristic [23, Section 6], modified to ensure identical steps in the primal and dual variables.

`doStatus`: Tests for termination. Unless the user supplies a specific termination routine, this method calls another method `defaultStatus`, which performs the tests (9), (10), (11), and (12) and returns a code indicating the current convergence status.

3.2. `Data` *Class*

The `Data` class stores the data defining the problem and provides methods for performing the operations with this data required by the interior-point algorithms. These operations include assembling the linear systems (5) or (7), performing matrix-vector operations with the data, calculating norms of the data, reading input into the data structure from various sources, generating random problem instances, and printing the data.

Since both the data structures and the methods implemented in `Data` depend so strongly on the structure of the problem, the parent class is almost empty. Our derived class of `Data` for the formulation (1) defines the vectors c, b, and d and the matrices A, C, and Q to be objects of the appropriate type from the linear algebra layer. The dimensions of the problem (n, m_A, and m_C) would be stored as integer variables.

Following (5) and (7), the general form of the linear system to be solved at each iteration is

$$\begin{bmatrix} Q & -A^T & -C^T & 0 \\ A & 0 & 0 & 0 \\ C & 0 & 0 & -I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = - \begin{bmatrix} r_Q \\ r_A \\ r_C \\ r_{z,s} \end{bmatrix}, \tag{13}$$

for some choice of $r_{z,s}$. Since the diagonal elements of Z and S are strictly positive, we can do a step of block elimination to obtain the following equivalent system:

$$\begin{bmatrix} Q & A^T & C^T \\ A & 0 & 0 \\ C & 0 & -Z^{-1}S \end{bmatrix} \begin{bmatrix} \Delta x \\ -\Delta y \\ -\Delta z \end{bmatrix} = \begin{bmatrix} -r_Q \\ -r_A \\ -r_C - Z^{-1} r_{z,s} \end{bmatrix}, \tag{14a}$$

$$\Delta s = Z^{-1}(-r_{z,s} - S\Delta y). \tag{14b}$$

Because of its symmetric indefinite form and the fact that formation of $Z^{-1}S$ is trivial, the system (14a) is convenient to solve in general. (Further reduction is possible for QPs with special structures, as we discuss in Section 5.) Storage for the matrix in (14a) is allocated in the `LinearSystem` class, but the methods for placing Q, A, and C into this data structure are implemented in the `Data` class.

3.3. `Variables` *Class*

The methods in the `Variables` class are defined as pure virtual functions because they strongly depend on the structure of the variables and the problem. They are essential in the implementation of the algorithms. The derived `Variables` class for the formulation (1) contains `int` objects that store the problem dimensions n, m_A, and m_C and vector objects from the linear algebra layer that store x, y, z, and s.

Methods in the `Variables` class include a method for calculating the complementarity gap μ (in the case of (1), this is defined by $\mu = z^T s/m_C$); a method for adding a scalar multiple of a given search direction to the current set of variables; a method for calculating the largest multiple of a given search direction that can be added before violating the nonnegativity constraints; a method for printing the variables in some format appropriate to their structure; and methods for calculating various norms of the variables.

3.4. `Residuals` *Class*

The `Residuals` class calculates and stores the quantities that appear on the right-hand side of the linear systems such as (5) and (7) that arise at each interior-point iteration. These residuals can be partitioned into two fundamental categories: the components arising from the linear equations in the KKT conditions, and the components arising from the complementarity conditions. For the formulation (1), the components r_Q, r_A, and r_C (which arise from KKT linear equations (3a), (3b), and (3c)) belong to the former class while $r_{z,s}$ belongs to the latter.

The main methods in the `Residuals` class are a method for calculating the "linear equations" residuals; a method for calculating the current duality gap (which we define for the formulation (1) by (8)); a method for calculating the residual norm; methods for zeroing the residual vectors; and methods for calculating and manipulating the "complementarity" residuals as required by the interior-point algorithm.

3.5. `LinearSystem` *Class*

The major operation at each iteration, computationally speaking, is the solution of a number of linear systems to obtain the predictor and corrector steps. For the formulation (1), these systems have the form (13). At each iteration of the interior-point method, such systems need to be solved two to six times, for different choices of the right-hand side components but the same coefficient matrix.

Accordingly, it makes sense to logically separate the operations of `factor`ing this matrix and `solve`ing for a specific right-hand side.

We use the term "factor" in a general sense, to indicate the part of the solution process that is *independent of the right-hand side*. The `factor` method could involve certain block-elimination operations on the coefficient matrix, together with an LU, LDL^T, or Cholesky factorization of a reduced system. Alternatively, when an iterative solver is used, the `factor` operation could involve computation of a preconditioner. The `factor` method may need to store data, such as a permutation matrix, triangular factors of a reduced system, or preconditioner information, for use in subsequent `solve` operations. We use the term "solve" to indicate that part of the solution process that takes a specific right-hand side and produces a result. Usually, the results of the "factor" method are used to facilitate or speed the solve process. Depending on the algorithm we employ, the `solve` method could involve triangular back-and-forward substitutions, matrix-vector multiplications, applications of a preconditioner, or permutation of vector components.

We describe possible implementations of `factor` for the formulation (1). One possibility is to apply a symmetric indefinite factorization routine directly to the formulation (14a). The `solve` would use the resulting factors and the permutation matrices to solve (14a) and then substitute into (14b) to recover Δs. Another possible approach is to perform another step of block elimination and obtain a further reduction to the form

$$\begin{bmatrix} Q + C^T Z S^{-1} C & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ -\Delta y \end{bmatrix} = \begin{bmatrix} -r_Q - C^T S^{-1}(Z r_C + r_{z,s}) \\ -r_A \end{bmatrix}. \qquad (15)$$

Again, `factor` could apply a symmetric indefinite factorization procedure to the coefficient matrix in this system. This variant is less appealing than the approach based on (14a), however, since the latter approach allows the factorization routine to compute its own pivot sequence, while in (15) we have partially imposed a pivot ordering on the system by performing the block elimination. However, if the problem (1) contained no equality constraints (that is, A and b null), the approach (15) might make sense, as it would allow a symmetric *positive definite* factorization routine to be applied to the matrix $Q + C^T Z S^{-1} C$.

An alternative approach would be to apply an iterative method such as QMR [13,14] or GMRES [25] (see also Kelley [19]) to the system (14a). Under this scenario, the role of the `factor` routine is limited to choosing a preconditioner. Since some elements of the diagonal matrix $Z^{-1}S$ approach zero while others approach ∞, a diagonal scaling that ameliorates this effect should be part of the preconditioning strategy.

The arguments of `factor` include instances of `Variables` and `Data`, which suffice to define the matrix fully. The information generated by `factor` is stored in the `LinearSystem` class, to be used subsequently by the `solve` method. The `solve` method accepts as input a `Data` object, a `Variables` object containing the current iterate, and a `Residuals` object containing the right-hand side of the linear system to be solved. It uses the information generated by `factor` to solve the linear system and returns an instance of `Variables` that contains the solution. Both `factor` and `solve` are pure virtual functions; their implementation is left to the derived class, since they depend strongly on the problem structure.

3.6. Linear Algebra Classes

In the preceding section, we discussed the structure-independent algorithmic classes of the QP solver layer and the structure-dependent problem-specific classes of the problem formulation layer. None of these classes, however, supplies the behavior that allows the user to perform the linear algebra operations needed to solve an optimization problem. These classes are supplied by the linear algebra layer. The problem formulations supplied with OOQP are written entirely in terms of abstract operation of this layer.

The same basic requirements for linear algebra operations and data structures recur in many different problem formulations. Regardless of the origin of the QP, the `Variable`, `Data`, and `LinearSystem` classes need to perform saxpy, dot product, and norm calculations. Furthermore, many sparse problems need to store and operate on matrices in a Harwell-Boeing format. If we chose to reimplement these linear algebra operations and operations and data structures inside each of the problem-dependent classes wherever they were needed, we would have faced an explosion in the size and complexity of our code.

Our approach to the linear algebra classes is to identify the basic operations that are used repeatedly in our problem-dependent implementations and provide these as methods. As far as possible, we use existing packages such as BLAS, LAPACK, MA27, and PETSc to supply the behavior needed to implement these methods. We are not striving to provide a complete linear algebra package, merely one that is useful in implementing interior-point algorithms. For this reason, we do not implement many BLAS operations, whereas certain operations common to interior-point algorithms, but rare elsewhere, are given equal status with the BLAS-type routines.

The primary abstract classes in the linear algebra layer are `OoqpVector`, `GenMatrix`, and `SymMatrix`, which represent mathematical vectors, matrices, and symmetric matrices, respectively. The `DoubleLinearSolver` class, which represents linear equation solvers, is also part of this layer. Because most of the methods of these classes represent mathematical operations and are named according to the nature of the operation, the interested reader can learn about the range of implemented methods by referring to the source code. We have provided concrete implementations of the linear algebra layer that perform operations on a single processor, using both dense and sparse representations of matrices, and an implementation that uses PETSc to represent vectors and matrices as objects on a distributed system.

3.7. Use of Classes and Layers in OOQP: An Illustration

We now give a specific example of how the three layers in OOQP interact with each other. The `start` method of the `Solver` class implements a heuristic to determine the starting point. Since this particular heuristic has proven to be effective regardless of the specific problem structure, it is part of the QP solver layer—the top layer. The code used to implement this method is as follows.

```
void Solver::start( Variables * iterate, Data * prob,
                    Residuals * resid, Variables * step  )
{
  double sdatanorm = sqrt(prob->datanorm());     /* 1  */
  double a  = sdatanorm, b  = sdatanorm;         /* 2  */

  iterate->interiorPoint( a, b );                /* 3  */
  resid->calcresids( prob, iterate );            /* 4  */
  resid->set_r3_xz_alpha( iterate, 0.0 );        /* 5  */

  sys->factor( prob, iterate );                  /* 6  */
  sys->solve( prob, iterate, resid, step );      /* 7  */
  step->negate();                                /* 8  */

  iterate->saxpy( step, 1.0 );                   /* 9  */
  double shift = 1.e3 + 2*iterate->violation();  /* 10 */
  iterate->shiftBoundVariables( shift, shift );  /* 11 */
}
```

We describe each line in this code by referring to the way in which it would be implemented for the particular formulation (1). We emphasize, however, that the matrices Q, A, and C specific to this formulation appear nowhere in the code for the `start` method. Rather, they are represented by the `prob` variable of the `Data` class. Likewise, this code does not refer to the residuals r_Q, r_A, and r_C but rather has a variable `resids` that represents the residuals with some unspecified structure.

Lines 1 and 2 set the scalar variables `a` and `b` to be the square root of the norm of the data. The norm of the data is computed by invoking the `datanorm` method on `prob`. For formulation (1) the data norm is defined to be magnitude of the largest element in the matrices Q, A, and C and the vectors c, b, and d.

In line 3, we invoke the `interiorPoint` method on `iterate` to compute an appropriate strictly feasible point. For formulation (1), this method call has the effect of setting x and y to zero, all the components of z to a, and all the components of s to b.

The call to `calcresids` in line 4 calculates the value of the residuals of the primal-dual system, through formulae similar to (6c)–(6e). Line 5 sets the complementarity part of the residuals to their affine scaling value, and lines 6–8 solve the affine scaling system, which for (1) has the form (5).

We next invoke the `saxpy` method on `iterate` to take the full affine scaling step, in other words to compute

$$(x, y, z, s) \leftarrow (x, y, z, s) + (\Delta x^{\text{aff}}, \Delta y^{\text{aff}}, \Delta z^{\text{aff}}, \Delta s^{\text{aff}}).$$

This step is likely to result in an iterate that is infeasible. The `violation` method in line 10 calculates the amount by which the variables violate their bounds; for (1) the formula is $\max_{i=1,2,\ldots,m_C} \max(-z_i, -s_i, 0)$. We calculate a shift large enough to make the iterate feasible, and apply this shift by invoking the `shiftBoundVariables` method, for formulation (1) setting $z \leftarrow z +$ `shift` and $s \leftarrow s +$ `shift`.

All the operations used in the `start` method are part of the abstract problem formulation layer. They refer to operations in the problem formulation layer, avoiding altogether references to the specific problem structure. The problem formulation layer is in turn built upon the abstract operations in the linear algebra layer. Take, for example, the implementation of the `negate` method for the formulation (1), which is defined as follows:

```
void QpExampleVars::negate()
{
  x->negate(); y->negate();
  z->negate(); s->negate();
}
```

This method specifically references the fact that the variables have an x, y, z and s component. On the other hand, it makes no reference to how these variables are stored on a computer. They may be all in the core memory of a single processor or distributed across many processors. Managing such low-level details is the responsibility of the linear algebra layer. The problem formulation layer need only invoke abstract operations from this layer, in this case the `negate` method of the `OoqpVector` class.

4. Other Classes

In this section, we describe some useful classes, also provided with OOQP, that don't fit into the framework described in the preceding section.

4.1. Status and Monitor Classes

OOQP is designed to operate both in a stand-alone context and as part of a larger code. Since different termination criteria and different amounts of intermediate output are appropriate to different contexts, we have designed the code to be flexible in these matters. An abstract `Monitor` class is designed to monitor the algorithm's progress, and an abstract `Status` class tests the status of the algorithm after each iteration, checking whether the termination criteria are satisfied.

The two implementations of the `Solver` class in OOQP each provide their own `defaultMonitor` and `defaultStatus` methods. Users who wish to modify the default functionality can simply create a subclass of the `Solver` class that overrides these default implementations. However, since OOQP delegates responsibility for these functions to `Monitor` and `Status` classes, an alternative mechanism is available. Users can create subclasses of `Monitor` and `Status`, redefining the `doIt` method in these classes to carry out the functionality they need.

4.2. `MpsReader` *Class*

The MPS format has been widely used since the 1950s to define linear programming problems. It is an ASCII file format that allows naming of the variables, constraints, objectives, and right-hand sides in a linear program, and assignment of numerical values that define the data objects. Extensions of the format to allow definition of quadratic programs have been proposed by various authors, most notably Maros and Mészáros [22]. The key extension is the addition of a section to the MPS file that defines elements of the Hessian. Though primitive by the standards of modeling languages, MPS remains a popular format for defining linear programming problems, and many test problems are specified in this format.

OOQP includes an `MpsReader` class that reads MPS files. The main input method in the `MpsReader` class is `readQpGen`, which reads a file in the extended MPS format described in [22] into the data structures of the class `QpGenData`, a derived class of `Data` for general sparse quadratic programs. The names assigned to primal and dual variables in the MPS input file are stored for later use in the output method `printSolution`.

5. Implementing Derived Classes for Structured QPs

In this section, we illustrate the use of the OOQP framework in implementing efficient solvers for some highly structured quadratic programming applications. We give a brief description of how some of the derived classes for `Data`, `Variables`, `Residuals`, and `LinearSystem` are implemented in a way that respects the structure of these problem types.

5.1. Huber Regression

Given a matrix $A \in \mathsf{R}^{\ell \times n}$ and a vector $b \in \mathsf{R}^{\ell}$, we seek the vector $x \in \mathsf{R}^n$ that minimizes the objective function

$$\sum_{i=1}^{\ell} \rho((Ax - b)_i), \tag{16}$$

where

$$\rho(t) = \begin{cases} \frac{1}{2}t^2, & |t| \le \tau, \\ \tau|t| - \frac{1}{2}\tau^2, & |t| > \tau, \end{cases}$$

where τ is a positive parameter. The function behaves like a least-squares loss function for small values of the residuals and like the more robust ℓ_1 function for larger residuals, so its minimizer is less sensitive to "outliers" in the data than is the least-squares function. By setting the derivative of (16) to zero, we can formulate this problem as a mixed monotone linear complementarity problem by introducing variables $w, \lambda^1, \lambda^2, \gamma^1, \gamma^2 \in \mathsf{R}^\ell$ and writing

$$\begin{aligned} w - Ax + b + \lambda^2 - \lambda^1 &= 0, & \text{(17a)} \\ A^T w &= 0, & \text{(17b)} \\ \gamma^1 &= w + \tau e, & \text{(17c)} \\ \gamma^2 &= -w + \tau e, & \text{(17d)} \\ \gamma^1 \ge 0 &\perp \lambda^1 \ge 0, & \text{(17e)} \\ \gamma^2 \ge 0 &\perp \lambda^2 \ge 0. & \text{(17f)} \end{aligned}$$

Mangasarian and Musicant [21, formula (9)] show that the conditions (17) are the optimality conditions of the following quadratic program:

$$\begin{aligned} &\min \tfrac{1}{2} w^T w + \tau e^T (\lambda^1 + \lambda^2), & \text{(18a)} \\ &\text{subject to } \; w - Ax + b + \lambda^2 - \lambda^1 = 0, \;\; \lambda^1 \ge 0, \;\; \lambda^2 \ge 0. \end{aligned}$$

Li and Swetits [20] derive an alternative linear program that yields the optimality conditions (17), namely,

$$\min \tfrac{1}{2} w^T w + b^T w, \quad \text{subject to } \; -A^T w = 0, \;\; -\tau e \le w \le \tau e. \qquad (19)$$

Both forms and their relationship are discussed by Wright [27]. Obviously, both have a highly specific structure: The Hessian is simply the identity matrix, the constraint matrix in (18) is sparse and structured, and the bounds in (19) can all be defined by a scalar τ.

OOQP contains an implementation of a solver for this problem in the directory `src/Huber`. The `HuberData` class, derived from `Data`, contains the dimensions of the matrix A and storage for τ as well as A and b. (The structures for both A and b are dense, since these quantities are expected to be dense in most applications.) The `HuberData` class also contains a method `textInput` that reads the contents of A and b from a file in a simple format. For benchmarking purposes, it also contains a method `datarandom` for defining a problem of specified dimensions with random data.

The `HuberVars` structure, which derives from `Variables`, contains vectors of `doubles` to store w, z, λ^1, λ^2, γ^1 and γ^2. The methods for `HuberVars` are defined in a way appropriate to the data structures; for example, μ is calculated as

$$\left[(\lambda^1)^T \gamma^1 + (\lambda^2)^T \gamma^2\right] / (2\ell).$$

In the `Residual` class `HuberResiduals`, four vectors are defined to hold the residuals corresponding to the first four equations in (17), while two more vectors hold residuals corresponding to the complementarity conditions $\lambda_i^1 \gamma_i^1 = 0$, $i = 1, 2, \ldots, \ell$ and $\lambda_i^2 \gamma_i^2 = 0$, $i = 1, 2, \ldots, \ell$, respectively.

The linear systems to be solved at each iteration of the primal-dual algorithm applied to this problem have the following general form:

$$\begin{bmatrix} I & -A & -I & I & 0 & 0 \\ A^T & 0 & 0 & 0 & 0 & 0 \\ -I & 0 & 0 & 0 & I & 0 \\ I & 0 & 0 & 0 & 0 & I \\ 0 & 0 & \Gamma^1 & 0 & \Lambda^1 & 0 \\ 0 & 0 & 0 & \Gamma^2 & 0 & \Lambda^2 \end{bmatrix} \begin{bmatrix} \Delta w \\ \Delta x \\ \Delta \lambda^1 \\ \Delta \lambda^2 \\ \Delta \gamma^1 \\ \Delta \gamma^2 \end{bmatrix} = \begin{bmatrix} r_w \\ r_x \\ r_{\lambda 1} \\ r_{\lambda 2} \\ r_{\gamma 1} \\ r_{\gamma 2} \end{bmatrix}.$$

By performing block elimination, we can reduce to a much smaller system of the form

$$A^T \left(I + (\Gamma^1)^{-1}\Lambda^1 + (\Gamma^2)^{-1}\Lambda^2 \right)^{-1} A \Delta x = \bar{r}_x. \tag{20}$$

(Note that the matrix $I + (\Gamma^1)^{-1}\Lambda^1 + (\Gamma^2)^{-1}\Lambda^2$ is diagonal and therefore easy to form and invert.) The `factor` method in the derived class `HuberLinsys` forms the coefficient matrix in (20) and performs a Cholesky factorization, storing the triangular factor L. The `solve` method performs the corresponding block eliminations on the right-hand side vector to obtain $\bar{r}_z$ in (20), solves this system to obtain Δz, and then recovers the other components of the solution. The cost of each `factor` is $O(n^2\ell + n^3)$, while the cost of each `solve` is $O(n\ell)$. Since n is typically small, both operations are economical.

5.2. Support Vector Machines

The following problem that arises in machine learning (Vapnik [24, Chapter 5]): Given a set of points $x_i \in \mathbb{R}^n$, $i = 1, 2, \ldots, \ell$, where each point is tagged with a label y_i that is either $+1$ or -1, we seek a hyperplane such that all points with label $y_i = +1$ lie on one side of the hyperplane while all points labeled with -1 lie on the other side. That is, we would like the following properties to hold for some $w \in \mathbb{R}^n$ and $\beta \in \mathbb{R}$:

$$y_i = +1 \Leftrightarrow w^T x_i - \beta \; > \; 0; \qquad y_i = -1 \Leftrightarrow w^T x_i - \beta \; < \; 0,$$

or, equivalently, $y_i(w^T x_i - \beta) > 0$, $i = 1, 2, \ldots, \ell$. By scaling w and β appropriately, we see that if such a hyperplane exists, we have without loss of generality that

$$y_i(w^T x_i - \beta) \geq 1, \quad i = 1, 2, \ldots, \ell. \tag{21}$$

If such a plane exists, the data is said to be *separable*. For nonseparable data, one may still wish to identify the hyperplane that minimizes the misclassification in some sense; we would like to have not too many points lying on the wrong side of the hyperplane. One formulation of this problem is as follows [24, p. 137]:

$$\min \tfrac{1}{2} w^T w + Ce^T \xi, \text{ subject to} \tag{22a}$$

$$y_i(w^T x_i - \beta) \geq 1 - \xi_i, \ \ \xi_i \geq 0, \ \ i = 1, 2, \ldots, \ell. \tag{22b}$$

The unknowns are the hyperplane variables (w, β) and the vector $\xi \in \mathsf{R}^\ell$ that measures violation of the condition (21). The positive parameter C weighs our desire to minimize the classification violations against a desire to keep $\|w\|$ of reasonable size. In a typical problem, the dimension n of the space in which each x_i lies is not very large (10–100, say), while the number of points ℓ can be quite large (10^3–10^7). Hence, the problem (22) can be a very large, highly structured quadratic program.

Denoting

$$Y = [y_i x_i]_\ell^{i=1}, \qquad b = [y_i]_\ell^{i=1},$$

we can rewrite (22) as follows:

$$\min_{w,\beta,\xi} \frac{1}{2} w^T w + Ce^T \xi \ \ \text{subject to} \ \ Yw + \xi - \beta b \geq e, \ \ \xi \geq 0. \tag{23}$$

By writing the optimality conditions for this system and applying the usual derivation of the primal-dual equations, we arrive at the following general form for the linear system to be solved at each interior-point iteration.

$$\begin{bmatrix} I & -Y^T & & & & \\ Y & & I & -I & 0 & -b \\ & -I & & & -I & \\ & b^T & & & & \\ & S & & V & & \\ & & T & & \Xi & \end{bmatrix} \begin{bmatrix} \Delta w \\ \Delta v \\ \Delta \xi \\ \Delta s \\ \Delta t \\ \Delta \beta \end{bmatrix} = \begin{bmatrix} r_w \\ r_\beta \\ r_C \\ r_b \\ r_{SV} \\ r_{T\Xi} \end{bmatrix}.$$

By performing successive block eliminations in the usual style, we arrive at a reduced system in the variables Δw and $\Delta \beta$ alone, with the following coefficient matrix:

$$\begin{bmatrix} I + Y^T D Y & -Y^T D b \\ -b^T D Y & b^T D b \end{bmatrix}, \qquad \text{where } D = (V^{-1} S + T^{-1} \Xi)^{-1}.$$

This matrix has dimension $n+1$ and requires $O(n^2 \ell)$ operations to form. It takes $O(n^3 + n\ell)$ operations to solve the reduced system and to recover the eliminated components.

The OOQP distribution contains an implementation of an SVM solver in directory `src/Svm`. The `SvmData` class, a subclass of `Data`, stores the dimensions `hyperplanedim` (n) and `nobservations` (ℓ), the objects Y and b stored as a dense matrix $[Y \,|\, b]$, the object b stored as a vector, and the penalty constant C. It also contains methods to multiply given vectors by $[Y \,|\, b]$ and its transpose, a method to read input from an ASCII file in a simple format, a method to form the inner product of a given vector with b, and methods to generate random data and to print the data objects. The subclass `SvmVars` of `Variables` consists of dense vectors containing w, β, ξ, v, s, and t, together with a method to print the solution, a method to print just the interesting part of the solution (w and β), and the pure virtual methods required by the parent `Variables` class. The subclasses `SvmResiduals` (of `Residuals`) and `SvmLinsys` (of `LinearSystem`) are defined in such as way as to facilitate the approach described in the previous paragraph for solving the linear systems.

5.3. Quadratic Programming with Bound Constraints

Consider the following QP in which the only constraints are upper and lower bounds on selected variables:

$$\min_x \tfrac{1}{2} x^T Q x + c^T x \ \text{ subject to} \tag{24a}$$

$$x_i \geq l_i, \ \ i \in \mathcal{L}, \quad x_i \leq u_i, \ \ i \in \mathcal{U}, \tag{24b}$$

where $\mathcal{L}$ and $\mathcal{U}$ are subsets of $\{1, 2, \dots, n\}$. We define the following row submatrices of I, corresponding to the constraint index sets $\mathcal{L}$ and $\mathcal{U}$:

$$E_{\mathcal{L}} = \left[e_i^T \right]_{i \in \mathcal{L}}, \quad E_{\mathcal{U}} = \left[e_i^T \right]_{i \in \mathcal{U}},$$

where e_i is the vector whose only nonzero element is a "1" in position i. Introducing slack variables s_i, $i \in \mathcal{L}$ for the lower bounds and t_i, $i \in \mathcal{U}$ for the upper bounds, and Lagrange multipliers v_i and z_i for the lower and upper bounds, respectively, we obtain the following optimality conditions:

$$\begin{aligned}
Qx - E_{\mathcal{L}}^T v + E_{\mathcal{U}}^T z &= -c, \\
E_{\mathcal{L}} x - s &= l, \\
E_{\mathcal{U}} x + t &= u, \\
s \geq 0 \perp v &\geq 0, \\
t \geq 0 \perp z &\geq 0,
\end{aligned}$$

where $l = [l_i]_{i \in \mathcal{L}}$, $v = [v_i]_{i \in \mathcal{L}}$, and so on. By writing the general form of the primal-dual linear system and performing the now familiar block elimination process, we arrive at a reduced system in the step Δw whose coefficient matrix is

$$\bar{Q} \overset{\text{def}}{=} Q + E_{\mathcal{L}}^T S^{-1} V E_{\mathcal{L}} + E_{\mathcal{U}}^T T^{-1} Z E_{\mathcal{U}}. \tag{25}$$

The second and third terms are diagonal matrices with nonzero elements occurring at diagonal locations corresponding to $\mathcal{L}$ (for the second term) and $\mathcal{U}$ (for the third term).

The matrix (25) is symmetric and positive semidefinite. One possibility therefore is to solve it with a sparse Cholesky factorization code, modified to allow for small pivots. A second possibility is to apply an iterative method, most suitably a preconditioned conjugate gradient approach. Some of the diagonals in the second and third terms of (25) approach ∞ as we near the solution, and the preconditioner should at a minimum ameliorate this effect. Specifically, defining a diagonal preconditioner D as follows:

$$D_{ii} \stackrel{\text{def}}{=} \max\left(\left(E_{\mathcal{L}}^T S^{-1} V E_{\mathcal{L}} + E_{\mathcal{U}}^T T^{-1} Z E_{\mathcal{U}} \right)_{ii}, 1 \right),$$

and applying the preconditioner symmetrically to obtain

$$D^{-1/2} \bar{Q} D^{-1/2}, \tag{26}$$

we would obtain a matrix that approaches a symmetric permutation of the following:

$$\begin{bmatrix} \hat{Q} & 0 \\ 0 & I \end{bmatrix},$$

where $\hat{Q}$ is the reduced Hessian (the submatrix of Q corresponding to the components of x that are away from their bounds at the solution). An additional level of preconditioning (for example, incomplete Cholesky) could be applied to the matrix in (26) to further enhance the convergence properties of conjugate gradient.

The OOQP distribution contains an implementation of a solver for (24) for a dense Hessian. The `QpBoundData` subclass of `Data` stores Q as a `SymMatrix` object, and c, l, and u as `SimpleVector` objects containing n elements—the bound vectors store even their zero elements. Two other `SimpleVector` objects `index_lower` and `index_upper` of length n represent the information in $\mathcal{L}$ and $\mathcal{U}$; they contain nonzero elements in locations corresponding to the elements of $\mathcal{L}$ and $\mathcal{U}$, respectively. Note that the class `QpBoundData` itself does not mandate a dense storage scheme; only when an instance of this class is created by the method `QpBoundDense::makeData()` is the storage scheme for Q actually defined to be dense. (We could implement an alternative method `QpBoundSparse::makeData()` that uses the same definition of `QpBoundData` but uses a sparse storage scheme for Q instead.)

The `QpBoundData` class also contains a `datarandom()` method to generate a random problem with specified dimension. The `QpBoundVars` subclass of `Variables` stores x, s, t, u, and v as `OoqpVector` objects of size n and uses the `index_lower` and `index_upper` vectors from the `QpBoundData` class to indicate which elements of s, t, u, and v are of real interest. `QpBoundResiduals` is a subclass of `Residuals` that implements the pure virtual methods in a straightforward way, while the `QpBoundLinsys` subclass of `LinearSystem` sets up the matrix (25) as a dense symmetric matrix and uses the LAPACK implementation of Cholesky factorization to solve it.

We have also implemented a solver for a sparse version of (24) that uses iterative methods from the PETSc library to solve the main linear system at each iteration. The PETSc version uses the same problem formulation classes as the dense version: `QpBoundData`, `QpBoundResiduals`, `QpBoundVars`, and `QpBoundLinsys`. It uses, however, a completely different linear algebra layer (see Section 3).

6. OOQP Distribution

The OOQP distribution archive can be obtained from

`http://www.cs.wisc.edu/~swright/ooqp/`

To install OOQP on a Unix system, follow the download procedure to obtain a gzipped tar file, and unpack to obtain a directory `OOQP`. Refer to the file `INSTALL` in this directory for information on setting up the environment required by OOQP (for example, ensuring that a BLAS library is available and obtaining the MA27 package from the HSL Archive) and building OOQP executables for the various solvers. The `README` file contains basic information about the contents of the distribution directory, the problems solved, and locations of the documentation. In particular, by pointing a browser at the file `doc/index.html`, one can obtain pointers to comprehensive documentation of various types.

By default, the configure-make process builds executables for the following solvers:

- two solvers for general sparse QPs, using Mehrotra's original algorithm and Gondzio's variant, respectively, and solving linear equations with MA27 in both cases;
- two solvers for general dense QPs, using Mehrotra's original algorithm and Gondzio's variant, respectively;
- a solver for the QP with bounds described in Section 5.3, with dense Hessian;
- a solver for Huber regression, described in Section 5.1;
- a solver for QPs arising from support vector machines, described in Section 5.2.

Interfaces that make some of the functionality of these solvers available via the AMPL modeling language and MATLAB are also included in the distribution but are not configured in the default build process. For AMPL, a solver for general sparse QP is available; instructions for building this solver are included in the `INSTALL` file. The OOQP distribution provides MATLAB functionality for reading MPS input files, calling a solver for general sparse QP, and calling solvers for SVM and Huber regression problems. Instructions for building the MATLAB interface can be found in the file `README_Matlab`.

Acknowledgments

We thank Jeff Linderoth, who collaborated on this project during his time at Argonne in 1998–2000, and Nate Brixius and Bjarni Halldorsson, who contributed to the early stages of this project in the summer of 1999. We also thank Iain Duff for his advice regarding the MA27 and MA57 codes, Alex Pothen for supplying Oblio, and Hans Mittelmann for his efforts in benchmarking this and many other optimization codes.

References

1. S. Balay, W. Gropp, L. Curfman McInnes, and B. Smith. *PETSc Users Manual.* Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, Ill. 60439, April 2001.
2. R. Bartlett. An introduction to rSQP++: An object-oriented framework for reduced-space successive quadratic programming. Report, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, Penn., October 1996.
3. S. Benson, L. Curfman McInnes, and J. J. Moré. TAO users manual. Technical Memorandum ANL/MCS-TM-249, Argonne National Laboratory, Argonne, Ill. 60439, March 2001.
4. A. M. Bruaset and H. P. Langtangen. Object-oriented design of preconditioned iterative methods in Diffpack. *ACM Transactions on Mathematical Software*, 23(1):50–80, 1997.
5. E. Chow and M. A. Heroux. An object-oriented framework for block preconditioning. *ACM Transactions on Mathematical Software*, 24(2):159–183, 1998.
6. J. Czyzyk, S. Mehrotra, M. Wagner, and S. J. Wright. PCx: An interior-point code for linear programming. *Optimization Methods and Software*, 11/12:397–430, 1999.
7. J. W. Demmel, J. R. Gilbert, and X. S. Li. *SuperLU User's Guide*, 1999. Available from `www.nersc.gov/ xiaoye/SuperLU/`.
8. H. L. Deng, W. Gouveia, and J. Scales. The CWP object-oriented optimization library. Technical report, Center for Wave Phenomena, Colorado School of Mines, June 1994.
9. F. Dobrian, G. Kumfert, and A. Pothen. The design of sparse direct solvers using object-oriented techniques. In A. M. Bruaset, H. P. Langtangen, and E. Quak, editors, *Modern Tools in Scientific Computing.* Springer-Verlag, 2000.
10. F. Dobrian and A. Pothen. Oblio: A sparse direct solver library for serial and parallel computations. Technical report, Department of Computer Science, Old Dominion University, 2000.
11. Iain S. Duff and J. K. Reid. MA27 – A set of Fortran subroutines for solving sparse symmetric sets of linear equations. Technical Report AERE R10533, AERE Harwell Laboratory, London, England, 1982.
12. M. C. Ferris and T. S. Munson. Interior-point methods for massive support vector machines. Data Mining Institute Technical Report 00-05, Computer Sciences Department, University of Wisconsin, Madison, May 2000.
13. R. Freund. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM Journal on Scientific Computing*, 14:470–482, 1993.
14. R. Freund and N. Nachtigal. QMR: A quasi-minimal residual method for non-Hermitian linear systems. *Numerische Mathematik*, 60:315–339, 1991.
15. E. M. Gertz and S. J. Wright. *OOQP User Guide.* Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., September 2001. Available from `http://www.cs.wisc.edu/~swright/OOQP/`.
16. M. Gockenbach and W. Symes. An overview of HCL1.0. *ACM Transactions on Mathematical Software*, 25:191–212, 1999.
17. J. Gondzio. Multiple centrality corrections in a primal-dual method for linear programming. *Computational Optimization and Applications*, 6:137–156, 1996.
18. HSL: A collection of Fortran codes for large scale scientific computation, 2000. Full details in http://www.numerical.rl.ac.uk/hsl.
19. C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations.* Number 16 in Frontiers in Applied Mathematics. SIAM Publications, Philadelphia, 1995.
20. W. Li and J. J. Swetits. The linear ℓ_1 estimator and the Huber M-estimator. *SIAM Journal on Optimization*, 8:457–475, 1998.
21. O. L. Mangasarian and D. R. Musicant. Robust linear and support vector machines. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(9):1–6, 2000.
22. I. Maros and C. Mészáros. A repository of convex quadratic programming problems. *Optimization Methods and Software*, 11 and 12:671–681, December 1999.
23. S. Mehrotra. On the implementation of a primal-dual interior point method. *SIAM Journal on Optimization*, 2:575–601, 1992.
24. V. N. Vapnik. *The Nature of Statistical Learning Theory.* Statistics for Engineering and Information Science. Springer, second edition, 1999.
25. H. Walker. Implementation of the GMRES method using Householger transformations. *SIAM Journal on Scientific and Statistical Computing*, 9:815–825, 1989.

26. S. J. Wright. *Primal-Dual Interior-Point Methods.* SIAM Publications, Philadelphia, 1997.
27. S. J. Wright. On reduced convex QP formulations of monotone LCPs. *Mathematical Programming*, 90:459–473, 2001.