Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-252

OOQP User Guide[*]

by

E. Michael Gertz[†] *and Stephen Wright*[‡]

Mathematics and Computer Science Division

Technical Memorandum No. 252

October 2001
Updated May 2004

[*]This work was 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 National Science Foundation Grants CDA-9726385 and ACI-0082065.

[†]Also Electrical and Computer Engineering Department, Northwestern University, Evanston, IL 60208; `gertz@ece.nwu.edu`

[‡]Computer Sciences Department, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, WI 53706; `swright@cs.wisc.edu`

Contents

OOQP User Guide

by

E. Michael Gertz and Stephen J. Wright

Abstract

OOQP is an object-oriented software package for solving convex quadratic programming problems (QP). We describe the design of OOQP, and document how to use OOQP in its default configuration. We further describe OOQP as a development framework, and outline how to develop custom solvers that solve QPs with exploitable structure or use specialized linear algebra.

1 Introduction

OOQP is a package for solving convex quadratic programming problems (QPs). These are optimization problems in which the objective function is a convex quadratic function and the constraints are linear functions of a vector of real variables. They have the following general form:

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

where Q is a symmetric positive semidefinite $n \times n$ matrix,; $x \in \mathsf{R}^n$ is a vector of variables; A and C are matrices of dimensions $m_a \times n$ and $m_c \times n$, respectively; and c, b, and d are vectors of appropriate dimensions.

Many QPs that arise in practice are highly structured. That is, the matrices that define them have properties that can be exploited in designing efficient solution techniques. For example, they may be general sparse matrices; diagonal, banded, or block-banded matrices; or low-rank matrices. A simple and common instance of structure occurs in applications in which the inequality constraints include simple upper or lower bounds on some components of x; the rows of C defining these bounds each contain a single nonzero element. A more extreme example of exploitable structure occurs in the QPs that arise in support vector machines. In one formulation of this problem, Q is dense but is a low-rank perturbation of a positive diagonal matrix.

In addition to the wide variations in problem structure, there is wide divergence in the ways in which the problem data and variables for a QP can be stored on a computer. Part of this variation may be due to the structure of the particular QP: it makes sense to store the problem data and variables in a way that is natural to the application context in which the QP arises, rather than shoehorning it into a form that is convenient for the QP software. Variations in storage schemes arise also because of different storage conventions for sparse matrices; because of the ways that matrices and vectors are represented on different parallel platforms; and because large data sets may necessitate specialized out-of-core storage schemes.

Algorithms for QP, as in many other areas of optimization, depend critically on linear algebra operations of various types: matrix factorizations, updates, vector inner products and “saxpy” operations. Sophisticated software packages may be used to implement the required linear algebra operation in a manner that is appropriate both to the problem structure and to the underlying hardware platform.

One might expect this wide variation in structure and representation of QPs to give rise to a plethora of algorithms, each appropriate to a specific situation. Such is not the case. Algorithms such as gradient projection, active set, and interior point all appear to function well in a wide variety of circumstances. Interior-point methods in particular appear to be competitive in terms of efficiency on almost all problem types, provided they are coded in a way that exploits the problem structure.

In OOQP, *object-oriented programming* techniques are used to implement a primal-dual interior-point algorithm in terms of abstract operations on abstract objects. Then, at a lower level of the code, the abstract operations are specialized to particular problem formulations and data representations. By reusing the top-level code across the whole space of applications, while exploiting structure and hardware capabilities at the lower level to produce tuned, concrete implementations of the abstract operations, users can produce efficient, specialized QP solvers with a minimum of effort.

This distribution of OOQP contains code to fully implement a solver for a number of standard OOQP formulations, including a version of the formulation (1) that assumes Q, A, and C to be general sparse matrices. The code in the distribution also provides a framework and examples for users who wish to implement solvers that are tailored to specific structured QPs and specific computational environments.

1.1 Different Views of OOQP

In this section, we describe different ways in which OOQP may be used.

Shrink-Wrapped Solution. The OOQP distribution can be used as an off-the-shelf, shrink-wrapped solver for QPs of certain types. Users can simply install it and execute it on their own problem data, without paying any attention to the structure of the code or the algorithms behind it. In particular, there is an implementation for solving general QPs (of the form (2) given in Section 2) in which the data matrices are sparse without any assumed structure. (The linear algebra calculations in the distributed version are performed with the codes MA27 [9], but we have also implemented versions that use MA57 [14], Oblio [7], and SuperLU [6].) The distribution also contains an implementation for computing a support vector machine to solve a classification problem; an implementation for solving the Huber regression problem; and an implementation for solving a quadratic program with simple bounds on a distributed platform, using PETSc [4]. These implementations each may be called via a command-line executable, using ascii input files for defining the data in a manner appropriate to the problem. Some of the implementations can also be called via the optimization modeling language AMPL or via MATLAB.

See the `README` file in the distribution for further details on the specialized implementations included in the distribution.

Embeddable Solver. Some users may wish to embed OOQP code into their own applications, calling the QP solver as a subroutine. This mode of use is familiar to users of traditional optimization software packages and numerical software libraries such as NAG or IMSL. The OOQP distribution supplies C and C++ interfaces that allow the users to fill out the data arrays for the formulation (2) themselves, then call the OOQP solver as a subroutine.

Development Framework. Some users may wish to take advantage of the development framework provided by OOQP to develop QP solvers that exploit the structure of specific problems. OOQP is an extensible C++ framework; and by defining their own specialized versions of the storage schemes and the abstract operations used by the interior-point algorithm, users may customize the package to work efficiently on their own applications.

Users may also modify one of the default implementations in the distribution by replacing the matrix and vector representations and the implementations of the abstract operations by their own variants. For example, a user may wish to replace the code for factoring symmetric indefinite matrices (a key operation in the interior-point algorithms) with some alternative sparse factorization code. Such replacements can be performed with relative ease by using the default implementation as an exemplar.

Research Tool. Researchers in interior point-methods for convex quadratic programming problems may wish to modify the algorithms and heuristics used in OOQP. They can do so by modifying the top-level code, which is quite short and easy to understand. Because of the abstraction and layering design features of OOQP, they will then be able to see the effect of their modifications on the whole gamut of QP problem structures supported by the code.

1.2 Obtaining OOQP

The OOQP Web page `www.cs.wisc.edu/~swright/ooqp/` has instructions on downloading the distribution. OOQP is also distributed by the Optimization Technology Center (OTC). See the page `www.ece.nwu.edu/OTC/software/` for information on obtaining OOQP and other OTC software.

Unpacking the distribution will create a single directory called `OOQP-X.XX`, where `X.XX` is the revision number. For simplicity, we will refer to this directory simply as `OOQP` throughout this document. The `OOQP` directory contains numerous files and subdirectories, which are discussed in detail in this manual. Whenever we refer to a particular directory in the text, we mean it to be taken as a subdirectory of `OOQP`. For example, when we discuss the directory `src/QpGen`, we mean `OOQP/src/QpGen`.

1.3 How to Read This Manual

This manual gives an overview of OOQP—its structure, the algorithm on which it is based, the ways in which the solvers can be invoked, and its utility as a development framework.

Section 2 is intended for those who wish to use the solver for general sparse quadratic programs (formulation (2)) that is provided with the OOQP distribution. It shows how to define the problem and invoke the solver in various contexts. Section 3 gives an overview of the OOQP development framework, explaining the basics of the layered design and details of the directory structure and makefile-based build process. Section 4 provides additional details on the top layer of OOQP—the QP solver layer—for the benefit of those who wish to experiment with variations on the two primal-dual interior-point algorithms supplied with the OOQP distribution. Section 5 describes the operations that must be defined and implemented in order to create a solver for a new problem formulation. Section 6 is a practical tutorial on OOQP's linear algebra layer. It describes the classes for vectors and sparse and dense matrices for the benefit of users who wish to use these classes in creating solvers for their own problem formulations. Finally, Section 7 is intended for advanced users who wish to specialize the linear algebra operations of OOQP by adding new linear solvers or using different matrix and vector representations.

Users who simply wish to use OOQP as a shrink-wrapped solver for quadratic programs formulated as general sparse problems (2) need read only Section 2. Those interested in learning a little more about the design of OOQP should read Sections 3.1 and 4.1, while those who wish to understand the design and motivation more fully should read Sections 3.1, 4, 5, and 6, in that order. Users who wish to implement a solver for their own QP formulation should read Sections 3, 4.1, 5, and 6 and then review Section 5 with code in hand. Those who wish to install new linear solvers should read Sections 2, 3, 6, and then focus on Section 7.

1.4 Other Resources

OOQP is distributed with additional documentation. In the top-level OOQP directory, the file README describes the contents of the distribution. This file includes the location of an html page that serves as an index of available documentation and may be viewed through a browser such as Netscape. This documentation includes the following items.

Online Reference Manual. We have extensively documented the source code , using the tool `doxygen` to create a set of html pages that serve as a comprehensive reference manual to the OOQP development framework. Details of the class hierarchy, the purposes of the individual data structures and methods within each class, and the meanings of various parameters are explained in these pages.

A Descriptive Paper. The archival paper [12] by the authors of OOQP contains further discussion of the motivation for OOQP, the structure of the code, and the way in which the classes are reimplemented for various specialized applications.

Manuals for Other Problem Formulations. Specialized QP formulations such as Svm, Huber, and QpBound have their own documentation. The documents describe the problems solved and how the solvers may be invoked.

OOQP Installation Guide. This document describes how to build and install OOQP.

Distribution Documents. These include files such as README that describe the contents of various parts of the distribution.

We also supply a number of sample problems and example programs in the `examples/` subdirectory. A README file in this subdirectory explains its contents.

2 Using the Default QP Formulation

The "general" quadratic programming formulation recognized by OOQP is as follows:

$$\begin{aligned} &\min \tfrac{1}{2}x^TQx + c^Tx \ \text{ subject to} \\ &Ax = b, \quad d \le Cx \le f, \quad l \le x \le u, \end{aligned} \tag{2}$$

where Q is an $n \times n$ positive semidefinite matrix, A is an $m_a \times n$ matrix, C is an $m_c \times n$ matrix, and all other quantities are vectors of appropriate size. Some of the elements of l, u, d, and f may be infinite in magnitude; that is, some components of Cx and x may not have upper and lower bounds.

The subdirectory `src/QpGen` in the OOQP distribution, together with the linear algebra subdirectories, contains code for solving problems formulated as (2), where Q, A, and C are general sparse matrices. In this section, we describe the different methods that can be used to define the problem data and, accordingly, different ways in which the solver can be invoked. We start with a command-line interface that can be used when the problem is defined via a text file (Section 2.1). We then describe several other interfaces: calling OOQP as a function from a C program (Section 2.2); calling it from a C++ program (Section 2.3); invoking OOQP as a solver from an AMPL process (Section 2.4); and invoking OOQP as a subroutine from a MATLAB program (Section 2.5).

2.1 Command-Line Interface

When the problem is defined in quadratic MPS ("QPS") format in an ascii file, the method of choice for solving the problem is to use an executable file that applies Mehrotra's predictor-corrector algorithm [19] with Gondzio's multiple corrections [13]. (The Installation Guide that is supplied with the OOQP distribution describes how to create this executable file, which is named `qpgen-sparse-gondzio.exe`.) We also provide `qpgen-sparse-mehrotra.exe`, an implementation of Mehrotra's algorithm that does not use Gondzio's corrections. These executables take their inputs from a text file in QPS format that describes the problem.

The QPS format proposed by Maros and Mészáros [17] is a modification of the standard and widely used MPS format for linear programming. The format is somewhat awkward and limited in the precision to which it can specify numerical data. We support it, however, because it is used by a number of QP solvers and is well known to users of optimization software.

A description of the MPS format, extracted from Murtagh [20], can be found at the NEOS Guide at

`www.mcs.anl.gov/otc/Guide/`

(search for "MPS"). The QPS format extends MPS by introducing a new section of the input file named `QUADOBJ` (alternatively named `QMATRIX`), which defines the matrix Q of the quadratic objective specified in the formulation (2). This section, if present, must appear after all other sections in the input file. The format of this section is the same as the format of the `COLUMNS` section except that only the lower triangle of Q is stored. As in the `COLUMNS` section, the nonzeros are specified in column major order.

We have relaxed the MPS definition so that strict limitations on field widths on each line are replaced by tokenization, in which fields are assumed to be separated by spaces. (Note that this parsing may introduce incompatibilities with files that are valid under the strict MPS definition, in which spaces may occur within a single numerical field between a minus sign and the digits it operates on.) Name records may now be up to 16 characters in length, and there is no restriction on the size of numerical fields, except those imposed by the maximum length of a line. The maximum line length is 150 characters.

A second deviation from MPS standard format is that an objective sense indicator may be introduced at the start of the file, to indicate either that the specified objetcive is to be minimized or maximized. This field has the form

```
OBJSENSE
MIN
```

when the intent is to minimize the objective, and

```
OBJSENSE
MAX
```

for maximization. The default is minimization. If this field is included, it must appear immediately after the `NAME` line.

Figure 1 shows a sample QPS file, taken from Maros and Mészáros [17]. This file describes the following problem:

$$\begin{array}{lcccl} \text{mimimize} & \multicolumn{4}{l}{4 + 1.5x_1 - 2x_2 + \frac{1}{2}(8x_1^2 + 4x_1x_2 + 10x_2^2)} \\ \text{subject to} & 2 & \leq & 2x_1 + x_2 & \leq \infty \\ & -\infty & \leq & -x_1 + 2x_2 & \leq 6 \\ & 0 & \leq & x_1 & \leq 20 \\ & 0 & \leq & x_2 & \leq \infty. \end{array} \tag{3}$$

If the file (1) is named `Example.qps` and is stored in the subdirectory `data`, and if the executable `qpgen-sparse-gondzio.exe` appears in the `OOQP` directory, then typing

```
qpgen-sparse-gondzio.exe ./data/Example.qps
```

will solve the problem and create the output file `OOQP/data/Example.out`.

Figure 2 shows the contents of `Example.out`. The `PRIMAL VARIABLES` are the components of the vector x in the formulation (2). The output shows that the optimal values are $x_1 = 0.7625$ and $x_2 = 0.475$. The bounds on each component of x, if any were specified, are also displayed. If neither bound is active, the reported value of the Lagrange multiplier in the final column should be close to zero. Otherwise, it may take a positive value when the lower bound is active or a negative value when the upper bound is active.

```
NAME          Example
ROWS
 N  obj
 G  r1
 L  r2
COLUMNS
    x1        r1                 2.0   r2                -1.0
    x1        obj                1.5
    x2        r1                 1.0   r2                 2.0
    x2        obj               -2.0
RHS
    rhs1      obj               -4.0
    rhs1      r1                 2.0   r2                 6.0
BOUNDS
 UP bnd1      x1                20.0
QUADOBJ
    x1        x1                 8.0
    x1        x2                 2.0
    x2        x2                10.0
ENDATA
```

Figure 1: A sample QPS file

The `CONSTRAINTS` section shows the values of the vectors Ax and Cx at the computed solution x, compares these values with their upper and lower bounds in the case of Cx, and displays Lagrange multiplier information in the final column, in a similar way to the `PRIMAL VARIABLES` section.

Note that the problem described in (1) contains no equality constraints (that is, A is null), so there is no `Equality Constraints` subsection in the `CONSTRAINTS` section of this particular output file.

A number of command-line options are available in calling `qpgen-sparse-gondzio.exe`. The current list of options can be seen by typing

```
qpgen-sparse-gondzio.exe --help
```

Current options are as follows:

`--print-level num`**:** (`num` is a positive integer) Larger values of `num` produce more output to the screen.

`--version`**:** shows the current version number and release date.

`--quiet`**:** suppresses output to the screen.

`--verbose`**:** produces maximal output to the screen.

`--scale`**:** scales the variables so that the diagonals of the Hessian matrix remain in a reasonable range.

```
Solution for 'Example '

Rows: 3,  Columns: 2

PRIMAL VARIABLES

  Name  Value           Lower Bound  Upper Bound  Multiplier

0 x1    7.62500000e-01  0.00000e+00  2.00000e+01  6.37776644e-15
1 x2    4.75000000e-01  0.00000e+00               2.83645544e-12


CONSTRAINTS

Inequality Constraints: 2

  Name  Value           Lower Bound  Upper Bound  Multiplier

0 r1    2.00000000e+00  2.00000e+00               4.27500000e+00
1 r2    1.87500000e-01               6.00000e+00  -8.81876986e-16

Objective value: 8.37188
```

Figure 2: Sample output from `qpgen-sparse-gondzio.exe`

The same options are available for `qpgen-sparse-mehrotra.exe`.

2.2 Calling from a C Program

OOQP supplies an interface to the default solver for (2) that may be called from a C program. This operation is performed by the function `qpsolvesp`, which has the following prototype.

```
void
qpsolvesp(          double    c[],   int      nx,
    int   irowQ[],  int    nnzQ,     int  jcolQ[],  double dQ[],
    double xlow[],  char  ixlow[],
    double xupp[],  char  ixupp[],
    int   irowA[],  int     nnzA,    int  jcolA[],  double dA[],
    double   bA[],  int       my,
    int   irowC[],  int     nnzC,    int  jcolC[],  double dC[],
    double clow[],  int       mz,    char iclow[],
    double cupp[],  char   icupp[],
    double    x[],  double gamma[],  double phi[],
    double    y[],
    double    z[],  double lambda[],  double pi[],
    double   *objectiveValue,
    int print_level, int * ierr );
```

This function uses an old-fashioned calling convention in which each argument is a native type (for example, an `int` or an array of `double`). While calling such a function can be tedious because of the sheer number of arguments, it is straightforward in that the relationship of each argument to the formulation (2) is fairly easy to understand.

Sparse matrices are represented by three data structures—two integer vectors and one vector of doubles, all of the same length. For the (general) matrix A, these data structures are `irowA`, `jcolA` and `dA`. The total number of nonzeros in the sparse matrix `A` is `nnzA`. The k nonzero element of `A` occurs at row `irowA[k]` and column `jcolA[k]` and has value `dA[k]`. Rows and columns are numbered starting at zero.

For the symmetric matrix Q, only the elements of the lower triangle of the matrix are specified in `irowQ`, `jcolQ`, and `dQ`.

The elements of each matrix must be sorted into row-major order before `qpsolvesp` is called. While this requirement places an additional burden on the user, it reduces the memory requirements of the `qpsolvesp` procedure significantly. OOQP provides a routine `doubleLexSort` that the user may call to sort the matrix elements in the correct order. To sort the elements of the matrix A, this routine can be invoked as follows:

```
doubleLexSort( irowA, nnzA, jcolA, dA )
```

We now show the correspondence between the input variables to `qpsolvesp` (which are not changed by the routine) and the formulation (2).

`c` is the linear term in the objective function, a vector of length `nx`.

`nx` is the number of primal variables, that is, the length of the vector x in (2). It is the length of the input vectors `c`, `xlow`, `ixlow`, `xupp`, `ixupp`, `x`, `gamma`, and `phi`.

`irowQ, jcolQ, dQ` hold the `nnzQ` lower triangular elements of the quadratic term of the objective function.

`xlow, ixlow` are the lower bounds on `x`. These contain the information in the lower bounding vector l in (2). If there is a bound on element k of x (that is, $l_k > -\infty$), then `xlow[k]` should be set to the value of l_k and `ixlow[k]` should be set to one. Otherwise, element k of both arrays should be set to zero.

`xupp, ixupp` are the upper bounds on `x`, that is, the information in the vector u in (2). These should be defined in a similar fashion to `xlow` and `ixlow`.

`irowA, jcolA, dA` are the `nnzA` nonzero elements of the matrix A of linear equality constraints.

`bA` contains the right-hand-side vector b for the equality constraints in (2). The integer parameter `my` defines the length of this vector.

`clow, iclow` are the lower bounds of the inequality constraints.

`cupp, icupp` are the upper bounds of the inequality constraints.

`print_level` controls the amount of output the solver prints to the terminal. Larger values of `print_level` cause more information to be printed. The following values of `print_level` are recognized:

0 operate silently.

≥ 10 print information about each interior point iteration.

≥ 100 print information from the linear solvers.

The remaining parameters are output parameters that hold the solution to the QP. The variables `objectiveValue` and `x` hold the values of interest to most users, which are the minimal value and solution vector x in (2). The parameter `ierr` indicates whether the solver was successful. The solver will return a nonzero value in `ierr` if it was unable to solve the problem. Negative values indicate that some error, such as an out of memory error, was encountered. For a description of the termination criteria of OOQP, and the positive values that might be returned in `ierr`, see Section 4.3.

The remaining output variables are vectors of Lagrange multipliers; the array `y` contains the Lagrange multipliers for the equality constraints $Ax = b$, while `lambda` and `pi` contain multipliers for the inequality constraints $Cx \geq d$ and $Cx \leq f$, respectively. The output variable `z` should satisfy

$$z = \lambda - \pi.$$

The multipliers for the lower and upper bounds $x \geq l$ and $x \leq u$, are contained in `gamma` and `phi`, respectively. Among other requirements (see our discussion of optimality conditions in the next section), these vectors should satisfy the following relationship on output:

$$c + Qx - A^T y - C^T z - \gamma + \phi = 0.$$

Because it is somewhat cumbersome to allocate storage for all the parameters of `qpsolvesp` individually, OOQP provides the following routine to perform all necessary allocations:

```
void
newQpGenSparse( double ** c,      int nx,
    int    ** irowQ,  int nnzQ,  int  ** jcolQ,  double ** dQ,
    double ** xlow,              char ** ixlow,
    double ** xupp,              char ** ixupp,
    int    ** irowA,  int nnzA,  int  ** jcolA,  double ** dA,
    double ** b,      int my,
    int    ** irowC,  int nnzC,  int  ** jcolC,  double ** dC,
    double ** clow,   int mz,    char ** iclow,
    double ** cupp,              char ** icupp,
    int    *  ierr );
```

The following routine frees all this allocated storage:

```
void
freeQpGenSparse( double ** c,
    int    ** irowQ,  int  ** jcolQ,  double ** dQ,
    double ** xlow,   char ** ixlow,
    double ** xupp,   char ** ixupp,
    int    ** irowA,  int  ** jcolA,  double ** dA,
    double ** b,
    int    ** irowC,  int  ** jcolC,  double ** dC,
    double ** clow,   char ** iclow,
    double ** cupp,   char ** icupp );
```

If `newQpGenSparse` succeeds, it returns `ierr` with a value of zero. Otherwise, it sets `ierr` to a nonzero value and frees any memory that it may have allocated to that point. We emphasize that users are not required to use these two routines; users can allocate arrays as they choose.

The distribution also contains a variant of `qpsolvesp` that accepts sparse matrices stored in the slightly more compact Harwell-Boeing sparse format (see Duff, Erisman, and Reid [8]), rather than the default sparse format described above. In the Harwell-Boeing format, the nonzeros are stored in row-major form, with `jcolA[l]` and `dA[l]` containing the column index and value of the l nonzero element, respectively. The integer vector `krowA[k]` indicates the index in `jcolA` and `dA` at which the first nonzero element for row `k` is stored; its final element `krowA[my+1]` points to the index in `jcolA` and `dA` immediately after the last nonzero entry. See [8] and Section 6.4 below for further details. The Harwell-Boeing version of `qpsolvesp` has the following prototype.

```
void
qpsolvehb( double    c[],  int  nx,
    int   krowQ[],  int  jcolQ[],  double dQ[],
    double xlow[],  char ixlow[],
    double xupp[],  char ixupp[],
    int   krowA[],  int  my,       int  jcolA[],  double dA[],
    double   bA[],
    int   krowC[],  int  mz,       int  jcolC[],  double dC[],
    double clow[],  char iclow[],
    double cupp[],  char icupp[],
    double    x[],  double gamma[],     double phi[],
    double    y[],
    double    z[],  double lambda[],  double pi[],
    double   *objectiveValue,
    int print_level, int * ierr );
```

The meaning of the parameters other than those that store the sparse matrices is identical to the case of `qpsolvesp`.

The prototypes of the preceding routines are located in the header file `cQpGenSparse.h`. Most users will need to include the line

```
#include "cQpGenSparse.h"
```

in their program. This header file is safe to include not only in a C program but also in a C++ program. Users who need more control over the solver than these functions provide should develop a C++ interface to the solver.

We refer users to the Installation Guide in the distribution for further information on building the executable using the OOQP header files and libraries.

2.3 Calling from a C++ Program

When calling OOQP from a C++ code, the user must create several objects and call several methods in sequence. The process is more complicated than simply calling a C function, but also more flexible. By varying the classes of the objects created, one can generate customized solvers for QPs of various types. In this section, we focus on the default solver for the formulation (2). The full sequence of calls for this case is shown in Figure 3. In the remainder of this section, we explain each call in this sequence in turn.

```
QpGenSparseMa27 * qp
    = new QpGenSparseMa27( nx, my, mz, nnzQ, nnzA, nnzC );

QpGenData      * prob
    = (QpGenData * ) qp->makeData( /* parameters here */);
QpGenVars      * vars
    = (QpGenVars *) qp->makeVariables( prob );
QpGenResiduals * resid
    = (QpGenResiduals *) qp->makeResiduals( prob );

GondzioSolver  * s     = new GondzioSolver( qp, prob );

s->monitorSelf();
int status = s->solve(prob,vars, resid);
```

Figure 3: The basic sequence for calling OOQP

The first method call in this sequence initializes a new problem formulation `qp` of class `QpGenSparseMa27`, which is a subclass of `ProblemFormulation`. The definition of this class determines how the problem data will be stored, how the problem variables will be stored and manipulated, and how linear systems will be solved. Our subclass `QpGenSparseMa27` implements the problem formulation (2), where the sparse matrices defining the problem are stored in sparse (not dense) matrices and that large linear systems that define steps of the interior-point method will be solved by using the `MA27` package from the Harwell Subroutine Library.

In the next method call in Figure 3, the `makeData` method in the object `qp` created in the first call creates the vectors and matrices that contain the problem data. In fact, `qp` contains different versions of the `makeData` method, which may be distinguished by their different parameter lists. Users whose matrix data is in row-major Harwell-Boeing sparse format may use the following form of this call.

```
QpGenData * prob
    = (QpGenData * ) qp->makeData( c,      krowQ,  jcolQ,  dQ,
                                   xlow,   ixlow,  xupp,   ixupp,
                                   krowA,  jcolA,  dA,     bA,
                                   krowC,  jcolC,  dC,
                                   clow,   iclow,  cupp, icupp);
```

(The meaning of the parameters is explained in Section 2.2 above.) In this method, data structure references in `prob` are set to the actual arrays given in the parameter list. This choice avoids copying of the data, but it requires that these arrays not be deleted until after deletion of the object `prob`.

For users whose data is in sparse triple format, a special version of `makeData` named `copyDataFromSparseTriple` may be called as follows.

```
QpGenData * prob
    = (QpGenData * ) qp->copyDataFromSparseTriple(
        c,      irowQ,  nnzQ,   jcolQ,  dQ,
        xlow,   ixlow,  xupp,   ixupp,
        irowA,  nnzA,   jcolA,  dA,     bA,
        irowC,  nnzC,   jcolC,
        clow,   iclow,  cupp,   icupp );
```

(The meaning of the parameters is explained in Section 2.2.) In this method, since the data objects in the argument list are actually copied into `prob`, they may be deleted immediately after the method returns.

There distribution includes several other version of `makeData` that will not be described here. In general, the preference is to fix references in `prob` to point to existing arrays of data, rather than copying the data into `prob`.

The calls to `makeVariables` and `makeResiduals` in Figure 3 create the objects that store the problem variables and the residuals that measure the infeasibility of a given point with respect to the various optimality conditions. The object `vars` contains both primal variables for (2) (including x) and dual variables (Lagrange multipliers). These variables are named `vars->x`, `vars->y`, and so on, following the naming conventions described in Section 2.2. The data and methods in the residuals class `resids` are typically of interest only to optimization experts. When an approximate solution to the problem (2) is found, all data elements in this object will have small values, indicating that the point in question approximately satisfies all optimality conditions.

The next step is to create the solver object for actually solving the QP. This is performed by means of the following call.

```
GondzioSolver  * s     = new GondzioSolver( qp, prob );
```

In our example, we then invoke the method `s->monitorSelf()` to tell the solver that it should print summary information to the screen as the solver is operating. (If this line is omitted, the solver will operate quietly.)

Finally, we invoke the algorithm to solve the problem by means of the call `s->solve(prob,vars, resid)`. The return value from this routine will be zero if the solver was able to compute an approximate solution, which will be found in the object `vars`. The solver will return a nonzero value if it was unable to solve the problem. Negative values indicate that some error, such as an out of memory error, was encountered. For a description of the termination criteria of OOQP, and the positive values that might be returned in `ierr`, see Section 4.3.

One must include certain header files to obtain the proper definitions of the classes used. In general, a class definition is in a header file with the same name as the class, appended with a ".h". For the example in Figure 3, the following lines serve to include all relevant header files.

```
#include "QpGenData.h"
#include "QpGenVars.h"
#include "QpGenResiduals.h"
#include "GondzioSolver.h"
#include "QpGenSparseMa27.h"
```

The OOQP Installation Guide explains how to build an executable using the OOQP header files and libraries.

2.4 Use in AMPL

OOQP may be invoked within AMPL, a modeling language for specifying optimization problems. From within AMPL, one must first define the model and input the data. If the model happens to be a QP, then an `option solver` command within the AMPL code can be used to ensure the use of OOQP as the solver.

An AMPL model file that may be used to describe a problem of the form (2) without equalities $Ax = b$ is as follows.

```
set I;  set J;
set QJJ within {J,J}; set CIJ within {I,J};

param objadd;  param g{J};   param Q{QJJ};
param clow{I}; param C{CIJ}; param cupp{I};
param xlow{J}; param xupp{J};

var x{j in J} >= xlow[j] <= xupp[j];

minimize total_cost: objadd + sum{j in J} g[j] * x[j]
         + 0.5 * sum{(j,k) in QJJ} Q[j,k] * x[j] * x[k];

subject to ineq{i in I} :
        clow[i] <= sum{(i,j) in CIJ } C[i,j] * x[j] <= cupp[i] ;
```

The data for the QP is normally given in a separate AMPL data file, which for the problem (3) is as follows.

```
param  objadd   := 4 ;

param: J : g    :=  col1    1.5         col2   -2 ;

param: QJJ : Q :=
       col1    col1    8    col1    col2    2
       col2    col1    2    col2    col2    10 ;

param xlow      :=  col1    0           col2    0 ;
param xupp      :=  col1    20          col2    Infinity ;

param: I : clow :=  row1    2           row2   -Infinity ;
param      cupp :=  row1    Infinity    row2    6 ;

param: CIJ : C :=
       row1    col1    2    row1    col2    1
       row2    col1   -1    row2    col2    2 ;
```

Suppose the model file was named `example.mod` and the data file was named `example.dat`. From within the AMPL environment, one would type the following lines to solve the problem and view the solution.

```
model example.mod;
data  example.dat;
option solver ooqp-ampl;
solve;
display x;
```

The following lines containing the optimal value of x would then be displayed.

```
x [*] :=
col1  0.7625
col2  0.475
;
```

2.5 Use in MATLAB

OOQP may be invoked from within the MATLAB environment. Instructions on how to obtain the necessary software may be found in the `README-Matlab` in the `OOQP` directory.

The prototype for the MATLAB function is as follows.

```
[x, gamma, phi, y, z, lambda, pi] = ...
    ooqp( c, Q, xlow, xupp, A, dA, C, clow, cupp, doPrint )
```

This function will solve the general QP formulation (2), re-expressed here in MATLAB notation.

```
minimize:      c' * x + 0.5 * x' * Q * x
subject to:    A * x = dA
               clow <= C * x <= cupp
               xlow <=     x <= xupp
```

This is the exactly the default QP formulation (2). The vectors and matrix objects in the argument list should be MATLAB matrices of appropriate size. Upper or lower bounds that are absent should be set to `inf` or `-inf`, respectively. (It is important to use these infinite values rather than large but finite values.)

The final parameter in the argument list, `doPrint`, is optional. If present, it should be set to one of the strings "yes," "on," "no," or "off." If the value is "yes" or "on," then progress information will be printed while the algorithm solves the problem. If `doPrint` is absent, the default value "off" will be assumed.

Help is also available within MATLAB. After you have followed the instruction in `README-Matlab` and installed the MATLAB interface in the local directory or on the MATLAB path, help can by obtained by typing `help ooqp` at the MATLAB prompt.

3 Overview of the OOQP Development Framework

In this section, we start by describing the layered design of OOQP, which is the fundamental organizing principle for the classes that make it up. We then discuss the directory structure of the OOQP distribution, and the makefile-based build process that is used to construct executables.

3.1 The Three Layers of OOQP

OOQP has a layered design in which each layer is built from abstract operations defined by the layer below it. We sketch these layers and their chief components in turn.

QP Solver Layer. The top layer of OOQP contains the high-level algorithms and heuristics for solving quadratic programming problems. OOQP implements primal-dual interior-point algorithms, that are motivated by the optimality (Karush-Kuhn-Tucker) conditions for a QP. We write these conditions for the formulation (1) by introducing Lagrange multiplier vectors y and z (for the equality and inequality constraints, respectively) and a slack vector s to yield the following system:

$$\begin{aligned} c + Qx - A^T y - C^T z &= 0, && \text{(4a)} \\ Ax - b &= 0, && \text{(4b)} \\ Cx - s - d &= 0, && \text{(4c)} \\ SZe &= 0, && \text{(4d)} \\ s, z &\geq 0, && \text{(4e)} \end{aligned}$$

where S and Z are diagonal matrices whose diagonal elements are the components of the vectors s and z, respectively. A primal-dual interior-point algorithm finds a solution to (1) by applying Newton-like methods to the nonlinear system of equations formed by (4a), (4b), (4c), and (4d), constraining all iterates (x^k, y^k, z^k, s^k), $k = 0, 1, 2, \ldots$ to satisfy the bounds (4e) *strictly* (that is, all components of z^k and s^k are strictly positive for all k).

OOQP implements the primal-dual interior point algorithm of Mehrotra [18] for linear programming, and the variant proposed by Gondzio [13] that includes higher-order corrections. See Section 4.1 below, and the text of Wright [23] for further description of these methods.

Problem Formulation Layer. Algorithms in the QP solver layer are built entirely from abstract operations defined in the problem formulation layer. This layer consists of several classes each of which represents an object of interest to a primal-dual interior-point algorithm. The major classes are as follows.

Data Stores the data (Q, A, C, c, b, d) defining the QP (1), in an economical format suited to the specific structure of the data and the operations needed to perform on it.

Variables Contains storage for the primal and dual variables (x, y, z, s) of the problem, in a format suited to the specific structure of the problem. Also implements various methods associated with the variables, including the computation of a maximum steplength, saxpy operations, and calculation of $\mu = (s^T z)/m_C$.

Residuals Contains storage for the residuals—the vectors that indicate infeasibility with respect to the KKT conditions—along with methods to calculate these residuals from formulae such as (4a–4d). This class also contains methods for performing the projection operations needed by the Gondzio approach, calculating residual norms, and calculating the current duality gap (see Section 5.2.3 for a discussion of the duality gap.)

LinearSystem Contain methods to factor the coefficient matrix used in the Newton-like iterations of the QP solver and methods that use the information from the factorization to solve the linear systems for different right-hand sides. The systems that must be solved are described in Section 4.1.

To be concrete in our discussion, we have referred to the QP formulation (1) given in the introduction, but the problem formulation layer provides abstract operations suitable to many different problem formats. For instance, the quadratic program that arises from classical support vector machine problems is

$$\min \|w\|^2 + \rho e^T u, \text{ subject to } D(Vw - \beta e) \geq e - u, \quad v \geq 0, \tag{5}$$

where V is a matrix of empirical observations, D is a diagonal matrix whose entries are ± 1, ρ is a positive scalar constant, and e is a constant vector of all ones. In OOQP's implementation of the solver for this problem, we avoid expressing the problem in the form (2) by forming the matrices Q and C explicitly. Rather, the problem formulation layer provides methods to perform operations involving Q, C, and the other data objects that define the problem. The QP solver layer implements a solver by calling these methods, rather than operating on the data and variables explicitly.

Since a solver for general problems of the form (2) is useful in many circumstances, OOQP provides a solver for this formulation, as well as for several specialized formulations such as (5). Users may readily specialize the abstract operations in this layer and thereby create solvers that are specialized to yet more problem formulations. Section 5 gives instructions on how to develop specialized implementations of this class.

Linear Algebra Layer. Many of the linear algebra operations and data structures in OOQP are shared by several problem types. For instance, regardless of the particular QP formulation, the Variable, Data, and LinearSystems classes will need to perform saxpy, dot product, and norm calculations on vectors of doubles. Furthermore, most sparse problems will need to store matrices in a Harwell-Boeing format. Reimplementing the linear algebra operations in each of the problem-dependent classes would result in an unacceptable explosion in code size and complexity. The solution we implemented in OOQP is to define another layer that provides the linear algebra functionality needed by many different problem formulations. An added advantage is that by confining linear algebra to its own layer, we can implement solvers for distributed platforms with little change in the code.

The linear algebra classes are somewhat a different from the classes in the QP solver and problem formulation layers. The two topmost layers of OOQP consist of small, abstract interfaces with no behavior whatsoever. We have provided concrete implementations based on these interfaces, but even our concrete classes tend to contain only a small number of methods. Hence, these classes are easy to understand and easy to override.

By contrast, implementations of linear algebra classes such as `DoubleMatrix` and `OoqpVector` must supply a relatively large amount of behavior. This complexity appears to be inevitable. The widely used BLAS library, which is meant to contain only the most basic linear algebra operations, consists of forty-nine families of functions and subroutines. As well as defining operations, the linear algebra classes also have to handle the storage of their component elements.

Our approach to the linear algebra classes is to identify and provide as methods the basic operations that are used repeatedly in our implementations of the problem formulation layer. As much as possible, we use existing packages such as BLAS [16], LAPACK [1] and PETSc [2, 3, 4] to supply the behavior needed to implement these methods. Since our goal is simplicity, we provide only the functionality that we use. We are not striving for a complete linear algebra package but for a package that may be conveniently used in the setting of interior point optimization algorithms. For this reason, many BLAS operations are not provided; and certain operations common in interior-point algorithms, but rare elsewhere, are given equal status with BLAS routines.

3.2 OOQP Directory Structure and Build Process

The OOQP installation process will generate compiled libraries in the directory `lib` and a directory named `include` containing header files. These libraries and headers may be copied into a more permanent system-dependent location. Users who wish to call OOQP code from within their own C or C++ programs may use any build process they wish to compile and link against the installed headers and libraries.

Users who wish to do more complex development with OOQP may find it more convenient to work within the source directory `src` and use the OOQP build system to compile their executables. OOQP has a modular directory structure in which source and header files that logically belong together are placed in their own subdirectory of `src`. For example, code that implements the solver for the formulation (2) can be found in `src/QpGen`, while code that defines classes for dense matrices and dense linear equation solvers can be found in `src/DenseLinearAlgebra`.

Any system of building executables in a complex project is necessarily complex. This is especially true for object-oriented code, as the most common methods for building executables are designed for use with procedural (rather than object-oriented) languages. In OOQP, we have designed a relatively simple process but one that requires some effort to learn and understand. Users who intend to develop a customized solver for a new QP formulation or to replace the linear algebra subsystem need to understand something of this process, and this section is aimed primarily at them. Users who do not have an interest in the details of the build process may safely skip this section.

OOQP is built by using the GNU version of the standard Unix `make` utility. GNU `make` is freely and widely available, yields predictable performance across a wide variety of platforms, and has a number of useful features absent in many vendor-provided versions of `make`. In this section, we assume that the user has a basic understanding of how to write makefiles, which are the files used as input to the `make` utility.

OOQP uses a `configure` script, generated by the GNU Autoconf utility, to set machine-dependent variables within the makefiles that appear in various subdirectories. In the top-level directory, `OOQP`, `configure` generates the global makefile `GNUmakefile` from an input file named `GNUmakefile.in`. The user who wishes to modify this makefile should alter `GNUmakefile.in` and then rerun `configure` to obtain a new `GNUmakefile`, rather than altering `GNUmakefile` directly. (Users will seldom have cause to alter this makefile or any other file under the control of Autoconf but should be aware of the fact that some makefiles are generated in this way.)

All subdirectories of the `src` that contain C++ code also contain a file named `Makefile.inc`. We give an example of such a file from the directory `src/QpExample`, which contains an example problem formulation based directly on (1). In the `src/QpExample` directory, the `Makefile.inc` reads as follows.

```
QPEXAMPLEDIR = $(srcdir)/QpExample

QPEXAMPLEOBJ = \
    $(QPEXAMPLEDIR)/QpExampleData.o \
    $(QPEXAMPLEDIR)/QpExampleVars.o \
    $(QPEXAMPLEDIR)/QpExampleResids.o  \
    $(QPEXAMPLEDIR)/QpExampleDenseLinsys.o \
    $(QPEXAMPLEDIR)/QpExampleDense.o

qpexample_dense_gondzio_OBJECTS = \
    $(QPEXAMPLEDIR)/QpExampleGondzioDriver.o \
    $(QPEXAMPLEOBJ) \
    $(libooqpgondzio_STATIC) \
    $(libooqpdense_STATIC) $(libooqpbase_STATIC)
```

This file contains three makefile variable definitions, specifying the subdirectory name (`QPEXAMPLEDIR`), the list of object files specific to the SVM solver (`QPEXAMPLEOBJ`), and the full list of object files that must be linked to create the executable for the solver (`qpexample_dense_gondzio_OBJECTS`). Every module of OOQP contains a similar `Makefile.inc` file to define variables relevant to that module. (Another example is the variable `libooqpgondzio_STATIC`, used in the definition of `qpexample_dense_gondzio_OBJECTS`, which is defined in `src/QpSolvers/Makefile.inc`.) Note that the variable `srcdir` in this example refers to the OOQP source directory and does not need to be defined in `src/QpExample/Makefile.inc`.

Some subdirectories of the `src` that contain C++ code also contain a file named `MakefileTargets.inc`. This file defines targets relevant to the build process. An example of such a file is `src/QpExample/MakefileTargets.inc`, which is as follows.

```
qpexample-dense-gondzio.exe: $(qpexample_dense_gondzio_OBJECTS)
    $(CXX) -o $@ $(CXXFLAGS) $(LDFLAGS) $(LIBS) \
        $(qpexample_dense_gondzio_OBJECTS) $(BLAS) $(FLIBS)
```

The `qpexample-dense-gondzio.exe` target specifies the dependency of the executable on the object list that was defined in the corresponding `Makefile.inc` file.

In using `Makefile.inc` and `MakefileTargets.inc` files, we separate target definitions from variable definitions because unpredictable behavior can occur if the targets are read before all variables are defined.

When a user invokes GNU `make` from the `OOQP` directory, the utility ensures that

- all variables defined in files named `Makefile.inc` in *direct* subdirectories of the `src` directory are made available in the build;
- all targets defined in similarly located files named `MakefileTargets.inc` are also made available;
- *direct* subdirectories of the `src` directory that contain a file that is named `Makefile.inc` are placed on the path on which to search for header (.h) files.

Thus, when the GNU `make` utility is named `gmake`, one may build the executable `qpexample-dense-gondzio.exe` by typing

```
    gmake qpexample-dense-gondzio.exe
```

from the command line from within the `OOQP` directory.

The makefile system can also be used to perform dependency checking. Typing

```
    gmake depend
```

will cause the Unix `makedepend` utility to generate dependency information for all source files in direct subdirectories of the `src` directory that contain a file named `Makefile.inc`. This dependency information will then be used in the next build to determine whether source files are up-to-date with respect to their included header files.

We emphasize that this process works only on direct subdirectories of the `src` directory. Files named `Makefile.inc` in more deeply nested subdirectories will not, without extra effort, be recognized. We deliberately restricted the search to direct subdirectories of the source directory in order to make the build process more predictable.

User-defined `Makefile.inc` and `MakefileTargets.inc` need be no more complicated than the example files given above. Some of the instances of these files that are included in the OOQP distribution contain more variables and targets than those shown above because they need to accomplish additional tasks. Moreover, they may contain conditional statements to disable certain targets, if these targets depend on external packages that are not present on the computer at the time of the build. These advanced issues may be ignored by all but developers of OOQP.

Finally, we mention that some external packages, such as PETSc, require specializations to the global makefile. When building executables that use these packages, one cannot use the default global makefile `GNUmakefile`. To build the executable `qpbound-petsc-mehrotra.exe`, for instance, one must type the following line.

```
gmake -f PetscMakefile qpbound-petsc-mehrotra.exe
```

We may include other such specialized makefiles in the OOQP distribution in the future. While inclusion of these files is a minor inconvenience, we consider it important to isolate changes to the global makefile in this manner, so that misconfiguration of a certain package is less likely to cause problems in an unrelated build.

4 Working with the QP Solver

In this section, we focus on the top layer of OOQP, the QP solver.

4.1 Primal-Dual Interior-Point Algorithms

We start by giving some details of the primal-dual interior-point algorithms that are implemented in the `Solver` class in the OOQP distribution. By design, the code that implements these algorithms is short, and one can see the correspondence between the code and the algorithm description below. Therefore, users who want to modify the basic algorithm will be able to do so after reading this section.

A primal-dual algorithm seeks variables (x, y, z, s) that satisfy the optimality conditions for the convex quadratic program (1), introduced in Section 3.1 but repeated here for convenience.

$$
\begin{aligned}
c + Qx - A^T y - C^T z &= 0, && \text{(6a)}\\
Ax - b &= 0, && \text{(6b)}\\
Cx - s - d &= 0, && \text{(6c)}\\
SZe &= 0, && \text{(6d)}\\
s, z &\geq 0. && \text{(6e)}
\end{aligned}
$$

The complementarity measure μ defined by

$$\mu = z^T s / m_c \tag{7}$$

(where m_c is the number of rows in C) is important in measuring the progress of the algorithm, since it measures violation of the complementarity condition $z^T s = 0$, which is implied by (6d). Infeasibility of the iterates with respect to the equality constraints (6a), (6b), and (6c) also makes up part of the indicator of nonoptimality.

The OOQP distribution contains implementations of two quadratic programming algorithms: Mehrotra's predictor-corrector method [19] and Gondzio's modification of this method that uses higher-order corrector steps [13]. (See also [23, Chapter 10] for a 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 convex quadratic programming. Mehrotra's algorithm can be specified as follows.

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

$$\begin{bmatrix} Q & -A^T & -C^T & 0 \\ A & 0 & 0 & 0 \\ C & 0 & 0 & -I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x^{\text{aff}} \\ \Delta y^{\text{aff}} \\ \Delta z^{\text{aff}} \\ \Delta s^{\text{aff}} \end{bmatrix} = - \begin{bmatrix} r_Q \\ r_A \\ r_C \\ ZSe \end{bmatrix}, \tag{8}$$

where

$$\begin{aligned} S &= \operatorname{diag}(s_1, s_2, \ldots, s_{m_c}), & \text{(9a)} \\ Z &= \operatorname{diag}(z_1, z_2, \ldots, z_{m_c}), & \text{(9b)} \\ r_Q &= Qx + c - A^T y - C^T z, & \text{(9c)} \\ r_A &= Ax - b, & \text{(9d)} \\ r_C &= Cx - s - d. & \text{(9e)} \end{aligned}$$

Compute α_{aff} to be the largest value in $(0, 1]$ such that

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

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

$$\begin{bmatrix} Q & -A^T & -C^T & 0 \\ A & 0 & 0 & 0 \\ C & 0 & 0 & -I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = - \begin{bmatrix} r_Q \\ r_A \\ r_C \\ ZSe - \sigma\mu e + \Delta Z^{\text{aff}} \Delta S^{\text{aff}} e \end{bmatrix}, \tag{10}$$

where ΔZ^{aff} and ΔS^{aff} are defined in an obvious way.
Compute $\alpha_{\max}$ to be the largest value in $(0, 1]$ such that

$$(z, s) + \alpha(\Delta z, \Delta s) \geq 0.$$

Choose $\alpha \in (0, \alpha_{\max})$ according to Mehrotra's step length heuristic.
Set

$$(x, y, z, s) \leftarrow (x, y, z, s) + \alpha(\Delta x, \Delta y, \Delta z, \Delta s).$$

until the convergence or infeasibility test is satisfied.

The direction obtained from (10) can be viewed as an approximate second-order step toward a point (x^+, y^+, z^+, s^+) at which the conditions (6a), (6b), and (6c) 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 [13] follows the Mehrotra algorithm in its computation of directions from (8) and (10). It may then go on to enhance the search direction further by solving additional systems similar to (10), with variations in the last m_C components of the right-hand side. Successive corrections attempt to increase the steplength α that can be taken along the final direction, and to bring the pairwise products $s_i z_i$ whose values are either much larger than or much smaller than the average into closer correspondence with the average. The maximum number of corrected steps we calculate is dictated by the ratio of the time taken to factor the coefficient matrix in (10) to the time taken to use these factors to produce a solution for a given right-hand side. When the marginal cost of solving for an additional right-hand side is small relative to the cost of a fresh factorization, and when the corrections appear to be improving the quality of the step significantly, we allow more correctors to be calculated, up to a limit of 5.

The algorithms implemented in OOQP use the step length heuristic described in Mehrotra [19, Section 6], modified slightly to ensure that the same step lengths are used for both primal and dual variables.

4.2 Monitoring the Algorithm: The `Monitor` Class

OOQP can be used both for solving a variety of stand-alone QPs and for solving QP subproblems as part of a larger algorithm. Different termination criteria may be appropriate to each context. For a simple example, the criteria used to declare success in the solution of a single QP would typically be more stringent than the criteria for a QP subproblem in a nonlinear programming algorithm, in which we can afford some inexactness in the solution. Accordingly, we have designed OOQP to be flexible as to the definition and application of termination criteria, and as to the way in which the algorithm's progress is monitored and communicated to the user. In some instances, a short report on each interior-point iteration is desirable, while in others, silence is more appropriate. In OOQP, an abstract `Monitor` class monitors the algorithm's progress, while an abstract `Status` class tests the termination conditions. We describe the `Monitor` class in this section, and the `Status` class in Section 4.3 below.

Our design assumes that each algorithm in the QP solver layer of the code has its own natural way of monitoring the algorithm and testing termination. Accordingly, the two derived `Solver` classes in the OOQP distribution each contain a `defaultMonitor` method to print out a single line of information to the standard output stream at each iteration, along with a suitable message at termination of the algorithm. The prototype of this method is as follows.

```
void Solver::defaultMonitor( Data * data, Variables * vars,
                             Residuals * resids,
                             int i, double mu,
                             int status_code, int stage )
```

The `data` argument contains the problem data, while `vars` and `resids` contain the values of the variables and residuals at the current iterate, which together depict the status of the algorithm. (See Sections 3.1 and 5 for further information about these objects.) The variable i is the current iteration number and mu is the complementarity measure (7). The integer status_code indicates the status of the algorithm at termination, if termination has occurred; see Section 4.3 below. The stage argument indicates to defaultMonitor what type of information it should print. In our implementations, the values stage=0 and stage=1 cause the routine to print out a single line containing iteration number, the value of μ, and the residual norm. The value stage=1 is used after termination has occurred, and additionally causes a message about the termination status to be printed.

One mechanism available to the user who wishes to alter the monitoring procedure is to create a new subclass of Solver that contains an implementation of defaultMonitor that overrides the existing implementation. This is the simplest way to proceed and will suffice in many circumstances. However, it has a disadvantage for users who work with several different implementations of Solver—versions that implement different primal-dual algorithms, for instance, or are customized to different applications—in that the new monitoring routine cannot be shared among the different QP solvers. A subclass of each QP solver that contains the overriding implementation of defaultMonitor would need to be created, resulting in a number of new leaves on the class tree. A second disadvantage is that some applications may require several monitor processes to operate at once, for example, one process like the defaultMonitor described above that writes minimal output to standard output, and another process that writes more detailed information to a log file. It is undesirable to create a new Solver subclass for each different set of monitor requirements.

In OOQP, we choose delegation, rather than subclassing, as our mechanism for customizing the monitor process. Delegation is a technique in which the responsibility for taking some action normally associated with an instance of a given class is delegated to some other object. In our case, although the Solver class would normally be responsible for displaying monitor information, we delegate responsibility to an associated instance of the Monitor class. The Solver class contains methods for establishing its defaultMonitor method as one of the monitor procedures called by the code and for adding monitor procedures supplied by the user.

The abstract definition of the Monitor class can be found in the OOQP distribution at src/Abstract/OOQPMonitor.h, along with the definitions of several subclasses. The only method of interest in the Monitor class is the doIt method, which causes the object to perform the operation that is its sole reason for being. Making these objects instances of a class rather than subroutines tends to be more natural in the C++ language and makes it far simpler to handle any state information that instances of Monitor may wish to keep between calls to doIt.

The doIt method has the following prototype, which is identical to the defaultMonitor method described above.

```
void OoqpMonitor::doIt( Solver * solver, Data * data,
                          Variables * vars, Residuals * resids,
                          int i, double mu,
                          int status_code, int stage );
```

Users who wish to implement their own monitor procedure should create a subclass of `OOQPMonitor`, for example by making the following definition:

```
class myMonitor : public OOQPMonitor {
public:
  virtual void doIt( Solver * solver,  Data * data,
                     Variables * vars, Residuals * resids,
                     int i, double mu,
                     int status_code, int level );
};
```

and then implementing their own version of the `doIt` method. Their code that creates the instance of the `Solver` class and uses it to solve the QP should contain the following code fragments:

```
OoqpMonitor * usermon = new myMonitor;
...
qpsolver->monitorSelf();
qpsolver->addMonitor( usermon );
```

The first statement creates an instance of the subclass `myMonitor`. The second and third statements should appear after the instance `qpsolver` of the `Solver` class has been created but before the method `qpsolver->solve()` has been invoked. The call to `monitorSelf` statement ensures that the `defaultMonitor` method is invoked at each interior-point iteration, while the call to `addMonitor` ensures that the user-defined monitor is also invoked. Users who wish to invoke only their own monitor procedure and not the `defaultMonitor` method can omit the second statement. The solver is responsible for deleting any monitors give to it via the `addMonitor` method.

The default behavior for an instance of `Solver` is to display no monitor information.

4.3 Checking Termination Conditions: The `Status` Class

In OOQP, the `defaultStatus` method of the `Solver` class normally handles termination tests. However, OOQP allows delegation of these tests to an instance of the `Status` class, in much the same way as the monitor procedures can be delegated as described above. Before describing how to replace the OOQP termination tests, let us describe the termination tests that OOQP uses by default.

The `defaultStatus` method of the `Solver` class uses termination criteria similar to those of PCx [5]. To discuss these criteria, we again refer to the problem formulation (1) (discussed in Section 4.1) and 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 may be defined for formulation (1) by the formula (16) below. We define the quantity ϕ_k as follows,

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

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 the termination status to be `SUCCESSFUL_TERMINATION` when

$$\mu_k \le \texttt{tol}_\mu, \qquad \|(r_Q^k, r_A^k, r_C^k)\|_\infty \le \texttt{tol}_r \|(Q, A, C, c, b, d)\|_\infty. \tag{11}$$

We declare the status to be `INFEASIBLE` if

$$\phi_k > 10^{-8} \text{ and } \phi_k \ge 10^4 \min_{0 \le i \le k} \phi_i. \tag{12}$$

(In fact, since this is not a foolproof test of infeasibility, the true meaning of this status is "probably infeasible.") Status `UNKNOWN` is declared if the algorithm appears to be making unacceptably slow progress, that is,

$$k \ge 30 \text{ and } \min_{0 \le i \le k} \phi_i \ge \frac{1}{2} \min_{1 \le i \le k-30} \phi_i, \tag{13}$$

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

$$\|(r_Q^k, r_A^k, r_C^k)\|_\infty > \texttt{tol}_r \|(Q, A, C, c, b, d)\|_\infty \tag{14a}$$

$$\text{and } \|(r_Q^k, r_A^k, r_C^k)\|_\infty / \mu_k \ge 10^8 \|(r_Q^0, r_A^0, r_C^0)\|_\infty / \mu_0. \tag{14b}$$

We declare status `MAX_ITS_EXCEEDED` when the number of iterations exceeds a specified maximum; the default is 100. If none of these conditions is satisfied, we declare the status to be `NOT_FINISHED`.

Users who wish to alter the termination test may simply create a subclass of `Solver` with their own implementation of `defaultStatus`. Alternatively, they may create a subclass of the `Status` class, whose abstract definition can be found in the file `src/Abstract/Status.h`. The sole method in the `Status` class is `doIt`, which has the following prototype.

```
int Status::doIt(  Solver * solver, Data * data,
                   Variables * vars, Residuals * resids,
                   int i, double mu, int stage );
```

The parameters to the `doIt` method have the same meaning as the correspondingly named parameters of the `OOQPMonitor::doIt` method. The return value of the `Status::doIt` method determines whether the algorithm continues or terminates. The possible values that may be returned are as follows.

```
enum TerminationCode
{
  SUCCESSFUL_TERMINATION = 0,
  NOT_FINISHED,
  MAX_ITS_EXCEEDED,
  INFEASIBLE,
  UNKNOWN
};
```

The meanings of these return codes in the `defaultStatus` method are described above. Users are advised to assign similar meanings in their specialized implementation.

Unlike the case of monitor procedures, it does not make sense to have multiple status checks in operation during execution of the interior-point algorithm; exactly one such check is required. Users who wish to use the `defaultStatus` method supplied with the OOQP distributions need do nothing; the default behavior of an instance of the `Solver` class is to call this method. Users who wish to supply their own method can create their own subclass of the `Status` class as follows.

```
class myStatus : public Status {
public:
        virtual void doIt(  Solver * solver, Data * data,
                            Variables * vars, Residuals * resids,
                            int i, double mu, int stage );
};
```

Then, they can invoke the `useStatus` method after creating their instance of the `Solver` class, to indicate to the solver object that it should use the user-defined status-checking method. The appropriate lines in the driver code would be similar to the following.

```
MyStatus * userstat = new myStatus;
...
qpsolver->useStatus( userstat );
```

The solver is responsible for deleting any `Status` objects given to it via the `useStatus` method.

5 Creating a New QP Formulation

Users who wish to construct a solver for a class of QPs with a particular structure not supported in the OOQP distribution may consider using the framework to build a new solver that represents and manipulates the problem data and variables in an economical, natural, and efficient way. In this section, we describe the major classes that must be implemented in order to develop a solver for a new problem formulation.

Most of the effort in developing a customized solver for a new class of structured QPs is in reimplementing the classes in the problem formulation layer. As described in Section 3, this layer consists of five main classes—`Data`, `Variables`, `Residuals`, `LinearSystem`, and `ProblemFormulation`—that contain data structures to store the problem data, variables, and residuals, and methods to perform the operations that are required by the interior-point algorithms.

As discussed in Section 4.1, the core algebraic operation in an interior-point solver is the solution of a Newton-like system of linear equations. For formulation (1), the general form of this system is as follows

$$\begin{bmatrix} Q & -A^T & -C^T & 0 \\ A & 0 & 0 & 0 \\ C & 0 & 0 & -I \\ 0 & 0 & S & Z \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta s \end{bmatrix} = - \begin{bmatrix} r_Q \\ r_A \\ r_C \\ r_{z,s} \end{bmatrix}, \tag{15}$$

where r_Q, r_A, and r_C are defined in equations (9c), (9d), and (9e), and $r_{z,s}$ is chosen in a variety of ways, as described in Section 4. Most of the objects that populate a problem formulation layer can be found in this system. The `Variables` in formulation (1) break down naturally into four components x, y, z, and s. Likewise, there are naturally four components to the `Residuals` of this formulation. For other problem formulations, such as SVM (5), this partitioning of the variables is not natural, and a scheme more suited to the particular formulation is used instead. However, to focus our discussion of the implementation of the problem formulation layer in this section, we will continue to refer to the particular formulation (1) and the system (15). The implementations of (1) discussed in this section may be found in the OOQP distribution in directory `src/QpExample`.

In reimplementing the problem formulation layer for a new QP structure, it may be helpful to make use of the classes from the *linear algebra layer*. As mentioned in Section 3, this layer contains classes for storing and operating on dense matrices, sparse matrices, and vectors. These classes can be used as building blocks for implementing the more complex storage schemes and arithmetic operations needed in the problem formulation layer.

We first elaborate on the use of the linear algebra layer and then describe in some detail the process of implementing the five classes in the problem formulation layer.

5.1 Linear Algebra Operations

Most implementations of the problem formulation layer that appear in the OOQP distribution (the `QpGen`, `QpExample`, `QpBound`, and `Huber`, and `Svm` implementations) all are built using the objects in OOQP's linear algebra layer. The classes in this layer represent objects such as matrices and vectors, and they provide methods that are especially useful for developing interior point QP solvers. By basing our problem formulation layer on the abstract operations of the linear algebra layer we gain another significant advantage: we can use the same problem formulation code for several quite varied representations of vectors and matrices. For instance, the implementation of the problem formulation layer for QPs with simple bounds is independent of whether the Hessian matrix is represented as a dense array on a single processor or as a sparse array distributed across several processors.

Use of OOQP's linear algebra layer in implementing the problem formulation layer is not mandatory. Users are free to define their own matrix and vector data structures and implement their own linear algebra operations (inner products, saxpys, factorizations, and so on) without referring to OOQP's linear algebra objects at all. The authors of OOQP recognize that there is a learning curve associated with the use of the abstract operations in OOQP's linear algebra objects and that the implementation might proceed more quickly if users define their own linear algebra in terms of concrete operations on concrete data.

For maximum effectiveness, we recommend a compromise approach. While the base classes for our linear algebra layer are defined only in terms of abstract operations, several of the classes (such as `SimpleVector`) may also be used concretely. Users can start by defining their problem formulation in terms of these simple classes but define their own concrete operations on the data. Later, they can replace their concrete operations by the abstract methods supplied with these classes. Finally, having gained proficiency in the use of these classes, they may then replace the entire class with a more appropriate one. Section 6 is a short tutorial on the linear algebra layer that can be consulted by those who wish to use the layer in this way.

5.2 Specializing the Problem Formulation Layer

We now detail how to implement the various classes in the problem formulation layer.

5.2.1 Specializing `Data`

The purpose of the `Data` class is to store the data defining the problem, in some appropriate format, to provide methods for performing operations with the data matrices (for example, matrix multiplications or insertion of problem matrices into the larger matrices of the form (8) or (10)), for calculating some norm of the data, for filling the data structures with problem data (read from a file, for instance, or passed from a modeling language or MATLAB), for printing the data, and for generating random problem instances for testing and benchmarking purposes.

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. It includes only two pure virtual functions, `datanorm` (of type `double`) and `print`, whose implementation *must* appear in any derived classes.

A derived class of `Data` for the formulation (1) in which the problem data is dense would include storage for the vectors c, b, and d as arrays of doubles; storage for A and C as two-dimensional arrays of doubles; and storage for the lower triangle of the symmetric matrix Q. In our implementation of the derived class `QpExampleData`, we have provided methods for multiplying by the matrices Q, A, and C and for copying the data into a larger structures such as the matrix in (15). We find it convenient to provide methods like this for manipulating the data in our `QpExampleData` class, rather than having code from other problem formulation classes manipulate the data structures directly; the extra generality that the added layer of encapsulation affords has sometimes proven useful.

Consider now the two pure virtual functions `datanorm` and `print`. One reasonable implementation of `datanorm` for the formulation (1) would simply return the magnitude of of the largest element in the matrices Q, A, and C, and the vectors c, b, and d that define (1). The implementation of `print` might print the data objects Q, A, C, c, b, and d to standard output in some useful format. Although not compulsory, we might also define a routine `datarandom` to generate an instance of (1), given user-defined dimensions n, m_A, and m_C, and possibly a desired level of sparsity for the matrices. Naturally, this method should take care that Q is positive semidefinite.

The derived `Data` class might also contain one or more implementations of a `datainput` method that allow the user to define the problem data. We could, for instance, have one implementation of `datainput` that reads the data in some simple format from ascii files and another implementation that reads a file in MPS format, appropriately extended for quadratic programming (Maros and Mészáros [17]). Since the MPS format allows for bounds and for constraints of the form $l_c \le Cx \le u_c$, the latter implementation generally would need to perform transformations to pose the problem in the form (1). (The data from a MPS file is more naturally represented by our "general" QP formulation (2).)

5.2.2 Specializing `Variables`

Instances of `Variables` class store the problem variables ((x, y, z, s) in the case of (1)) in whatever format is appropriate to the problem structure. The class includes a variety of methods essential in the implementation of Algorithm MPC. Most of them defined as pure virtual functions, because they strongly depend on the structure of the problem.

We now sketch the main methods for the `Variables` class, illustrating each one by specifying its implementation for the formulation (1).

`mu`: Calculate the complementarity gap: $\mu = z^T s/m_C$.

`mustep`: Calculate the complementarity gap that would be obtained from a step of length α along a specified direction from the current point. For (1), given the search direction $(\Delta x, \Delta y, \Delta z, \Delta s)$ (supplied in an argument of type `Variables`) and a positive scalar α, this method would calculate

$$(z + \alpha \Delta z)^T (s + \alpha \Delta s)/m_C.$$

`negate`: Multiply the current set of variables by -1. For (1), we would replace (x, y, z, s) by $-(x, y, z, s)$.

`saxpy`: Given another set of variables and a scalar, perform a saxpy operation with the current set of variables. For (1), we would pass a second instance of a `Variables` class containing (x', y', z', s'), together with the scalar α as arguments, and perform the replacement

$$(x, y, z, s) \leftarrow (x, y, z, s) + \alpha(x', y', z', s').$$

`stepbound`: Calculate the longest step in the range $[0, 1]$ that can be taken from the current point in a specified direction without violating nonnegativity of the complementary variables. For (1), the argument would be the direction (x', y', z', s') (stored in another instance of the`Variables` class), and this function would return the largest value of α in $[0, 1]$ such that the condition $(z + \alpha z', s + \alpha s') \geq 0$ is satisfied.

`findBlocking`: Similar to `stepbound` but returns additional information. Besides returning the maximum step α in the range $(0, 1]$ that can be taken without violating the appropriate nonnegativity constraint, the method indicates whether a primal or dual variable was the “blocking” variable (the one that will violate nonnegativity if the step α is any longer) by setting its last argument to 1 for a primal blocking variable, to 2 for a dual blocking variable, and to 0 if a full steplength $\alpha = 1$ can be taken without violating nonnegativity. In its second argument, the method returns the component of the primal variable vector that corresponds to the blocking index, while in its third argument, the method returns the same component of the primal *step* vector. In its fourth and fifth arguments, it returns the corresponding components of the dual variable vector and the dual step vector, respectively. To illustrate this functionality, suppose in the case of (1) that the step bound is α and the blocking variable is the ith primal variable; that is, $s_i + \alpha s'_i = 0$, while $(z + \alpha z', s + \alpha s') \geq 0$. Then the final argument of `findBlocking` returns 1, while the second through fifth arguments return the real numbers s_i, s'_i, z_i, and z'_i, respectively. The return value of the method itself would be α.

When both a primal and a dual index are “blocking,” the method reports the dual variable, by setting the final argument to 2 and reporting the components corresponding to the dual index. Subject to the latter condition, when there is a tie between different indices, the smaller index is reported.

`interiorPoint`: Set all components of the complementary variables to specified positive constants α and β. In the case of (1), we would set $s \leftarrow \alpha e$ and $z \leftarrow \beta e$, where e is the vector whose elements are all 1.

`shiftBoundVariables`: Add specified positive constants α and β to the complementary variables. For (1), this method would perform the replacements $s \leftarrow s + \alpha e$ and $z \leftarrow z + \beta e$.

`print`: Print the variables in some intelligible problem-dependent format.

`copy`: Copy the data from one instance of the `Variables` class into another.

`onenorm, infnorm`: Compute the ℓ_1 and ℓ_∞ norms of the variables. For (1), these quantities would be $\|(x, y, z, s)\|_1$ and $\|(x, y, z, s)\|_\infty$, respectively.

The usefulness of some of these methods in implementing Algorithm MPC is obvious. For instance, `saxpy` is used to take a step along the eventual search direction; `stepbound` is used to compute α_{aff} and $\alpha_{\max}$; `mustep` is used to compute μ_{aff}. The methods `interiorPoint` and `shiftBoundVariables` can be used in the heuristic to determine the starting point, while `findBlocking` plays an important role in Mehrotra's heuristic for determining the step length.

5.2.3 Specializing `Residuals`

The `Residuals` class calculates and stores the quantities that appear on the right-hand side of the linear systems that are solved at each iteration of the primal-dual method. 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 (9c), (9d), and (9e)) belong to the former class, while $r_{z,s}$ belongs to the latter. As above, we describe the roles of the main methods in the `Residuals` class with reference to the formulation (1).

`calcresids`: Given a `Data` object and a `Variables` object, calculate the residual components arising from the KKT linear equations. For (1), this method calculates r_Q, r_A, and r_C using the formulae (9c), (9d), and (9e), respectively.

`dualityGap`: Calculate the duality gap, which we define for the formulation (1) as follows:

$$\text{gap}_k \overset{\text{def}}{=} (x^k)^T Q x^k - b^T y^k + c^T x^k - d^T z^k. \qquad (16)$$

See the discussion below for guidance in formulating an expression for this parameter.

`residualNorm`: Calculate the norm of the components arising from the KKT linear equations. For (1), this method returns $\|(r_Q, r_A, r_C)\|$ for some norm $\|\cdot\|$.

`clear_r1r2`: Zero the components arising from the KKT linear equations. (Gondzio's method requires the solution of linear equations in which these residual components are replaced by zeros.)

`clear_r3`: Set the complementarity components to zero. In the case of (1), for which the general form of the linear system is (15), this operation sets $r_{z,s} \leftarrow 0$. (This operation is needed only in Gondzio's algorithm.)

`add_r3_xz_alpha`: Given a scalar α and a `Variables` class, add a complementarity term and a constant to each of the complementarity components of the residual vector. For (1), given variables (x, y, z, s), we would set

$$r_{z,s} \leftarrow r_{z,s} + ZSe + \alpha e,$$

where Z and S are the diagonal matrices constructed from the z and s variables.

`set_r3_xz_alpha`: As for `add_r3_xz_alpha`, but overwrite the existing value of $r_{z,s}$; that is, set $r_{z,s} \leftarrow ZSe + \alpha$.

`project_r3`: Perform the projection operation used in Gondzio's method on the $r_{z,s}$ component of the residual, using the scalars $\rho_{\min}$ and $\rho_{\max}$.

As discussed in Section 4.3, the `residualNorm` and `dualityGap` functions are used in termination and infeasibility tests. Users familiar with optimization theory will recognize the concept of the duality gap and will also recognize that the formula $x^TQx - b^Ty + c^Tx - d^Tz$ used in (16) is one of a number of expressions that are equivalent when the residuals r_Q, r_A, and r_C are all equal to zero. One such equivalent expression is the formula s^Tz, used in the definition of μ in the `Variables` class. We find it useful, however, to use a definition of the duality gap from which the slack variables have been eliminated and all the linear equalities in the KKT conditions have been taken into account. Such a definition can be obtained by starting with the definition of μ and successively substituting from each of the KKT conditions. For the case of (1), we start with s^Tz, substitute for s from the equation $Cx - s - d = 0$ (see (4c)) to obtain $z^T(Cx - d)$, then substitute for C^Tz from $c + Qx - A^Ty - C^Tz = 0$ (see (4a)) to obtain $c^Tx + x^TQx - x^TAy - d^Tz$, and finally substitute for Ax from $Ax - b = 0$ (see (4b)) to obtain the final expression.

In Algorithm MPC, the method `set_r3_xz_alpha` is called with the current `Variables` and $\alpha = 0$ to calculate the right-hand side for the affine-scaling system (8). Once σ has been determined and the affine-scaling step is known, `add_r3_xz_alpha` is called with $\alpha = -\sigma\mu$ and the `Variables` instance that contains the affine-scaling step, to add the necessary terms to the $r_{z,s}$ component to obtain the system (10).

5.2.4 Specializing `LinearSystem`

As mentioned above, major algebraic operations at each interior-point iteration are solutions of linear systems to obtain the predictor and corrector steps. For the formulation (1), these systems have the form (15). Such systems need to be solved two to six times per iteration, for different choices of the right-hand side components but the same coefficient matrix. Accordingly, it makes sense to logically separate the `factor` method that operates only on the matrix and the `solve` method that operates on 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 we use an iterative solver, the `factor` operation could involve computation of a preconditioner. The `factor` class may need to include storage—for a permutation matrix, for triangular factors of a reduced system, or for a preconditioner—for use in subsequent `solve` operations. We use the term "solve" to indicate that part of the solution process depends on the specific right-hand side. Usually, the results of applying methods from the `factor` class are used to facilitate or speed the process. Depending on the algorithm we employ, the `solve` method could involve triangular back-and-forward substitutions, matrix-vector multiplications, applications of a preconditioner, and/or permutation of vector components.

Both `factor` and `solve` are pure virtual functions; their implementation is left to the derived class because they depend entirely on the problem structure. For problems with special structure, the `factor` method is the one in OOQP that gives the most scope for exploitation of the structure and for computational savings over naive strategies. The SVM formulation is one case in which an appropriate implementation of the `factor` class yields significant savings over an implementation that is not aware of the structure. Another instances in which an appropriate implementation of `factor` can produce large computational savings include the case in which Q, A, and C have a block-diagonal structure, as in optimal control problems, allowing (17a) to be reordered and solved with either a banded matrix factorization routine or a discrete Riccati substitution (Rao, Wright, and Rawlings [21]).

We now describe possible implementations of `factor` for the formulation (1). Direct factorization of the matrix in (15) is not efficient in general as it ignores the significant structure in this system—the fact that S and Z are diagonal and the presence of a number of zero blocks. 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:

$$\begin{aligned} \begin{bmatrix} Q & A^T & C^T \\ A & 0 & 0 \\ C & 0 & -Z^{-1}S \end{bmatrix} \begin{bmatrix} \Delta x \\ -\Delta y \\ -\Delta z \end{bmatrix} &= \begin{bmatrix} -r_Q \\ -r_A \\ -r_C - Z^{-1} r_{z,s} \end{bmatrix}, \quad &(17a) \\ \Delta s &= Z^{-1}(-r_{z,s} - S\Delta y). \quad &(17b) \end{aligned}$$

Application of a direct factorization code for symmetric indefinite matrices to this equivalent form is an effective strategy. The `factor` routine would perform symmetric ordering, pivoting, and computation of the factors, while `solve` would use these factors to solve (17a) and then substitute into (17b) to recover Δs.

Another possible approach is to perform another step of block elimination and obtain a further reduction to the form

$$\begin{bmatrix} Q + C^T Z S^{-1} C & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ -\Delta y \end{bmatrix} = \begin{bmatrix} -r_Q - C^T S^{-1}(Z r_C + r_{z,s}) \\ -r_A \end{bmatrix}. \quad (18)$$

The main operation in `factor` would then be to apply a symmetric indefinite factorization procedure to the coefficient matrix in this system, while `solve` would perform triangular substitutions to solve (18) and then substitute to recover Δz and Δs in succession. This variant is less appealing than the approach based on (17a), however, since the latter approach allows the factorization routine to compute its own pivot sequence, while in (18) we have partially imposed a pivot ordering on the system by performing the additional step of block elimination. However, if the problem (1) contained no equality constraints (that is, A and b null), the approach (18) might be useful, as it would allow a symmetric *positive definite* factorization routine to be applied to the matrix $Q + C^T Z S^{-1} C$.

Alternative implementations of the `factor` and `solve` classes for (1) could apply iterative methods such as QMR [10, 11] or GMRES [22] (see also Kelley [15]) to the system (17a). 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 avoids the resulting ill conditioning should form part of the preconditioning strategy.

5.2.5 Specializing `ProblemFormulation`

Once a user has created new subclasses of `Data`, `Variables`, `Residuals`, and `LinearSystem` appropriate to the new QP formulation, he or she must create a subclass of `ProblemFormulation` to assemble a compatible set objects to be used by a QP solver. Assembly might seem to be a simple task not requiring the use of an additional assembly class, but in practice the process of creating a compatible set of objects can become quite involved, as we now discuss.

Consider our example QP formulation (1). Even in this simple case, one must create all vectors and matrices so that they have compatible sizes and so that they are able to copy or wrap the given problem data. The more abstract and flexible a problem formulation is, the more options tend to be present when the objects are created. If we wish to create a subclass of `Variables` for our new QP formulation in which the code is independent of whether the solver is executed on a uniprocessor platform or on a multiprocessor platform with distributed data, we must make some other arrangements to ensure that when the instance of `Variables` is created, the storage for the variables is allocated and distributed in the appropriate way. A traditional approach for managing this kind of complexity is to isolate the code for creating a compatible set of components in a separate subroutine. In OOQP, we use the same principle, isolating the code for managing the complexity in the methods of a subclass of `ProblemFormulation`.

The abstract ProblemFormulation class has the following prototype.

```
class ProblemFormulation {
public:
  // makeData will often take parameters.
  //  virtual Data          * makeData()      = 0;
  virtual Residuals     * makeResiduals( Data * prob_in ) = 0;
  virtual LinearSystem  * makeLinsys( Data * prob_in )    = 0;
  virtual Variables     * makeVariables( Data * prob_in ) = 0;
  virtual ~ProblemFormulation() {};
};
```

The makeVariables method is responsible for creating an instance of a subclass of Variables that is appropriate for this problem structure and for the computational platform. The other methods have similar purposes for instances of the other subclasses in the problem formulation layer. An advantage to encapsulating the creation code in a ProblemFormulation class is that it is not necessary to specify how many copies of each object need be created. This additional flexibility is useful because different QP algorithms need different numbers of instances of variable and residual classes.

Normally, an instance of ProblemFormulation will be given any parameters that it needs to build a compatible set of objects when it is created. Take, for example, the class QpExampleDense, which is a subclass of ProblemFormulation used to create objects for solving QPs of the form (1) using dense linear algebra. A partial prototype for the QpExampleDense class is as follows.

```
class QpExampleDense : public ProblemFormulation {
 protected:
  int mNx, mMy, mMz;
 public:
  QpExampleDense( int nx, int my, int mz );
};
```

When a QpExampleDense is created by code of the form

```
QpExampleDense * qp = new QpExampleDense( nx, my, mz );
```

it records the problem dimensions n, m_A, and m_C, allowing it subsequently to create objects of the right size.

Note that the ProblemFormulation class does not contain the declaration of an abstract makeData method. One normally needs additional information to create Data objects, namely, the problem data itself. A makeData method with no parameters is normally useless; on the other hand, no one set of parameters would be useful for all formulations. Therefore, there is no appropriate abstract definition of makeData.

6 Using Linear Algebra Objects

This section takes the form of a tutorial on elements of OOQP's linear algebra layer. It is intended for those who wish to use these linear algebra objects and operations concretely to define a new problem formulation. We have found these objects useful in implementing solvers for the problem formulations supplied with the OOQP distribution, and we believe they will also be useful to users who wish to implement solvers for their own special QP formulations. Users are not, however, compelled to use the OOQP linear algebra layer in implementing their own problem formulation layer; they may write their own code to store the data objects and to perform the linear algebra operations that are required by the interior-point algorithm.

The QP formulations and interior-point algorithms supplied with the OOQP distribution are written in terms of linear algebra operations in abstract classes, such as `OoqpVector`, `GenMatrix`, and `SymMatrix`. When we speak of using linear algebra objects "concretely," we mean accessing the data contained in these objects directly, in a manner that depends explicitly on how the data is stored. A code development process using concrete objects and operations is as follows. The user starts by creating objects that are instances of specific concrete subclasses of the abstract linear algebra classes, and manipulates these objects accordingly. Then, the user migrates to an abstract interface by systematically replacing the data-structure-dependent code in the problem formulation with mathematical operations from the abstract base classes. Finally, the user changes the type declarations of the variables from the concrete classes to abstract base classes such as `OoqpVector`, causing the compiler to disallow any remaining data-structure-dependent code. This development process of migrating from a working concrete QP formulation to an abstract QP formulation may be simpler than trying to use the abstract interface on the first pass. The material in this section will be helpful for users that follow this path.

We start in Section 6.1 by describing the reference counting scheme used to manage memory in OOQP. In Section 6.2, we describe `SimpleVector`, a class that can be used in place of arrays of double-precision numbers. Section 6.3 describes classes for storing and manipulating dense matrices, while Section 6.4 discusses classes for sparse matrices.

6.1 Reference Counting

Reference counting is a powerful technique for managing memory that helps prevent objects from being deleted accidentally or more than once. The technique is not limited to C++ code and, despite its name, is unrelated to the C++ concept of reference variables. Rather, the term means that we maintain a count of all "owning references" to an object and delete the object when this count becomes zero. An owning reference is a typically a pointer to an object that is a data member of an instance of another class. Consider, for instance, the following class.

```
class MyVariables : Variables {
    SimpleVector * mV;
public:
    SimpleVector&  v();
    SimpleVector * getV();
    void copyV( SimpleVector& w );
    MyVariables();
    MyVariables( SimpleVector * v );
    ~MyVariables();
};
```

Instances of `MyVariables` would hold an owning reference to a `SimpleVector` in the variable `mV`. In the reference counting scheme, the destructor for this class would be as follows.

```
MyVariables::~MyVariables()
{
    IotrRelease( &mV );
};
```

Rather than deleting `mV`, the destructor signals that it is no longer holding a reference to the object, so the reference count associated with this object is decremented. In correct code, every object has at least one owning reference. When the number of owning references has decreased to zero through calls to `IotrRelease`, the reference counting scheme deletes the object.

Usually, objects are created in the constructors of other objects and are released when the creating object no longer needs them, typically in the destructor. For instance, the constructor

```
MyVariables::MyVariables()
{
    mV = new SimpleVector(5);
}
```

creates a new `SimpleVector` object, and the corresponding destructor will release the owning reference to this object when the `MyVariables` object is finished with it.

Another common scenario is that a pointer to an object may be passed as a parameter to a method or constructor for another object, which may then wish to establish its own owning reference for the parameter object. This scenario arises in the following constructor.

```
MyVariables::MyVariables(SimpleVector * v_in )
{
    mV = v_in;
    IotrAddRef( &mV );
}
```

The call to `IotrAddRef` informs the reference counting scheme that a new owning reference to the `SimpleVector` has been established, so the counter associated with this object is incremented. If `IotrAddRef` had not been called, the reference counting scheme would assume that the object had declined to establish a new owing reference.

When objects are passed into methods as C++–style reference variables, rather than via pointers, owning references must not be established. For instance, the method

```
void MyVariables::copy( SimpleVector& w )
{
...
}
```

may not establish a new owing reference for its parameter `w`. A similar convention exists for the return values of functions. A return value that is a C++–style reference variable needs no special attention

```
SimpleVector& MyVariables::v()
{
    return *mV;
}
```

but if the return value is a pointer, then a new owning reference is *always* established, and so the reference count must be incremented via a call to `IotrAddRef`:

```
SimpleVector * MyVariables::getV()
{
    IotrAddRef( &mV );
    return mV;
}
```

A typical program makes few calls to `IotrAddRef` and `IotrRelease`. For the most part, one may simply call the `IotrRelease` function instead of the C++ operator `delete`.

Finally, we mention that OOQP contains a `SmartPointer` class that handles calls to `IotrAddRef` and `IotrRelease` automatically. This class has proven useful to the OOQP developers and is present in the OOQP distribution for others who wish to use it. We will not, however, describe it further in this document.

6.2 Using `SimpleVector`

`SimpleVector` is a class whose instances may be used in place of arrays of double precision numbers. It is a subclass of OOQP's abstract base vector class, `OoqpVector`, and all abstract operations of an `OoqpVector` are implemented in `SimpleVector`. However, there is one important additional feature: The operator `[]` has been defined for `SimpleVector`, which allows indexing to be used to access individual elements in the `SimpleVector` object. For example, the following piece of code involving `SimpleVector` objects `a`, `b`, and `c` is legal, provided that these vectors have compatible lengths.

```
void add( SimpleVector& a, SimpleVector& b, SimpleVector& c )
{
     for( int i = 0; i < a->length(); i++ ) {
         c[i] = a[i] + b[i];
     }
}
```

The elements of a `SimpleVector` may be passed to a legacy C routine in the manner demonstrated in the following code fragment, which calls the C routine `norm` on the elements of `a`.

```
extern "C"
double norm( double a[], int len );
double mynorm( SimpleVector& a )
{
     return norm( a->elements(), a->length() );
}
```

(Indeed, in most cases, we could use the calling sequence

```
norm( &a[0], a->length() );
```

but this call will fail for vectors of length zero.)

`SimpleVector` objects may be created via calls to a constructor of the following form:

```
    // Create a vector of length 5
    SimpleVector * a = new SimpleVector( 5 );
```

When interfacing with non-OOQP code, however, it may be preferable to invoke an alternative constructor that uses an existing array of doubles to store the elements of the new `SimpleVector` instance. Use of this constructor is demonstrated by the following code fragment.

```
    double * v = new double[5];
    SimpleVector * b = new SimpleVector( v, 5 );
```

The array `v` will be used as the storage location for the elements of `b` and will not be deleted when `b` is deleted.

We recommend that users always use operator `new` to create new instances of `SimpleVector`. Creating `SimpleVector` on the stack is not supported and may cause unforeseen problems. In other words, users should not create variables of type `SimpleVector`, but rather should create pointers and references to instances of `SimpleVector`, as in the examples above.

6.3 Using `DenseGenMatrix` and `DenseSymMatrix`

`DenseGenMatrix` is a class that represents matrices stored as a dense array in row-major order. `DenseSymMatrix` also stores matrix elements in a dense array but represents symmetric (rather than general) matrices. Row and columns indices for the matrices start at zero, following C and C++ conventions.

The indexing operator `[]` is defined appropriately for both `DenseGenMatrix` and `DenseSymMatrix`. The following code fragment, for example, is legal.

```
int myFunc( DenseGenMatrix& M )
{
  for( int i = 0; i < M.rows(); i++ ) {
    for( int j = 0; j < M.columns(); j++ ) M[i][j] = i * 10 + j;
  }
}
```

`DenseSymMatrix` stores its elements in the lower triangle of the matrix; the result of accessing the upper triangle is undefined. An example of code to fill a `DenseSymMatrix` is the following.

```
int mySymFunc( DenseSymMatrix& M )
{
  for( int i = 0; i < M.size(); i++ ) {
    for( int j = 0; j <= i; j++ ) M[i][j] = i * 10 + j;
  }
}
```

The elements of a dense matrix may be passed to legacy C code by invoking the method `elements`, which returns a pointer to the full matrix laid out in row major order. An example is as follows.

```
void myFactor( DenseGenMatrix& M )
{
    factor( M.elements(), M.rows(), M.columns() );
}
```

Both `DenseGenMatrix` or `DenseSymMatrix` provide the method `mult`, which performs matrix-vector multiplication. For instance, if `M` is an instance of either class, the function

```
void func(double beta,  SimpleVector& y,
          double alpha, SimpleVector& x)
{
    M.mult( beta, y, alpha, x )
}
```

perform the computation $y \leftarrow \beta y + \alpha M x$. Similarly, `transMult` computes $y \leftarrow \beta y + \alpha M^T x$.

These classes contain no member functions to factor the matrices. Users may either program their own factorization on the elements of the matrix or use one of the linear solvers from the OOQP distribution. For a `DenseSymMatrix` an appropriate linear solver is `DeSymIndefSolver`. We demonstrate the use of this solver in the following sample code, which solves a linear system with a coefficient matrix `M` (an instance of `DenseSymMatrix`) and right-hand side `x` (an instance of `SimpleVector`). The result is returned in the `SimpleVector` object `y`.

```
void mySolve( SimpleVector& y, DenseSymMatrix * M,
              SimpleVector& x )
{
     DeSymIndefSolver * solver = new DeSymIndefSolver( M );

     solver->matrixChanged();
     y.copyFrom( x );
     solver->solve( y );

     IotrRelease( &solver );
}
```

The `matrixChanged` method performs an in-place factorization on the values of `M`, overwriting the original values of this matrix with the values of its factors. The `solve` method uses the factors to compute the solution to the system.

If it is known that `M` is positive definite, the solver `DeSymPSDSolver` should be used in place of `DeSymIndefSolver`. OOQP does not supply linear solvers for instances of `DenseGenMatrix`.

A `DenseGenMatrix` may be created by using the operator `new`. The following code will create a `DenseGenMatrix` with five rows and three columns.

```
DenseGenMatrix * pgM = new DenseGenMatrix( 5, 3 );
```

Instances of `DenseSymMatrix` are necessarily square, so only one argument is needed for the constructor. The following code creates a `DenseSymMatrix` with five rows and columns.

```
DenseSymMatrix * psM = new DenseSymMatrix( 5 )
```

As in the `SimpleVector`class, other constructors can be invoked to use an existing array of doubles as storage space for the new `DenseGenMatrix` or `DenseSymMatrix` instances, as demonstrated in the following code fragment.

```
double * gen = new double[5 * 3]
double * sym = new double[5 * 5];
DenseGenMatrix * pgM = new DenseGenMatrix( gen, 5, 3 );
DenseSymMatrix * psM = new DenseSymMatrix( sym, 5 )
```

The arrays `gen` and `sym` will not be deleted when the matrices `pgM` and `psM` are created or freed.

6.4 Using `SparseGenMatrix` and `SparseSymMatrix`

In many practical instances, the matrices used to formulate the QP are large and sparse. General sparse matrices and sparse symmetric matrices are represented by `SparseGenMatrix` and `SparseSymMatrix`, respectively. Unlike their dense counterparts, `SparseGenMatrix` and `SparseSymMatrix` cannot be used as drop-in replacements for an array of doubles because they do not define the indexing operator `[]`.

The data elements of these matrices are stored in a standard compressed format known as *Harwell-Boeing* format. In the `SparseGenMatrix` class, the elements are stored in row-major order, and, as in the dense case, the row and column indices start at zero. Harwell-Boeing format encodes the matrix in three arrays—two arrays of integers and one array of doubles. For an `m` × `n` general matrix containing `len` nonzero elements, these arrays are represented by three data structures within the sparse matrix classes, as follows.

```
int     krowM[m+1];
int     jcolM[len];
double      M[len];
```

For each index `i` $= 0, 1, \ldots, m-1$, the nonzero elements from row `i` are stored in locations `krowM[i]` through `krowM[i+1]-1` of the vector `M`. (Recall that we index the rows and columns by $0, 1, \ldots, m-1$ and $0, 1, \ldots, n-1$, respectively.) The column index of each nonzero is stored in the corresponding location of `jcolM`. In other words, for any `k` between `krowM[i]` and `krowM[i+1]-1` (inclusive) the `(i,jcolM[k])` element of the matrix is stored in `M[k]`.

For a symmetric matrix, an instance of `SparseSymMatrix` stores only the nonzero elements in the lower triangle of the matrix. Otherwise the format is identical to that described above for general matrices.

Perhaps the simplest way to understand the format is to study the following code sample, which prints out the elements of the matrix in row-major order.

```
for( int i = 0; i < m; i++ ) {
   for( int k = krowM[i]; k < krowM[i+1]; k++ ) {
       cout << "Row: "   << i    << "column: " << jcolM[k]
            << "value: " << M[k] << endl;
   }
}
```

As for the dense classes, the `SparseGenMatrix` and `SparseSymMatrix` classes provide `mult` and `transMult` methods, which perform matrix-vector multiplications.

No methods within the sparse matrix classes perform factorizations of the matrices. Classes with this functionality are supplied elsewhere in the OOQP distribution, however. The default sparse direct linear equation solver in the OOQP distribution is the code MA27 from the Harwell Sparse Library, wrapped in a way that makes it callable from C++ code. The following code fragment solves a system of linear equations involving a sparse symmetric indefinite matrix. On input, `M` contains the coefficient matrix while `x` contains the right-hand side. Neither `M` nor `x` is changed in the call, but `y` is replaced by the solution of the linear system.

```
void mySparseSolve( SimpleVector& y, SparseSymMatrix * M,
                    SimpleVector& x )
{
     Ma27Solver * solver = new Ma27Solver( M );

     solver->matrixChanged();
     y.copyFrom( x );
     solver->solve( y );

     IotrRelease( &solver );
}
```

Users are also free to supply their own sparse solvers. If the solver accepts Harwell-Boeing format, the three arrays that encode the matrix can be passed individually as arguments, as can the elements of the right-hand side and the solution. If `M` is a `SparseGenMatrix` or `SparseSymMatrix` object and `x` and `y` are `SimpleVector` objects, then the interface to the user-supplied solve routine may be as follows.

```
mySparseSolver( M.krowM(), int m, M.jcolM(), M.M(),
                x.elements(), y.elements() );
```

In interior-point algorithms, one frequently must solve a sequence of linear systems in which the matrices differ from each other only in the diagonal elements. Consequently, we supply the methods `fromGetDiagonal` and `atPutDiagonal`, whose function is to transfer the diagonal elements between an instance of a sparse matrix class and an instance of the `SimpleVector` class. For example, the following code copies diagonal elements $(4,4)$ through $(8,8)$ inclusive from the `SparseSymMatrix` object `M` into the `SimpleVector`object `d`.

```
SimpleVector * getMe( SparseSymMatrix& M )
{
    SimpleVector * d = new SimpleVector(5);
    M.fromGetDiagonal( 4, *d );
    return d;
}
```

To copy the elements from `d` into the diagonals of `M`, one would use a call of the form

```
M.atPutDiagonal( 4, *d );
```

which overwrites diagonal elements $(4,4)$ through $(3+r, 3+r)$ with the elements of `d`, where r is the number of elements in `d`.

Instances of `SparseGenMatrix` or `SparseSymMatrix` can be created by first filling the three arrays that encode the matrix in Harwell-Boeing format and then calling a constructor. For general sparse matrices, this call has the following form:

```
SparseGenMatrix * sgm
   = new SparseGenMatrix( m, n, len, krowM, jcolM, M );
```

where `m` and `n` are the number of rows and columns, respectively, `len` is the number of nonzero elements, and `krowM`, `jcolM`, and `M` are the three arrays discussed above. For sparse matrices, the corresponding call is

```
SparseSymMatrix * ssm
    = new SparseSymMatrix( m, len, krowM, jcolM, M );
```

We emphasize that the arrays `krowM`, `jcolM`, and `M` are not copied but rather are used directly. They are not deleted when the sparse matrix instances `sgm` or `ssm` are freed.

Alternative constructors can be used when a description of the matrix is available in *sparse triple* format. In this simple format, the matrix is encoded in two integer arrays and one double array, all of which have length equal to the number of nonzeros in the matrix. (In the case of a symmetric matrix, only the lower triangle of the matrix is stored.) By defining `nnz` to be the number of stored nonzeros, and defining the three arrays as follows,

```
int irow[nnz];
int jcol[nnz];
double A[nnz];
```

we have for any `k` in the range `0,...,nnz-1` that the element at row `irow[k]` and column `jcol[k]` has value `A[k]`. The elements in this format can be sorted into row-major order by calling another routine from the OOQP distribution, `doubleLexSort`, in the following way.

```
doubleLexSort( irow, nnz, jcol, A );
```

Given the matrix in this form, with the arrays sorted into row-major form, we can build an instance of `SparseGenMatrix` or `SparseSymMatrix` by first calling a constructor with the matrix dimensions and the number of non-zeros as arguments, as follows.

```
SparseGenMatrix * sgm = new SparseGenMatrix( m, n, nnz );
SparseSymMatrix * ssm = new SparseSymMatrix( m, nnz );
```

We can then call the method `putSparseTriple`, available in both classes, to place the information in `irow`, `jcol`, and `A` into `sgm` or `ssm`. This call has the following form.

```
sgm.putSparseTriple( irow, nnz, jcol, A, info );
```

The output parameter `info` will be set to zero if `sgm` is large enough to hold the elements in `irow`, `jcol`, and `A`. Otherwise it will be set to one.

7 Specializing Linear Algebra Objects

The solver supplied in the OOQP distribution for the formulation (2) with sparse data uses the MA27 [9] sparse indefinite linear equation solver from the Harwell Subroutine Library to solve the systems of linear equations that arise at each interior-point iteration. Some users may wish to replace MA27 with a different sparse solver. (Indeed, we implemented a number of different solvers during the development of OOQP.) Users also may want to make other modifications to the linear algebra layer supplied with the distribution. For example, it may be desirable to alter the representations of matrix and vectors that are implemented in OOQP's linear algebra layer, by creating new subclasses of `OoqpVector`, `SymMatrix`, `GenMatrix`, and `DoubleStorage`. One motivation for doing so might be to embed OOQP in an applications code that defines its own specialized matrix and vector storage schemes.

In Section 7.1, we describe the process of replacing MA27 by a new linear solver. Section 7.2 discusses the subclassing of objects in OOQP's linear algebra layer that may be carried out by users who wish to specialize the representations of matrices and vectors.

7.1 Using a Different Linear Equation Solver

The MA27 solver for symmetric indefinite systems of linear equations is an efficient, freely available solver from the Harwell Subroutine Library that is widely used to solve the linear systems that arise in the interior-point algorithm applied to sparse QPs of the form (2). By the nature of OOQP's design, however, an advanced user can substitute another solver without much trouble. This section outlines the steps that must be taken to do so. We focus on replacing a sparse linear solver because this operation is of greater practical import than replacing a dense solver and because there are a greater variety of sparse factorization codes than of dense codes.

7.1.1 Creating a Subclass of `DoubleLinearSolver`

The first step is to create a subclass of the `DoubleLinearSolver` class. A typical subclass will have the following prototype.

```
#include "DoubleLinearSolver.h"
#include "SparseSymMatrix.h"
#include "OoqpVector.h"

class MyLinearSolver : public DoubleLinearSolver {
  SparseSymMatrix * mStorage;
public:
  MyLinearSolver( SparseSymMatrix * storage );
  virtual void diagonalChanged( int idiag, int extent );
  virtual void matrixChanged();
  virtual void solve ( OoqpVector& vec );
  virtual ~MyLinearSolver();
};
```

Each `DoubleLinearSolver` object is associated with a matrix. Therefore, a typical constructor for a subclass `MyLinearSolver` of `DoubleLinearSolver` would be as follows.

```
MyLinearSolver::MyLinearSolver( SparseSymMatrix * ssm )
{
    IotrAddRef( &ssm );
    mMat = ssm; // Here mMat is a data member of MyLinearSolver.
}
```

The call to `IotrAddRef` establishes an owning reference to the matrix (see Section 6.1). It must be balanced by a call to `IotrRelease` in the destructor, as follows.

```
MyLinearSolver::~MyLinearSolver()
{
    IotrRelease( &mMat );
}
```

When the linear solver is first created, the matrix with which it is associated will not typically contain any data of interest to the linear solver. Once the contents of the matrix have been loaded, the interior-point algorithm may call the `matrixChanged` method, which triggers a factorization of the matrix. Subsequently, the algorithm performs one or more calls to the `solve` method, each of which uses the matrix factors produced in `matrixChanged` to solve the linear system for a single right-hand side.

Calls to `matrixChanged` typically occur once at each interior-point iteration. It is assumed that the sparsity structure of the matrix does not change between calls to `matrixChanged`; only the data values will be altered. This assumption, which holds for all popular interior-point algorithms, allows subclasses of `DoubleLinearSolver` to cache information about the sparsity structure of the matrix and its factors and to reuse this information throughout the interior-point algorithm.

The `diagonalChanged` method supports those rare solvers that take a different action if only the diagonal elements of the matrix are changed (while off-diagonals are left untouched). Most solvers cannot do anything interesting in this case; a typical implementation of `diagonalChanged` simply calls `matrixChanged`, as follows.

```
void MyLinearSolver::diagonalChanged( int idiag, int extent )
{
   this->matrixChanged();
}
```

The implementation of `matrixChanged` and `solve` depends strongly on the sparse linear system solver in use, as well as on the data format used to store the sparse matrices. Section 6.4 describes the data format used by our sparse matrix classes. The convention in OOQP is that sparse linear solvers must not act destructively on the matrix data. In some instances, this restriction requires a copy of part of the matrix data to be made before factorization begins. Typically, however, this restriction is not too onerous because the fill-in that occurs during a typical factorization would make it necessary to allocate additional storage in any case.

The opposite convention is in place for subclasses of `DoubleLinearSolver` that operate on dense matrices. These invariably perform the factorization in place, overwriting the matrix data. While having two different conventions is far from ideal, we felt it unwise to enforce unnecessary copying of matrices in the dense case for the sake of conformity.

7.1.2 Creating a Subclass of `ProblemFormulation`

Having defined and implemented a new subclass of `DoubleLinearSolver`, the user must now arrange so that the new solver, rather than the default linear solver, is created and used by the quadratic programming algorithm.

In Section 5.2.4 we described how subclasses of `LinearSystem` are used to solve the linear systems arising in interior point algorithms. We give specific examples of how an instance of `LinearSystem` designed to handle our example QP formulation (1) assembles a matrix and right-hand side of a system to be passed to a general-purpose linear solver, which would normally be an instance of a subclass of `DoubleLinearSolver`. In this manner, we have separated the problem-specific reductions and transformations, which are the responsibility of instances of `LinearSystems`, from the solution of matrix equations, which are the responsibility of instances of `DoubleLinearSolver`.

On the other hand, the nature and properties of the `DoubleLinearSolver` will affect the efficiency and feasibility of problem-specific reductions and transformations. Moreover, when the `LinearSystem` assembles the matrix equations to be solved, it must assemble the matrix in a format acceptable to the linear solver. To ensure that a compatible set of objects is created, the `DoubleLinearSolver`, the matrix it operates on, and `LinearSystem` are created in the same routine.

As we discussed in Section 5.2.5, OOQP contains classes—specifically, subclasses of `ProblemFormulation`—that exist for the express purpose of creating a compatible set of objects for implementing solvers for QPs with a given formulation. The `makeLinsys` methods of these classes is, naturally, the place in which appropriate instances of subclasses of `LinearSystem` are created. As we discussed in the earlier section, code for creating a compatible collection of objects can become quite involved, so it is natural to collect this code in one place. OOQP's approach is to place this code in the methods of subclasses of `ProblemFormulation`.

To use a new `DoubleLinearSolver` with an existing problem formulation, one must create a new subclass of `ProblemFormulation`. Since the code needed to implement a subclass of `ProblemFormulation` depends strongly on the specific data structures of the problem formulation, it is difficult to give general instructions on how to write such code. However, we describe below the appropriate procedure for users who wish to work with a sparse variant of the `QpGen` formulation (2), changing only the `DoubleLinearSolver` object and retaining the data structures and other aspects of the formulation that are used in the default (MA27-based) solver supplied with the OOQP distribution. To accommodate such users, we have created a subclass of `ProblemFormulation` called `QpGenSparseSeq`, which holds the code common to all formulations of `QpGen` that uses sparse sequential linear algebra. Users can create a subclass of `QpGenSparseSeq` in the following way.

```
class QpGenSparseMySolver : public QpGenSparseSeq {
public:
  QpGenSparseMySolver( int nx, int my, int mz,
               int nnzQ, int nnzA, int nnzC );
  LinearSystem * makeLinsys( Data * prob_in );
};
```

The constructor may be implemented by simply passing its arguments through to the parent constructor.

```
QpGenSparseMySolver::QpGenSparseMySolver( int nx, int my, int mz,
                      int nnzQ, int nnzA, int nnzC ) :
  QpGenSparseSeq( nx, my, mz, nnzQ, nnzA, nnzC )
{
}
```

The implementation of the `makeLinsys` method is too solver-specific to be handled by generic code, but the following code fragment, which is based on the file `src/QpGen/QpGenSparseMa27.C`, may give a useful guide.

```
LinearSystem * QpGenSparseMySolver::makeLinsys( Data * prob_in )
{
  QpGenData * prob = (QpGenData *) prob_in;
  int n = nx + my + mz;

  // Include diagonal elements in the matrix, even if they are
  // zero. Enforce by inserting a diagonal of zeros.
  SparseSymMatrix *  Mat =
     new SparseSymMatrix( n, n + nnzQ + nnzA + nnzC );

  SimpleVector * v = new SimpleVector(n);
  v->setToZero();
  Mat->setToDiagonal(*v);
  IotrRelease( &v );

  prob->putQIntoAt( *Mat, 0, 0 );
  prob->putAIntoAt( *Mat, nx, 0);
  prob->putCIntoAt( *Mat, nx + my, 0 );
  // The lower triangle is now [ Q   * ]
  //                           [ A   C ]

  MyLinearSolver  * solver = new MyLinearSolver( Mat );
  QpGenSparseLinsys  * sys
      = new QpGenSparseLinsys( this, prob,
                              la, Mat, solver );

  IotrRelease( &Mat );

  return sys;
}
```

We emphasize that users who wish to alter the MA27-based implementation of the solver for the sparse variant of (2) only by substituting another solver with similar capabilities to MA27 will be able to use these examples directly, by inserting the names they have chosen for their solver into these code fragments.

7.2 Specializing the Representation of Vectors and Matrices

Although the OOQP linear algebra layer provides a comprehensive set of linear algebra classes, as described in Section 6, some users may wish to use a different set of data structures to represent vectors and matrices. This could happen, for instance, when the user needs to embed OOQP in a larger program with its own data structures already defined. The design of OOQP is flexible enough to accommodate user-defined linear algebra classes. In this section, we outline how such classes can be written and incorporated into the code.

The vector and matrix classes need to provide methods that, for the most part, represent simple linear algebra operations, such as inner products and saxpy operations. The names are often self-explanatory; those that are specific to the needs of the interior-point algorithm are described in the class documentation accompanying the OOQP distribution. We note, however, that efficient implementation of these operations can require a significant degree of expertise, especially when the data structures are complex. We recommend that users search for an existing implementation that is compatible with their data storage needs before attempting to implement the methods themselves. As a rule, it is easier to create OOQP vectors and matrix classes that wrap existing libraries than to write efficient code from scratch.

To specialize the representation of vectors and matrices, one must create subclasses of the following abstract classes:

OoqpVector: Represents mathematical vectors.

GenMatrix: Represents nonsymmetric and possibly nonsquare matrices as mathematical operators.

SymMatrix: Represents symmetric matrices as mathematical operators.

DoubleStorage: Contains the concrete code for managing the data structures that hold the matrix data.

DoubleLinearSolver: Solves linear systems with a specific type of matrix as its coefficient.

LinearAlgebraPackage: Creates instances of vectors and matrices.

We have outlined how to create a new subclass of `DoubleLinearSolver` in the preceding section. The remainder of this section will focus on the other new subclasses. We will not describe the methods of these classes in detail, because the majority of them are familiar mathematical operations. We refer the reader to the class documentation accompanying the OOQP distribution for a description of these methods.

The code in the problem formulation layer is implemented b using the abstract linear algebra classes described above. Objects in the problem formulation layer can be created by using instances of user-defined subclasses to represent linear algebra objects. We have discussed in the preceding section and in Section 5.2.5 the use of the `ProblemFormulation` class in creating a compatible set of objects in the problem formulation layer. Users who wish to specialize the representation of vectors and matrices will also need to create at least one new subclass of `ProblemFormulation`.

The header file `src/Vector/OoqpVector.h` defines the abstract vector class. The header files defining the other abstract classes may be found in the subdirectory `src/Abstract`. As a rule, the files needed to define a particular implementation of the linear algebra layer are given their own subdirectory. Some existing implementations are located in the following directories.

```
src/DenseLinearAlgebra/
src/SparseLinearAlgebra/
src/PetscLinearAlgebra/
```

Users may wish to refer to these implementations as sample code. Because `DenseLinearAlgebra` and `SparseLinearAlgebra` share the same vector implementation, `SimpleVector`, this code is located in its own directory, named `src/Vector`. Several linear solvers have also been given their own subdirectories below the directory `src/LinearSolvers`.

OOQP does not attempt to force matrices and vectors that are represented in significantly different ways to work together properly. For instance, the distribution contains no method that multiplies a matrix stored across several processors by a vector whose data is stored on a tape drive attached to a single processor. Nor do we perform any compile-time checks that only compatible linear algebra objects are used together in a particular implementation. Such checks would require heavy use of the C++ template facility, and we were wary of using templates because of the portability issues and other costs that might arise. Rather, we endeavored to design our problem formulation classes in a way that makes it difficult to mix representations of linear algebra objects accidentally. (We suggest that users who are modifying the matrix and vector representations follow this design.) Commonly, we downcast at the start of a method. For example, the following code fragment downcasts from the abstract `OoqpVector` class to the `MyVector` class, which the `mult` method in `MySymMatrix` is intended to use.

```
void MySymMatrix::mult ( double beta,  OoqpVector& y_in,
                double alpha, OoqpVector& x_in )
{
  MyVector & y = (MyVector &) y_in;
  MyVector & x = (MyVector &) x_in;
}
```

Subclasses of `DoubleStorage` are responsible for the physical storage of matrix data on a computer. The physical data structure might be as simple as a dense two-dimensional array. In a distributed-computing setting, it could be much more complex. Instances of `DoubleStorage` are rarely used in an abstract setting. The code will know precisely what type of `DoubleStorage` is being used and what concrete data structures are being used to implement it. Thus, many of the methods of a subclass of `DoubleStorage` will be data-structure specific.

By contrast, each subclass of `DoubleStorage` will be associated with subclasses of `GenMatrix` and `SymMatrix` that are used primarily in an abstract, data-structure-independent fashion. Subclasses of `GenMatrix` and `SymMatrix` generally implement their methods by calling the structure-specific methods of a subclass of `DoubleStorage`. By using this design in OOQP, we were able to separate abstract mathematical manipulations of matrices and vectors from details of their representation. Accordingly, in creating their subclasses, users should feel free to implement any structure-dependent methods they need in their implementation of the `DoubleStorage` subclass, whereas their implementations of the `GenMatrix` and `SymMatrix` subclasses should adhere more closely to the abstract interface.

We emphasize the following points for users who wish to create subclasses from the matrix classes: Matrices in OOQP are represented in row-major form, and row and column indices start at zero. Adherence to these conventions will make it easier to refer to existing implementations in designing new versions of the linear algebra layer. Symmetric matrices in OOQP store their elements in the lower triangle of whatever data structure is being used. For some linear algebra implementations, it might be desirable to symmetrize the structure, explicitly storing all elements of the matrix, despite the redundancy this entails. If this approach is chosen, one should be careful to treat the matrix as if only the lower triangle were significant, as subtle bugs may arise otherwise.

Subclasses of `OoqpVector` represent mathematical vectors and should adhere closely to the abstract vector interface. The methods of `OoqpVector` typically operate on the entire vector. Access to individual elements of the vector should be avoided.

Users who implement their own representation of vectors and matrices will also need to specialize the `LinearAlgebraPackage` class. This class has the following interface (see `src/Abstract/LinearAlgebraPackage.h`).

```
class LinearAlgebraPackage {
protected:
  LinearAlgebraPackage() {};
  virtual ~LinearAlgebraPackage() {};
public:
  virtual SymMatrix * newSymMatrix( int size, int nnz ) = 0;
  virtual GenMatrix * newGenMatrix( int m, int n, int nnz ) = 0;
  virtual OoqpVector * newVector( int n ) = 0;
  // Access the type name for debugging purposes.
  virtual void whatami( char type[32] ) = 0;
};
```

Instances of `LinearAlgebraPackage` do nothing more than create vectors and matrices on request. Our reason for including this class in the OOQP design is to provide a mechanism by which abstract code can create new vectors and matrices that are compatible with existing objects. The code cannot call the operator `new` on a type name and still remain abstract. Use of `LinearAlgebraPackage`, on the other hand, allows users to create new vectors and matrices, without referring to specific vector and matrix types, by invoking the `newVector`, `newSymMatrix`, and `newGenMatrix` methods of an instance of `LinearAlgebraPackage`.

Instances of `LinearAlgebraPackage` are never deleted. Because these instances are small, the memory overhead is normally insignificant. However, it is customary to arrange so that each subclass of `LinearAlgebraPackage` has at most one instance, as in the following code fragment.

```
class MyLinearAlgebraPackage : public LinearAlgebraPackage {
protected:
  DenseLinearAlgebraPackage() {};
  virtual ~DenseLinearAlgebraPackage() {};
public:
  static MyLinearAlgebraPackage * soleInstance();
  // ...
}

MyLinearAlgebraPackage * MyLinearAlgebraPackage::soleInstance()
{
  static
  MyLinearAlgebraPackage * la = new MyLinearAlgebraPackage;

  return la;
}
```

The use of such a scheme is optional.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide.* SIAM, Philadelphia, 1992.

[2] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Lois Curfman McInnes, and Barry F. Smith. PETSc home page. `www.mcs.anl.gov/petsc`, 2001.

[3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202, Boston, 1997. Birkhauser Press.

[4] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.0, Argonne National Laboratory, 2001.

[5] 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.

[6] J. W. Demmel, J. R. Gilbert, and X. S. Li. *SuperLU User's Guide*, 1999. Available from `www.nersc.gov/ xiaoye/SuperLU/`.

[7] 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.

[8] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Clarendon Press, Oxford, 1986.

[9] 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.

[10] R. Freund. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM Journal on Scientific Computing*, 14:470–482, 1993.

[11] R. Freund and N. Nachtigal. QMR: A quasi-minimal residual method for non-Hermitian linear systems. *Numerische Mathematik*, 60:315–339, 1991.

[12] E. M. Gertz and S. J. Wright. Object-oriented software for quadratic programming. Preprint ANL/MCS-P999-0901, Mathematics and Computer Science Division, Argonne National Laboratory, September 2001.

[13] J. Gondzio. Multiple centrality corrections in a primal-dual method for linear programming. *Computational Optimization and Applications*, 6:137–156, 1996.

[14] HSL: A collection of Fortran codes for large scale scientific computation, 2000. Full details in http://www.numerical.rl.ac.uk/hsl.

[15] C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations.* Number 16 in Frontiers in Applied Mathematics. SIAM Publications, Philadelphia, 1995.

[16] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *Application for Computing Machinery Transactions on Mathematical Software*, 5:308–323, 1979.

[17] I. Maros and C. Mészáros. A repository of convex quadratic programming problems. *Optimization Methods and Software*, 11 and 12:671–681, December 1999.

[18] S. Mehrotra. Asymptotic convergence in a generalized predictor-corrector method. Technical Report, Dept. of Industrial Engineering and Management Science, Northwestern University, Evanston, Ill., October 1992.

[19] S. Mehrotra. On the implementation of a primal-dual interior point method. *SIAM Journal on Optimization*, 2:575–601, 1992.

[20] B. A. Murtagh. *Advanced Linear Programming: Computation and Practice.* McGraw-Hill, New York, 1981.

[21] C. V. Rao, S. J. Wright, and J. B. Rawlings. Application of interior-point methods to model predictive control. *Journal of Optimization Theory and Applications*, 99:723–757, 1998.

[22] H. Walker. Implementation of the GMRES method using Householger transformations. *SIAM Journal on Scientific and Statistical Computing*, 9:815–825, 1989.

[23] S. J. Wright. *Primal-Dual Interior-Point Methods.* SIAM Publications, Philadelphia, 1997.

A COPYRIGHT

COPYRIGHT NOTIFICATION

E. MICHAEL GERTZ gertz@mcs.anl.gov
Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL 60439-4844

STEPHEN J. WRIGHT swright@cs.wisc.edu
Computer Sciences Department
University of Wisconsin
1210 West Dayton Street
Madison, WI 53706 FAX: (608)262-9777

GOVERNMENT LICENSE

DISCLAIMER