An Object-oriented Approach to Semidefinite Programming

Yuzhen Ge[†] Layne T. Watson[‡] Emmanuel G. Collins, Jr.[*]

Abstract

An object-oriented design and implementation of a primal-dual algorithm for solving the semidefinite programming problem is presented. The advantages of applying the object-oriented methodology to numerical computations, in particular to an interior point algorithm for semidefinite programming, or for solving other types of linear matrix inequalities are discussed. One object-oriented design of the primal-dual algorithm and its implementation using C++ is presented. The performance of the C++ implementation is compared with that of a procedural C implementation, and while the performance of the C++ implementation is comparable to that of the C implementation, the resulting code is easier to read, modify, and maintain.

Key words: linear matrix inequality, object oriented, primal dual algorithm, scientific computation, semidefinite programming

CR categories: D.2.2, G.1.6

1 Introduction

Object-oriented design and programming has been a major theme in software engineering in recent years. Traditional design, the main software design paradigm until about the mid 1980s, concentrates on the actions that a system has to take and decomposes the system into separate units or modules according to their functionalities. In object-oriented design a system to be modeled is viewed as a collection of objects, each of which has its own attributes and the operations performed on an object or functions acting on an object are also defined in one syntactic unit. Objects communicate by passing messages or by calling functions from other objects which provide services. Object-oriented design is developing an object-oriented model of a system and can be realized (implemented) by object-oriented programming using languages such as C++, FORTRAN 90, or Smalltalk.

The advantages of object-oriented design and programming have been described widely elsewhere [1]. A short summary will be provided here. First, an object is an independent entity that is encapsulated in one syntactic unit. The definition of an object consists of the definition of the attributes of the object along with operations that can be performed on the object and the services or function calls provided by the object. Encapsulation hides the implementation details of an object and makes the program easier to read and modify. Any subsequent change to the program can be localized, making the resulting program more easily maintained.

The work of Yuzhen Ge and Emmanuel G. Collins, Jr. was supported in part by Air Force Office of Scientific Research grant F49620-95-1-0244 and the work of Layne T. Watson was supported in part by Air Force Office of Scientific Research grant F49620-92-J-0236 and Department of Energy grant DE-FG05-88ER25068/A004.

[†] Department of Mathematics and Computer Science, Butler University, Indianapolis, IN 46208.

[‡] Departments of Computer Science and Mathematics, Virginia Polytechnic Institute & State University, Blacksburg, VA 24061-0106.

[*] Department of Mechanical Engineering, Florida A&M-Florida State University, Tallahassee, FL 32310-6046.

The second advantage is information hiding. Definitions of an object which need not be known to other objects are inaccessible to other objects, preventing them from being changed accidentally. In other words, information hiding makes implementation details of an object inaccessible to other objects. However, the designer has the freedom to decide what to hide and what not to hide.

The third advantage is code reuse. Inheritance enables the definition of a new object, which can be viewed as a subclass of an existing object, without having to repeat some of the details. The new object can inherit attributes or operations from its ancestor. Inheritance is one way to support reuse of existing objects. There are different kinds of reuse in object-oriented programming; inheritance is only one of them.

One of the most popular object-oriented programming languages is C++ [11], which is used to implement the algorithm of this paper. Some of the reasons why C++ is so widely used are upward compatibility with C, design emphasis on efficiency and performance, and the availability of many useful libraries and tools. For instance, the Gnu C++ compiler and other tools are available on a wide range of platforms and provide good performance, programming environments, and reasonable compliance with ANSI standards.

There are many available libraries such as IML++[6], SparseLib++ [5] [9], STL[10] [8], and others which emphasize numerical computation. One notable package is LAPACK++, developed by Dongarra et al. [4], which is a C++ interface to LAPACK and BLAS. [4] has shown that performance of programs using the package is comparable to calling LAPACK and BLAS directly, and can at the same time reap the benefits of object-oriented programming.

This paper contains the result of object-oriented design and implementation of an algorithm for semidefinite programming. Semidefinite programming refers to minimizing a linear function subject to a linear matrix inequality [12]. That is,

$$\begin{aligned} &\underset{x \in \mathbf{R}^m}{\text{minimize}} \ \ c^T x \\ &\text{subject to } F(x) \geq 0, \end{aligned} \tag{1}$$

where

$$F(x) \equiv F_0 + \sum_{i=1}^{m} x_i F_i,$$

$c \in \mathbf{R}^m$, and $F_0, \ldots, F_m \in \mathbf{R}^{n \times n}$ are symmetric matrices. $F(x) \geq 0$ means that $F(x)$ is positive semidefinite.

Many problems in controls engineering can be cast in terms of a semidefinite programming problem [12]. Since a semidefinite programming problem is a convex optimization problem, which can be solved by interior point methods [7], it has attracted the attention of many researchers in interior point methods. There is a C implementation of a primal-dual algorithm for solving the semidefinite programming problem [13]. A C++ implementation of that primal-dual algorithm for the semidefinite programming problem is developed here to explore the possible benefits of object-oriented design. Because of the similarity of the primal-dual algorithm with other interior point algorithms for solving the semidefinite programming problem, the design and implementation methodology developed here can be easily modified and applied to other interior point algorithms.

The performance of a C++ implementation of the primal-dual algorithm for semidefinite programming is compared with the existing C implementation of the same algorithm from [13]. While the CPU times of the two implementations are comparable to each other, the

C++ version offers the advantages mentioned earlier in this section. Segments of the code will be used to illustrate object-oriented features of the implementation.

Section 2 briefly sketches the primal-dual algorithm for semidefinite programming that will be used to illustrate the object-oriented design methodology. In Section 3, the details of an object-oriented design of the primal-dual algorithm will be given. The implementation using C++ will be described in Section 4. Comparison and discussion of the C and C++ results will be given in Section 5.

2 Primal-dual algorithm for semidefinite programming

The primal-dual algorithm for solving the semidefinite programming program given in detail in [12] will be described briefly here. The dual problem associated with the semidefinite program (1) is

$$\begin{aligned} &\underset{Z\in\mathbf{R}^{n\times n}}{\text{maximize}} \quad -\text{tr } F_0 Z \\ &\text{subject to tr } F_i Z = c_i, \quad i = 1,\ldots,m \\ &\qquad Z = Z^T, Z \geq 0. \end{aligned} \tag{2}$$

The primal-dual method can be interpreted as solving the primal-dual optimization problem

$$\begin{aligned} &\underset{x\in\mathbf{R}^m, Z\in\mathbf{R}^{n\times n}}{\text{minimize}} \quad c^T x + \text{tr } F_0 Z \\ &\text{subject to tr } F_i Z = c_i, \quad i = 1,\ldots,m \\ &\qquad F(x) \geq 0, Z \geq 0, Z = Z^T, \end{aligned} \tag{3}$$

where the objective function $c^T x + \text{tr } F_0 Z \equiv \eta$ is called the duality gap, which has the known optimum value of zero for a convex problem. The advantage of using the primal-dual formulation is that at each step information from the dual problem can be used to obtain a good update for the primal variables.

One of the methods to solve the primal-dual optimization problem is the potential reduction method. Define a potential function

$$\phi(x,Z) \equiv (n+\nu\sqrt{n})\log(\text{ tr } F(x)Z) - \log\det\, F(x) - \log\ \det\, Z - n\log n,$$

where $\nu \geq 1$ is a weighting parameter in the potential. The duality gap is

$$\eta \leq \exp\left(\frac{\phi}{\nu\sqrt{n}}\right),$$

which approaches 0 as the potential function ϕ approaches $-\infty$.

The whole algorithmic process can be described as follows. Starting from a strictly feasible x_0 and Z_0, find x_k and Z_k so that the potential ϕ is reduced at each step by at least a fixed amount $\delta > 0$,

$$\phi\big(x^{(k+1)}, Z^{(k+1)}\big) \leq \phi\big(x^{(k)}, Z^{(k)}\big) - \delta,$$

until the duality gap η is smaller than some specified $\epsilon > 0$. The first x_k and Z_k which make $\eta < \epsilon$ constitute the approximate numerical solution of the primal-dual problem.

The primal-dual potential reduction method for solving the semidefinite program (1) can be summarized as follows. Given strictly feasible x and Z, while duality gap $\eta = c^T x + \text{ tr } F_0 Z > \epsilon$ do

1. compute a direction δx for the primal variable x and a direction δZ for the dual variable Z;

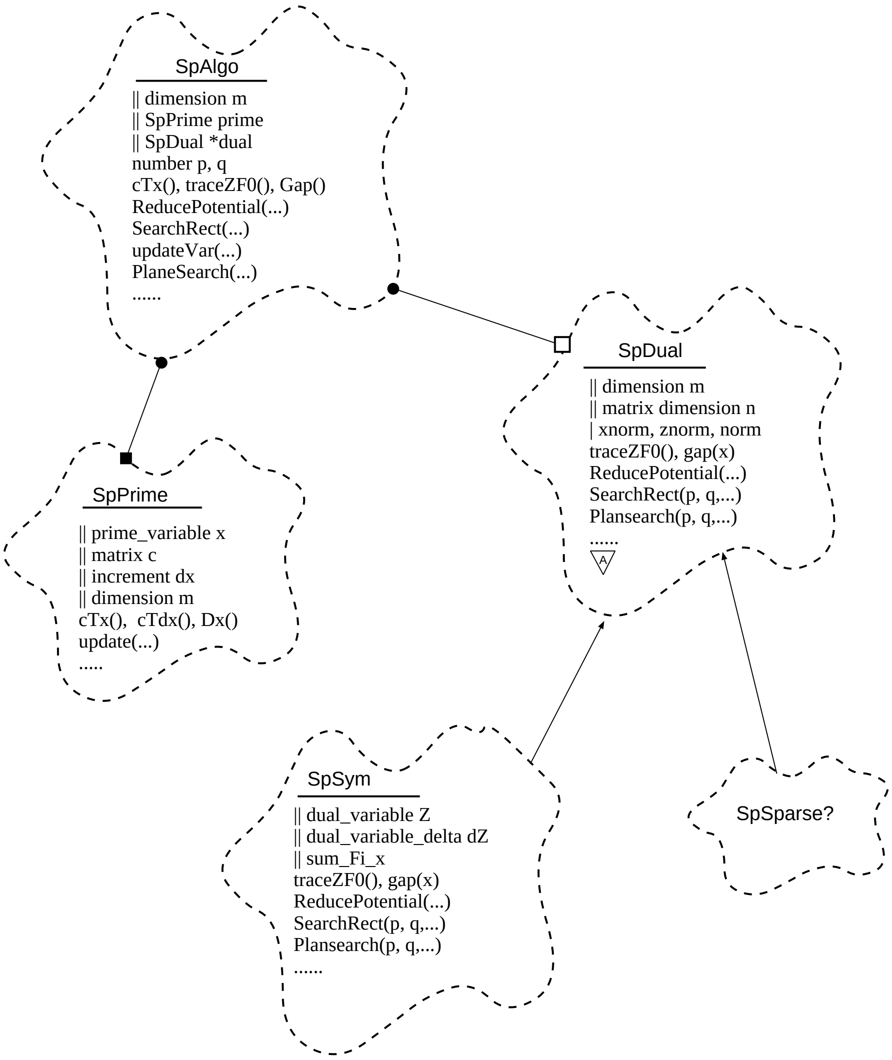

Fig. 1. Class diagram.

2. find p, q that minimize $\phi(x + p\delta x, Z + q\delta Z)$, where (p, q) are constrained to some search rectangle in the plane;
3. update $x := x + p\delta x$ and $Z := Z + q\delta Z$.

Details for each of these steps are given in [12].

3 The object-oriented design

In C++ terminology, the word "class" is used to mean a type of object. For example, a class *Symm* can be defined for symmetric matrices, while a specific symmetric matrix is an object of type *Symm* or an instance of the class *Symm*. C++ terminology will be followed in the rest of the paper.

Object-oriented analysis and design is one of the most active research areas for both academics and industrial practitioners. When applied in different circumstances, different analysis and design techniques may be emphasized. One of the most influential analysis and design techniques is due to Booch[1], whose conventions will be loosely followed in this paper.

First we will describe the classes used in the design and their relationship. The class diagram for the problem is shown in Fig. 1, where all the classes are defined with their attributes and functions. For clarity, only main attributes and functions are shown and the functions names may be different from that used in real implementation.

In Fig. 1, each dotted cloud-shaped figure describes a class, with the private members of the class preceded by $||$ and the protected members of the class preceded by $|$. *Protected* properties are inherited by and accessible to subclasses, whereas *private* properties are not. All other properties are considered public, i.e., accessible by other classes. Protected and private properties are not accessible to other classes. For example, in the class *SpPrime*, the private attributes are *prime_variable* x, *matrix* c, and *increment* dx, the main public functions are *cTx()* which computes $c^T x$, *cTdx()* which computes $c^T dx$, *Dx()* which computes dx, and *update()* to update x.

The connection ●——■ from *SpAlgo* to *SpPrime* denotes the physical containment of *SpPrime* in *SpAlgo*, while the connection ●——□ from *SpAlgo* to *SpDual* denotes a pointer reference to the class *SpDual* by *SpAlgo*, which can be illustrated by the corresponding part of the definition of *SpAlgo*

```
......
   private:
      SpPrime prime;    // primal space
      SpDual  *dual;    // dual space, dynamically allocated
      ......
```

This code declares *prime* as an instance of the class *SpPrime*, and *dual* as an instance of the class *SpDual*.

An arrow $\rightarrow$ from *SpSym* to *SpDual* denotes that *SpSym* is a subclass of *SpDual* and inherits the public and protected attributes and functions from *SpDual*. An upside-down triangle with A in *SpDual* denotes that *SpDual* is an *abstract class*. Many functions in *SpDual* are defined as *virtual* so that dynamic binding is used for these functions, i.e., the decision on which implementation to use is made at run time, as it cannot be determined at compile time. Consequently, if a class other than *SpSym* is used to implement *SpDual*, then the run time system will choose the right function depending on the parameters passed.

The symmetry between the primal and dual spaces in the objective function in (3) suggests that these two spaces should be fundamental classes in the problem, and so two classes, *SpPrime* and *SpDual*, are defined. *SpPrime* is relatively simple. The primal variable x is the independent variable. The only important actions on the class are to calculate $c^T x$ and update x. The class *SpDual* will contain more functions and is more complicated. It will not only calculate $\mathrm{tr} F_0 Z$, but also compute δZ, p, q, and update Z. The dual variable Z is a symmetric matrix. In the future we may want to exploit any special structure the matrix Z may have. For example, we could exploit the sparsity of the matrix Z by defining and using another class *SpSparse*. So *SpDual* is designed as an "envelope" class [3] to provide a generic dual space interface and to isolate it from the "letter" class that implements the dual space for a particular representation. As of this writing, we implemented *SpSym*, a symmetric matrix representation not exploiting any other special structure that the matrix may have.

SpAlgo is the base class for algorithm implementation. It mediates the communication between *SpPrime* and *SpDual*, executes the computational tasks, and changes the "states" of the *SpPrime* and *SpDual* objects.

Finally *main* is a utility class, a "free" program not tied to any particular class, that stores the convergence criterion and is the driver for the algorithm. It is denoted with a dotted cloud with a shadow in Fig. 2 and Fig. 3.

The major advantage of this way of thinking is that the implementation of the dual space is completely independent of the program structure and state transition. This is desirable because the dual space implementation is where algorithmic changes are likely to occur, and this separation of the interface and implementation localizes changes in the program. The envelope class *SpDual* provides a clean and effective interface for the dual space; *SpSym* is one of the possible subclasses of *SpDual* that does nothing but implement the specification of *SpDual*. The matrix and vector classes are from LAPACK++, and are not shown in the diagram.

There are two execution scenarios, whose outlines are shown in Figs. 2 and 3, corresponding to different stages of the primal-dual algorithm.

A small square with F inside on the border of *SpDual* and *SpPrime* on the line from *SpAlgo* denotes that *SpPrime* and *SpDual* are part of the client object, *SpAlgo*. The two square boxes with L inside on the border of *SpAlgo* on the line from *main* denote that *SpAlgo* is a locally declared object in *main*. An arrow with an empty circle indicates that results or services are returned to the object pointed to by the arrow. An arrow denotes the direction of a message from one object to another. The number in front of the message denotes the execution sequence of the message.

In the first scenario (Fig. 2):

Step 1. If a search rectangle exists, *main* sends a message to *SpAlgo* to do a plane search, i.e., find p and q such that $\phi(x+p\delta x, Z+q\delta Z)$ is minimized, where p and q are constrained in the search rectangle. Otherwise *main* sends a message to *SpAlgo* to calculate the duality gap.

Step 2. *SpAlgo* sends a message to *SpPrime* to get the primal variable.

Step 3. *SpAlgo* sends the primal variable to *SpDual* along with a message to calculate the duality gap.

Step 4. *main* checks the convergence criterion. If the criterion is met, execution stops. Otherwise, go to the second scenario.

In the second scenario (Fig. 3):

Step 1. *main* sends a message to *SpAlgo* to do a potential reduction calculation.

Step 2. *SpAlgo* sends a message to *SpDual*, which returns dx, the increment for the primal variable.

Step 3. *SpAlgo* sends dx to *SpPrime* to store it in *SpPrime*.

Step 4. *main* sends a message to *SpAlgo* to calculate the search rectangle.

Step 5. *SpAlgo* sends a message to *SpPrime* to calculate $c^T dx$.

Step 6. *SpAlgo* sends $c^T dx$ along with message *SearchRect(...)* to *SpDual* to calculate and return several intermediate variables that mark the search rectangle.

Step 7. *main* checks the convergence criterion using the corners of the search rectangle. If the convergence criterion is met, execution stops. Otherwise, go to the first scenario.

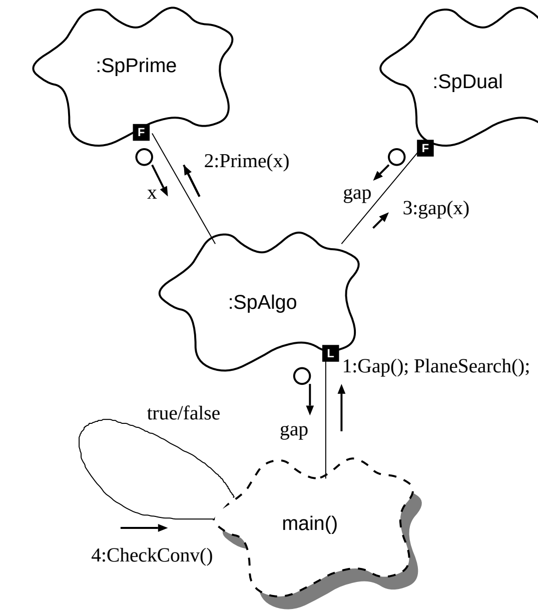

Fig. 2. Execution Scenario 1.

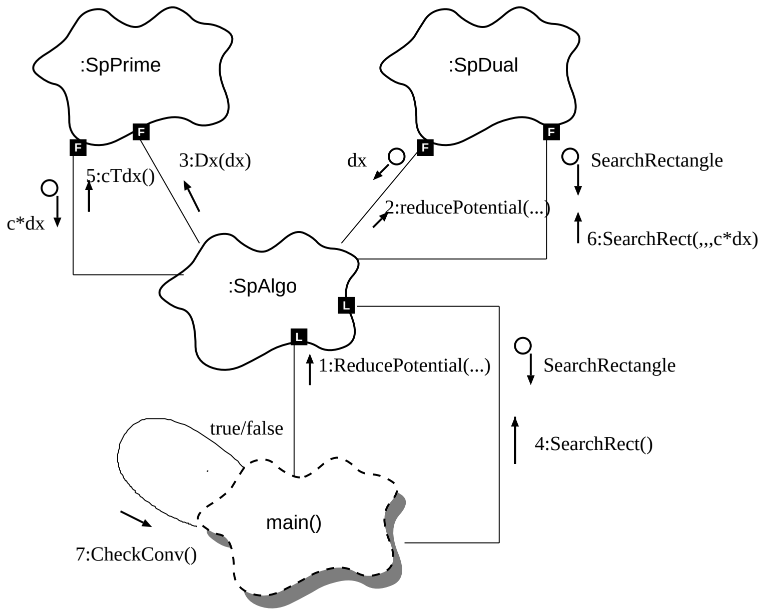

Fig. 3. Execution Scenario 2.

4 The C++ implementation

The program is built upon the LAPACK++ v1.0 package, especially the LaVectorDouble, LaGenMat, LaSymmMat classes, and BLAS++ is used extensively. Several special purpose routines are added to the LAPACK++ package to accommodate the primal-dual

```
for (k=0, pos4=pos;  k<blck_szs[i];  pos4+=blck_szs[i]-k, k++)
 {
      scal = sigx[k];
      rhs[pos4] = (1.0/scal + rho*scal)/sqrt2;
 }
for (j=0, pos=0;  j<m;  j++)
   for (i=0, pos2=0;  i<L;
         pos += blck_szs[i]*(blck_szs[i]+1)/2,
         pos2 += blck_szs[i]*blck_szs[i], i++)
     {
        /* compute V' * Fj(i) * V, store in Fsc+pos, V is scaled.*/
        cngrncb(2, blck_szs[i], F+sz+pos, R+pos2, Fsc+pos, temp);
        /* correct diagonal elements */
        for (k=0, pos4=pos;  k<blck_szs[i];  pos4 += blck_szs[i]-k, k++)
        Fsc[pos4] /= sqrt2;
     }
/*
 * solve least-squares problem; need workspace of size m + nb*sz
 * - rhs is overwritten by dx
 * - in first iteration, estimate condition number of Fsc
 */
dgels_("N", &sz, &m, &int1, Fsc, &sz, rhs, &sz, temp, &ltemp,
       &info2);
```

Fig. 4. A segment of C code.

```
for ( int i = 0, pos=0; i < j; pos += j-i, i++)
{
   double scal = sigx(i);
   dx(pos)=  ( 1.0/scal +rho*scal) * sq2;
}
/* loop over Fi,  compute V' * Fi * V, store it in Fsc */
for ( vector < LaVectorDouble>::iterator i = Fi->begin()+1;
     i < Fi->end(); i++, n++)
{
      LaVectorDouble tmp(Fsc.addr()+pd_sz*n, 1, pd_sz);
      DualScale(0, *i, vecx, tmp);
      /* correct diagonal elements */
      for ( int k = 0, pos=0; k < j; pos += j-k, k++)
        tmp(pos) *= sq2;
}
LaLeastSquare(dx, Fsc, &n);
```

Fig. 5. A segment of C++ code

algorithm for semidefinite programming. The program also uses the iterator object in STL (Standard Template Library)[10] [8] to traverse arrays of objects.

The first major difference between the C and C++ implementations is the way the initial data is read in. Unlike C/Fortran style subroutines, in which one can pass a pointer/address for a piece of storage and let the subroutine split the storage into pieces to get the data, C++ objects' constructors have no such scheme. Initialization is done by reading a data file.

The second major difference is that because C++'s objects are higher level abstractions, the implementation in C++ is less dependent upon pointer arithmetic, as shown by the code segments in C and C++ (Figs. 4 and 5) for doing the same computation. There is overhead associated with this higher level of abstraction, but we will show that the effect on performance is negligible.

5 Comparison and discussion of results

Two sets of data are obtained by randomly generating all the matrices F_0, F_1, $\cdots$, F_m, and the vector c. Strictly feasible initial points x_0 and Z_0 are also generated. The timing results are shown in Table 1. All the timings are done on a HP 712/60 workstation. Both the C implementation from [13] and the C++ implementation are compiled using the Gnu C/C++ compiler version 2.7.2 with the same compiler options. It is clear from Table 1 that the performance penalty for using C++ is only a few percent and decreases as the problem size increases.

Table 1. Comparison of implementations.

	Example 1, $n = 40$	
	C++ implementation	C implementation
m	time (sec)	time (sec)
20	4.7	4.4
30	6.9	6.5
	Example 2, $n = 100$	
50	229	222
75	390	381

We have shown an objected-oriented design and implementation of a semidefinite programming algorithm. Even though object-oriented technology is being used more and more widely in industry now, there are not many realistic applications to numerical computation. The programming environments and tools seem to be mature enough to apply this new methodology, and the performance seems to be comparable to a non-object-oriented implementation.

However, there are other considerations that have to be taken into account when applying object-oriented technology. First, it takes time and effort to learn the new methodology. Second, it is not a trivial task to set up the environment: compiling all the C++ packages, and verifying that they work correctly, especially when most of the C++ packages for numerical computation are still in the testing stage. Third, the resulting code size, i.e., the size of the C++ executable, is about 2.5 times that of the C executable. With continuing development of object-oriented technology and of compilers for object-oriented languages, the second problem, will likely be alleviated. The hardware considerations of the third problem are becoming less of a hindrance with the advances in the computer industry. We do believe that the benefits of using object-oriented methodology outweight the currently existing disadvantages.

The design and analysis in this paper can be generalized to apply to object-oriented design and implementation of other interior point algorithms which use potential reduction to find the optimum.

References

[1] G. Booch, *Object Oriented Design with Applications, second edition*, Benjamin/Cummings, Redwood City, CA, 1994.

[2] S. Boyd, L. Vandenberghe, and M. Grant, *Efficient Convex Optimization for Engineering Design*, Proc. of the IFAC Symposium on Robust Control Design, Rio de Janeiro, Brazil, Sept. 1994, pp. 14–23.

[3] J. O. Coplien, *Advanced C++, Programming Styles and Idioms*, Addison-Wesley, Redwood City, CA, 1992.

[4] J. Dongarra, R. Pozo, and D. Walker, *LAPACK++: A Design Overview of Object-Oriented Extensions for High Performance Linear Algebra*, Proc. Supercomputing '93, IEEE Press, 1993, pp. 162–171.

[5] J. Dongarra, A. Lumsdaine, R. Pozo, K. Remington, *A Sparse Matrix Library in C++ for High Performance Architectures*, Proc. Second Annual Object-Oriented Numerics Conference, 1994, pp. 214–218.

[6] J. Dongarra, A. Lumsdaine, R. Pozo, and K. A. Remington, *IML++ Iterative Methods Library Reference Guide*, http://gams.nist.gov/acmd/Staff/RPozo/sparselib++.html, 1994.

[7] Y. Nesterov and A. Nemirovsky, *Interior-point Polynomial Methods in Convex Programming*, Studies in Applied Mathematics, vol. 13, SIAM, Philadelphia, PA, 1994.

[8] D. R. Musser and A. Saini, *C++ Programming with the Standard Template Library*, Addison-Wesley Professional Computing Series, Reading, MA, 1996.

[9] R. Pozo, K. A. Remington, and A. Lumsdaine, *SparseLib++ v. 1.3, Reference Guide*, http://gams.nist.gov/acmd/Staff/RPozo/sparselib++.html, 1994.

[10] A. Stepanov and M. Lee, *The Standard Template Library*, http://www.cs.rpi.edu/projects /STL/stl-new/stl-new.html, 1993.

[11] B. Stroustrup, *The C++ Programming Language, 2nd ed.*, Addison-Wesley, Reading, MA, 1991.

[12] L. Vandenberghe and S. Boyd, *Semidefinite Programming*, SIAM Review, vol. 38, 1996, pp. 49–95.

[13] L. Vandenberghe and S. Boyd, *Software for Semidefinite Programming, User's Guide*, preprint, 1996.