Inexact Solution of the Schur Complement Equation in a Primal-Dual Interior-Point Method for Semidefinite Programming

Cand. Scient. Thesis
May 7, 2002

Lennart Frimannslund

Department of Informatics
University of Bergen
Norway

Notation Legend

n	Number of LP variables, dimension of SDP variables
m	Number of constraints
μ	Central path parameterisation variable
σ	Centering parameter
α	LP variables step length, primal SDP variable step length
β	Dual SDP variables step length
$\mathcal{X}$	Diagonal matrix with the vector x on its diagonal.
x, $\mathcal{X}$	Primal LP variable
x, X	Primal SDP variable
y, z, $\mathcal{Z}$	Dual LP variables
y, z, Z	Dual SDP variables

The context will make it clear when the lowercase versions of these variables pertain to LP or SDP.

b	LP and SDP dual objective function vector
c	LP primal objective function vector
C	SDP primal objective function matrix
r	Residual associated with Schur Complement Equation
e	Vector of all ones
Δx, ΔX	Variables associated with Newton method step equation.
h_d, h_p, h_c	Variables associated with dual, primal and complementarity KKT conditions, respectively.
X^P, X^C	Variables associated with predictor and corrector step, respectively.
$\widetilde{X}$	Variable computed inexactly.

Contents

1 Introduction to Semidefinite Programming **7**
- 1.1 A special case — Linear Programming . . . 7
- 1.2 Extension to Semidefinite Programming . . . 9
- 1.3 Example of an SDP Problem . . . 10
- 1.4 Duality Theory — LP vs SDP . . . 11

2 Solving Semidefinite Programs **13**
- 2.1 A special case revisited — Solving Linear Programs . . . 13
  - 2.1.1 Interior-Point Methods . . . 13
- 2.2 Extension to Semidefinite Programs . . . 18

3 Analysis of the Interior-Point Algorithm **22**
- 3.1 Initial Iterates . . . 22
- 3.2 Solving the Step Equation . . . 22
- 3.3 Calculation of Step Lengths . . . 25
- 3.4 The centering Parameter σ . . . 26
  - 3.4.1 Algorithm PD-SDP . . . 26
  - 3.4.2 The Predictor-Corrector (PC) Scheme . . . 26
- 3.5 Stopping Criteria . . . 28

4 Inexact Schemes **30**
- 4.1 Initial Observations . . . 31
  - 4.1.1 Inexact Solution of the SCE . . . 31
  - 4.1.2 Conservation of Primal Feasibility . . . 32
  - 4.1.3 Inexact Solution of the SCE within a PC-framework . . . 34
- 4.2 Implicit Representation of the SCE . . . 37
- 4.3 Description of the Schemes . . . 40
  - 4.3.1 Scheme 1 — Inexact SCE Solution with PC . . . 40
  - 4.3.2 Scheme 2 — Inexact SCE Solution with PC and Conservation of primal Feasibility . . . 41

5 Numerical Results **42**
- 5.1 Computer Code used . . . 42
- 5.2 Heuristics and Convergence Criteria used . . . 44

5.3 A typical Run 44
5.4 Listing of Results 46

6 Conclusions **50**
6.1 Experiment Evaluation 50
6.2 Possible Modifications 50
6.3 Topics and Issues not addressed 51
6.4 Summary 52

A Notation and Terminology **53**
A.1 Basic definitions 53
A.2 Spaces and Sets 55
A.3 Functionals and Operators 56
A.4 Notation and Terminology specific to LP and SDP 59

B Derivation of the Central Path Equations for LP **61**
B.1 Reformulation of the LP 61
B.2 Lagrange Multipliers 62
B.3 Lagrange Multipliers applied to LP 63
B.4 The unconstrained Problem vs LP 66
B.5 Extension to SDP 66

C Source Code **67**
C.1 Listing of Scheme 2 67

Introduction

Semidefinite programming (SDP), which can be considered a generalisation of linear programming (LP), has been the topic of broad research the last ten years. It can be applied to many different problems such as minimising the maximum eigenvalue of matrix, optimising a linear function subject to convex quadratic constraints, control theory, logarithmic Chebychev approximation and obtaining tight bounds for hard combinatorial optimisation problems, to name a few [15, 22]. Much of the previous research has been concentrated on developing search directions for primal-dual interior-point methods [1, 17, 13]. What we wish to do is try to modify such a method. In other words, we are not so much interested in the aspect of modeling; we assume that the semidefinite programming problem has been presented to us in some standard form, and we are primarily interested in the process of solving it.

Primal-dual interior-point algorithms are based on the application of Newton's method [7, 14]. Newton's method takes an approximation to the solution as input and refines it through iteration until it (subject to certain conditions) converges to the solution sought. It is implemented through solving a series of linear equation systems, each based on the solution to the foregoing equations. What we wish to investigate is whether or not it is beneficial to solve these systems, or at least some of them *inexactly*, that is, to approximate their solution. Solving inexactly typically means less work in the case of a single equation system, but we don't know what effect this has on a series of systems, where error propagation from one system to the next as well as global convergence are important concerns. We expect that some of the equation systems, probably the last few, will have to be solved not by approximation but using direct methods which yield high-accuracy solutions. The exact circumstances surrounding this are what we wish to investigate. The motivation behind this idea is twofold.

First of all, it has been done in the field of unconstrained optimisation [9, 14]. Secondly, this has also been tried for linear programming [5, 23], which as mentioned is as special case of semidefinite programming. Our question is if these ideas can be applied to semidefinite programming as well.

Throughout, effort has been made to first introduce and state results

for linear programming, and then to generalise them to semidefinite programming. We hope that this will make the text understandable for the reader familiar with linear programming, but not necessarily with knowledge interior-point methods or semidefinite programming.

The rest of the text is organised as follows. In chapter 1 we first outline what linear programming is, extend it to semidefinite programming, and give an example of a semidefinite program. In chapter 2 we present an algorithm for solving linear programs, and again extend to semidefinite programming. The algorithm is more thoroughly described in chapter 3, and in chapter 4 we outline our ideas to improve it. Experimental results are presented in chapter 5, and evaluated in chapter 6. The three appendices present notation and terminology and outline the motivation behind the algorithm of chapter 2 for the reader unfamiliar with these ideas, and list source code we have used.

Chapter 1

Introduction to Semidefinite Programming

1.1 A special case — Linear Programming

In linear programming we are searching for a nonnegative n-vector x, such that an objective (or cost) function, $c^T x$, is minimised while satisfying m linear constraints, which can be written $Ax = b$. In other words, we have the problem:

$$\begin{array}{rc} \min & c^T x \\ \text{such that} & Ax = b \\ \text{where} & x \geq 0. \end{array}$$

$x \geq 0$ means that $x_i \geq 0,\ i = 1 \ldots n$. This is often referred to as the primal problem, or a problem in primal form. Associated with each primal problem is its uniquely defined dual problem, given as:

$$\begin{array}{rc} \max & b^T y \\ \text{such that} & A^T y + z = c \\ \text{where} & z \geq 0. \end{array}$$

Here $A \in \mathbb{R}^{m \times n}$, $c,\ x$ and $z \in \mathbb{R}^n$, b and $y \in \mathbb{R}^m$. A vector $x \geq 0$ which satisfies the constraints $Ax = b$ is called a primal *feasible solution*. A pair of vectors (y, z) which satisfy the constrains $A^T y + z = c$ and where $z \geq 0$ is called a dual feasible solution. The primal and dual problems have the same objective value at the optimum, the primal approaching it from above, the dual from below.

The optimal solution vectors (x^*, y^*, z^*) have the property that the pairwise products $x_i^* z_i^* = 0,\ i = 1 \ldots n$. Since all components in x and z are required to be nonnegative, this is equivalent to the otherwise weaker property that

$$(x^*)^T z^* = 0. \tag{1.1}$$

This property is referred to as *complementarity* of the vectors x^* and z^*. Necessary and sufficient conditions for optimality are satisfaction of the primal, dual and non-negativity constraints, as well as complementarity [21, 14].

Linear Programming problems such as these can be solved in many ways, two of which are the classical simplex method, described in any introductory book on LP (for example [21]), or the younger class of so-called interior-point methods. We shall be most concerned with the latter.

Reformulation of the Linear Program Consider the following inner product between two $n \times n$-matrices

$$A \bullet B = \sum_{i=1}^{n} \left(\sum_{j=1}^{n} A_{ij} B_{ij} \right),$$

that is, multiply the matrices entry-wise and sum all the elements of the resulting matrix. We will call this the *bullet product*. It turns out to be equal to $\mathbf{trace}(B^T A)$. (Recall that the **trace** of a square matrix is the sum of its diagonal elements.) When A and B are symmetric it is equal to $\mathbf{trace}(AB)$. If x is a vector, then let the square diagonal matrix $\mathcal{X}$ be equal to $\mathbf{diag}(x)$:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \Rightarrow \mathcal{X} = \begin{bmatrix} x_1 & 0 & 0 \\ 0 & x_2 & 0 \\ 0 & 0 & x_3 \end{bmatrix}.$$

If we name the matrix with the first row of A on its diagonal $\mathcal{A}_1$, the matrix with the second row on its diagonal $\mathcal{A}_2$ and so on, we can write the primal linear program as follows:

$$\begin{array}{ll} \min & \mathcal{C} \bullet \mathcal{X} \\ \text{s.t.} & \mathcal{A}_i \bullet \mathcal{X} = b_i, \quad i = 1 \ldots m \\ & \mathcal{X} \succeq 0. \end{array}$$

$X \succeq 0$ means that $\mathcal{X}$ is *positive semidefinite*, that is $v^T \mathcal{X} v \geq 0$, for all v. This property follows from the fact that $\mathcal{X}$ is diagonal and that all its entries are nonnegative. We can write the dual program as:

$$\begin{array}{ll} \max & b^T y \\ \text{s.t.} & \sum_{i=1}^{m} y_i \mathcal{A}_i + \mathcal{Z} = \mathcal{C} \\ & \mathcal{Z} \succeq 0. \end{array}$$

To see that this makes sense, note that $A^T y$ is the linear combination of the columns of A^T with coefficients y_i. $\sum_{i=1}^{m} y_i \mathcal{A}_i$ is therefore the same as $\mathbf{diag}(A^T y)$. The complementarity condition (1.1) becomes:

$$\mathcal{X}^* \bullet \mathcal{Z}^* = 0.$$

1.2 Extension to Semidefinite Programming

So far all matrices involved have been diagonal. If we now allow them to be general real *symmetric* matrices (let $\mathcal{S}^n$ be the space of real symmetric $n \times n$-matrices, see appendix A), we arrive at the general semidefinite programming problem: (Note the slight change in notation as matrices are no longer restricted to be diagonal.)

Primal SDP problem:

$$\begin{array}{ll} \min & C \bullet X \\ \text{s.t.} & A_i \bullet X = b_i, \quad i = 1 \dots m, \\ & X \succeq 0. \end{array}$$

Here $\{C,\ X,\ A_i,\ i = 1 \dots m\} \in \mathcal{S}^n,\ b \in \mathbb{R}^m$.

Dual SDP problem:

$$\begin{array}{ll} \max & b^T y \\ \text{s.t.} & \sum_{i=1}^m y_i A_i \ + Z = C, \\ & Z \succeq 0. \end{array}$$

Here $Z \in \mathcal{S}^n,\ y \in \mathbb{R}^m$.

Necessary and sufficient optimality conditions are satisfaction of all the primal and dual constraints as well as

$$XZ = 0.$$

This is outlined in [15].

Matrix structure (that is, locations of zero elements) is important when it comes to solving SDP problems, as we shall see. From the problem definition above we can see that the structure of Z is determined from the structure of the constraint matrices $A_i,\ i = 1 \dots m$ and C. No such restrictions apply to X, but given the nature of the bullet product, the only elements of X which contribute to the objective function $C \bullet X$ are the ones that correspond to nonzero entries in C. In other words, restricting X to have the same structure as C does not affect the range of the objective function. However, X must still satisfy the m constraints $A_i \bullet X = b_i,\ i = 1 \dots m$, so the structure of X should be the union of the structure of C and $A_i,\ i = 1 \dots m$.

Typically, the matrices C and A_i will be *block diagonal*, or equivalently, there will be a block diagonal pattern which encapsulates all their nonzero elements. An example of a symmetric block diagonal matrix with block sizes

one, three and two is:

$$\begin{bmatrix} a_1 & & & & & \\ & b_1 & b_2 & b_3 & & \\ & b_2 & b_4 & b_5 & & \\ & b_3 & b_5 & b_6 & & \\ & & & & c_1 & c_2 \\ & & & & c_2 & c_3 \end{bmatrix}.$$

1.3 Example of an SDP Problem

Many different problems can be cast as SDP problems, see for example [22, 15]. We will give one example here, regarding undirected graphs. First, we need some terminology about graphs.

An *independent set* is a set of mutually nonadjacent nodes, that is, a set of nodes where no two members are neighbours. A *clique* is a set of mutually adjacent nodes, that is, a set of nodes where *all* members are neighbours. Given a graph G, then $\alpha(G)$ is the size of its largest independent set, and $\bar{\chi}(G)$ is the smallest number of cliques it takes to include all the nodes of the graph. Both of these problems are NP-hard[1] to solve, but we can obtain an approximation to their solutions with the *Lovász θ function*, $\theta(G)$ [11]. It can be defined as the solution to the following SDP:

$$\max ee^T \bullet X,$$

such that

$$I \bullet X = 1,$$

where in addition $X_{ij} = 0$ if there is an edge in G from node i to node j, and

$$X \succeq 0,$$

where e is a vector of all ones.

This is a maximisation problem, but can be turned into a minimisation problem by instead minimising the negative of the objective function. The requirement of zero elements in X can be enforced by constraints of the form

$$A_i \bullet X = 0,$$

where each A_i forces a pair of elements in X to be zero. Noting that these constraints only place restrictions on the structure of X, it is possible to eliminate them and instead have the solution algorithm deal with structure concerns. We will return to this possibility in chapter 2.

For a general graph G we have the following relation:

$$\alpha(G) \leq \theta(G) \leq \bar{\chi}(G).$$

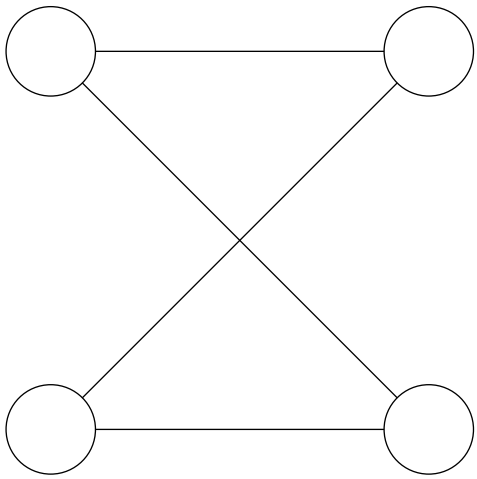

Figure 1.1: A perfect graph.

If the three are equal, G is said to be a *perfect* graph. A concrete example, using the graph in figure 1.1 and numbering its nodes left to right, top to bottom would give the following optimal solution:

$$\theta(G) = ee^T \bullet \begin{bmatrix} \frac{1}{4} & & \frac{1}{4} & \\ & \frac{1}{4} & & \frac{1}{4} \\ \frac{1}{4} & & \frac{1}{4} & \\ & \frac{1}{4} & & \frac{1}{4} \end{bmatrix} = 2.$$

The graph turns out to be perfect, as can be verified by inspection. The maximum number of mutually nonadjacent nodes is two, as is the minimum number of cliques to include all nodes, for example the clique of the two upper nodes combined with the clique of the two lower nodes.

1.4 Duality Theory — LP vs SDP

If we use our definition of the primal and dual LP problems (and disregard the very special case of infeasible problems which are their own dual), the following statements about LP are true:

- If the primal problem has no feasible solutions, then the dual objective function is unbounded.
- If the dual problem has no feasible solutions, then the primal objective function is unbounded.
- If both the primal and dual problem have feasible solutions, then *any* primal feasible solution corresponds to an objective function value

[1]More information on NP-hard and NP-complete problems can be found in [8].

greater than or equal to *all* dual objective function values stemming from dual feasible solutions. This is referred to as *weak duality*.

- If feasible solutions to both the primal and dual problems exist, then their optimal objective function value is the same. This is referred to as *strong duality*.

The first three apply to SDP as well, but unfortunately this is not necessarily the case with the last statement. As discussed in [15], we may encounter cases where the primal and dual problems have optimal objective function values which are *not* the same. A theorem guaranteeing strong duality, whose proof can also be found in [15], follows below. Let a primal feasible point be an $X \in \mathcal{S}^n$ such that $A_i \bullet X = b_i,\ i = 1 \dots m$ and $X \succeq 0$, and a dual *strictly* feasible point be a pair of $(y, Z),\ y \in \mathbb{R}^m,\ Z \in \mathcal{S}^n$ such that $\sum_{i=1}^{m} y_i A_i + Z = C$ and $Z \succ 0$.

Theorem 1.1 *If there exists a primal feasible point and a dual strictly feasible point, then the optimal solution to the primal and dual problems are the same.*

Note the subtle difference that X should be positive semidefinite while Z must be positive definite.

This is not as strict a requirement as it might seem, and for the remainder of the text we shall assume strong duality when it comes to SDP.

Chapter 2

Solving Semidefinite Programs

In this chapter we introduce a method for solving linear programs called a primal-dual interior-point method. We then extend it to semidefinite programming, and briefly describe an algorithmic framework. The motivation for the method is outlined in appendix B.

2.1 A special case revisited — Solving Linear Programs

A Note on the Simplex Method The simplex method [21, 14] is the classical algorithm for solving linear programs. It makes use of the fact that the boundary of the feasible region, on which the optimal solution lies, is an n-dimensional convex polyhedron (or *simplex*), e.g. a diamond in 3-space. Because both the objective function and constraints are linear, the solution lies at one of the vertices. The simplex method starts out at one such vertex, and visits neighbouring vertices with decreasing objective function values until it has reached the optimum.

In semidefinite programming however, the boundary of the feasible set is not necessarily a polyhedron (that is, its surfaces are not necessarily linear) [22]. Therefore the optimal value of the objective function is not necessarily located at a vertex. If we want to generalise an LP algorithm to SDP, we need a different approach.

2.1.1 Interior-Point Methods

Interior-point methods make use of the fact that the optimality conditions for LP (known as the Karush-Kuhn-Tucker or KKT conditions) constitute

a nonlinear system of equations:

$$\begin{aligned} A^T y + z - c &= 0, \\ Ax - b &= 0, \\ x_i z_i &= 0,\ i = 1 \dots n, \end{aligned} \tag{2.1}$$

where

$$(x, z) \geq 0.$$

To solve this system we make use of Newton's method [7, 14], which is given as follows:

To solve

$$F(v) = 0, \tag{2.2}$$

where v is a vector and v and $F(v)$ belong to the same spaces, repeat

$$v^i = v^{i-1} - J^{-1}(v^{i-1})F(v^{i-1})$$

(where J is the Jacobian matrix of the function F and v^i is the approximation to v after i iterations of Newton's method) until a convergence criterion is satisfied, ideally $F(v^k) = 0$ for some k. This can be rewritten as a *step equation*:

$$J(v^{i-1})\Delta v^{i-1} = -F(v^{i-1}), \tag{2.3}$$

so that

$$v^i = v^{i-1} + \Delta v^{i-1}.$$

If we want to apply Newton's method to our problem (2.1) it is again useful to define $\mathcal{X} = \mathbf{diag}(x)$, $\mathcal{Z} = \mathbf{diag}(z)$ and e, a vector of all ones. Our equation (2.2) becomes

$$F(x, y, z) = \begin{bmatrix} A^T y + z - c \\ Ax - b \\ \mathcal{X}\mathcal{Z}e \end{bmatrix} = 0, \tag{2.4}$$

where

$$(x, z) \geq 0.$$

To make the text more readable, we will from now on omit superscripts signifying iteration number in step equations. Defining the shorthand $h_d = c - z - A^T y$ and $h_p = b - Ax$, the step equation corresponding to (2.4) becomes

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ \mathcal{Z} & 0 & \mathcal{X} \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} h_d \\ h_p \\ -\mathcal{X}\mathcal{Z}e \end{bmatrix}. \tag{2.5}$$

h_p, called the *primal infeasibility* is a measure of how close x is to satisfy the constraints of the primal problem. h_d is called the *dual infeasibility*, and

is a measure of how close y and z are to satisfy the constraints of the dual problem. We will use the term primal infeasibility for both h_p and $\|h_p\|$, and the term dual infeasibility for both h_d and $\|h_d\|$. Both h_p and h_d are zero if the current iterate is feasible.

The problem now is that Newton's method doesn't take the constraint $(x, z) \geq 0$ into account and may therefore very well converge to another solution than the one desired. To try and solve this problem we will modify the pure Newton method in two ways. First, starting with strictly positive initial iterates x^0 and z^0, we will perform a line search along the proposed search direction in order to keep x and z nonnegative. This modification alone doesn't help much, as we risk aiming for a point we are not allowed to reach over and over again, and hence stagnate. We will therefore secondly try to modify the Newton direction, guiding it towards the nonnegative solution we seek.

The central path Let us modify (2.4) slightly and write it as follows:

$$F(x, y, z) = \begin{bmatrix} A^T y + z - c \\ Ax - b \\ \mathcal{X}\mathcal{Z}e - \mu e \end{bmatrix} = 0, \tag{2.6}$$

where

$$(x, z) \geq 0.$$

In other words, let the pairwise products $x_i z_i,\ i = 1 \dots n$ be equal to μ rather than 0. The solution to (2.6) is uniquely defined as long as the set of *strictly feasible* points, that is, feasible points where in addition $x > 0,\ z > 0$, is nonempty [21, 14]. The solution lies within the feasible region, and if we solve (2.6) for decreasing values of μ we see that the solutions follow a path, the *central path*, which (for $\mu = \infty$) has an endpoint at the center of the feasible region and goes towards a solution of (2.4) as μ goes to zero (see figure 2.1). The step equation corresponding to (2.6) is

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ \mathcal{Z} & 0 & \mathcal{X} \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} h_d \\ h_p \\ -\mathcal{X}\mathcal{Z}e + \mu e \end{bmatrix}. \tag{2.7}$$

At each iteration we now have two possible search directions. The solution to (2.5) gives us the pure Newton or *affine scaling* direction, while the solution to (2.7) is known as a *centering direction*. Instead of choosing one over the other, we will introduce a centering parameter $\sigma \in [0, 1]$ which turns (2.6) into

$$F(x, y, z) = \begin{bmatrix} A^T y + z - c \\ Ax - b \\ \mathcal{X}\mathcal{Z}e - \sigma\mu e \end{bmatrix} = 0, \tag{2.8}$$

where

$$(x, z) \geq 0.$$

We see that if we set $\sigma = 0$ we obtain (2.4) and that $\sigma = 1$ gives (2.6). In practice, a typical value for σ is 0.1. The step equation becomes

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ \mathcal{Z} & 0 & \mathcal{X} \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} h_d \\ h_p \\ -\mathcal{X}\mathcal{Z}e + \sigma\mu e \end{bmatrix}. \tag{2.9}$$

If a full step is taken along the primal or dual search directions the corresponding right-hand side h_p or h_d becomes and stays equal to zero. This is because (in the case of h_p)

$$A\Delta x = b - Ax,$$

so when we take a full step the new h_p becomes

$$b - A(x + \Delta x) = b - Ax - (Ax - b) = 0.$$

The same applies to h_d, where a full step gives us

$$h_d = c - (z + \Delta z) - A^T(y + \Delta y) = c - z - A^T y - (A^T \Delta y + \Delta z) = 0,$$

since $A^T\Delta y + \Delta z = c - z - A^T y$.

At each interior-point iteration we set the μ-value we aim for to be $x^T z/n$. This value will be the current value of μ if we are on the central path. The constant updating of μ however causes Newton's method to aim for a "moving target", so we are unlikely to actually obtain an iterate on the central path until we converge on the solution to the original problem (2.4). This situation is not a problem, but part of the process of steering the algorithm away from solutions which don't satisfy the non-negativity requirements regarding the variables x and z.

Outline of the Algorithm We can now describe a general algorithm for an interior-point method:

Given (x^0, y^0, z^0), where $(x, z) > 0$
While (not converged)
  Solve (2.9), where $\sigma \in [0, 1]$ and $\mu^k = \frac{(x^k)^T z^k}{n}$.

  Set $(x^{k+1}, y^{k+1}, z^{k+1}) = (x^k, y^k, z^k) + \alpha_k(\Delta x^k, \Delta y^k, \Delta z^k)$
  where α_k is chosen such that $(x^{k+1}, z^{k+1}) > 0$.

Since we demand that $(x^{k+1}, z^{k+1}) > 0$ (which we have to, otherwise the coefficient matrix in (2.9) might be singular), then even if we're converging on a solution, we will never produce an iterate which actually lies *on* the

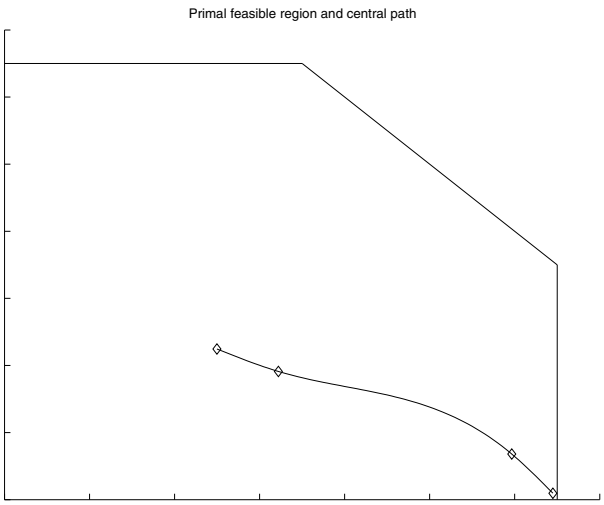

Figure 2.1: The primal feasible region and points on the central path for which $\mu = 50, 10, 1, 0.1$ for the problem $\min -2x_1 + x_2$, s.t. $x_1 \leq 6.5,\ x_2 \leq 6.5,\ x_1 + x_2 \leq 10$. The points have been interpolated with a cubic spline. The change in curvature shown in this case typically not present. Each point on the primal central path corresponds to a unique point on the dual central path, and hence the path shown can be considered a projection of the higher-dimensional central path containing all three variables $x,\ y$ and z.

boundary of the feasible region, although the solution itself always will. All feasible iterates will lie in the interior of the feasible region, and hence the name primal-dual interior-point methods. The term "primal-dual" refers to the fact that we solve both the primal and dual problems at the same time.

2.2 Extension to Semidefinite Programs

Extending the interior-point formulation for LP to SDP can be quite labourious. There are a few considerations to be made, as well as a few tricks to be employed. The KKT conditions for SDP are [15]:

$$\begin{array}{rcll} \sum_{i=1}^{m} y_i A_i \ + Z - C & = & 0, & \\ A_i \bullet X - b_i & = & 0, & i = 1 \dots m, \\ XZ & = & 0, & \end{array} \tag{2.10}$$

where as before

$$X \succeq 0 \text{ and } Z \succeq 0.$$

The SDP equivalent of (2.6), which defines the central path is:

$$F(X, y, Z) = \begin{bmatrix} \sum_{i=1}^{m} y_i A_i \ + Z - C \\ A_1 \bullet X - b_1 \\ \vdots \\ A_m \bullet X - b_m \\ XZ - \mu I \end{bmatrix} = 0. \tag{2.11}$$

From these equations we wish to derive a step equation resembling (2.9), but we have to deal with the following problem first: While the matrix XZ is diagonal on the central path, it in general is neither diagonal nor symmetric. If we differentiate (2.11) to obtain a step equation of the form (2.3) and subsequently iterate, we will obtain X-iterates which are not necessarily symmetric. (Z on the other hand is forced to be symmetric by the first KKT condition.) In other words, v and $F(v)$ in (2.2) won't necessarily belong to the same space and Newton's method is therefore not applicable. In order to be able to use Newton's method we replace

$$XZ - \mu I = 0$$

with

$$S_P(XZ) - \mu I = 0,$$

where S_P is a symmetrisation operator given as:

$$S_P(A) = \frac{1}{2}(PAP^{-1} + P^{-T}A^T P^T). \tag{2.12}$$

This reformulation guarantees symmetric X-iterates. The choice of the matrix P here is important. Different Ps lead to different search directions with different properties. We will use $P = Z^{\frac{1}{2}}$, which results in the HKM[1]

[1] Helmberg-Rendl-Vanderbei-Wolkowicz, Kojima-Shindoh-Hara, Monteiro.

search direction. Numerical results [18] have shown the HKM direction to converge faster (that is, use less CPU time) than other well-known directions such as the AHO ($P = I$), NT and GT directions. (The NT and GT directions correspond to less obvious choices of the matrix P [18].) Using the symmetrisation operator to modify (2.11) we get:

$$\begin{bmatrix} \sum_{i=1}^{m} y_i A_i \ + Z - C \\ A_1 \bullet X - b_1 \\ \vdots \\ A_m \bullet X - b_m \\ Z^{\frac{1}{2}} X Z^{\frac{1}{2}} - \mu I \end{bmatrix} = 0. \tag{2.13}$$

Here we have used the fact that Z (and therefore also $Z^{\frac{1}{2}}$ and $Z^{-\frac{1}{2}}$, see appendix A) is symmetric to eliminate the need for matrix transposes. Adding a centering parameter σ as discussed in the previous section, the step equation for the HKM direction is defined as the equations [13]:

$$\sum_{i=1}^{m} \Delta y_i A_i + \Delta Z = -\sum_{i=1}^{m} y_i A_i \ - Z + C, \tag{2.14}$$

$$A_i \bullet \Delta X = -A_i \bullet X + b_i \ , \ i = 1 \dots m, \tag{2.15}$$

which are shared by all the aforementioned step directions, and in addition the equation

$$Z^{\frac{1}{2}}(X\Delta Z + \Delta X Z)Z^{-\frac{1}{2}} + Z^{-\frac{1}{2}}(\Delta Z X + Z \Delta X)Z^{\frac{1}{2}} = 2(\sigma\mu I - Z^{\frac{1}{2}} X Z^{\frac{1}{2}}). \tag{2.16}$$

Although neat on paper, these equations are impractical when it comes to computing the desired Δ-variables. It is much easier to solve equations where the variables are vectors rather than matrices. This we can do, and to arrive at such a formulation we need to introduce a few operators originally introduced by Alizadeh, Haeberly and Overton [1].

The operator svec Let **svec** be an operator from $\mathcal{S}^n$ to $\mathbb{R}^{\overline{n}^2}$, where

$$\overline{n}^2 = \frac{n(n+1)}{2},$$

that is, the number of distinct elements in an $n \times n$ real symmetric matrix. **svec** stacks the columns of the lower (or upper, depending on the implementation) triangle of the matrix in a vector, and multiplies the off-diagonal entries by $\sqrt{2}$ so that

$$\mathbf{svec}(A)^T \mathbf{svec}(B) = A \bullet B,$$

for $A, B \in \mathcal{S}^n$.

The operator smat Let **smat** be the inverse operator of **svec**, that is the operator from $\mathbb{R}^{\overline{n}^2}$ to $\mathcal{S}^n$ which takes a vector and constructs a symmetric matrix, multiplying off-diagonal entries by $\frac{1}{\sqrt{2}}$.

Note that if the matrices in question are not full, but block diagonal with the same block structure, the operators can be modified to take this structure into account. This can in fact be extended to any general sparsity structure. Such a custom **svec** operator could replace the constraints of the graph example in section 1.3, which forced X to have a certain structure. As for block-diagonal matrices, if a symmetric block diagonal $n \times n$-matrix consists of k blocks, with block i of size

$$n_i \times n_i,$$

then its vector representation can be stored in a vector of length

$$q = \sum_{i=1}^{k} \frac{n_i(n_i+1)}{2}. \tag{2.17}$$

Provided that the block diagonal pattern in question encapsulates all the nonzero elements of $C,\ X,\ Z$ and $A_i,\ i = 1 \dots m$ whichever version of these operators is used has no mathematical implications, but when it comes to implementation the latter is clearly preferable. The obvious reason for this would be memory considerations, but there are also other aspects such as operation count, which we will return to in chapter 4.

The symmetric Kronecker product $\circledast$ Let $M, N \in \mathbb{R}^{n\times n}$ and $K \in \mathcal{S}^n$. The symmetric Kronecker product $\circledast$ is a product between the two $n \times n$ matrices M and N, and results in an $\overline{n}^2 \times \overline{n}^2$ matrix $(M \circledast N)$. It is implicitly defined by:

$$(M \circledast N)\mathbf{svec}(K) = \mathbf{svec}\left(\frac{1}{2}(NKM^T + MKN^T)\right).$$

We now have the necessary tools to construct a more desirable formulation at hand. Let

$$A = \begin{bmatrix} \mathbf{svec}(A_1)^T \\ \mathbf{svec}(A_2)^T \\ \vdots \\ \mathbf{svec}(A_m)^T \end{bmatrix}. \tag{2.18}$$

In other words, row i of A is the vector representation of A_i.

Let $x = \mathbf{svec}(X)$, and $z = \mathbf{svec}(Z)$. We will use this relation throughout for all variables, that is, if v is a vector then V is the corresponding

matrix $V = \mathbf{smat}(v)$ and vice versa. Equations (2.14) and (2.15) can then be written

$$\begin{aligned} A^T\Delta y + \Delta z &= \mathbf{svec}(C) - z - A^T y \\ A\Delta x &= b - Ax. \end{aligned}$$

Using the implicit definition of the symmetric Kronecker product and employing some algebra, (2.16) can be written as:

$$(Z^{\frac{1}{2}} \circledast Z^{\frac{1}{2}})\Delta x + (Z^{\frac{1}{2}} X \circledast Z^{-\frac{1}{2}})\Delta z = \mathbf{svec}(\sigma\mu I - Z^{\frac{1}{2}} X Z^{\frac{1}{2}}).$$

Now let $E = (Z^{\frac{1}{2}} \circledast Z^{\frac{1}{2}})$, $F = (Z^{\frac{1}{2}} X \circledast Z^{-\frac{1}{2}})$, and we finally get:

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ E & 0 & F \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} \mathbf{svec}(C) - z - A^T y \\ b - Ax \\ \mathbf{svec}(\sigma\mu I - Z^{\frac{1}{2}} X Z^{\frac{1}{2}}) \end{bmatrix}, \tag{2.19}$$

which is a linear system of equations with $2\overline{n}^2 + m$ unknowns and equations. If the **svec**-operator takes block structure into account, it becomes a system of $2q + m$ equations and unknowns.

Outline of the Algorithm On the central path we have

$$\frac{X \bullet Z}{n} = \mu.$$

To see this, consider: $XZ - \mu I = 0 \Rightarrow X = \mu Z^{-1}$. Now bullet-multiply by Z to obtain $(X \bullet Z) = \mu(Z^{-1} \bullet Z)$, which by the second definition of the bullet product is given as $\mu\ \mathbf{trace}(Z^{-1}Z) = \mu n$, from which the relation follows.

We use $\mu = (X \bullet Z)/n$ at each interior-point iteration, similar to what we did for LP. Furthermore, introducing separate step lengths for the primal and dual variables (thereby hopefully obtaining longer steps), the algorithm based on the above step equation can be stated as:

Algorithm PD-SDP

Given (X^0, y^0, Z^0) where X and $Z \succ 0$

While (not converged)
  Solve (2.19), where $\sigma \in [0, 1]$ and $\mu = \frac{X \bullet Z}{n}$.

  Set $X^{k+1} = X^k + \alpha \Delta X^k$
  where α is chosen such that $X^{k+1} \succ 0$.

  Set $(y^{k+1}, Z^{k+1}) = (y^k, Z^k) + \beta(\Delta y^k, \Delta Z^k)$
  where β is chosen such that $Z^{k+1} \succ 0$.

Chapter 3

Analysis of the Interior-Point Algorithm

In this chapter we take a closer look at the algorithm PD-SDP from the previous chapter, considering implementation issues and technical details regarding each individual step.

3.1 Initial Iterates

As with all incarnations of Newton's method, our algorithm needs an initial iterate to get started. If available, an estimate of the solution is likely to speed up convergence. If not, some initial value will have to be provided. A natural choice might be zero, but this choice won't work in our case, as the coefficient matrix in (2.19) will be singular.

A better choice, in the case of X and Z at least, would be the identity matrix, or a (strictly positive) multiple thereof. This ensures strict positive definiteness of the matrix variables and non-singularity of the step equation coefficient matrix. Now of course any initial iterate, well-considered or not, is still a guess as far as the algorithm is concerned. Therefore what multiple will be a good starting value is highly problem dependent.

As for an initial estimate of the vector y, no restrictions apply, so zero is fine here. A more advanced heuristic for determining starting values can be found in [18].

3.2 Solving the Step Equation

At the heart of the interior-point iteration is the determination of the step direction, the solution to (2.19). We could of course try to solve this equation system as it is written, but when m and n become large, storage of the coefficient matrix is impractical. For a problem with a primal variable size of 100×100 and 100 constraints the matrix will be $2\overline{n}^2 + m \times 2\overline{n}^2 + m$, or

10200×10200. (This is if the matrices are full. If they are block diagonal, the term $\overline{n}^2$ should as before be replaced by q defined in (2.17).) Instead, we will perform two steps of *block Gaussian elimination* in order to reduce the problem size, and back-substitute to obtain the entire solution. It is again useful to introduce shorthand for the right-hand side of the Newton step equation, for SDP (2.19). Define:

$$\begin{aligned} h_d &= \mathbf{svec}(C) - z - A^T y, \\ h_p &= b - Ax, \\ h_c &= \mathbf{svec}(\sigma\mu I - Z^{\frac{1}{2}} X Z^{\frac{1}{2}}), \end{aligned}$$

so that (2.19) becomes

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ E & 0 & F \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} h_d \\ h_p \\ h_c \end{bmatrix}. \tag{3.1}$$

As for LP, h_p is the primal infeasibility, h_d the dual infeasibility.

Now let R_i denote block of rows number i. To eliminate Δz we perform the following (block) row operations: $R_1 \leftarrow R_1 - F^{-1}R_3$, delete R_3. This gives us:

$$\begin{bmatrix} -F^{-1}E & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} h_d - F^{-1}h_c \\ h_p \end{bmatrix}. \tag{3.2}$$

Now, eliminate Δx: $R_2 \leftarrow R_2 + (AE^{-1}F)R_1$, delete R_1. Define

$$M = AE^{-1}FA^T. \tag{3.3}$$

We call M the *Schur Complement Matrix*. The reduced problem becomes:

$$M\Delta y = h_p + (AE^{-1})(Fh_d - h_c). \tag{3.4}$$

In the example mentioned above, M (which is of size $m \times m$) is 100×100, a significant reduction of the problem size. (3.4) is called the *Schur Complement Equation* (SCE) and is usually solved by means of a direct method, e.g. LU-factorisation. From (3.2) we have

$$\Delta x = -E^{-1}(F(h_d - A^T\Delta y) - h_c),$$

and from (3.1) that

$$\Delta z = h_d - A^T\Delta y.$$

The definitions of E and F are implicit, but using the definition of the symmetric Kronecker product we can arrive at explicit statements. From (3.3) we can obtain an expression for the ij-th element of M, which is given as

$$e_i^T M e_j,$$

where e_i is the vector with 1 in position i and zero elsewhere. We now have

$$M_{ij} = e_i^T A E^{-1} F A^T e_j,$$

which is the same as

$$\mathbf{svec}(A_i)^T E^{-1} F \ \mathbf{svec}(A_j),$$

which once E^{-1} and F are multiplied with $\mathbf{svec}(A_j)$ is a product between two vectors. The equivalent product between two matrices is

$$M_{ij} = A_i \bullet \mathbf{smat}\left(E^{-1} F \ \mathbf{svec}(A_j)\right). \tag{3.5}$$

Using three properties of the symmetric Kronecker product (see appendix A) this expression can be simplified. First of all, we use the fact

$$(G \circledast G)^{-1} = (G^{-1} \circledast G^{-1}),$$

which implies that

$$E^{-1} = (Z^{\frac{1}{2}} \circledast Z^{\frac{1}{2}})^{-1} = (Z^{-\frac{1}{2}} \circledast Z^{-\frac{1}{2}}).$$

In addition, we use

$$(H \circledast H)(G \circledast K) = (HG \circledast HK),$$

so that

$$E^{-1} F = (Z^{-\frac{1}{2}} \circledast Z^{-\frac{1}{2}})(Z^{\frac{1}{2}} X \circledast Z^{-\frac{1}{2}}) = (X \circledast Z^{-1}). \tag{3.6}$$

(3.5) then turns into

$$M_{ij} = A_i \bullet \mathbf{smat}\left((X \circledast Z^{-1})\mathbf{svec}(A_j)\right),$$

which is the same as

$$M_{ij} = A_i \bullet \frac{1}{2}(Z^{-1} A_j X + X A_j Z^{-1}).$$

Using this expression we can construct M element by element. The last fact we use is that if G and K are symmetric and positive definite, then so is $(G \circledast K)$. In our case this means that $(X \circledast Z^{-1})$ has these properties, and hence

$$\begin{aligned} M_{ij} &= \mathbf{svec}(A_i)^T (X \circledast Z^{-1}) \mathbf{svec}(A_j) \\ &= \mathbf{svec}(A_j)^T (X \circledast Z^{-1})^T \mathbf{svec}(A_i) = M_{ji}. \end{aligned}$$

M is in fact symmetric, which makes way for the use of specialised algorithms when solving (3.4). Performing similar analysis we can derive explicit expressions for the right-hand side in (3.4) and the matrix ΔX. The results we need are:

$$\begin{aligned} M\Delta y &= h_p + A \ \mathbf{svec}\left(\frac{1}{2}(Z^{-1} H_d X + X H_d Z^{-1}) - \sigma\mu Z^{-1} + X\right), \\ \Delta X &= \sigma\mu Z^{-1} - X - \frac{1}{2}(Z^{-1}\Delta Z X + X \Delta Z Z^{-1}), \end{aligned}$$

where $H_d = \mathbf{smat}(h_d)$. An important point here, from a computational point of view, is that although the different expressions of M_{ij} are all mathematically equivalent (the same applies to intermediate expressions regarding $M\Delta y$ and ΔX), their *numerical* properties may vary, and the expression ensuring the most stability need *not* be the simplest one. A discussion of this (though regarding the AHO step direction) can be found in [1].

3.3 Calculation of Step Lengths

Computing the longest permissible step length is a little trickier when we are dealing with matrices rather than vectors as in LP, but the following procedure gives us the answer we need.

Given $X \in \mathcal{S}^n,\ X \succeq 0$ and $\Delta X \in \mathcal{S}^n$, assume we want to find the largest $\alpha > 0$ such that

$$X + \alpha \Delta X \succ 0. \tag{3.7}$$

In addition, assume that X is positive definite and that its Cholesky factorisation is given by

$$X = R^T R.$$

(3.7) then is the same as

$$\alpha \Delta X \succ -R^T R.$$

Multiply from the left with R^{-T}, from the right with R^{-1} and divide by α (assuming it is positive) to obtain

$$R^{-T} \Delta X R^{-1} \succ -\frac{1}{\alpha} I.$$

Now, since the matrix on the left side of the expression has the same eigenvectors as the one on the right (every vector being an eigenvector of the identity matrix) all the eigenvalues of the matrix $R^{-T}\Delta X R^{-1}$ are larger than $-\alpha^{-1}$ and in particular

$$\lambda_{\min}(R^{-T} \Delta X R^{-1}) > -\frac{1}{\alpha}.$$

If $\lambda_{\min}$ is negative it follows that

$$\alpha < \frac{1}{-\lambda_{\min}(R^{-T} \Delta X R^{-1})}. \tag{3.8}$$

If we now set α equal to τ times the right-hand side, where $\tau \in (0, 1)$, what might happen is that we will get a negative α-value. The cause of this is that $\lambda_{\min}$ is positive, and means that the step direction points away from

the boundary of the semidefinite cone. We can then in principle take as long a step as we want, usually $\alpha = 1$. If $\lambda_{\min} < 0$, then the formula for α is:

$$\alpha = \min\left(1, \frac{\tau}{-\lambda_{\min}(R^{-T}\Delta X R^{-1})}\right).$$

A suitable τ-value might be for example 0.999 or 0.99, depending on the stability properties of the chosen step direction.

3.4 The centering Parameter σ

3.4.1 Algorithm PD-SDP

As pointed out in the previous chapter, a constant value for σ which has proven to work well in practice, is 0.1. Theoretically speaking, a successful choice of σ depends on the iterate we are working with. ("Successful" here meaning "ensuring quick convergence".) If we are close to the solution then setting σ as close to zero as possible and hence biasing the search direction towards the solution itself rather than a point on the central path would be desirable. As for points far from the solution, we would like them to approach the feasible region and the central path as quickly as possible, which can be done with a larger value for σ. Now $\sigma = 0.1$ certainly works well, but all in all, an adaptive σ would be the most desirable.

3.4.2 The Predictor-Corrector (PC) Scheme

When solving a semidefinite program, or a linear program for that matter, what we ideally want is to obtain a feasible point, and then follow the central path until we are so close to the solution that a pure Newton method would give us the solution we seek. An adaptive σ as outlined above would hopefully achieve this, and such a σ is obtained through the Predictor-Corrector scheme.

The idea, introduced by Mehrotra [12], is: First, we calculate the pure Newton step direction, from (3.1) with σ set to 0. If we can take long steps along this direction without violating the conditions $X \succ 0$ and $Z \succ 0$ then little centering is needed, and so we choose a small value for σ. If we, on the other hand can't make much progress along the Newton direction we select a larger value for σ. This is done using the following algorithm. Terms with a superscript P means terms associated with the predictor step, terms with superscript C are associated with the corrector step.

Algorithm PC

Given (X^0, y^0, Z^0) where X and $Z \succ 0$
While (not converged)

Set $\mu = \frac{X \bullet Z}{n}$.
Solve (3.1) with $\sigma = 0$,
to obtain the *predictor step*, $(\Delta X^P, \Delta y^P, \Delta Z^P)$.

Compute step lengths α^P and β^P such that $X + \alpha^P \Delta X^P \succ 0$,
and $Z + \beta^P \Delta Z^P \succ 0$.

Set $\mu^P = \frac{(X+\alpha^P \Delta X^P) \bullet (Z+\beta^P \Delta Z^P)}{n}$.

Set $\sigma = \left(\frac{\mu^P}{\mu}\right)^{expon}$, for an appropriate value of *expon*.

Solve (3.1), but replace h_c with
$h_c^C = \mathbf{svec}(\sigma\mu I - S_P(XZ) - S_P(\Delta X^P \Delta Z^P))$,
with S_P defined in (2.12) to obtain the *corrector step*,
$(\Delta X^C, \Delta y^C, \Delta Z^C)$.

Compute step lengths α^C and β^C such that
$X + \alpha^C \Delta X^C \succ 0$ and $Z + \beta^C \Delta Z^C \succ 0$.

Update iterates using the corrector step:

$$\begin{aligned} X^{k+1} &= X^k + \alpha^C \Delta X^C, \\ y^{k+1} &= y^k + \beta^C \Delta y^C, \\ Z^{k+1} &= Z^k + \beta^C \Delta Z^C. \end{aligned}$$

An "appropriate" value for *expon* is often determined experimentally. In LP *expon* = 3 is common, a common choice for the HKM direction in SDP is *expon* = 1. Other update formulae for σ are of course also possible. The corrector step Schur complement equation is

$$\begin{aligned} M\Delta y^C &= h_p + A\ \mathbf{svec}\left(\frac{1}{2}(Z^{-1}H_d X + X H_d Z^{-1}) - \sigma\mu Z^{-1} + X \right. \\ &\qquad \left. + \frac{1}{2}(\Delta X^P \Delta Z^P Z^{-1} + Z^{-1}\Delta Z^P \Delta X^P)\right), \end{aligned} \tag{3.9}$$

where as before $H_d = \mathbf{smat}(h_d)$. We need an expression for ΔX^C as well,

namely

$$\begin{aligned}\Delta X^C &= \sigma\mu Z^{-1} - X - \frac{1}{2}(Z^{-1}\Delta Z^C X + X\Delta Z^C Z^{-1}) \\ &\quad -\frac{1}{2}(\Delta X^P \Delta Z^P Z^{-1} + Z^{-1}\Delta Z^P \Delta X^P). \qquad (3.10)\end{aligned}$$

That the above choice of σ leads to the desired effect can be deduced by studying its formula. When we can take long steps along the pure Newton (or predictor) direction, μ^P is much smaller than μ, resulting in a small σ. When little progress is made, μ^P and μ differ little in magnitude, so the chosen σ is close to 1. The extra work involved in having to solve (3.1) two times is relatively modest — we can reuse an LU or Cholesky-factorisation when solving the second time.

The adaptive choice of σ is however not the only feature of PC-scheme. The addition of the term

$$-S_P(\Delta X^P \Delta Z^P)$$

in the right hand side of the corrector step corresponds utilising second-order (i.e. curvature) information about the central path. The combination of these two features makes the PC-scheme a very effective tool as far as quick convergence is concerned. Multiple corrections are also possible, an idea pursued for LP in [10].

Unfortunately, there is no convergence theory available for Mehrotra's scheme [14]. It usually outperforms PD-SDP, but there are examples for which it in fact diverges.

3.5 Stopping Criteria

With the main loop now described, the question becomes when to stop. If all goes well and we converge on a solution, there are three measures that all should be zero. These are the three KKT conditions for SDP, (2.10). Since we're usually working in floating-point arithmetic, such absolute accuracy would be too much to hope for and, depending on the computer used, values of a certain magnitude (for example 10^{-12}) must simply be considered to be equal to zero.

Having made our minds up about the error tolerance, a simple stopping rule would be to stop when

$$\max\left\{\|b - Ax\|,\ \|\mathbf{svec}(C) - z - A^T y\|,\ X \bullet Z\right\} \leq \text{ error tolerance},$$

for some norm $\|\cdot\|$.

This criterion of course assumes that all goes well, which need certainly not be the case. Any implementation should incorporate safeguards against

stagnation (i.e. two consecutive iterates are the same or step length becoming extremely small) as well as indefinite iterates. Especially when solving the step equation inexactly the latter is prone to happening.

Chapter 4

Inexact Schemes

In this chapter we will modify the approach described in chapter 3 by solving (3.1) inexactly. More specifically, inexact solution in practice means that we at each interior-point iteration solve the SCE (3.4) inexactly via an iterative equation solver, which performs a number of *inner* iterations to arrive at an approximate solution. The condition number of the Schur complement equation, $\kappa(M)$, grows rapidly as μ gets smaller, but hopefully we can make good progress along inexactly computed directions in early interior-point iterations, and switch to direct solution in the final stages of convergence.

The reason for the growth of $\kappa(M)$ is linked to the fact that Z and X and hence E and F become closer and closer to being singular as $X \bullet Z$ gets smaller. Solving (3.4) inexactly with the same error tolerance will consequently give larger and larger errors as the interior-point iterations progress. As shown in figure 4.1 $\kappa(M)$ is modest in early interior-point iterations, and arguably solving inexactly is a good idea.

For Newton's method inexact solution of the step equation was introduced in [9]. It has also been done in the case of LP (see [5] and the references therein), but little has been done to our knowledge when it comes to SDP.

We will show how the the error we make when solving (3.4) inexactly is propagated in the setting of PD-SDP (which is outlined in [16] as well) and make new observations about what happens in a predictor-corrector framework, as well as estimate the operation count of both inexact and direct solution. Based on this information, we present possible modifications which suggest a hybrid algorithm — one that switches between inexact and direct solution of the Schur complement equation.

Throughout this chapter, in order to make the algebra as simple as possible, we assume that full steps are taken when iterating, i.e.

$$\alpha = \beta = 1.$$

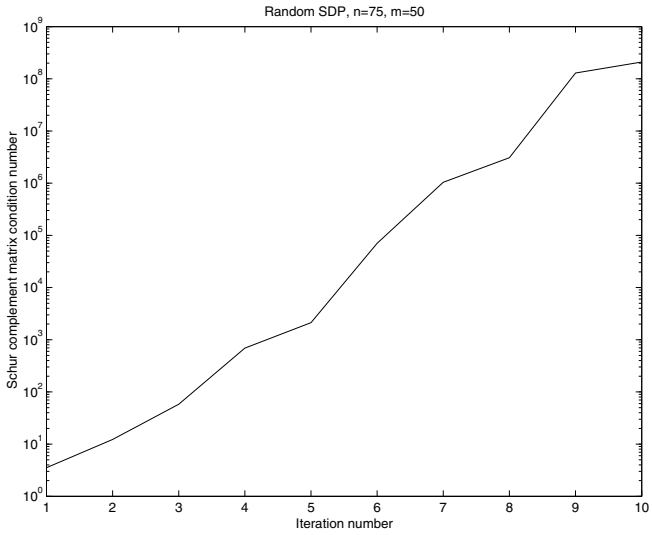

Figure 4.1: The condition number $\kappa(M)$ of the Schur complement matrix M for a random SDP problem with matrix sizes $n \times n$, $n = 75$, and number of constraints $m = 50$. As the interior-point iterations progress and the solution is approached, the conditioning of M steadily worsens.

4.1 Initial Observations

4.1.1 Inexact Solution of the SCE

We now investigate what happens if we solve (3.4) inexactly.

Let $\widetilde{\Delta y}$ be an inexact solution to (3.4), that is

$$M\widetilde{\Delta y} = h_p + (AE^{-1})(Fh_d - h_c) + r. \tag{4.1}$$

We now have

$$\widetilde{\Delta x} = -E^{-1}(F(h_d - A^T\widetilde{\Delta y}) - h_c),$$

and

$$\widetilde{\Delta z} = h_d - A^T\widetilde{\Delta y}.$$

To see how the error in $\widetilde{\Delta y}$ is propagated, we insert the inexact solutions $(\widetilde{\Delta x}, \widetilde{\Delta y}, \widetilde{\Delta z})$ into (3.1).

Row 1:

$$A^T\widetilde{\Delta y} + \widetilde{\Delta z} = A^T\widetilde{\Delta y} + (h_d - A^T\widetilde{\Delta y}) = h_d.$$

Row 2:

$$A\widetilde{\Delta x} = A(-E^{-1}(F(h_d - A^T\widetilde{\Delta y}) - h_c)).$$

Eliminating parentheses we get

$$\begin{aligned} & -AE^{-1}(Fh_d - FA^T\widetilde{\Delta y} - h_c) \\ = \quad & -AE^{-1}Fh_d + AE^{-1}FA^T\widetilde{\Delta y} + AE^{-1}h_c. \end{aligned}$$

$AE^{-1}FA^T$ is the same as M, so we can write this as

$$-AE^{-1}Fh_d + M\widetilde{\Delta y} + AE^{-1}h_c.$$

Substituting $M\widetilde{\Delta y}$ with the right-hand side in (4.1) we get

$$-AE^{-1}Fh_d + (h_p + (AE^{-1})(Fh_d - h_c) + r) + AE^{-1}h_c \quad = \quad h_p + r.$$

Row 3:

$$E\widetilde{\Delta x} + F\widetilde{\Delta z} = E(-E^{-1}(F(h_d - A^T\widetilde{\Delta y}) - h_c)) + F(h_d - A^T\widetilde{\Delta y}),$$

which is the same as

$$-(Fh_d - FA^T\widetilde{\Delta y} - h_c) + Fh_d - FA^T\widetilde{\Delta y} = h_c.$$

The right hand sides of row one and row three are the same as in (3.1), the right hand side in row two gets a contribution from the residual r. All in all, this means that when we solve (3.4) inexactly to obtain the solution to (3.1), what we really end up doing is solving is the following system:

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ E & 0 & F \end{bmatrix} \begin{bmatrix} \widetilde{\Delta x} \\ \widetilde{\Delta y} \\ \widetilde{\Delta z} \end{bmatrix} = \begin{bmatrix} h_d \\ h_p + r \\ h_c \end{bmatrix}. \tag{4.2}$$

We see that the residual ends up in the equation regarding primal feasibility [16]. If a full step is taken along this direction, the second KKT condition gives us:

$$A(x + \widetilde{\Delta x}) - b = Ax + h_p + r - b = Ax + (b - Ax) + r - b = r,$$

so the residual r actually becomes the new primal infeasibility.

4.1.2 Conservation of Primal Feasibility

As shown above, the residual (if present) from the Schur complement equation affects $\widetilde{\Delta x}$ so that it doesn't satisfy the step equation exactly. It is possible to have the residual affecting complementarity instead of primal feasibility, by at each interior-point iteration replacing $\widetilde{\Delta x}$ with

$$\overline{\Delta x} = \widetilde{\Delta x} - A^T(AA^T)^{-1}r,$$

and leaving $\widetilde{\Delta y}$ and $\widetilde{\Delta z}$ unchanged [16]. As before, r is the residual from the Schur complement equation. This is an inexpensive step, since we only need to compute the inverse (or Cholesky factorisation) of AA^T once. In addition, the dimension of AA^T is $m \times m$, so it is not prohibitely large to store. The row in the step equation regarding primal feasibility becomes

$$A\overline{\Delta x} = A\widetilde{\Delta x} - r,$$

which since $r = A\widetilde{\Delta x} - h_p$ is the same as

$$A\overline{\Delta x} = h_p.$$

The row in the step equation regarding complementarity becomes

$$E\overline{\Delta x} + F\widetilde{\Delta z} = E(I - A^T(AA^T)^{-1}A)\widetilde{\Delta x} + F\widetilde{\Delta z} = h_c + EA^T(AA^T)^{-1}r.$$

The row regarding dual feasibility, that is

$$A^T\widetilde{\Delta y} + \widetilde{\Delta z} = h_d,$$

does not contain any $\widetilde{\Delta x}$-terms and hence is not affected. Putting everything together, we get:

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ E & 0 & F \end{bmatrix} \begin{bmatrix} \overline{\Delta x} \\ \widetilde{\Delta y} \\ \widetilde{\Delta z} \end{bmatrix} = \begin{bmatrix} h_d \\ h_p \\ h_c + EA^T(AA^T)^{-1}r \end{bmatrix}. \tag{4.3}$$

The size of the term $EA^T(AA^T)^{-1}r$ can be analysed to some extent.
$E = (Z^{\frac{1}{2}} \circledast Z^{\frac{1}{2}})$, where $Z^{\frac{1}{2}}$ is symmetric, so we have:

$$EA^T(AA^T)^{-1}r = \mathbf{svec}\left(Z^{\frac{1}{2}}\mathbf{smat}(A^T(AA^T)^{-1}r)Z^{\frac{1}{2}}\right),$$

which is the same as

$$\mathbf{svec}\left(Z^{\frac{1}{2}}\left[\sum_{i=1}^{m}\left[(AA^T)^{-1}r\right]_i A_i\right]Z^{\frac{1}{2}}\right).$$

$\left[(AA^T)^{-1}r\right]_i$ denotes component i of the vector $(AA^T)^{-1}r$. Taking norms and using (A.1) we have

$$\|EA^T(AA^T)^{-1}r\|_2 = \left\|Z^{\frac{1}{2}}\left[\sum_{i=1}^{m}\left[(AA^T)^{-1}r\right]_i A_i\right]Z^{\frac{1}{2}}\right\|_F.$$

Using the relation $\|A + B\| \leq \|A\| + \|B\|$, an upper bound on the norm of the bracketed sum is the norm of the largest element of $(AA^T)^{-1}r$ multiplied

with the A_i of largest norm m times. The largest element of $(AA^T)^{-1}r$ can't be larger than $\|(AA^T)^{-1}r\|$, so we have

$$\left\| \sum_{i=1}^{m} \left[(AA^T)^{-1}r \right]_i A_i \right\| \leq m \ \|(AA^T)^{-1}r\| \ \max_i \|A_i\|.$$

Using this relation, as well as the relation $\|AB\| \leq \|A\| \ \|B\|$ successively, we arrive at the relation

$$\|EA^T(AA^T)^{-1}r\|_2 \leq m \ \|(AA^T)^{-1}\| \ \|r\| \ \|Z^{\frac{1}{2}}\|_F^2 \ \max_i \|A_i\|_F.$$

The only term here apart from $\|r\|$ which isn't constant is $\|Z^{\frac{1}{2}}\|_F^2$. We could scale the A_is in advance, but they would still be constant once we start solving. We cannot expect $\|Z^{\frac{1}{2}}\|_F^2$ to become smaller and smaller as we approach the solution, so the size of the entire residual may well depend only on $\|r\|$ throughout the interior-point iterations. The condition number of the equation system itself steadily increases, so the relative difference between the inexact step and its exact counterpart is likely to become bigger and bigger if $\|r\|$ is of about the same magnitude throughout. Hence, performing the projection of $\widetilde{\Delta x}$ to $\overline{\Delta x}$ with $\|r\|$ constant may well prevent us from obtaining a μ below a certain level.

4.1.3 Inexact Solution of the SCE within a PC-framework

A Fundamental Obstacle As mentioned earlier, when the condition number of the Schur complement equation is small, solving inexactly conceivably is a good idea. Predictor-corrector methods are very effective, so utilising this framework would be desirable. Unfortunately, we will then encounter the following inconvenient problem. While direct solvers perform little extra work to solve the same system of equations for two or more different right hand sides once the coefficient matrix is factorised, iterative methods have to start over from the very beginning each time. This means that even if we really can make use of inexactly computed search directions, doing so within a PC-framework means that we will have to perform work as if we were solving twice as many equation systems as with a direct approach. However, it is conceivably still a better idea than using an inexact version of algorithm PD-SDP (which would only involve one equation system per interior-point iteration, but requires more interior-point iterations to converge), since the condition number only increases every two equation systems we solve rather than each time.

Error Propagation As before, let a superscript P denote a term associated with the predictor step, and let a superscript C denote a term associated with the corrector step. Terms without superscripts are step invariant.

The step equation for the inexact predictor step without projection of $\widetilde{\Delta x}$ to $\overline{\Delta x}$ is:

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ E & 0 & F \end{bmatrix} \begin{bmatrix} \widetilde{\Delta x^P} \\ \widetilde{\Delta y^P} \\ \widetilde{\Delta z^P} \end{bmatrix} = \begin{bmatrix} h_d \\ h_p + r^P \\ h_c^P \end{bmatrix}. \tag{4.4}$$

The predictor step is used to calculate h_c^C in the corrector step step equation right-hand side. We have:

$$h_c^C = \mathbf{svec}\left(\sigma\mu I - S_P(XZ) - S_P(\Delta X^P \Delta Z^P)\right).$$

Since h_c^C is computed from the inexact predictor solution $(\widetilde{\Delta x^P}, \widetilde{\Delta y^P}, \widetilde{\Delta z^P})$ the calculated $\widetilde{h_c^C}$ will be inexact as well, that is, $\widetilde{h_c^C} = h_c^C + \varepsilon$. This is also the case if we do perform the projection of $\widetilde{\Delta x}$ to $\overline{\Delta x}$. By solving the resulting system of equations, again inexactly, we get the following inexact corrector step:

$$\begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ E & 0 & F \end{bmatrix} \begin{bmatrix} \widetilde{\Delta x^C} \\ \widetilde{\Delta y^C} \\ \widetilde{\Delta z^C} \end{bmatrix} = \begin{bmatrix} h_d \\ h_p + r^C \\ h_c^C + \varepsilon \end{bmatrix}. \tag{4.5}$$

The residual ε unfortunately turns out to be quite difficult to analyse, so we are forced to make experimental observations. Let us define:

$$G = \begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ E & 0 & F \end{bmatrix}$$

$$h^P = \begin{bmatrix} h_d \\ h_p \\ h_c^P \end{bmatrix} \tag{4.6}$$

$$h^C = \begin{bmatrix} h_d \\ h_p \\ h_c^C \end{bmatrix} \tag{4.7}$$

$$\widetilde{h^P} = \begin{bmatrix} h_d \\ h_p + r^P \\ h_c^P \end{bmatrix} \tag{4.8}$$

$$\widetilde{h^C} = \begin{bmatrix} h_d \\ h_p + r^C \\ h_c^C + \varepsilon \end{bmatrix}. \tag{4.9}$$

Let $\widetilde{\Delta^P}$ and $\widetilde{\Delta^C}$ be the solutions to (4.4) and (4.5) so that we can write (4.4) as

$$G\widetilde{\Delta^P} = \widetilde{h^P}$$

and (4.5) as

$$G\widetilde{\Delta^C} = \widetilde{h^C}.$$

Let Δ^P and Δ^C be their exact counterparts, so that

$$G\Delta^P = h^P$$

and

$$G\Delta^C = h^C.$$

Note that Δ^C is the exact solution to the corrector step step equation corresponding to an exact predictor step.

Then, an upper bound on the relative error we make when we end up solving (4.4) instead of its exact counterpart $G\Delta^P = h^P$ is given by the relation

$$\frac{\|\widetilde{\Delta^P} - \Delta^P\|}{\|\Delta^P\|} \leq \kappa(G) \frac{\|\widetilde{h^P} - h^P\|}{\|h^P\|},$$

where $\kappa(G)$ is the condition number of the matrix G. (See appendix A.) In the case of the inexact corrector step (4.5), we have

$$\frac{\|\widetilde{\Delta^C} - \Delta^C\|}{\|\Delta^C\|} \leq \kappa(G) \frac{\|\widetilde{h^C} - h^C\|}{\|h^C\|}.$$

The question becomes what the relation between the relative errors themselves is. If we assume r^C is zero and plot

$$\frac{\|r^P\|/\|h^P\|}{\|\varepsilon\|/\|h^C\|},$$

we get curves like in figure 4.2. Here we solved the problem with direct SCE solution as usual, and at each interior-point iteration tried inexact solution of the predictor step SCE with error tolerance

$$\frac{\|r^P\|}{\|M\widetilde{\Delta y^P} - r^P\|} \leq \left\{10^{-3},\ 10^{-6},\ 10^{-9}\right\},$$

to compare $\|r^P\|/\|h^P\|$ and $\|\varepsilon\|/\|h^C\|$. (The denominator $\|M\widetilde{\Delta y} - r^P\|$ is just the right-hand side of the exact predictor step Schur complement equation, (3.4) with $h_c = h_c^P$.) Note that in early interior-point iterations where we plan to take inexact steps, the upper bound on the relative error in the corrector step is larger than that of the predictor step, even if $r^C = 0$.

In later interior-point iterations this relation seems to be reversed, but now the condition number of Schur complement equation is large and we don't expect an inexact step to bring us closer so the solution of the SDP itself. Hence, this information about later interior-point iterations is not very useful. Since the corrector step residual ε contains the actual errors in $\widetilde{\Delta X^P}$

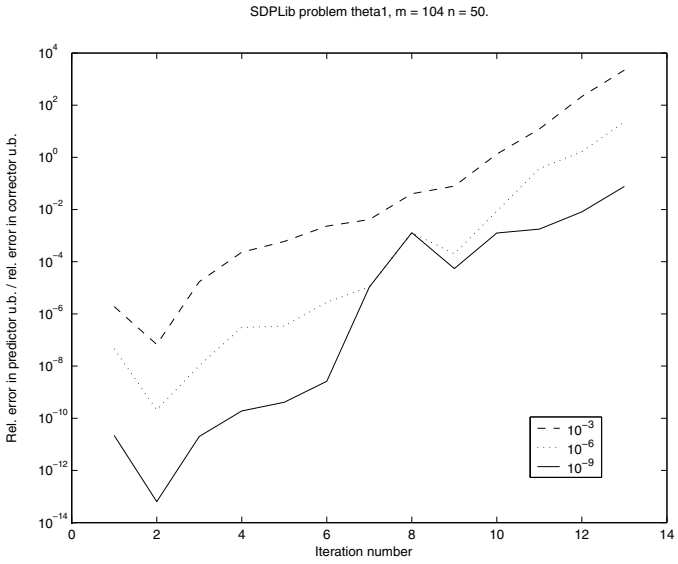

Figure 4.2: Plot of the upper bound of the relative error of an inexact predictor step divided by the upper bound of the relative error of its *exact* corrector step. The problem is SDPLIB problem theta1 [6].

and $\widetilde{\Delta Z^P}$ (and not just the residual r^P), it doesn't sound implausible that the error we make in the corrector step is larger than that of the predictor step. As we plan to solve the corrector step step equation inexactly, causing an even larger error to be made, we should consider tightening the error tolerance when solving the corrector step Schur complement equation.

Of course the discussion regarding the projection of $\widetilde{\Delta x}$ to $\overline{\Delta x}$ applies here in the corrector step as well, with the Schur complement equation residual either affecting primal feasibility or complementarity.

4.2 Implicit Representation of the SCE

When choosing which method to use to solve the SCE inexactly, *Krylov subspace methods*, such as for example the conjugate gradient (CG) method is a natural choice. One of the advantages of these methods is that when solving a system of equations they do not need an explicit representation of the coefficient matrix, but instead only the ability to compute the matrix-vector product for any vector [20]. This property turns out to be potentially useful in our case — the Schur complement equation (3.4). Element-by-element construction of the Schur complement matrix M can be quite expensive, so when solving inexactly there might be advantages to the implicit approach at least as far as computation time is concerned. In any event, if we use the

formula:

$$M_{ij} = \mathbf{svec}(A_i)^T \ \mathbf{svec}\left(\frac{1}{2}(Z^{-1}A_jX + XA_jZ^{-1})\right),$$

we can generate M using the following algorithm:

Given A defined as in (2.18), X, and R such that $Z = R^TR$,

for $(j = 1 \dots m)$,

$tmp \leftarrow R^{-1}R^{-T}\mathbf{smat}(A_{(j,:)})X$
$tmp_{sym} \leftarrow \frac{1}{2}(tmp + tmp^T)$
$w_j \leftarrow \mathbf{svec}(tmp_{sym})$

for $(i = j \dots m)$,
$M_{ij} \leftarrow A_{(i,:)}w_j$
$M_{ji} \leftarrow M_{ij}$
end

end

Here we have used a Matlab-like notation $A_{(i,:)}$ to denote row i of the matrix A. We take the Cholesky factor R of Z for granted, since we will need it to compute step lengths whether we generate the Schur complement matrix or not.

In the three first lines of the outermost for-loop the second vector in the formula for M_{ij} is constructed, and in the innermost for-loop the product is carried out. Assume now that the A_i-matrices, X and Z are dense matrices. The innermost loop is run one time for each distinct element in M, $\frac{1}{2}m(m+1)$ times. Each time it performs $\overline{n}^2 - 1$ additions and $\overline{n}^2$ multiplications. The outermost loop performs one matrix multiplication ($O(n^3)$), one **svec** and one **smat** (each $O(\overline{n}^2)$) and two instances of back-substitution (corresponding to the triangular matrices R^{-1} and R^{-T}), both also $O(n^3)$ of work. The symmetrisation of tmp corresponds to $\overline{n}^2 - n$ additions and $\overline{n}^2 - n$ multiplications. All in all, noting that $\overline{n}^2 = O(\frac{1}{2}n^2)$ and keeping only the highest-order terms, the construction of M takes

$$O(3mn^3 + \frac{1}{2}m^2n^2)$$

operations. If we are dealing with block diagonal matrices this bound on the operation can be modified, as software packages like Matlab can be expected to optimise matrix multiplications and back-substitutions based on block structure. Therefore, as commented in [18], the terms n^3 and n^2 should be replaced by the sum of the cubes and squares of the individual

block sizes (let block i be of size $n_i \times n_i,\ i = 1 \dots k$), giving us an operation count of

$$O(3m \sum_{i=i}^{k} n_i^3 + \frac{1}{2} m^2 \sum_{i=1}^{k} n_i^2).$$

In addition the **svec** and **smat** operations can be made more effective as discussed earlier. As for implicit representation of M, we for a given vector v have:

$$Mv = AE^{-1}FA^T v = AE^{-1}F(A^T v).$$

Using (3.6), this is the same as

$$A(X \circledast Z^{-1})(A^T v).$$

Each row of A contains the vector version of a constraint matrix. Therefore $\mathbf{smat}(A^T v)$ is the same as

$$\sum_{i=1}^{m} v_i A_i,$$

and is symmetric since the matrices $A_i,\ i = 1 \dots m$ are. We finally get

$$Mv = A\ \mathbf{svec}\left(\frac{1}{2}(Z^{-1}\mathbf{smat}(A^T v)X + X\mathbf{smat}(A^T v)Z^{-1})\right), \qquad (4.10)$$

here taking into account the symmetry of X and Z^{-1} as well to eliminate transposes.

The operation count here consists of the product $A^T v$ which is $O(2m\overline{n}^2)$ operations, **smat** ($O(\overline{n}^2)$), multiplication by X and two instances of back-substitution corresponding to multiplication of Z^{-1} (all three put together $O(3n^3)$), one symmetrisation ($O(2\overline{n}^2)$), one **svec** ($O(\overline{n}^2)$), and finally multiplication by A, which is $O(2m\overline{n}^2)$. Discarding the pure $\overline{n}^2$-terms and again noting that $\overline{n}^2 = O(\frac{1}{2}n^2)$, we all in all get an operation count for the product Mv of

$$O(3n^3 + 2mn^2)$$

operations. If we are dealing with block-diagonal matrices the correct operation count is

$$O(3 \sum_{i=i}^{k} n_i^3 + 4mq)$$

operations, with q defined as in (2.17).

Roughly speaking, we can from these O-expressions expect inexact solution to be cheaper than direct solution if the sum of inner iterations in the predictor and corrector steps is less than m.

4.3 Description of the Schemes

First of all we make a few assumptions and observations:

- Assume that the initial iterates X^0 and Z^0 are multiples of the identity matrix, and are likely to be far from the solution. This results in a large duality gap, and a well-conditioned Schur complement matrix.
- Observe that the accuracy used in the corrector step should be more restrictive than that used in the predictor step, to minimise error propagation.
- Observe that an inexact interior-point iteration using the predictor-corrector scheme where

$$\frac{\|\widetilde{h^P} - h^P\|}{\|h^P\|} \ll \frac{X \bullet Z}{n}, \quad \text{as well as} \quad \frac{\|\widetilde{h^C} - h^C\|}{\|h^C\|} \ll \frac{X \bullet Z}{n}, \tag{4.11}$$

  using the definitions from (4.6)-(4.9), is equally effective as an interior-point iteration employing direct solution. That is, if we solve inexactly under the condition above, the total number of interior-point iterations it takes to solve the entire problem is about the same as with direct solution throughout. This hypothesis is supported by preliminary numerical testing.

Using these bullet points as well as the other observations made in this chapter, we devised two simple schemes for testing.

4.3.1 Scheme 1 — Inexact SCE Solution with PC

The idea behind this scheme is very simple: Using the predictor-corrector approach, solve the SCE inexactly (two times per interior-point iteration) with a loose error tolerance for the predictor step, and a more restrictive error tolerance for the corrector step. In our experiments we used an SCE relative residual of norm 10^{-4} in the predictor step and a relative residual of norm 10^{-8} in the corrector step, that is

$$\frac{\|r^P\|}{\|M\widetilde{\Delta y^P} - r^P\|} \leq 10^{-4}, \quad \text{and} \quad \frac{\|r^C\|}{\|M\widetilde{\Delta y^C} - r^C\|} \leq 10^{-8}.$$

The denominators here are just the right hand side of (3.4) with $h_c = h_c^P$ in case of the predictor step, and $h_c = \widetilde{h_c^C}$ in case of the corrector step. The Schur complement matrix is not generated explicitly, but represented by a function which returns Mv for any v using (4.10). Keep solving inexactly until one of the following occurs:

- The operation count for the previous interior-point iteration was more than 85% of the cost of an iteration employing direct solution. Operation counts are estimated using the O-expressions above. An interior-point iteration with inexact solution is assumed to cost the number of inner iterations in the predictor step plus the number of inner iterations in the corrector step multiplied by the cost of the product Mv. An interior-point iteration with direct solution is assumed to cost the number of operations used to generate M, as well as the cost of computing its Cholesky factorisation ($O(\frac{1}{3}m^3)$, [20]) and back-substitution. The threshold of 85% may of course be at the user's discretion, this is only a suggestion.

- The equation solver used fails to return a sufficiently approximate solution within the maximum number of allowed inner iterations. In our experiments we set this number to m, so this never occurred.

- (4.11) no longer holds. If this happens, we expect inexact solution of the SCE to yield large errors in both the predictor and corrector steps, that is, that $\widetilde{\Delta^P}$ and $\widetilde{\Delta^C}$ will differ much from their exact counterparts. Our codes actually don't check this, as our choice of corrector step accuracy causes inexact solution to be abandoned due to cost before it becomes a problem. It should however not be overlooked.

When inexact solution is deemed undesirable, switch to generating M and solving (3.4) directly.

4.3.2 Scheme 2 — Inexact SCE Solution with PC and Conservation of primal Feasibility

This scheme is essentially the previous scheme only with projection of $\widetilde{\Delta x}$ to $\overline{\Delta x}$ to obtain and preserve primal feasibility in both the predictor and corrector steps (but only when solving inexactly). The projection adds slightly to the cost of an inexact interior-point iteration since we have to perform $O(2m^2 + 2mq)$ work when projecting. As discussed previously, we thereby transfer the error we make to the part of the inexact step equation regarding complementarity. If the error we make when solving (4.3) compared to exact solution of the step equation is large, we risk stagnation, since we might end up aiming over and over again for a point corresponding to a much larger value for $X \bullet Z$ than intended. However, a large error in the solution to (4.3) usually corresponds to a large condition number in the coefficient matrix, and therefore also in the coefficient matrix of the Schur complement equation. It will then be difficult to solve, and we are likely to switch to direct solution. Indeed, in our experiments this scheme performed well and seemingly without any such stagnation problems.

Chapter 5

Numerical Results

5.1 Computer Code used

Many excellent software packages for solving SDP problems are available, and when trying to improve existing algorithms modifying existing code is certainly a feasible option. To maintain complete control over the code with which we were to conduct experiments however, we decided to create code from scratch[1], resulting in a package of Matlab m-files given the name lfsdp.

To check the validity and stability of our code we tried several sample runs on randomly generated problems to see if our code would produce the same iterates as existing software. We chose to compare ourselves to the SDPT3 v2.3 software package [19] which implements the HKM direction. Below are graphs of three sample runs, on one small ($n = 75,\ m = 75,\ A_i$ sparse) problem, as well as a larger ($n = 500,\ m = 500,\ A_i$ sparse) problem and a problem with dense coefficient matrices ($n = 200,\ m = 100$), all randomly generated. Both codes use the same heuristics when updating σ, primal and dual step lengths, and have the same stopping criteria. In the first example the iterates are almost indistinguishable on these plots, and in the second example they closely follow each other. From these two runs we could conclude that the two codes are equivalent, but if we look at the problem with dense constraint matrices, we get a somewhat different result. Here SDPT3 uses one more interior-point iteration to converge, and the iterates of the two codes do not match each other as well as before. The question becomes if this is due to numerical properties or an actual error in our implementation. Since, on sparse problems, the two different codes behave in a similar fashion, (the behaviour on the two sparse problems shown here is quite typical) we are satisfied that ours is indeed a valid implementation of the HKM direction, but with different numerical properties than SDPT3.

It should be added that on challenging problems, SDPT3 outperforms lfsdp when it comes to computation time, probably due to its use of spe-

[1] Four subroutines from [2, 3] were used. We are grateful to its authors.

cialised MEX routines when it comes to computing the Schur complement matrix.

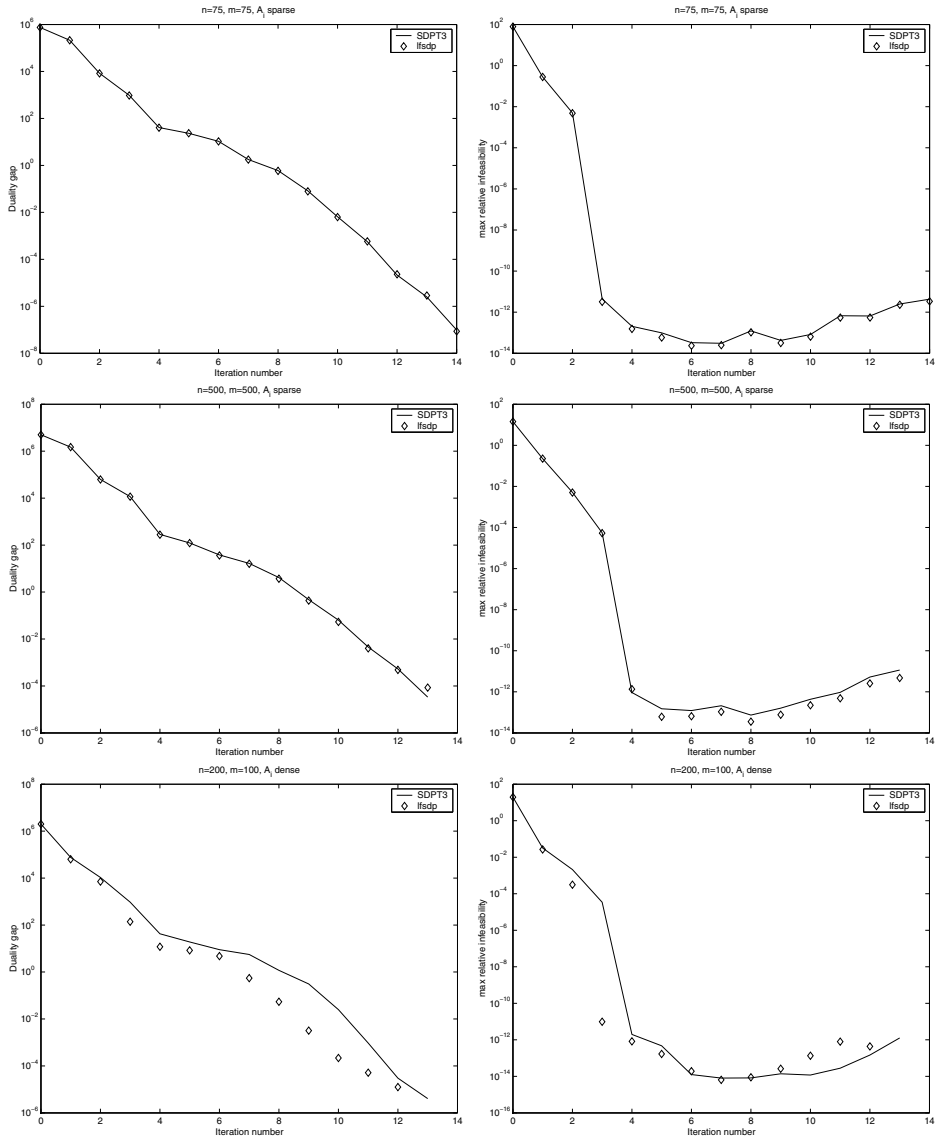

Table 5.1: Plots of the duality gap and maximum relative infeasibility of three randomly generated problems. The first two problems involve sparse matrices, the third dense matrices.

5.2 Heuristics and Convergence Criteria used

As described in the previous section, we adopted stopping criteria as well as heuristics for determining σ, α and β from SDPT3 when trying to replicate its iterates. As it turned out, these heuristics helped making lfsdp more stable, so they were adopted permanently, and are outlined below:

$$\sigma = \min\left(1, \left[\frac{\mu^P}{\mu}\right]^{expon}\right),$$

where as before the superscripts P and C correspond to the predictor and corrector step, respectively and *expon* is chosen as

$$expon = \left\{ \begin{array}{ll} 1 & \text{if } \mu > 10^{-6} \text{ and } \min(\alpha^P, \beta^P) < 1/\sqrt{3}, \\ \max[1, 3\min(\alpha^P, \beta^P)^2] & \text{if } \mu > 10^{-6} \text{ and } \min(\alpha^P, \beta^P) \geq 1/\sqrt{3}, \\ 1 & \text{if } \mu \leq 10^{-6}. \end{array} \right\}$$

α and β are computed as before, only with

$$\tau^P = 0.9 + 0.09\min(\alpha^C, \beta^C),$$

where α^C and β^C are from the previous interior-point iteration, and

$$\tau^C = 0.9 + 0.09\min(\alpha^P, \beta^P),$$

with the initial value of τ being 0.9. The convergence criterion is:

$$\max\left\{\frac{\|b - Ax\|}{\max(1, \|b\|)}, \frac{\|\mathbf{svec}(C) - z - A^T y\|}{\max(1, \|\mathbf{svec}(C)\|)}, \frac{X \bullet Z}{n}\right\} \leq 10^{-8}.$$

The main difference here from the convergence criterion outlined in chapter 3 is that we here consider relative rather than absolute magnitudes when it comes to all three error measures.

5.3 A typical Run

Our experiments consisted of comparing the two schemes against the traditional approach of solving the SCE directly on the same problem and on the same computer. The output from a typical run looked like this:

```
knott.ii.uib.no
Fri Mar 29 19:24:33 CET 2002
```

This is the name of the computer, which is a 1000MHz workstation running Linux, as well as a time stamp. Next follows the problem dimensions:

```
m = 300
n = 400
q = 80200
```

q means the number of distinct elements in the $n \times n$-matrices, so these are full in this instance. Now we solve the problem (which here is a random problem generated using routines from SDPPack v0.8 beta [2, 3]), using the first scheme, without projection:

```
iter    pstep        dstep      pinfeas     dinfeas       gap         val        sigma

  0   0.000e+00    0.000e+00   1.583e+02   1.315e+00   4.000e+06  -5.758e+02
  1   1.000e+00    9.742e-01   1.179e-05   3.399e-02   1.236e+05   9.466e+03     0.024
  2   1.000e+00    9.877e-01   1.050e-05   4.191e-04   1.351e+04   6.149e+03     0.003
  3   9.822e-01    9.838e-01   9.990e-07   6.798e-06   3.164e+02   2.579e+01     0.001
  4   8.891e-01    1.000e+00   1.842e-07   1.775e-16   3.853e+01  -1.126e+02     0.013
  5   3.729e-01    1.000e+00   1.238e-07   1.775e-16   3.324e+01  -1.150e+02     0.644
  6   9.071e-01    1.000e+00   1.720e-08   1.765e-16   5.372e+00  -1.288e+02     0.076
  7   1.000e+00    9.063e-01   4.437e-09   1.703e-16   2.252e+00  -1.302e+02     0.416
  8   9.132e-01    1.000e+00   6.030e-09   1.778e-16   6.688e-01  -1.309e+02     0.232
  9   9.423e-01    9.522e-01   4.128e-09   1.728e-16   6.408e-02  -1.312e+02     0.041
 10   9.599e-01    9.742e-01   5.801e-09   1.749e-16   5.121e-03  -1.312e+02     0.042
Operation count becoming too large for inexact solution...
Solving SCE directly from now on
 11   9.858e-01    1.000e+00   8.244e-11   1.785e-16   2.688e-04  -1.312e+02     0.039
 12   1.000e+00    1.000e+00   7.641e-14   1.777e-16   1.643e-05  -1.312e+02     0.061
 13   1.000e+00    1.000e+00   1.655e-13   1.766e-16   1.456e-06  -1.312e+02     0.089
 14   1.000e+00    1.000e+00   2.656e-13   1.777e-16   3.456e-08  -1.312e+02     0.024
 15   1.000e+00    1.000e+00   2.007e-12   1.771e-16   1.117e-09  -1.312e+02     0.032
Success, error reduced to value desired.
Cpu time spent: 19570.49 seconds.
```

The output columns are designed to mimic those of SDPT3 so they display, from left to right:

iter	Interior-point iteration number.
pstep	Length of the primal corrector step, that is, α^C from algorithm PC.
dstep	Length of the dual corrector step, β^C from algorithm PC.
pinfeas	Relative primal infeasibility $\frac{\|b-Ax\|}{\max(1,\|b\|)}$.
dinfeas	Relative dual infeasibility $\frac{\|\mathbf{svec}(C)-z-A^T y\|}{\max(1,\|\mathbf{svec}(C)\|)}$.
dgap	$X \bullet Z$
val	The average of the primal and dual objective function values, $\frac{1}{2}(C \bullet X + b^T y)$.
sigma	σ used in the corrector step.

Note that when a full primal step is taken, the new primal infeasibility becomes the residual from the Schur complement equation instead of zero.

Next, we solve the same problem from the same starting point using the scheme with projection:

```
iter    pstep        dstep      pinfeas     dinfeas       gap         val        sigma

  0   0.000e+00    0.000e+00   1.583e+02   1.315e+00   4.000e+06  -5.758e+02
  1   1.000e+00    9.742e-01   8.215e-12   3.399e-02   1.236e+05   9.466e+03     0.024
  2   1.000e+00    9.877e-01   6.277e-12   4.191e-04   1.351e+04   6.149e+03     0.003
  3   9.822e-01    9.836e-01   7.575e-13   6.864e-06   3.166e+02   2.590e+01     0.001
  4   8.891e-01    1.000e+00   9.265e-14   1.772e-16   3.856e+01  -1.126e+02     0.013
  5   3.734e-01    1.000e+00   6.048e-14   1.772e-16   3.325e+01  -1.150e+02     0.643
  6   9.071e-01    1.000e+00   7.083e-15   1.779e-16   5.375e+00  -1.288e+02     0.076
  7   1.000e+00    9.067e-01   2.900e-15   1.700e-16   2.253e+00  -1.302e+02     0.416
```

```
  8   9.136e-01    1.000e+00   2.664e-15   1.781e-16   6.691e-01  -1.309e+02     0.232
  9   9.423e-01    9.528e-01   2.562e-15   1.733e-16   6.431e-02  -1.312e+02     0.041
 10   9.698e-01    9.807e-01   4.730e-15   1.752e-16   5.168e-03  -1.312e+02     0.052
Operation count becoming too large for inexact solution...
Solving SCE directly from now on
 11   9.697e-01    9.788e-01   3.693e-14   1.754e-16   2.600e-04  -1.312e+02     0.021
 12   1.000e+00    1.000e+00   9.708e-14   1.789e-16   1.674e-05  -1.312e+02     0.064
 13   1.000e+00    1.000e+00   1.890e-13   1.766e-16   1.697e-06  -1.312e+02     0.101
 14   1.000e+00    1.000e+00   2.289e-13   1.767e-16   3.822e-08  -1.312e+02     0.023
 15   1.000e+00    1.000e+00   1.926e-12   1.779e-16   1.295e-09  -1.312e+02     0.034
Success, error reduced to value desired.
Cpu time spent: 19816.75 seconds.
```

The main difference here is that when a full primal step is taken, the relative primal infeasibility becomes and stays zero. This scheme does however not behave with notable difference from the previous one when it comes to the number of inexact interior-point iterations or computation time.

Finally we solve the same problem using the traditional approach:

```
iter    pstep        dstep      pinfeas     dinfeas       gap         val        sigma

  0   0.000e+00    0.000e+00   1.583e+02   1.315e+00   4.000e+06  -5.758e+02
  1   1.000e+00    9.742e-01   1.016e-11   3.399e-02   1.236e+05   9.466e+03     0.024
  2   1.000e+00    9.877e-01   7.217e-12   4.192e-04   1.351e+04   6.149e+03     0.003
  3   9.822e-01    9.836e-01   7.447e-13   6.870e-06   3.167e+02   2.592e+01     0.001
  4   8.892e-01    1.000e+00   9.147e-14   1.775e-16   3.856e+01  -1.126e+02     0.013
  5   3.737e-01    1.000e+00   5.943e-14   1.786e-16   3.324e+01  -1.150e+02     0.643
  6   9.071e-01    1.000e+00   7.696e-15   1.761e-16   5.374e+00  -1.288e+02     0.076
  7   1.000e+00    9.067e-01   3.207e-15   1.688e-16   2.255e+00  -1.302e+02     0.416
  8   9.133e-01    1.000e+00   3.368e-15   1.791e-16   6.690e-01  -1.309e+02     0.231
  9   9.424e-01    9.522e-01   3.931e-15   1.742e-16   6.419e-02  -1.312e+02     0.041
 10   9.597e-01    9.742e-01   1.190e-14   1.753e-16   5.130e-03  -1.312e+02     0.042
 11   9.860e-01    1.000e+00   4.940e-14   1.765e-16   2.714e-04  -1.312e+02     0.040
 12   1.000e+00    1.000e+00   8.251e-14   1.778e-16   1.656e-05  -1.312e+02     0.061
 13   1.000e+00    1.000e+00   1.614e-13   1.794e-16   1.453e-06  -1.312e+02     0.088
 14   1.000e+00    1.000e+00   2.626e-13   1.778e-16   3.473e-08  -1.312e+02     0.024
Success, error reduced to value desired.
Cpu time spent: 22630.97 seconds.
```

Once a full step is taking along either the primal or dual search direction, the corresponding infeasibility term becomes zero, as expected. The number of interior-point iterations is roughly the same as before, but the total computation time is about 14% longer.

5.4 Listing of Results

We tested the codes on both problems with full coefficient matrices, and on problems with 20 equal blocks of dimension $(n/20) \times (n/20)$, referred to as "dense" and "sparse" problems, respectively, as well as on some selected SDPLIB [6] problems. The results are listed below. S1 means the scheme without projection, S2 the one with projection. The columns S1 vs lfsdp and S2 vs lfsdp indicate solution time relative to that of lfsdp. A relative solution time of less than 100% means that S1 or S2 was faster than lfsdp.

Dense problems				
m	n	lfsdp sol. time	S1 vs lfsdp	S2 vs lfsdp
100	100	102s	116%	118%
100	200	692s	103%	98%
100	300	2735s	103%	104%
100	400	7130s	103%	103%
100	500	11599s	81%	82%
200	100	311s	104%	106%
200	200	1829s	87%	83%
200	300	6122s	75%	76%
200	400	13573s	82%	83%
200	500	24798s	74%	76%
300	100	519s	103%	106%
300	200	3484s	87%	88%
300	300	12985s	89%	90%
300	400	22630s	86%	88%
400	100	906s	91%	93%
400	200	5774s	80%	82%
400	300	19538s	75%	76%
500	100	1435s	100%	101%
500	200	8871s	84%	86%
500	300	26578s	75%	69%

Figure 5.1: Experimental results of the three codes on 20 different dense randomly generated problems, generated with [2].

The pattern in the results seems to be that when the problems reach a certain size, the inexact schemes arrive at the solution faster than the direct approach, in terms of cpu time. On small problems they on the other hand usually get outperformed by a significant relative margin, though this for the most part doesn't mean much in terms of seconds. An example: The biggest relative increase in cpu time spent was when scheme 1 used 169% of the time used by lfsdp on a small problem, trailing by about 28 seconds. The biggest relative reduction in cpu time spent was when scheme 2 used 69% of the time lfsdp used on a large problem, which amounted to a solution time reduction of two hours and 17 minutes.

It is difficult to announce a "winner" between the two schemes themselves. Scheme 1 wins most of the time, but on the largest problem scheme 2 is significantly faster than scheme one.

Sparse problems					
m	n	q	lfsdp sol. time	S1 vs lfsdp	S2 vs lfsdp
100	100	300	17.84s	108%	107%
100	200	1100	49.52s	104%	104%
100	300	2400	105.92s	99%	101%
100	400	4200	237.15s	90%	92%
100	500	6500	456.57s	87%	89%
200	100	300	45.52s	110%	110%
200	200	1100	111.08s	102%	104%
200	300	2400	272.17s	93%	94%
200	400	4200	576.09s	91%	91%
200	500	6500	1041.56s	83%	85%
300	100	300	40.07s	169%	157%
300	200	1100	199.2s	118%	114%
300	300	2400	411.55s	93%	95%
300	400	4200	1038.97s	90%	92%
300	500	6500	1905.25s	79%	82%
400	200	1100	297.66s	104%	105%
400	300	2400	832.6s	83%	85%
400	400	4200	1494.22s	76%	77%
400	500	6500	2554.23s	87%	88%
500	200	1100	403.23s	106%	100%
500	300	2400	921.68s	87%	88%
500	400	4200	2208.1s	82%	82%
500	500	6500	4228.68s	73%	78%

Figure 5.2: Experimental results of the three codes on 23 different sparse randomly generated problems, generated with [2].

Selected SDPLIB problems						
Name	m	n	q	lfsdp sol. time	S1 vs lfsdp	S2 vs lfsdp
arch0	174	335	13215	2293.64s	94%	94%
arch4	174	335	13215	2168.99s	88%	88%
mcp100	100	100	5050	128.57s	92%	93%
mcp124-1	124	124	7750	294.64s	87%	88%
ss30	132	426	43497	5470.72s	N/A	87%
theta1	104	50	1275	29.61s	89%	89%
theta2	498	100	5050	2044.7s	84%	77%
truss2	58	133	331	11.29s	107%	106%
truss5	208	331	1816	228.82s	93%	94%

Figure 5.3: Experimental results of the codes on selected SDPLIB [6] problems. The "N/A"-entry signifies scheme one failing to converge.

Chapter 6

Conclusions

6.1 Experiment Evaluation

Although we have only tested our schemes on about 50 concrete problems, most of which which in addition are random and hence unlikely to exhibit special structure real-world problems might have, the results obtained must be said to be encouraging. As the size of the problems grow, the two proposed schemes outperform the direct approach with an increasing relative margin, as much as 31% of the total computation time in one of our experiments. The direct approach is the most effective on smaller problems, but this is not unexpected.

What is puzzling, though, is the relationship between the performances of the two schemes themselves. In most cases they perform the same number of both inexact and direct interior-point iterations, and spend roughly the same amount of time arriving at a solution even though their individual iterates are different, at least when it comes to primal feasibility. The reason for this similar behaviour remains unclear, as we know little about the global convergence properties of any our three solution methods. Both schemes must be said to have performed well in practice.

It should of course be noted that the convergence criteria are a factor when it comes to total computation time. If the tighten the global error tolerance then we will have to perform more interior-point iterations and this will favour the direct approach compared to our results. If we loosen the error tolerance, it will favour our schemes.

6.2 Possible Modifications

Our codes are relatively rudimentary, and might well benefit from some refinement. Possibilities are:

- *Code stability, problem scaling.* In our preliminary experiments, the codes used including lfsdp frequently failed to converge on non-random

problems. There can be several causes to this, for example that our algorithms are sensitive to problem scaling. By scaling we mean multiplying objective function coefficients and problem constraints with constants in a way which doesn't alter the optimal solution but results in better numerical properties. Implementing this technique in one way or the other could possibly enhance the performance of our codes.

Another possible cause is that even if the expressions for right hand sides, Schur complement matrix elements, step lengths and so on may be written in many mathematically equivalent ways, their numerical properties may vary. It may well be that our software would benefit from experimenting with different formulations to see which would result in the most stability.

- *Adaptive decision-making.* Often, the hallmark of a good algorithm is the ability to adapt to the problem at hand. In our case, this could be the ability to determine whether or not projection of $\widetilde{\Delta x}$ is a good idea, whether or not we should use preconditioning (see below) when solving the Schur complement equation inexactly, what error tolerance should be used, and so on. It is quite possible that such adaptivity could improve performance.

6.3 Topics and Issues not addressed

There are a number of issues we haven't given attention or explored to full depth:

- *Preconditioning.* Preconditioning is a very important tool when it comes to inexact solution of linear equation systems, and is outlined in [20]. Its main goal is to make the iterative solver converge to the solution of the equation system in as few iterations as possible. In our case, this translates to obtaining a good PC-step with as few inner iterations as possible. We have based the desirability of a PC-step largely on computation time, and hence preconditioning of the Schur complement equation might be of use to us.

- *Matlab MEX files.* Many Matlab SDP solvers use specialised routines often written in C++ (MEX files) to compute the Schur complement matrix. This we could do as well, but we haven't as it would favour the direct approach. Of course one might argue that doing so would show that our schemes are less useful than we claim, but this would not be "fair" in our view as the operations used to construct the SCE would take less time than those used to perform the product Mv. Such unfairness could namely also be made to favour our schemes; if SCE generation is written in C++ and the routine performing the product Mv is written in pure machine language, then the tables have turned.

- *Global convergence analysis.* Global convergence theory is not fully developed when it comes to the predictor-corrector approach, as outlined in [14], and consequently we haven't been able to shed light on the global convergence properties of inexact PC. We can of course rely on our experimental results when claiming our schemes have the right to life, but the theory behind it all remains incomplete.

6.4 Summary

We set out to investigate whether inexact solution of the Schur complement equation when solving SDPs could be beneficial. We have surveyed research already done in this field, and made new observations about inexact SCE solution within a predictor-corrector framework, as well as observations regarding the operation count involved in the individual steps of Newton's method.

Using this information we devised schemes which implement direct and inexact SCE solution in a hybrid fashion, starting out with inexact solution and switching to direct solution once inexact solution becomes expensive, in terms of operation count.

Through experiments we have shown that on both random and non-random problems of a certain size, the suggested schemes can reduce the solution time without losing accuracy. A number of questions remain unanswered, but our schemes seem to perform well, and the motivation behind them seems to be a good idea.

Appendix A

Notation and Terminology

In this appendix we define some of the notation and terminology which is used throughout the text. It is mainly intended as reference material.

A.1 Basic definitions

Vectors and Matrices We denote vectors by lowercase letters, for example x. If x is made up of n real numbers, then we say that x belongs to the space $\mathbb{R}^n$, or simply $x \in \mathbb{R}^n$. Such a vector can also be referred to as an n-vector, although this doesn't say anything about the nature of its components. Matrices are denoted by uppercase letters. For a matrix A with m rows and n columns made up of real numbers we say that $A \in \mathbb{R}^{m \times n}$. If $m = n$ we say that the matrix is *square*.

Individual elements are denoted by a subscript, so x_i means the i-th element of x, and A_{ij} means the element found at row i, column j.

Linear Independence Given i vectors $\{x^1, x^2, \ldots, x^i\}$ of length n, where $i \leq n$, if the relation

$$x^i = \sum_{j=1}^{i-1} c_j x^j$$

doesn't hold for *any* real numbers $\{c_1, \ldots, c_{i-1}\}$ or *any* ordering of the vectors x^j, then we say that $\{x^1, \ldots, x^j\}$ are *linearly independent*. If the relation does hold, the vectors are said to be linearly *dependent*. In words, this means that none of the vectors can be constructed as a linear combination of any of the other vectors if they are linearly independent.

If the columns (or rows) of a square matrix are linearly independent, then the matrix is said to be *nonsingular*, otherwise it is said to be *singular*.

Eigenvalues and Eigenvectors For an $n \times n$-matrix A, if for some vector x

$$Ax = \lambda x,$$

where λ is a scalar (real or complex), we say that x is an *eigenvector* of A and that λ is the corresponding *eigenvalue*. If x is an eigenvector, then so is cx for any $c \in \mathbb{R}$, but it is common and useful to say that eigenvectors are the vectors scaled so that $\|x\| = 1$. An $n \times n$-matrix may have up to n distinct eigenvalues and eigenvectors. An eigenvalue may correspond to more than one eigenvector, but not the other way around. The determinant of a matrix is equal to the product of its eigenvalues, so a matrix is singular if and only if it has one or more zero eigenvalues. If the $n \times n$-matrix A has n linearly independent eigenvectors (which need not be the case even if A is nonsingular), then there exists a factorisation

$$A = X^{-1}\Lambda X,$$

where the columns of X are the scaled eigenvectors of A, and Λ is a diagonal matrix with the corresponding eigenvalues of A on its diagonal. Powers and roots of matrices can now easily be computed using the observation that the factors X and X^{-1} cancel each other when A is multiplied by itself, so that

$$A^n = X^{-1}\Lambda^n X.$$

If A is symmetric, then

$$X^{-1}\Lambda X = X^T\Lambda^T X^{-T} \Rightarrow X^{-1} = X^T.$$

Combining these observations, we can deduce that if A is symmetric, then so is A^{-1}, A^2, and in general A^n for any integer n.

An additional observation is that if two matrices have the same set of eigenvectors, they commute.

Big O-Notation This notation is used to provide an upper bound on the number of operations it takes to perform a certain task. One definition is that if a task takes $f(n)$ operations to complete, then an upper bound on the operation count is $O(g(n))$, where $g(n)$ is a function such that

$$0 \leq f(n) \leq cg(n),$$

for some $c > 0$ and $n \geq n_0$, for some n_0. An example:

$$f(n) = \frac{1}{2}n^3 + 4n^2 + n$$

is $O(n^3)$. The definition is satisfied if $c > 1/2$ and n_0 is the value of n for which the two functions are equal in value. Constants are often dropped when it comes to giving upper bounds, but we have kept them in the text since we are dealing with functions of more than one variable. For more information on big O-notation and related material, see for example [8].

A.2 Spaces and Sets

We have already mentioned the spaces $\mathbb{R}^n$ and $\mathbb{R}^{m\times n}$. In $\mathbb{R}^n$ each element (that is, each n-vector) has n independent components, so we say that the *dimension* of this space is n. The dimension of $\mathbb{R}^{m\times n}$, is mn.

The Space $\mathcal{S}^n$ Let $\mathcal{S}^n$ denote the space of real symmetric matrices. Here the individual components of each member matrix are not independent, since for a matrix $A \in \mathcal{S}^n$, we must have $A_{ij} = A_{ji}$. The dimension of this space we will denote by $\overline{n}^2$, and is given by

$$\overline{n}^2 = \frac{n(n+1)}{2}.$$

Positive definite Matrices Matrices for which $x^T Ax > 0$ for all $x \neq 0$ are said to be *positive definite*. This property is indicated using the notation $A \succ 0$. These matrices have the property that all their eigenvalues are strictly positive. Positive definite matrices encountered in practice are often symmetric, but they need not be, for example the matrix

$$\begin{bmatrix} a & -b \\ b & a \end{bmatrix}$$

is positive definite, but not symmetric. For symmetric and positive definite matrices there exists a factorisation

$$A = R^T R,$$

where R is upper-triangular. (Or equivalently, $A = LL^T$, where L is lower-triangular.) This factorisation is known as the *Cholesky* factorisation, and is cheaper to compute than the standard LU-factorisation.

It is meaningful to talk about the set of positive definite matrices. If A and B are positive definite, then so is $cA + dB$ for scalars c and d such that $c + d > 0$. This set is therefore a cone.

Positive Semidefinite Matrices Positive semidefinite matrices are an extension of positive definite matrices. These are matrices for which $x^T Ax \geq 0$ for all x. The main difference in this definition compared to the definition of positive definite matrices is that a positive semidefinite matrix need not have full rank. Therefore, for a positive semidefinite matrix we have that for all eigenvalues, $\lambda_i \geq 0$. All positive definite matrices are positive semidefinite, but the opposite is not true.

The semidefinite cone is the set which arises from linear combination of semidefinite matrices where the sum of the coefficients is nonnegative.

A.3 Functionals and Operators

trace The *trace* of an $n \times n$-matrix is defined to be the sum of its diagonal elements:

$$\mathbf{trace}(A) = \sum_{i=1}^{n} A_{ii}.$$

The trace of a matrix interestingly turns out to be equal to the sum of its eigenvalues.

The Bullet Product $\bullet$ We define $\bullet$ to be an inner product between two matrices in $\mathbb{R}^{n \times n}$. Let

$$A \bullet B = \mathbf{trace}(B^T A) = \sum_{i=1}^{n} \left(\sum_{j=1}^{n} A_{ij} B_{ij} \right).$$

If A and B both belong to $\mathcal{S}^n$, $A \bullet B$ is equal to $\mathbf{trace}(AB)$.

Condition Numbers The condition number of a matrix A, $\kappa(A)$, is given as the number

$$\kappa(A) = \|A\| \|A^{-1}\|,$$

for some norm $\|\cdot\|$. It is useful in light of the relation

$$\frac{1}{\kappa(A)} \cdot \frac{\|r\|}{\|b\|} \leq \frac{\|e\|}{\|x\|} \leq \kappa(A) \frac{\|r\|}{\|b\|},$$

where x is the solution to $Ax = b$ and given an approximate solution $\widetilde{x}$,

$$\|e\| = \|x - \widetilde{x}\| \text{ and } \|r\| = \|b - A\widetilde{x}\|.$$

A proof of this relation can be found in [7]. What it means is that when approximately solving a system of equations, the relative error $\|e\|/\|x\|$ which we can say little about without solving the system exactly can be as large as $\kappa(A)$ times the relative residual $\|r\|/\|b\|$, which can easily be computed. An important implication is that if we want to solve a system of equations with a high condition number, also called an *ill-conditioned* system, we have to use high accuracy to solve even if we just want moderate accuracy in the solution. An example: If a matrix has a condition number of 10^8, then a solution with relative residual 10^{-10} could still have a relative error of 10^{-2}.

The Jacobian Matrix of a Function F If F is a function from $\mathbb{R}^n$ to $\mathbb{R}^m$, that is, it transforms an n-vector v to an m-vector $F(v)$, then its

Jacobian matrix is the $m \times n$-matrix:

$$\begin{bmatrix} \frac{\partial F_1}{\partial v_1} & \frac{\partial F_1}{\partial v_2} & \dots & \frac{\partial F_1}{\partial v_n} \\ \frac{\partial F_2}{\partial v_1} & \frac{\partial F_2}{\partial v_2} & \dots & \frac{\partial F_2}{\partial v_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial F_m}{\partial v_1} & \frac{\partial F_m}{\partial v_1} & \dots & \frac{\partial F_m}{\partial v_n} \end{bmatrix},$$

where v_i is the i-th element of v and F_j is the j-th component of the vector function F. See for example [4] for more details.

svec and smat Let **svec** be an operator from $\mathcal{S}^n$ to $\mathbb{R}^{\overline{n}^2}$. **svec** stacks the distinct elements of the matrix in a vector, and multiplies the off-diagonal entries by $\sqrt{2}$ so that

$$\mathbf{svec}(A)^T\mathbf{svec}(B) = A \bullet B,$$

for $A, B \in \mathcal{S}^n$. Let **smat** be the inverse operator of **svec**, that is, the operator from $\mathbb{R}^{\overline{n}^2}$ to $\mathcal{S}^n$ which takes a vector and constructs a symmetric matrix, multiplying off-diagonal entries by $\frac{1}{\sqrt{2}}$. An example might clarify things:
Let

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 6 \\ 3 & 6 & 9 \end{bmatrix}, \; B = \begin{bmatrix} 5 & 7 & 2 \\ 7 & 3 & 9 \\ 2 & 9 & 11 \end{bmatrix}.$$

We now have

$$A \bullet B = \mathbf{trace}(AB) = \mathbf{trace}\left(\begin{bmatrix} 25 & 40 & 53 \\ 57 & 83 & 115 \\ 75 & 120 & 159 \end{bmatrix}\right) = 267.$$

Another way to obtain the bullet-product between A and B we get by first multiplying each component in A with the corresponding component in B to obtain:

$$\begin{bmatrix} 5 & 14 & 6 \\ 14 & 15 & 54 \\ 6 & 54 & 99 \end{bmatrix}.$$

$A \bullet B$ is now the sum of the elements of this matrix, again 267. As stated above, we can also perform this product using **svec**. We let $\mathbf{svec}(A)$ stack the columns of the lower-triangular part of A in a vector. (In theory we can order the elements any way we want, as long as we are consistent.) Multiplying the off-diagonal elements by $\sqrt{2}$, we get:

$$\mathbf{svec}(A) = \begin{bmatrix} 1 \\ 2\sqrt{2} \\ 3\sqrt{2} \\ 5 \\ 6\sqrt{2} \\ 9 \end{bmatrix}, \; \mathbf{svec}(B) = \begin{bmatrix} 5 \\ 7\sqrt{2} \\ 2\sqrt{2} \\ 3 \\ 9\sqrt{2} \\ 11 \end{bmatrix}.$$

Note that the length of these vectors is 6, which is $\overline{n}^2$, the number of distinct elements in A and B. Performing the inner product between these two vectors is to give us $A \bullet B$, and indeed

$$\mathbf{svec}(A)^T \mathbf{svec}(B) = 267.$$

Exploiting Block Structure with svec We can make **svec** and **smat** more effective by taking block diagonal structure, if present, into account. If, for example we have the matrix

$$A = \begin{bmatrix} E & & & \\ & F & & \\ & & G & \\ & & & H \end{bmatrix},$$

where E through H are symmetric blocks, not necessarily of equal size, then instead of implementing the **svec** operation as in the example above, we can store the distinct nonzero elements more economically in the vector

$$\begin{bmatrix} \mathbf{svec}(E) \\ \mathbf{svec}(F) \\ \mathbf{svec}(G) \\ \mathbf{svec}(H) \end{bmatrix}.$$

This makes the economy-version of svec an operator from $\mathcal{S}^n$ to $\mathbb{R}^q$, where

$$q = \sum_{i=1}^{k} \frac{n_i(n_i+1)}{2},$$

and $n_i \times n_i$ is the dimension of block number i. Of course, such a modified **svec** is only applicable if we want to bullet-multiply two matrices with identical block structure.

svec, smat and Norms For a vector x the 2-norm $\|x\|_2$ is given as

$$\|x\|_2 = \sqrt{x^T x}.$$

For a matrix A the *Frobenius norm* $\|A\|_F$ is given as

$$\|A\|_F = \mathbf{trace}(A^T A) = A \bullet A.$$

If we're dealing with a symmetric matrix, say V, then

$$V \bullet V = \mathbf{svec}(V)^T \mathbf{svec}(V).$$

From these relations we can see that if V is symmetric and $v = \mathbf{svec}(V)$, then

$$\|v\|_2 = \|V\|_F. \tag{A.1}$$

The symmetric Kronecker Product $\circledast$ Let $M, N \in \mathbb{R}^{n\times n}$ and $K \in \mathcal{S}^n$. The symmetric Kronecker product $\circledast$ is implicitly defined by:

$$(M \circledast N)\mathbf{svec}(K) = \mathbf{svec}\left(\frac{1}{2}(NKM^T + MKN^T)\right).$$

What happens is that we from two $n \times n$-matrices M and N, which are *not* necessarily symmetric, construct the matrix $(M \circledast N)$ which is of dimension $\overline{n}^2 \times \overline{n}^2$. This new matrix is not necessarily symmetric either, see the properties listed below. When multiplied to $\mathbf{svec}(K)$, where K is symmetric, we get the vector version of a symmetric matrix as can be seen from the implicit definition above. To construct to matrix $(M \circledast N)$ explicitly from our definition we could for example multiply it with the individual columns of the $\overline{n}^2 \times \overline{n}^2$ identity matrix. Some properties of $\circledast$ are:

$$\begin{aligned}
G \circledast K &= K \circledast G, \\
(G \circledast K)^T &= G^T \circledast K^T, \\
(G \circledast G)^{-1} &= G^{-1} \circledast G^{-1}, \\
(G \circledast K)(H \circledast H) &= (GH \circledast KH), \\
(H \circledast H)(G \circledast K) &= (HG \circledast HK).
\end{aligned}$$

If G and K are symmetric and positive definite, then so is $(G \circledast K)$. For additional properties, see for example [17].

A.4 Notation and Terminology specific to LP and SDP

Inequality Constraints In our standard LP problem we have the constraints

$$x \geq 0 \text{ and } z \geq 0.$$

This is non-standard notation when it comes to mathematics in general. What is meant is,

$$x_i \geq 0, i = 1 \dots n, \text{ and } z_i \geq 0, \ i = 1 \dots n,$$

respectively. The corresponding relation when it comes to matrices is, as mentioned above

$$X \succeq 0, \text{ and } Z \succeq 0,$$

which means that X and Z are positive semidefinite. If X and Z are positive definite, we write

$$X \succ 0, \text{ and } Z \succ 0.$$

Sometimes, however, we might encounter an expression like this:

$$A \succeq B.$$

The meaning of this expression is that the matrix $A - B$ is positive semidefinite. The same applies to such expressions involving positive definite matrices.

Terminology In both LP and SDP we work with functions which are subject to constraints. Consider the primal LP problem. Any n-vector x is said to be a *solution*, whether it satisfies the constraints in question or not. A vector x which does satisfy the constraints is said to be a (primal) *feasible* solution. Similarly, a pair of vectors (y, z) satisfying the constraints of the dual problem are said to be a (dual) feasible solution. If no primal or dual feasible solutions exist, the problem is said to be infeasible. The constraints themselves are referred to as the primal and dual *feasibility constraints*.

Primal and dual *infeasibility* are the expressions

$$b - Ax,$$

and

$$c - z - A^T y,$$

respectively. These are measures of how close the points x and y are to satisfying the constraints of the LP or SDP problem. The term primal/dual infeasibility is often used to mean the norms of the two expressions as well.

Appendix B

Derivation of the Central Path Equations for LP

In this appendix we outline the motivation behind the interior-point method for linear programming in chapter 2. It is mainly intended as a brief introduction for the reader unfamiliar with these ideas. A more thorough discussion can be found in [14].

B.1 Reformulation of the LP

If we're only interested in the optimal values of the variables of the primal and dual LP problems rather than the optimal objective function value, the linear program we use as a standard (both primal and dual) can be written as follows:

$$\min c^T x - b^T y$$

such that

$$Ax = b$$

and

$$A^T y + z = c$$

where

$$x \geq 0,\ z \geq 0.$$

We assume that $A \in \mathbb{R}^{m \times n}$ has full rank, that is, that its *rows* are linearly independent, and consequently that $m \leq n$.

Let's say now that we want to try to convert this constrained problem into a problem without constraints. Such problems are usually much easier to solve. If we wanted to eliminate the condition

$$(x, z) \geq 0,$$

we could replace it by terms in the objective function which are small when x and $z \geq 0$, and prohibitely large otherwise. This can be done in many ways, but an obvious choice would be the logarithm function, or more precisely the negative thereof:

$$-\sum_{i=1}^{n} \mathbf{log}(x_i) - \sum_{i=1}^{n} \mathbf{log}(z_i).$$

(A function like this is known as a logarithmic *barrier function.*) This function has a modest growth rate when x and z are positive, and goes to infinity as x and z tend to zero. Negative (x, z)-values are thus not a problem anymore, but as we would like x and z to able to be very close to zero should the original problem demand it, we introduce a scaling parameter μ. The reason for this is to compensate for the rapid decrease in the logarithm function as its argument approaches zero, giving us:

$$-\mu \sum_{i=1}^{n} \mathbf{log}(x_i) - \mu \sum_{i=1}^{n} \mathbf{log}(z_i).$$

We can now reformulate the LP problem to give us an intermediate problem:

$$\min c^T x - b^T y - \mu \sum_{i=1}^{n} \mathbf{log}(x_i) - \mu \sum_{i=1}^{n} \mathbf{log}(z_i),$$

such that

$$Ax = b, \text{ and } A^T y + z = c.$$

It is important to note that this problem is *not* the same problem as the original one, but if we choose smaller and smaller values for μ it will become an increasingly better approximation to it.

Turning our attention to the remaining constraints we see that if they had been of the type $Ax \leq b$ we could have reused the logarithmic technique, but unfortunately it doesn't work for equality constraints. As it turns out, we can't replace for example $Ax = b$ by

$$Ax \leq b, \ Ax \geq b$$

and use logarithms on each individual constraint either. If we do so, all the constraints will still have to be satisfied in the is-equal sense, and the logarithms will shoot the objective function off to infinity. Consequently, the feasible set will be empty. What we need is a more versatile tool.

B.2 Lagrange Multipliers

Lagrange multipliers are a tool in analysis which deal with the problem of finding maxima and/or minima of (not necessarily linear) functions subject to one or more constraints. The idea stems from the following theorem:

Theorem B.1 *If a function*

$$f(x_1, \ldots, x_n) : \mathbb{R}^n \mapsto \mathbb{R}$$

has a local maximum/minimum when subject to m constraints

$$\left\{ \begin{array}{c} g_1(x_1, \ldots, x_n) \\ \vdots \\ g_m(x_1, \ldots, x_n) \end{array} \right\} = 0,$$

where $m < n$, then there exist m scalars λ_i, $i = 1 \ldots m$, such that at each maximum/minimum

$$\nabla f = \sum_{i=1}^{m} \lambda_i \nabla g_i.$$

If we look at the so-called *Lagrangian function*:

$$L(x, \lambda) = f(x) + \sum_{i=1}^{m} \lambda_i g(x),$$

where here $x = (x_1, \ldots, x_n)$ and $\lambda = (\lambda_1, \ldots, \lambda_m)$, then setting the partial derivatives of $L(x, \lambda)$ equal to zero satisfy the original m constraints as well as the relation in theorem B.1. In other words, to find the maximum or minimum of a constrained function (provided that such a point exists), all we need to do is to find the critical points of the corresponding Lagrangian function. More information on Lagrange multipliers can be found in [4].

B.3 Lagrange Multipliers applied to LP

Applying the technique to our problem, we get:

$$\min f = c^T x - b^T y - \mu \sum_{i=1}^{n} \mathbf{log}(x_i) - \mu \sum_{i=1}^{n} \mathbf{log}(z_i) + \lambda^T (Ax - b) + \eta^T (A^T y + z - c).$$

To obtain a critical point what we need to do is to set all the individual partial derivatives to zero. Differentiating, we get:

$$\begin{aligned}
\frac{\partial f}{\partial x_j} &= c_j - \frac{\mu}{x_j} + \sum_{i=1}^{m} \lambda_i A_{ij} \\
\frac{\partial f}{\partial y_i} &= -b_i + \sum_{j=1}^{n} \eta_j A_{m+1-i,j} \\
\frac{\partial f}{\partial z_j} &= -\frac{\mu}{z_j} + \eta_j \\
\frac{\partial f}{\partial \lambda_i} &= \sum_{j=1}^{n} A_{ij} x_j - b_i \\
\frac{\partial f}{\partial \eta_j} &= \sum_{i=1}^{m} A_{m+1-i,j} y_i + z_j - c_j
\end{aligned}$$

$$\text{for } i = 1 \ldots m,\ j = 1 \ldots n,$$

all of which are to be zero. If we let

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}, \ \frac{\partial f}{\partial y} = \begin{bmatrix} \frac{\partial f}{\partial y_1} \\ \vdots \\ \frac{\partial f}{\partial y_m} \end{bmatrix}$$

and so on, we can write these equations more compactly as:

$$\begin{aligned}
\frac{\partial f}{\partial x} &= c - \mu \mathcal{X}^{-1} e + A^T \lambda = 0, \\
\frac{\partial f}{\partial y} &= -b + A\eta = 0, \\
\frac{\partial f}{\partial z} &= -\mu \mathcal{Z}^{-1} e + \eta = 0, \\
\frac{\partial f}{\partial \lambda} &= Ax - b = 0, \\
\frac{\partial f}{\partial \eta} &= A^T y + z - c = 0.
\end{aligned} \tag{B.1}$$

The two last equations here are the familiar LP constraints. The three first can be manipulated further. If we multiply the first equation by $\mathcal{X}$ and the third by $\mathcal{Z}$, the first three rows become:

$$\begin{aligned}
\mathcal{X}c - \mu e + \mathcal{X}A^T\lambda &= 0, \\
A\eta - b &= 0, \\
\mathcal{Z}\eta - \mu e &= 0.
\end{aligned} \tag{B.2}$$

If we now set

$$\lambda = -y, \text{ and } \eta = x,$$

then the first equation turns into

$$\mathcal{X}c - \mu e - \mathcal{X}A^T y = 0.$$

Since $-A^T y = z - c$ this is the same as

$$\mathcal{X}c - \mu e + \mathcal{X}(z - c) = 0,$$

or

$$\mathcal{X}z = \mu e,$$

which is the same as

$$x_i z_i = \mu,\ i = 1 \dots n.$$

The second equation turns into the primal feasibility constraints, and the third turns into

$$\mathcal{Z}x - \mu e = 0,$$

which again is the same as $x_i z_i = \mu,\ i = 1 \dots n$. So all in all we end up with

$$\begin{aligned} Ax - b &= 0, \\ A^T y + z - c &= 0, \\ \mathcal{X}\mathcal{Z}e - \mu e &= 0. \end{aligned} \tag{B.3}$$

With the given choice of η and λ (B.1) certainly implies (B.3), but we don't know if the converse also is true. That is, if the relation

$$\mathcal{X}\mathcal{Z}e = \mu e \Rightarrow \lambda = -y,\ \eta = x$$

holds. If we take a look at (B.2) and substitute μe with $\mathcal{X}\mathcal{Z}e$ in the first row, we get

$$\mathcal{X}c - \mathcal{X}\mathcal{Z}e + \mathcal{X}A^T\lambda = 0.$$

Multiplying by $\mathcal{X}^{-1}$ gives us

$$c - \mathcal{Z}e + A^T\lambda = 0,$$

which is the same as

$$-A^T\lambda + z - c = 0.$$

Now $A^T\lambda$ is the linear combination of the columns of A^T with coefficients λ_i, and since those columns are linearly independent, we must have $\lambda = -y$. If we look at the third row of (B.2) and substitute μe with $\mathcal{Z}\mathcal{X}e$, we get

$$\mathcal{Z}\eta - \mathcal{Z}\mathcal{X}e = 0.$$

Multiplying by $\mathcal{Z}^{-1}$ gives us

$$\eta - \mathcal{X}e = 0,$$

which is the same as $\eta = x$, so (B.1) with $\eta = x$ and $\lambda = -y$, and (B.3) are equivalent. From second-order information it can be gathered that, if it exists, there is in fact only one solution to (B.1), so we can safely solve (B.3) instead.

B.4 The unconstrained Problem vs LP

As noted above, the problem we obtain when introducing barrier functions is not the same problem as the original LP problem, but we hoped that it would be a good approximation for small values of μ. A more precise relation between the two can be obtained as follows:

We have

$$\mathcal{X}\mathcal{Z}e = \mu e \;\Rightarrow x^T z = \mu n.$$

If both x and z are feasible, we have from the relation $A^T y + z - c = 0$ that this is the same as

$$x^T(-A^T y + c) = \mu n$$

which can be rearranged as

$$-y^T(Ax) + c^T x = \mu n.$$

Now, since x is feasible we have $Ax = b$, and therefore

$$-y^T b + c^T x = \mu n,$$

or, rearranging a little again:

$$c^T x - b^T y = \mu n.$$

In other words, when x and z are feasible, μn is equal to the duality gap.

B.5 Extension to SDP

Generalisation of these results to SDP is possible, but as one might expect is quite complicated, even though the resulting equations are not all that different from the ones pertaining to LP. A full derivation of the existence and uniqueness of the SDP central path can be found in [15]. The generalised relations we are interested in are as follows:

For points on the central path we have:

$$\begin{aligned} \textstyle\sum_{i=1}^m y_i A_i \; + Z - C &= 0, \\ A_1 \bullet X - b_1 &= 0, \\ &\vdots \\ A_m \bullet X - b_m &= 0, \\ XZ - \mu I &= 0. \end{aligned}$$

For feasible points, the duality gap is given by:

$$X \bullet Z = \mu n.$$

Appendix C

Source Code

C.1 Listing of Scheme 2

Listed below is the source code for scheme 2 (with projection), from which scheme 1 and lfsdp can be recovered. The function "Mfun" takes as input a vector v and returns the product Mv, using the formula (4.10).

```
% scheme2/lfsdp - Primal-Dual SDP solver using the HKM (P = Z^(1/2))
% direction and Mehrotra's Predictor-Corrector approach.
%
% Written by Lennart Frimannslund, lennart@ii.uib.no, 2002.
%
% MODIFICATION SCHEME 2:
%     Solves the Schur Complement equation Mdy = rhs
%     using CG and a relative error tolerance of 1e-4 (pred) / 1e-8 (corr)
%     until CG becomes too expensive or fails
%     Projects dX at each iteration so that a full step results
%     in Ax-b = 0. This projection affects complementarity
%
% Solves the problem
%
% min trace(C*X)
% s.t.
% trace(A_i*X) = b_i, i=1...m, X >= 0,
%
% as well as its associated dual problem
%
% max b'y
% s.t.
% sum (i=1..m) y_i * A_i + Z = C, Z >= 0.
%
% Needs the m-file Mfun (implicit def of SCE matrix M) to be
% available
%
% Needs the following m-files from sdppack to be available
%
% svec    -  Conversion from matrix to vector
% smat    -  Conversion from vector to matrix
% sdbound -  Step length computer
% blkeig  -  Eigenvalue computer
%
% Needs the following variables to exist in the workspace:
%
```

```
% A - Matrix containing [svec(A1)'; ... ; svec(Am)']
% C - primal problem cost matrix
% b - dual problem cost vector
% blk - block structure vector
%
% X - initial iterate, must be positive definite
% Z - initial iterate, must be positive definite
% y - initial iterate
%
cput = cputime;

% some booleans for decision making, see scheme description above
solveinexactly = 1;
doneinexact =0;
pflag=0;
cflag=0;

% some constants used for calculating complexity
q = 0.5* sum(blk.*(blk+1));
sumnisquare = sum(blk.^2);
sumnicube   = sum(blk.^3);


% Compute chol(A*A'), we'll need it later for projecting dx
% onto the nullspace of A
[cholAAT,indef] = chol(A*A');
if (indef~=0),
   disp('Matrix A*A^T not positive definite, aborting...');
   return;
end;

m = length(y);
n = length(C(:,1));
M = zeros(m,m);
sigma = 0.1;

[cholX,indef] = chol(X);
if (indef~=0),
   disp('Initial X not positive definite, aborting...');
   return;
end;

[cholZ, indef] = chol(Z);
if (indef~=0),
   disp('Initial Z not positive definite, aborting...');
   return;
end;

pinfeas = norm(b - A*svec(X,blk)) / max(1,norm(b));
dinfeas = norm(svec(C-Z,blk)-A'*y,'fro') /max(1,norm(C,'fro'));
dgap    = sum(sum(X.*Z));

err = max(max(pinfeas,dinfeas),dgap);
tol = 1e-8;

mu = dgap / n;
iter = 0;
alpha = 0;
beta = 0;
tau = 0.9;

% Print info
```

```
disp(['iter    pstep        dstep      pinfeas     dinfeas' ...
      '       gap         val        sigma']);
disp(' ');
disp(sprintf('%3d %11.3e  %11.3e %11.3e %11.3e %11.3e %11.3e',...
      iter,alpha,beta,full(pinfeas),full(dinfeas),...
      full(dgap),0.5*(full(sum(sum(C.*X)))+full(b'*y))));


%Main loop
while (err >= tol),
   iter = iter +1;

   Rd = C - Z - smat(A'*y,blk);
   rp = b - A*svec(X,blk);

   if (solveinexactly == 0),
       % Construct Schur Complement
       for (j = 1:m),
           invZAjX = cholZ \ (cholZ' \ (smat(A(j,:),blk) * X));
           invEFAj = 0.5 *( invZAjX + invZAjX');
           svec_invEFAj = svec(invEFAj,blk);
           for (i = j:m),
               M(i,j) = A(i,:) * svec_invEFAj;

               if (i ~= j),
                   M(j,i) = M(i,j);
               end;

           end;
       end;

       [R,indef] = chol(M);
       if indef~=0,
           disp(['Schur Complement matrix not positive definite -' ...
                ' aborting...']);
           disp(['Cpu time spent: ' num2str(cputime - cput) ' seconds.']);
          return;
       end;
   end;


   %
   % PREDICTOR STEP
   %

   % Construct right hand side
   invZRdX =  cholZ \ (cholZ' \ (Rd * X));
   rhs = rp + A*(svec( 0.5*(invZRdX + invZRdX') + X,blk));

   if (solveinexactly == 0),
       % Solve M dy = rhs
       dy = R \ (R' \ rhs);
   else,
       [dy,pflag,relres,piter] = pcg('Mfun',rhs,1e-3,m,[],[],[],X,cholZ,A,blk);
       r = Mfun(dy,X,cholZ,A,blk) - rhs;
   end;


   % Back-substitute. Obtaining dX this way might not be numerically optimal
   % way, see AHO section 5 for discussion
   dZ = Rd - smat(A'*dy,blk);
```

```
invZdZX = cholZ \ ( cholZ' \ (dZ * X));
dX = - X - 0.5*(invZdZX + invZdZX');

% Symmetrize dX,dZ, according to AHO good idea because of rounding
dZ = 0.5*(dZ + dZ');
dX = 0.5*(dX + dX');

% If solving inexactly, project dx onto nullspace of A
if ( solveinexactly ==1),
    dX = dX - smat(A'*(cholAAT \ (cholAAT'\r)), ...
   blk);
end;

% Now compute steplengths
alpha = min(1,tau*sdbound(cholX,dX,blk));
beta  = min(1,tau*sdbound(cholZ,dZ,blk));

%Store predictor step
Xnew = X +  alpha * dX;
Znew = Z +  beta  * dZ;

% We choose choose centering parameter sigma dynamically
% based on heuristic presented in SDPT3 v2.1 user's guide

exponent = 1;

if (mu > 1e-6),
    if (min(alpha,beta) < 1/sqrt(3)),
        exponent = 1;
    else,
        exponent = max(1, 3 * min(alpha,beta)^2);
    end;
else,
    exponent = 1;
end;

sigma = min(1, power(sum(sum(Xnew.*Znew))/dgap,exponent));


%
% CORRECTOR STEP
%

tau = 0.9 + 0.09*min(alpha,beta);

% Construct corrector right hand side

dXdZinvZ = ((dX * dZ ) / cholZ) / cholZ';
rhscorr = rp + A*(svec( 0.5*(invZRdX + invZRdX') - sigma * mu * ...
inv(cholZ)*inv(cholZ') + X + 0.5*(dXdZinvZ + dXdZinvZ'), ...
blk));

if (solveinexactly == 0),
    % Solve M dy = rhs
    dy = R \ (R' \ rhscorr);
else,
    [dy,cflag,relres,citer] = pcg('Mfun',rhscorr,1e-9,m,[],[],[],X,cholZ,A,blk);
    r = Mfun(dy,X,cholZ,A,blk) - rhscorr;
end;


%Back-substitute as before
```

```
dZ = Rd - smat(A'*dy,blk);

invZdZX = cholZ \ ( cholZ' \ ( dZ * X));
dX = sigma * mu * inv(cholZ)*inv(cholZ') - X - 0.5*(invZdZX + invZdZX') - ...
                                              0.5*(dXdZinvZ + dXdZinvZ');

% Symmetrize dX,dZ, according to AHO good idea because of rounding
dZ = 0.5*(dZ + dZ');
dX = 0.5*(dX + dX');

% If solving inexactly, project dx onto nullspace of A
if ( solveinexactly ==1),
    dX = dX - smat(A'*(cholAAT \ (cholAAT'\r)), ...
   blk);
end;

% Now compute steplengths
alpha = min(1,tau*sdbound(cholX,dX,blk));
beta  = min(1,tau*sdbound(cholZ,dZ,blk));

%Update iterates for testing, we'll accept
%or reject them shortly
Xnew = X + alpha * dX;
Znew = Z + beta  * dZ;


[cholX,indef] = chol(Xnew);
if (indef~=0),
   disp('New X-iterate indefinite, terminating...');
   disp(['Cpu time spent: ' num2str(cputime - cput) ' seconds.']);
   return;
end;

[cholZ, indef] = chol(Znew);
if (indef~=0),
   disp('New Z-iterate indefinite, terminating...');
   disp(['Cpu time spent: ' num2str(cputime - cput) ' seconds.']);
   return;
end;


%Accept new point
X = Xnew;
Z = Znew;
y = y + beta * dy;

pinfeas = norm(b - A*svec(X,blk)) / max(1,norm(b));
dinfeas = norm(svec(C-Z,blk)-A'*y,'fro') /max(1,norm(C,'fro'));
dgap    = sum(sum(X.*Z));

err = max(max(pinfeas,dinfeas),dgap);
mu = dgap/n;
tau = 0.9 + 0.09*min(alpha,beta);

%Display info
disp(sprintf('%3d %11.3e  %11.3e %11.3e %11.3e %11.3e %11.3e     %5.3f',...
   iter,alpha,beta,full(pinfeas),full(dinfeas),...
   full(dgap),0.5*(full(sum(sum(C.*X)))+full(b'*y)),full(sigma)));

%decide if we're to abandon inexact computation
if (pflag ~= 0 | cflag ~= 0),
    solveinexactly = 0;
```

```
        %set these two zero to satisfy this if-test when done inexactly.
        pflag=0;
        cflag =0;
    end;

    % test of whether or not to quit based on operation count
    % Check if operations carried out to solve inexactly
    % are close to the cost of solving directly. If so, quit
    % Don't count cost of projecting as it is close
    % to the cost of SCE back-substitution which we dont count either
    if (solveinexactly == 1 & (piter+citer)*(3*sumnicube+4*m*q) >= ...
            0.85*(3*m*sumnicube+0.5*m^2*sumnisquare+(1/3)*m^3)),

        solveinexactly = 0;
        %set these two zero to satisfy if-test when done inexactly.
        pflag=0;
        cflag =0;
        disp('Operation count becoming too large for inexact solution...');
    end;

    if (solveinexactly ==0 & doneinexact ==0),
        disp('Solving SCE directly from now on');
        doneinexact =1;
    end;

end;
disp('Success, error reduced to value desired.');
disp(['Cpu time spent: ' num2str(cputime - cput) ' seconds.']);
```

Bibliography

[1] F. Alizadeh, J.-P. A. Haeberly and M. L. Overton: *Primal-Dual Interior-Point Methods for Semidefinite Programming: Convergence Rates, Stability and Numerical Results*, SIAM J. Optimization, 8 1998, p. 746-768.

[2] F. Alizadeh, J.-P. A. Haeberly, M.V. Nayakkankuppam and M. L. Overton: *SDPPACK v0.8 beta*, http://www.cs.nyu.edu/cs/faculty/overton/sdppack/sdppack.html

[3] F. Alizadeh, J.-P. A. Haeberly, M.V. Nayakkankuppam and M. L. Overton: *SDPPACK User's guide v0.8 beta*, NYU Computer Science Department Technical Report 734, March 1997.

[4] T.M. Apostol: *Calculus Volume II Second Edition*, John Wiley & Sons, Inc. 1969, ISBN 0-471-00008-6.

[5] V. Baryamureeba and T. Steihaug: *On the Convergence of an Inexact Primal-Dual Interior Point Method for Linear Programming*, Report no. 188, Department of Informatics, University of Bergen, March 2000. ISSN 0333-3590.

[6] B. Borchers: *SDPLIB 1.2, A Library of Semidefinite Programming Test Problems*, http://www.nmt.edu/∼sdplib/

[7] S.D. Conte and C. de Boor: *Elementary Numerical Analysis — An Algorithmic Approach*, McGraw-Hill Book Company 1981, ISBN 0-07-066228-2.

[8] T.H. Cormen, C.E. Leiserson and R.L. Rivest: *Introduction to Algorithms*, MIT Press 1997, ISBN 0-262-53091-0.

[9] R.S. Dembo, S.C. Eisensat and T. Steihaug: *Inexact Newton Methods*, SIAM J. Numerical Analysis, 19 1982, p. 400-408.

[10] J. Gondzio: *Multiple Centrality Corrections in a Primal-Dual Method for Linear Programming*, Computational Optimization and Applications, 6 1996, p. 137-156.

[11] L. Lovász: *On the Shannon Capacity of a Graph*, IEEE Transactions on information Theory, 25:1-7, 1979.

[12] S. Mehrotra: *On the Implementation of a Primal-Dual Interior Point Method*, SIAM J. Optimization, 2 1992, p. 575-601.

[13] R.D.C. Monteiro: *Primal-Dual Path-Following Algorithms for Semidefinite Programming*, SIAM J. Optimization 7 1997, p. 663-678.

[14] J. Nocedal and S.J. Wright: *Numerical Optimization*, Springer-Verlag 1999. ISBN 0-387-98793-2.

[15] M.J. Todd: *Semidefinite Optimization*, Acta Numerica 10 2001, p. 515-560.

[16] K.C. Toh and M. Kojima: *Solving some large scale semidefinite programs via the conjugate residual method*, SIAM J. Optimization vol 12, No 3 p. 669-691.

[17] K.C. Toh, M.J. Todd and R.H. Tütüncü: *On the Nesterov-Todd Direction in Semidefinite Programming*, SIAM J. Optimization vol. 8, No. 3 1998, p. 769-796.

[18] K.C. Toh, M.J. Todd, R.H. Tütüncü: *SDPT3 — a Matlab software package for semidefinite programming, version 2.1*, Optimization Methods and Software, 11 1999, p. 545-581.

[19] K.C. Toh, R.H. Tütüncü and M.J. Todd: *SDPT3 version 2.3 — a MATLAB software package for Semidefinite Programming*, http://www.math.nus.edu.sg/∼mattohkc/

[20] L.N. Trefethen and D. Bau III, *Numerical Linear Algebra*, SIAM 1997, ISBN 0-89871-361-7.

[21] R.J. Vanderbei: *Linear Programming: Foundations and Extensions*, Kluwer Academic Publishers 1997, ISBN 0-7923-8141-6.

[22] L. Vandenberghe and S. Boyd: *Semidefinite Programming*, SIAM Rev. 38 1996, p. 49-95.

[23] W. Wang and D.P. O'Leary: *Adaptive Use of Iterative Methods in Predictor-Corrector Interior Point Methods for Linear Programming*, University of Maryland CS Technical Report 3560, November 1995.