AN OBJECT ORIENTED PLATFORM FOR IMPLEMENTING INTERIOR-POINT ALGORITHMS

AN OPEN SOURCE OBJECT ORIENTED PLATFORM FOR RAPID DESIGN OF HIGH-PERFORMANCE PATH FOLLOWING INTERIOR-POINT METHODS

by

VOICU CHIŞ, M.Sc.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree
Master of Science

McMaster University
© Copyright by Voicu Chiş, 2008

MASTER OF SCIENCE (2008)
(Mathematics and Statistics)

McMaster University
Hamilton, Ontario

TITLE: An open source object oriented platform for rapid design of high-performance path following interior-point methods
AUTHOR: Voicu Chiş, M.Sc.
SUPERVISOR: Dr. Tamás Terlaky, Dr. Yuriy Zinchenko
NUMBER OF PAGES: ix,126

Acknowledgements

I would like to acknowledge the persons who have extensively contributed, directly or indirectly, to this thesis: my supervisors dr. Tamás Terlaky and dr. Yuriy Zinchenko, the AdvOL group, my colleagues in the Math Department and my extended family in Canada and Romania. I appreciate everything that they did for me.

Abstract

Interior point methods (IPMs) is a powerful tool in convex optimization. From the theoretical point of view, the convex set of feasible solutions is represented by a so-called barrier functional and the only information required by the algorithms is the evaluation of the barrier, its gradient and Hessian. As a result, IPM algorithms can be used for many types of convex problems and their theoretical performance depends on the properties of the barrier. In practice, performance depends on how the data structure is exploited at the linear algebra level. In this thesis, we make use of the object-oriented paradigm supported by C++ to create a platform where the aforementioned generality of IPM algorithms is valued and the possibility to exploit the data structure is available. We will illustrate the power of such an approach on optimization problems arrising in the field of Radiation Therapy, in particular Intensity Modulated Radiation Therapy.

Contents

I Interior Point Methods **1**

1 Problem formulation **1**

2 Main ingredients in Interior Point Algorithms **8**

2.1 The central path 8

2.2 Barriers on convex sets 12

2.3 Several algorithms for convex optimization problems 18

2.4 Barriers in conic optimization 23

2.5 Predictor step in conic optimization
Symmetry and primal-dual algorithms 28

2.6 A primal-dual predictor-corrector algorithm
for nonsymmetric cones 35

3 Semidefinite optimization in matrix variable **46**

4 Barriers for some well-structured sets **52**

4.1 The non-negative orthant: logarithmic barrier 52

4.2 The second-order cone: logarithmic barrier 52

4.3 The cone of semidefinite positive matrices:
logarithmic barrier 54

4.4 The p-cone 54

II The design of YAS **58**

5 Overview **58**

5.1 LA Layer 58

5.2 IPM layer 61

6 The Linear Algebra Layer **64**

6.1 Low-level routines 66

6.1.1 BLAS routines 70

6.1.2 LAPACK routines 77

6.2 YAS_K_block 84

6.2.1 Properties of YAS_K_Block 87

6.2.2 Methods of YAS_K_block 94

6.3 YAS_K_mb 104

7 Appendix A. YAS defined types **105**

III Optimization in IMRT **108**

8 Problem formulation **108**

9 A MATLAB prototype **117**

10 Benchmarking **118**

11 Conclusions **121**

Introduction

Interior point methods (IPMs) provide important tools to solve convex optimization problems. In [11], Nesterov and Nemirovskii have shown that theoretically efficient IPM algorithms can be developed in a very general setting. For a convex optimization problem such as

$$\inf_{x \in D} \langle c, x \rangle$$

where $\langle \cdot, \cdot \rangle$ is an inner product and D is a closed convex set in a finite-dimensional real vector space, the only information about the convex set D needed by IPM algorithms is a so-called barrier functional on the interior of D. Theoretical efficiency of the algorithms depends on the properties of the barrier which result from the structure of D. For example, when D is the intersection of an affine space with a special type of a convex cone, a so-called self-scaled cone, a barrier with particularly useful properties is available for D. As a result, so-called primal-dual algorithms can be developed in this setting. Their observed practical behaviour is better than the behaviour of pure primal algorithms that only use a barrier with no other appealing properties.

The properties of the barrier determine the set of algorithms that can be used to solve the problem. In this thesis, we set the basis of a new optimization solver with a modular design that follows this general pattern, and, in particular, utilizes the power of primal-dual barriers in the nonsymmetric setting.

A number of very powerful IPM-based packages already exist: SDPT3 [19], SeDuMi [16], CSDP [2], DSDP [1], SDPA [6], to name just a few. How-

ever, most of these solvers are very specialized to a particular problem type and data structure, e.g., CSDP is designed to solve only the so-called positive semidefinite optimization problems and performs exceptionally well primarily on problem instances with dense data structures. These packages exhibit a somewhat rigid design: modifying these solvers to accommodate convex optimization problems of other types by extending the implemented algorithms becomes extremely difficult, if at all possible. In addition, many optimization problems have an easily identifiable block-density pattern, while few optimization packages are capable of taking full advantage of this structure at the linear algebra level. The main goal of our work is to propose a modular framework that can be used to create an optimization engine capable to overcome the above mentioned shortcomings of existing IPM-based solvers. In particular, we focus on implementing primal-dual path-following algorithms in the non-symmetric cone setting, with the underlying hypothesis being that such optimization problems offer end-users more adequate modelling capabilities, while the primal-dual algorithms allow very efficient computational strategies similar to those already exhibited in practice by the path-following algorithms in the symmetric cone setting.

Through this design we aim to

- allow easy development of IPM algorithms for optimization problems beyond standard symmetric cone optimization problems;
- allow easy switch between different linear algebra packages that supply routines required by IPM algorithms; in particular, one can use platform-tuned linear algebra packages;

- allow the user to exploit the type/structure of the matrices involved in the linear algebra and so
  - speed up additions/multiplications;
  - speed up factorizations/inversions, for example for $k-$update matrices or block-structured matrices;
  - save memory when storing sparse, symmetric, diagonal etc. matrices;
  - obtain better numerical accuracy through customized linear algebra techniques such as factorization of block matrices;
- allow easy switch between different numerical precisions of data;
- allow accomodation of techniques motivated by numerical accuracy such as storing the iterates of the algorithms in a scaled space (see YAS_k_EVS in Section 5.2);
- allow the optimization problem to be given in either primal or dual form;
- allow modelling of optimization problems in their natural formulation (see Semidefinite optimization in matrix variable in Chapter 3);
- provide an appropriate framework for the development of an open-source library of derived classes that are tuned for optimization problems with specific data structures.

Using the object-oriented paradigm supported in C++, the algorithms implemented in this framework make use of a special class named "barrier".

Through the methods of the barrier class, the algorithms can evaluate the barrier, compute its gradient or Hessian. In particular, this framework allows the implementation of primal-dual algorithms for optimization over nonsymmetric cones, see [8], where the barrier is log-homogenous but not self-scaled. Geometric optimization and optimization over p-cones are situations where log-homogenous barriers are the natural barriers. Interior point methods in this setting have drawn recent attention with hopes that their performance for large problems might be better then first-order methods applied to the original optimization problem, or classical IPMs applied to an equivalent optimization problem over symmetric cones. In particular, our interest in optimization problems over p-cones stems from the fact that problems of this kind arise in optimal radiation therapy treatment planning. The large-scale and nearly dense nature of these optimization problems make them inaccessible to any of the state-of-the-art solvers available today. However, there is a clear need to advance our capacities in solving such problems as dictated by this important application.

In what follows we will go into a bit more details of our design formalism. The YAS design is split in two levels:

- Basic Linear Algebra layer;
- Interior Point Methods layer.

The goal of the Basic Linear Algebra layer is to provide a transparent access to hardware-tuned linear algebra routines. This layer consists of:

- the $low - level\ routines$, further grouped into:

  - BLAS routines include matrix-matrix multiplications and additions for different types of matrices;
  - LAPACK routines are used for inverting, factorizing, solving linear systems with different types of matrices.

- the YAS_K_block class allows the storage of one or more blocks of the same type and dimension and provides methods to do linear algebra operations. By a block of a certain type we refer to a matrix with an exploitable structure, such as a matrix that is sparse, symmetric, diagonal etc.

- the YAS_K_mb class allows the storage of one or more matrices of blocks of the same type and dimension and provides methods to do linear algebra operations. By a matrix of blocks we refer to a matrix that can be splited into blocks.

The Interior Point Methods layer consists of:

- YAS_k_EVS is a class that is a container of a k-tuple of elements of a vector space. The class provides methods such as adding, scaling or computing the norm of objects of this type.

- $YAS_barrier$ is a class which is of obvious importance for interior point methods. Through the methods of $YAS_barrier$, one can evaluate the barrier, its gradient or its Hessian.

- YAS_LO is a class used to replicate a linear operator.

- YAS_norm_eq is a class that is derived from YAS_LO allowing the user to form a compressed and expanded versions of normal equations and to solve the latter.

The thesis is divided in three parts: *Interior Point Methods*, *The Design* of YAS and *Optimization in* $IMRT$.

In the first part, we collect some theoretical results about interior point methods concluding with an algorithm for nonsymmetric cones due to Nesterov [10]. We start with the formulation of the problems that we want to tackle with YAS. Next, we present the theory of interior point methods as applied to such problems, starting from the general form

$$\inf_{x \in D} \langle c, x \rangle,$$

where D a closed convex set and c, x vectors, and building up to conic optimization.

The second part of the thesis is about the design of YAS. Up to this point, the base layer of the solver, the linear algebra layer, is completely established. The top layer, the interior point methods layer, is roughly described emphasizing the resulting advantages. This second part of the thesis has the style of a software manual, where the classes and the routines of the solver are well documented. We start with an *Overview* where we say what can be done with the solver, the resulting advantages, and not how to do it. We then proceed to describing the Linear Algebra Layer. We follow the same pattern starting with more general ideas, slightly more specific than the ones in Overview, but still not enough for the actual use of the

software. We are avoiding technicalities at the beginning. Then we proceed with detailed documentation for each routine/class.

Finally, the third part of the thesis concerns with optimization problems arising from IMRT. We present a model and then isolate the resulting optimization problem and conclude with remarks about its computational tractability. We then describe a prototype code in MATLAB that is implementing Nesterov's algorithm for nonsymmetric cones, and use this to tackle the optimization problem presented before. We compare our result with SeDuMi and SDPT3. We want to point out that real situations like this ask for enough flexibility to model the data with different types, at least dense and sparse, and so a solver with a modular design is preferable. While working on the prototype code, we have also found an unexpected situation where the flexibility that we provide in the IPM layer might prove useful. However, at this point we can not confirm this with strong numerical results since the software is in incipient form.

Part I

Interior Point Methods

1 Problem formulation

Let X be a finite dimensional real vector space endowed with an inner product $\langle \cdot, \cdot \rangle : X \times X \to \mathbb{R}$. Let $c \in X$ and let $D \subseteq X$ be a closed convex set, i.e., a closed set such that if $x, y \in D$ and $\alpha \in [0, 1]$ then $\alpha x + (1 - \alpha) y \in D$. We focus on solving convex optimization problems of the following type

$$\inf_{x \in D} \langle c, x \rangle \tag{1}$$

In particular, we are interested in problems of type (1) that are amenable to the so-called interior point methods. For the later to hold, one has to be able to equip $\mathring{D}$, the interior of D, with a certain barrier functional $f : \mathring{D} \to \mathbb{R}$. See Section 2.2 for a detailed discussion of the barrier functionals.

Theoretically, a barrier exists for any closed convex set, see [11]. But in practice we need to be able to compute this barrier efficiently. Fortunately, for many well-structured closed convex sets such computable barriers are already known. Even if such a formulation is not readily available, in many cases a linear transformation of the decision variable $x \in X$ suffices to put an optimization problem into the desirable equivalent form. To this extend,

in addition to (1) we introduce four more types of well-structured convex optimization problems.

In what follows, X and Y are finite dimensional real vector spaces each endowed with an inner product $\langle \cdot, \cdot \rangle$, with the underlying space being clear from the context. Let $c \in X$, $b \in Y$ and $\mathcal{A} : X \to Y$ be a linear operator. Denote with $\mathcal{A}^* : Y \to X$ its adjoint, i.e., the unique linear operator such that $\langle \mathcal{A}x, y \rangle = \langle x, \mathcal{A}^* y \rangle$, for all $x \in X$, $y \in Y$. Taking the adjoint of a linear operator is an involution, i.e., $(\mathcal{A}^*)^* = \mathcal{A}$. Denote $K \subset X$ a closed convex cone, i.e., a closed convex set such that if $x \in K$ and $t \in [0, \infty)$ then $tx \in K$. Recall that the dual cone $K^* \subset X$ is the closed convex cone defined by $K^* = \{s \in X : \langle s, x \rangle \geq 0 \ \forall x \in K\}$; taking the dual of the closed convex cone is an involution, i.e., $(K^*)^* = K$.

Convex conic optimization problem. Consider the problem

$$\begin{cases} \inf\limits_{x} \ \langle c, x \rangle \\ \text{s.t. } \mathcal{A}x = b \\ x \in K \end{cases} \tag{2}$$

and its *dual* problem

$$\begin{cases} \sup\limits_{y,s} \langle b, y \rangle \\ \text{s.t. } \mathcal{A}^* y + s = c \\ s \in K^* \end{cases} \tag{3}$$

The points $x \in K \cap \{x : \mathcal{A}x = b\}$ are called primal feasible and $(y, s) \in Y \times K^* \cap \{(y, s) : \mathcal{A}^* y + s = c\}$ are called dual feasible. Feasible points that are in the interior of the cone are called strictly feasible.

Problems (2) and (3) are strongly connected. If x and (y, s) are feasible points then weak duality holds, i.e.,

$$\langle c, x \rangle - \langle b, y \rangle = \langle x, s \rangle \geq 0$$

In particular, if one denotes with val the optimal value of (2) and val^* the optimal value of (3) it follows that $val \geq val^*$. Under fairly mild assumptions val and val^* are equal, a property referred to as *strong duality*. For example, if (2) and (3) have strictly feasible points then strong duality holds. If one has strong duality, in many cases it is possible to recover an optimal solution to (3) from a solution to (2) and vice-versa. Thus, for most practical applications (2) and (3) are thought of as equivalent problems.

If the cone K or K^* is equipped with a particularly nice barrier, the so-called log-homogenous barrier, the connection between (2) and (3) goes well beyond the weak or strong duality as above. Primal-dual methods are a class of algorithms that exploit this connection. They are believed to be the most efficient in practice. One such algorithm is discussed in Section 2.6.

Examples:

1. Linear optimization (LO). Let $X = \mathbb{R}^n$, $Y = \mathbb{R}^m$ and the inner products on $\mathbb{R}^n$ and $\mathbb{R}^m$ be both given by $\langle x, y \rangle = x^T y$. Let $K = \mathbb{R}^n_+$ be the nonnegative orthant in $\mathbb{R}^n$. One can show $K^* = \mathbb{R}^n_+$. Let $c \in \mathbb{R}^n, b \in \mathbb{R}^m$, $\mathcal{A} : \mathbb{R}^n \to \mathbb{R}^m$ be a linear operator with its adjoint $\mathcal{A}^* : \mathbb{R}^m \to \mathbb{R}^n$. Note that

if $\mathcal{A}$ is represented by a $m \times n$ matrix A so that $\mathcal{A}(x) = Ax$ then $\mathcal{A}^* = A^T$. Then problem (2) and (3) are referred as linear optimization problems in the primal and, respectively, dual form.

2. Second-Order Cone Optimization (SOCO). Let $X = \mathbb{R}^n$, $Y = \mathbb{R}^m$ and the inner products on $\mathbb{R}^n$ and $\mathbb{R}^m$ be both given by $\langle x, y \rangle = x^T y$. Let $K = \{x \in \mathbb{R}^n : x_n \geq \sqrt{x_1^2 + ... + x_{n-1}^2}\}$ be the second-order cone (also called Lorentz cone or ice-cream cone) in $\mathbb{R}^n$. One can show $K^* = K$. Let $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $\mathcal{A} : \mathbb{R}^n \to \mathbb{R}^m$ be a linear operator with its adjoint $\mathcal{A}^* : \mathbb{R}^m \to \mathbb{R}^n$. Then problem (2) and (3) are referred as second-order conic optimization problems in the primal and, respectively, dual form.

3. Semidefinite Optimization (SDO). Let $X = S^{n\times n}$ the vector space of real symmetric $n \times n$ matrices. Let the inner product on $S^{n\times n}$ be $\langle x, y \rangle = tr(xy)$, $\forall x, y \in S^{n\times n}$. Let $Y = \mathbb{R}^m$ with the inner product $\langle x, y \rangle = x^T y$. Let $K = S_+^{n\times n}$, the cone of positive-semidefinite symmetric matrices. One can show $K^* = K$. Let $c \in S^{n\times n}, b \in \mathbb{R}^m$. Let $\mathcal{A} : S^{n\times n} \to \mathbb{R}^m$ be a linear operator. One can show that there exists $A_i \in S^{n\times n}$, $i = \overline{1, m}$ such that $\mathcal{A}(X) = (\langle A_i, X \rangle, ..., \langle A_m, X \rangle)$. In this case, its adjoint $\mathcal{A}^* : \mathbb{R}^m \to S^{n\times n}$ is given by $\mathcal{A}^*(y) = \sum_{i=1}^m y_i A_i$, $y \in \mathbb{R}^m$. Then problem (2) and (3) are referred as semidefinite optimization problems in the primal and, respectively, dual form.

In many applications the natural formulation has the form of the dual problem. For example, in control theory the sufficient stability criteria is expressed in terms of a linear matrix inequality such as $F^T P F - P \preceq I$. Also, in linear optimization, a constraint of the form $Ax \geq b$ is obviously a dual constraint because it is equivalent with $Ax - s = b$, $s \in \mathbb{R}_+^n$. Although

one can rephrase it as $A(x_1 - x_2) - s = b$, $x_1, x_2, s \in \mathbb{R}^n_+$, this procedure doubles the dimension of the problem. Most solver such as SeDuMi, MOSEK, SDPT3 implementing interior point algorithms require the input to be in the form of (2). We allow input in both primal and dual formats.

The conic optimization modelling framework has several limitations. First, conic constraints do not arise naturally in applications. Convex optimization problems can be put in the form of a conic optimization problem using the, so-called, lifting procedure. However, during this transformation the complexity of the new problem to be solved is increased. Second, primal-dual algorithms require certain information, to be made more precise at a latter point, regarding *both* K and K^*. In many cases, this information is available for K but not for K^*. Therefore, we are interested in solving problems of the following types:

Convex optimization problem in *dual form.* Let $D \subset X$ be a closed convex set, not necessarily a cone. Consider the problem

$$\begin{cases} \sup\limits_{y} \ \langle b, y \rangle \\ \text{s.t. } \ c - \mathcal{A}^* y \in D \end{cases} \tag{4}$$

Convex optimization problem in *primal form.* Let $D \subset X$ be a closed convex set. Consider the problem

$$\begin{cases} \inf_x \ \langle c, x \rangle \\ \text{s.t. } \mathcal{A}x = b \\ x \in D \end{cases} \tag{5}$$

Remark 1.1 *Note that problems (1), (5) and (4) are equivalent. The equivalence follows from the fact that interior point algorithms can be developed for (1) if a barrier on $\mathring{D}$ is available. A barrier f on $\mathring{D}$ naturally gives a barrier on $\{y : c - \mathcal{A}^* y \in \mathring{D}\}$, namely $y \mapsto f(c - \mathcal{A}^* y)$. Therefore (5) can be seen as a particular case for (1). Consider now a convex optimization problem in primal form (4). Assume N is a basis for the null space of $\mathcal{A}$ and x_0 is such that $\mathcal{A}x_0 = b$. It follows that $\{x : \mathcal{A}x = b\} \cap D = \{Ny + x_0\} \cap D$ and from here the equivalence:*

$$\begin{cases} \inf_x \ \langle c, x \rangle \\ \text{s.t. } \mathcal{A}x = b \\ x \in D \end{cases} \Leftrightarrow \begin{cases} \inf_y \langle b, Ny + x_0 \rangle \\ \text{s.t. } x_0 + Ny \in D \end{cases} \Longleftrightarrow \begin{cases} \sup_y - \langle N^* b, y \rangle \\ \text{s.t. } x_0 + Ny \in D \end{cases}$$

Therefore, (1), (5) and (4) are equivalent.

In the rest of this thesis we use "convex optimization problem" to refer problems formulated as (1), "primal convex conic optimization problem" for (2), "dual convex conic optimization problem" for (3), "convex optimization problem in dual form" for (5), "convex optimization problem in primal form" for (4).

We emphasize that we are interested in solving problems in their natural formulation. For this purpose, we choose to distinguish between (1), (5) and (4) although they are equivalent. Also, the use of abstract linear

operators mapping vector spaces into vector spaces allows us to model the linear operator $x \mapsto yx + xy$ (x and y are matrices of appropriate size) instead of replacing it with a linear operator that acts on the vector obtained by stacking the lines of x.

2 Main ingredients in Interior Point Algorithms

The goal of this part is to present several interior point algorithms. We start with the ideas of the interior point methods that go back to the 1950s. The key concept is the central path, a curve that leads to the optimal set and whose definition depends on a certain functional. After this we present Nesterov's and Nemirovskii's [11] choice for the functional, the so-called barriers, followed by several algorithms relying on the properties required for the functional. Then we consider the barriers in the context of conic optimization and quote results from [11] and [13] showing how the duality theory is enriched. We finish with the presentation of a primal-dual predictor-corrector algorithm due to Nesterov [10] that is motivated by the previous results.

2.1 The central path

In 1968, Fiacco and McCormick authored a book called "Nonlinear programming: sequential unconstrained minimization techniques" [5]. They introduce it to the reader as a book that provides "a unified body of theory on methods of transforming a constrained minimization problem into a sequence of unconstrained minimizations of an appropriate auxiliary function" and also "some historical perspective for the basic approach with an effort toward synthesis". Following their remarks, it is safe to say that such ideas go back at least to the 1950s. The problem they consider is a general nonlinear optimization problem and the two main techniques presented are

called interior point methods and exterior point methods. In the current literature of nonlinear optimization, these methods are also referred to as barrier methods and, respectively, penalty methods.

We will illustrate the ideas of general interior point methods in our setting.

Consider a convex optimization problem

$$\begin{cases} \inf \ \langle c, x \rangle \\ \text{s.t. } x \in D \end{cases}$$

We will construct a family of optimization problems by adding to the objective a term which approaches infinity when the boundary is approached. In this way, the minimum of the new objective is pushed in the interior of the feasible region. We control this term with a parameter that allows us to increase or decrease the term's role in the objective.

Let $f : \overset{\circ}{D} \to R$ be such that $\lim_{x \to \partial D} f(x) = +\infty$. Consider the family of optimization problems parametrized by $\mu > 0$:

$$\inf_{x \in D} \langle c, x \rangle + \mu f(x) \tag{6}$$

To preserve the convexity of the problem and guarantee uniqueness of minimizers we will assume f is strict convex. Assume that for every $\mu > 0$ there exists $x_\mu \in \overset{\circ}{D}$ minimizer of $\langle c, x \rangle + \mu f(x)$. One can prove that:

Theorem 2.1 *1.* $0 < \mu < \sigma \Leftrightarrow \langle c, x_\mu \rangle \leq \langle c, x_\sigma \rangle$

2. Let $(\mu_k)_{k \geq 1}$ *be a strictly decreasing sequence of positive numbers such that*

$\mu_k \to 0$. Denote with x_k the unique solution of $\inf\limits_{x\in D} \langle c, x\rangle + \mu_k f(x)$. Since D is bounded, we can assume that (x_k) converges to some $x^* \in D$. Then

$$\lim_{k\to\infty} \langle c, x_k\rangle = \langle c, x^*\rangle = \inf_{x\in D} \langle c, x\rangle$$

Proof.

1. By definitions:

$$\langle c, x_\mu\rangle + \mu f(x_\mu) \leq \langle c, x_\sigma\rangle + \mu f(x_\sigma)$$

and

$$\langle c, x_\sigma\rangle + \sigma f(x_\sigma) \leq \langle c, x_\mu\rangle + \sigma f(x_\mu)$$

Multiplying the first inequality with $\frac{\sigma}{\mu}$, then adding and rearranging the terms we obtain:

$$(1 - \tfrac{\sigma}{\mu}) \langle c, x_\mu\rangle \leq (1 - \tfrac{\sigma}{\mu}) \langle c, x_\sigma\rangle$$

And so $\mu < \sigma \Leftrightarrow \langle c, x_\mu\rangle \leq \langle c, x_\sigma\rangle$.

2. The first part of the statement follows by continuity. For the second part, denote $L := \{x : \langle c, x\rangle = \langle c, x^*\rangle\}$. Assume $\langle c, x^*\rangle > \inf\limits_{x\in D} \langle c, x\rangle$. Then $L \cap \mathring{D}$ is nonempty. Therefore the optimization problem

$$\begin{cases} \inf\limits_{x} f(x) \\ \text{s.t. } x \in L \cap \mathring{D} \end{cases}$$

has a unique solution $\bar{x} \in L \cap \mathring{D}$. It satisfies $g(\bar{x}) \perp L$ i.e. $g(\bar{x}) = \mu c$ for some $\mu \in \mathbb{R}$. But this implies that $\bar{x}$ is also the solution of $\min \mu \langle c, x\rangle + f(x)$ and so $\mu > 0$.

Because $\langle c, \bar{x}\rangle = \langle c, x^*\rangle \leq \langle c, x_k\rangle$ for all k, using the first part of this theorem, we have $\mu_k \geq \mu$ contradicting $\mu_k \to 0$. ■

The previous theorem shows that the curve $\{x_\mu : \mu > 0\}$ is leading to a solution of the initial problem.

Having such a curve in the interior of the domain, one way to follow it is the following. Start with $\mu_0 > 0$ and with a point x_0 that is a good approximation for x_{μ_0}. Decrease μ_0 to μ_1. If μ_1 is not much smaller than μ_0, we expect that x_0 will be farther from x_{μ_1} than it is from x_{μ_0} but will still be a relatively good approximation for it. Now do one step of some minimization algorithm of your choice to get x_1 which will be a better approximation for x_{μ_1}. To be able to repeat this process, one should make sure that x_1 is as close to x_{μ_1} as was x_0 to x_{μ_0}.

Note that the description above allows freedom to choose f and freedom to choose a minimization algorithm.

In the optimization literature, Newton's method is a minimization algorithm that is attractive due to its local quadratic convergence rate. However, in the general analysis, in order to check if the convergence rate is quadratic we need to be able to measure the distance between our current point and the minimizer. Of course this is not practical because the minimizer is not known. A fundamental result in nonlinear analysis that is eliminating this inconvenient is due to Kantorovich, see [17]. It contains a set of assumptions under which Newton's method performs good. Nesterov and Nemirovskii [11] proposed their own type of functionals on which Newton's method performs well. They also have all the properties needed to generate a curve as before that leads to a solution of the problem.

In the rest of this thesis by a barrier we mean the functional introduced by Nesterov and Nemirovskii. If f is a barrier, then for each $\mu > 0$ we denote

with x_μ the unique solution of $\min_x \mu \langle c, x\rangle + f(x)$ and call the resulting curve, $\{x_\mu : \mu > 0\}$, the central path. Note that this definition for x_μ is slightly different than before. In this case, the curve leads to a solution of the problem when $\mu \to \infty$ and not to 0. Also note that the definition depends on c.

We continue with the presentation of the barriers and focus on the analysis of Newton's method as it applies to them.

2.2 Barriers on convex sets

In this section we present the machinery needed for interior point algorithms for convex optimization problems. Given a closed bounded convex set $D \subset X$ with nonempty interior $\mathring{D}$, our goal is to explain what is a barrier having $\mathring{D}$ as its domain and state important properties that are used extensively by the algorithms. Following Renegar [13] and Nesterov and Nemirovskii [11] we present them using local norms. Therefore, we start by explaining what do we mean by local norms. We continue by introducing self-concordant functionals and stating results regarding the analysis of Newton's method as it applies to them. After this we introduce barrier functionals. We closely follow Renegar [13].

In what follows, when we refer to f we also assume the following properties. First, $f \in C^2(D)$, i.e., f is twice continuously differentiable on D. Denote with $g(x) \in X$ the gradient of g in x and with $H(x) : X \to X$ the Hessian of f in x which is a linear operator. Second, we assume that $H(x)$ is self-adjoint and positive definite for any $x \in D$, i.e., $\langle H(x)y, z\rangle = \langle y, H(x)z\rangle$ for any $y, z \in X$ and $\langle H(x)y, y\rangle > 0 \; \forall y \in X\backslash\{0\}$. In particular, this implies

that f is strict convex.

Such a functional f gives rise to a family of inner products on X. For any $x \in D$, define the inner product $\langle \cdot, \cdot \rangle_x$ by:

$$\langle y, z \rangle_x = \langle y, H(x)z \rangle, \ \forall y, z \in X$$

As usual, the inner product $\langle \cdot, \cdot \rangle_x$ induces a norm on X denoted with $||\cdot||_x$ and given by:

$$||y||_x = \langle y, y \rangle_x^{1/2}, \ \forall y \in X$$

It is known that the definitions of the gradient and Hessian depend on the norm. We denote with $g_x(y)$ and $H_x(y)$ the gradient in y with respect to the norm induced by x and, respectively, the Hessian in y with respect to the norm induced by x. One can show that

$$g_x(y) = H(x)^{-1} g(y) \text{ and } H_x(y) = H(x)^{-1} H(y)$$

Also denote with $B_x(x, 1)$ the ball of radius 1 centered in x where the distance is measured in the norm induced by x.

We use the definition in Renegar [13] to introduce self-concordant functionals instead of the apparently more technical definition used in Nesterov and Nemirovskii [11]. In [13] it is proved that they are equivalent.

Definition 2.1 *We call f self-concordant if, in addition, the following conditions are satisfied:*

1. $B_x(x, 1) \subset D, \ \forall x \in D$
2. $1 - ||y - x||_x \leq \frac{||v||_y}{||v||_x} \leq \frac{1}{1 - ||y - x||_x}$, for all $v \neq 0$

Two results of obvious interest are

Theorem 2.2 *If D_i are two closed convex sets and $f_i : \mathring{D}_i \to \mathbb{R}$, $i = 1, 2$, are two self-concordant functionals then $f_1 + f_2 : \mathring{D}_1 \cap \mathring{D}_2 \to \mathbb{R}$ is a self-concordant functional.*

Theorem 2.3 *Assume $\mathcal{A} : \mathbb{R}^n \to \mathbb{R}^m$ is an injective linear operator, $b \in \mathbb{R}^m$, $D \subset \mathbb{R}^m$ is a closed convex set and $f : \mathring{D} \to \mathbb{R}$ is self-concordant. Then $x \longmapsto f(\mathcal{A}x - b)$ is self-concordant if the set $\{x : \mathcal{A}x - b \in D\}$ is not empty.*

Theorem 2.4 *If $f : \mathring{D} \to \mathbb{R}$ is self-concordant and $c \in X$ then $x \longmapsto f(x) + \langle c, x \rangle$ is self-concordant (on $\mathring{D}$).*

Self-concordant functionals have appealing properties if we apply Newton's method to find their minimum. Recall that an iteration of Newton's method applied to a functional f consists of moving into the minimizer of the quadratic approximation considered in the current point.

Assuming our current point is x, the quadratic approximation for f in x is given by

$$q_x(y) = f(x) + \langle g(x), y - x \rangle + \tfrac{1}{2} \langle y - x, H(x)(y - x) \rangle$$

Since $H(x)$ is positive definite it follows that q_x is strict convex. Its minimizer y^* is the critical point of q_x. Therefore, y^* is computed from

$$g(x) + H(x)(y^* - x) = 0$$

as $y^* = x - H(x)^{-1}g(x)$. We use the notation $n(x) := -H(x)^{-1}g(x)$ and refer to it as the Newton step in x. Remark that $g_x(x) = -n(x)$. Therefore the Newton direction is exactly the steepest descent direction if

the gradient is computed w.r.t. the norm induced by x. We now present the results regarding Newton's method for self-concordant functionals.

The first result refers to the local approximation of the functional by its quadratic approximation.

Theorem 2.5 *Assume f is self-concordant, $x \in D$ and $y \in B_x(x, 1)$. Then*

$$|f(y) - q_x(y)| \leq \frac{||y-x||_x^3}{3(1-||y-x||_x)}$$

The second result refers to the progress done by Newton's method.

Theorem 2.6 *Assume f is self-concordant and $x \in D$. If z minimizes f and $z \in B_x(x, 1)$ then*

$$||x_+ - z||_x \leq \frac{||x-z||_x^2}{1-||x-z||_x}$$

where x_+ is the next iteration of the Newton algorithms i.e. $x_+ := x - H(x)^{-1}g(x)$.

Corollary 2.7 *If $||x - z||_z < \frac{1}{4}$ then*

$$||x_+ - z||_z < 4\,||x - z||_z^2$$

Theorem 2.8 *Assume f is self-concordant. If $||n(x)||_x < 1$ then*

$$||n(x_+)||_{x_+} \leq \left(\frac{||n(x)||_x}{1-||n(x)||_x}\right)^2$$

In the general case, the convergence results for Newton's method require x to be sufficiently close to the minimizer. The only way to decide if we are sufficiently close to the minimizer is to know the minimizer. In the case of self-concordant functionals we can decide if we are close enough to the minimizer by looking at the size of the Newton step.

Theorem 2.9 *Assume f is self-concordant. If $||n(x)||_x < \frac{1}{4}$ for some $x \in D$ then f has a minimizer z and*

$$||z - x_+||_x \leq \frac{3||n(x)||_x^2}{(1-||n(x)||_x)^3}$$
$$||z - x||_x \leq ||n(x)||_x + \frac{3||n(x)||_x^2}{(1-||n(x)||_x)^3}$$

Now we introduce barrier functionals.

Definition 2.2 *A self-concordant functional f is called a barrier if the quantity $\nu_f := \sup_x ||g_x(x)||_x^2$ is finite. We refer to ν_f as the complexity value of f.*

As in the case of self-concordant functionals, the following results are of interest

Theorem 2.10 *If D_i are two closed convex sets and $f_i : \mathring{D}_i \to \mathbb{R}$, $i = 1, 2$, are two barrier then $f := f_1 + f_2 : \mathring{D}_1 \cap \mathring{D}_2 \to \mathbb{R}$ is a barrier and $\nu_f \leq \nu_{f_1} + \nu_{f_2}$.*

Theorem 2.11 *Assume $\mathcal{A} : \mathbb{R}^n \to \mathbb{R}^m$ is an injective linear operator, $b \in \mathbb{R}^m$, $D \subset \mathbb{R}^m$ closed convex set, and $f : \mathring{D} \to \mathbb{R}$ is a barrier. Then $x \longmapsto f(\mathcal{A}x - b)$ is a barrier if the set $\{x : \mathcal{A}x - b \in D\}$ is not empty with complexity value at most ν_f.*

Remark 2.1 *Adding a linear functional to a barrier results in a self-concordant functional (see Theorem 2.4) but not necessarily a barrier. As an example consider $x \longmapsto x - \ln x$.*

Practical experience and complexity analysis of interior point methods show that we are interested in barriers with small complexity value. Nesterov and Nemirovskii [11] show that $\nu_f \geq 1$ for any barrier f. They also show that each open convex set containing no line is the domain of a barrier functional. If the set is in $\mathbb{R}^n$, they also prove that there exists a universal constant C and a barrier on the set with complexity value less then $C \cdot n$. Unfortunately, the proof is not constructive and therefore it has only theoretical value because, as we will see, interior point algorithms require computable gradients and Hessians for the barriers.

It is worth mentioning that the objective that define the central path, see (6), are not barriers but self-concordant.

We now state one important property of barriers:

Theorem 2.12 *Assume f is a barrier and $x, y \in D$. Then* $\langle g(x), y - x \rangle < \nu_f$

This result can be used to prove that the central path is leading to a solution of the initial problem. We see that x_μ, the unique solution of $\inf_x \mu \langle c, x \rangle + f(x)$, is given by $g(x_\mu) = -\mu c$. Therefore, for any $y \in D$, we have $\langle c, x_\mu \rangle - \langle c, y \rangle = \langle c, x_\mu - y \rangle = -\frac{1}{\mu} \langle g(x_\mu), x_\mu - y \rangle = \frac{1}{\mu} \langle g(x_\mu), y - x_\mu \rangle < \frac{1}{\mu} \nu_f$. It follows that $\langle c, x_\mu \rangle < \frac{1}{\mu} \nu_f + \langle c, y \rangle$ and from here:

Theorem 2.13 $\langle c, x_\mu \rangle \leq \frac{1}{\mu} \nu_f + \inf_{x \in D} \langle c, x \rangle$.

One can also prove an analog relation for the case when the point is not on the central path:

Theorem 2.14 *For any $x \in D$ we have*

$$\langle c, x \rangle < \frac{1}{\mu}\nu_f(1 + ||x - x_\mu||_{x_\mu}) + \inf_{x \in D} \langle c, x \rangle.$$

2.3 Several algorithms for convex optimization problems

We now have all the tools we need to state several algorithms for convex optimization problems

$$\begin{cases} \min\limits_{x} \ \langle c, x \rangle \\ \text{s.t. } \ x \in D \end{cases}$$

We assume a barrier f on $\mathring{D}$ with complexity value ν_f is available. Denote with $f_\mu(x) := \mu \langle c, x \rangle + f(x)$, $\mu > 0$, and with $n_\mu(x)$ the Newton step for f_μ in x.

The Short-Step Barrier Method. For a given $\epsilon > 0$, the algorithm will return x^* such that $\langle c, x^* \rangle < \epsilon + \inf\limits_{x \in D} \langle c, x \rangle$.

Input: $\epsilon > 0$ (the desired accuracy)

$\mu_1 > 0$ and x_1 such that $||n_{\mu_1}(x_1)||_{x_1} \leq \alpha$ (a point close enough to the central path)

Let $\alpha := \frac{1}{9}$, $\beta := 1 + \frac{1}{8\sqrt{\nu_f}}$

Repeat: For $k \geq 1$:

Let $\mu_{k+1} := \mu_k \beta$ (we increase μ_k with factor β)

Let $x_{k+1} := x_k + n_{\mu_{k+1}}(x_k)$

$k := k + 1$

Until: $\mu_k > \frac{6\nu_f}{5\epsilon}$

The fact that at each iteration $||n_{\mu_k}(x_k)||_{x_k} \leq \frac{1}{9}$ implies $||x_k - x_{\mu_k}||_{x_{\mu_k}} \leq \frac{1}{5}$ (Theorem 2.9). Therefore, we can get a bound on how far we are from

the optimal value (Theorem 2.14):

$$\langle c, x_k \rangle < \frac{1}{\mu_k}\nu_f(1+\tfrac{1}{5}) + \inf_{y\in D} \langle c, y\rangle$$

This relation gives us the stopping condition. Also note that if we use $1+\frac{1}{8\sqrt{\nu_f}}$ as the multiplying factor, we need $O(\sqrt{\nu_f}\log(\mu_*/\mu_1))$ iterations to get from the initial value μ_1 to μ_*. Therefore, for given $\epsilon > 0$ we need $O(\sqrt{\nu_f}\log(\frac{\nu_f}{\epsilon\mu_1}))$ iterations to produce a point x^* such that $\langle c, x^*\rangle \leq \epsilon + \inf_{y\in D} \langle c, y\rangle$.

Note that one could replace $\alpha = \frac{1}{9}$ and $\beta = 1 + \frac{1}{8\sqrt{\nu_f}}$ with any $\alpha > 0$, $\beta > 1$ such that if we define $\gamma := \alpha\beta + (\beta-1)\sqrt{\nu_f}$ then $\gamma < 1$ and $(\frac{\gamma}{1-\gamma})^2 \leq \alpha$. The restrictions on α and β guarantee that at for each k we have $||n_{\mu_k}(x_k)||_{x_k} \leq \alpha$, so the algorithm stays on track. The only thing that is changing in this situation is the stopping condition.

The Long-Step Barrier Algorithm. This algorithm is build on the idea that one should use for μ_{k+1} a much larger value then the safer $\mu_k\left(1+\frac{1}{8\sqrt{\nu_f}}\right)$ that is used by the barrier method. To get close to $x_{\mu_{k+1}}$ the algorithm is not taking Newton steps. Instead is doing exact line search along the Newton directions until it gets close enough to $x_{\mu_{k+1}}$. As usual, the distance between x and $x_{\mu_{k+1}}$ is represented by the quantity $||n_{\mu_{k+1}}(x)||_x$. If $||n_{\mu_{k+1}}(x)||_x < \frac{1}{4}$ the algorithm considers that x is close enough to $x_{\eta_{k+1}}$. The value $\frac{1}{4}$ is the biggest value that allows one to apply Theorem 2.9.

Input: $\epsilon > 0$ (the desired accuracy)

$\mu_1 > 0$ and x_1 such that $||n_{\mu_1}(x_1)|| < \frac{1}{4}$ (a point close enough to the central path)

$r > 1$ (the factor to be used for increasing μ_k)

Repeat: For $k \geq 1$:

Let $\mu_{k+1} := r\mu_k$ Denote with $y_1 := x_k$

$i := 1$

Repeat: compute $t_i \geq 0$ such that $f_{\mu_{k+1}}(y_i + t_i n_{\mu_{k+1}}(y_i)) := \min_{t \geq 0} f_{\mu_{k+1}}(y_i + t n_{\mu_{k+1}}(y_i))$

$y_{i+1} := y_i + t_i n_{\mu_{k+1}}(y_i)$

$i := i + 1$

Until: $||n_{\mu_{k+1}}(y_i)|| < \frac{1}{4}$

$x_{k+1} := y_i$

$k := k + 1$

Until: $\mu_{k+1} \geq 4\frac{\nu_f}{\epsilon}$

The fact that at each iteration $||n_{\mu_k}(x_k)||_{x_k} \leq \frac{1}{4}$ implies, from Theorem 2.9, $||x_k - x_{\mu_k}||_{x_k} < \frac{3}{4}$ and further, from the definition of self-concordancy (Definition 2.1), $||x_k - z(\mu_k)||_{z(\mu_k)} \leq 3$. Therefore, we can get a bound on how far x_k is from the optimal value (Theorem 2.14):

$$\langle c, x_k \rangle < 4\frac{1}{\mu_k}\nu_f + \inf_{y \in D} \langle c, y \rangle$$

From here we get the stopping criteria, i.e. $\mu_k \geq 4\frac{\nu_f}{\epsilon}$.

The analysis of the complexity for the algorithm depends on the number of line searches needed to get from x_k having the property $||n_{\mu_k}(x_k)|| < \frac{1}{4}$ to x_{k+1}, the point such that $||n_{\mu_{k+1}}(x_{k+1})|| < \frac{1}{4}$. Obviously, it depends on r, the factor by which μ_k is increased. We skip the details, but mention the main ideas. First, one can find an upper bound for the difference $f_{\mu_{k+1}}(x_k) - f_{\mu_{k+1}}(x_{\mu_{k+1}})$ in terms of r. Then, one can show that there exists a constant $\tau > 0$, the same for all line searches, such that every line search decreases the value of $f_{\mu_{k+1}}$ with at least τ, i.e. $f_{\mu_{k+1}}(y_i) - f_{\mu_{k+1}}(y_{i+1}) \geq \tau$ for any i.

If follows that it takes $O(r\nu_f \log_r(\frac{\nu_f}{\epsilon\mu_1}))$ line searches for the algorithm to finish. Although the theoretical complexity of the long step algorithm is worse by a factor of $\sqrt{\nu_f}$ then the one of the short-step method, people agree that it is somehow more efficient in practice.

A Predictor-Corrector Algorithm. The idea of the predictor-correct algorithm is to follow the central path by using a direction that approximates a tangent to the central path. After moving in this direction a fixed fraction of the distance to the boundary of the feasible region, the algorithm returns to the central path and repeats.

Remember that the points on the central path $(x_\mu)_{\mu>0}$ are given by $g(x_\mu) + c\mu = 0$. Differentiation with respect to μ gives $H(x_\mu)x'_\mu = -c$. Therefore $x'_\mu = -H(x_\mu)^{-1}c$ and so the tangent to the central path in x_μ is the vector $-H(x_\mu)^{-1}c$. If the current iterate x is not on the central path we call the vector $c_x := -H(x)^{-1}c$ the predictor direction in x. Intuitively, if x is close enough to a point x_μ on the central path, the vector $-H(x)^{-1}c$ will be a good approximation for $-H(x_\mu)^{-1}c$. As a side remark, the predictor direction in x is also the direction of steepest descent for the functional $\langle c, \cdot\rangle$ computed in x with respect to the norm induced by x.

If x is a feasible point, presumably not close to the central path, the corrector steps are iterations that move towards x_μ the unique point on the central path such that $\langle c, x\rangle = \langle c, x_\mu\rangle$. Denote with $L(x) = \{y : \langle c, y\rangle = \langle c, x\rangle\}$. The corrector steps minimize the restriction of f to the affine space $L(x)$. Each step is doing an exact line search along the Newton direction, denoted with $n_{|L(x)}$, of $f_{|L(x)}$. We stop the correction steps at the point x^* that satisfies $||n_{|L(x)}(x^*)||_{x^*} \leq \frac{1}{14}$. The choice of $\frac{1}{14}$ comes from the

implication $\left|\left|n_{|L(x)}(x^*)\right|\right|_{x^*} \leq \frac{1}{14} \Longrightarrow ||n(x^*)||_{x^*} \leq \frac{1}{9}$ that makes the analysis similar with the analysis for the short-step barrier method.

Input: $\mu > 0$ (desired accuracy; μ is rather big as convergence is achieved when $\mu \to \infty$)

x_1 such that $n_{|L(x_1)}(x_1) \leq \frac{1}{14}$ (a point close enough to the central path)

$\sigma \in (0,1)$ (the predictor step length, i.e., the fraction from the distance to the boundary)

Start:

$k := 0$

Repeat:

$k := k+1$

compute $s_k := \sup\{s : s > 0, x_k - sc_{x_k} \in D\}$.

take the predictor step: $x_k^P := x_k - \sigma s_k c_{x_k}$

$y_1 := x_k^P$; $i := 1$;

denote $L(y_1) = \{y : \langle c, y\rangle = \langle c, y_1\rangle\}$

Repeat:

compute $t_i \geq 0$ such that

$$f(y_i + t_i n_{|L(y_1)}(y_i)) := \min_{t \geq 0} f(y_i + t n_{|L(y_1)}(y_i))$$

$y_{i+1} := y_i + t_i n_{|L(y_1)}(y_i)$

Until: $\left|\left|n_{L(y_1)}(y_{i+1})\right|\right| \leq \frac{1}{14}$

$x_{k+1} := y_i$

compute μ_{aprox} from $\mu_{aprox} c + g(x_{k+1}) = 0$;

Until: $\mu_{aprox} > \mu$

2.4 Barriers in conic optimization

We will now look at the conic convex optimization problem i.e. the primal-dual pair introduced in Chapter 1 as (2) and (3):

$$\begin{cases} \min \ \langle c, x\rangle \\ \text{s.t. } \mathcal{A}x = b \quad (2) \\ x \in K \end{cases} \quad \text{and} \quad \begin{cases} \max \ \langle b, y\rangle \\ \text{s.t. } \mathcal{A}^*y + s = c \quad (3) \\ s \in K^* \end{cases}$$

Recall that X, Y are finite dimensional spaces, $K \subset X$ is a closed convex cone, $K^* \subset X$ is the dual of K, $c \in X$, $b \in Y$ and $\mathcal{A} : X \to Y$ is a linear operator with adjoint $\mathcal{A}^* : Y \to X$. A standard assumption is that $\mathcal{A}$ is surjective and so $\mathcal{A}^*$ is injective. Therefore y is uniquely determined by s from $\mathcal{A}^*y + s = c$. We also assume that both problems are strictly feasible.

Assume f is a barrier on $\mathring{K}$ with complexity ν_f. We will state results showing how f is building more connections between the primal and the dual problem. We will be dealing with three types of barriers. From general to particular they are: barriers (as defined for convex sets), log-homogenous barriers and self-scaled barriers. Unless otherwise specified f will be a barrier.

Remark 2.2 *Saying that the cone is self-scaled is the same as saying that there exists a self-scaled barrier on it. The formulation is emphasizing that the cone has enough structure to support a self-scaled barrier.*

We can now define log-homogenous barriers and present their properties. Before introducing self-scaled barriers we need to develop more theory.

Definition 2.3 *A barrier $f : \mathring{K} \to R$ is called log-homogenous if for all $x \in \mathring{K}$, $t > 0$ we have:*

$$f(tx) = f(x) - \nu_f \ln t$$

Theorem 2.15 *f is log-homogenous iff for all $x \in K$ and $t > 0$*

$$g(tx) = \tfrac{1}{t} g(x)$$

Theorem 2.16 *If f is log-homogenous then the following hold:*

$$1.\ H(tx) = \tfrac{1}{t^2} H(x)$$

$$2.\ ||g_x(x)||_x = \sqrt{\nu_f}$$

$$3.\ H(x)x = -g(x) \iff -x = g_x(x)$$

$$4.\ \langle -g(x), x \rangle = \nu_f \iff \langle H(x)x, x \rangle = \nu_f$$

$$5. \langle g(x), (H(x))^{-1} g(x) \rangle = \nu_f$$

$$6.\ f(u) \le f(x) + \langle g(x), u - x \rangle + \omega(r)$$

$$H(u) \le \tfrac{1}{(1-r)^2} H(x)$$

$$\text{where } r = ||u - x||_x < 1 \text{ and } \omega(r) = -r - \ln(1 - r)$$

We will now illustrate the idea that a barrier on $\mathring{K}$ generates a barrier on $\mathring{K}^*$ and show more connections between the primal and the dual problems that result from this.

Definition 2.4 *Define the conjugate functional of f as:*

$$f^*(s) := -\inf_{x \in D} [\langle x, s \rangle + f(x)]$$

Theorem 2.17 *If f is a barrier then f^* is a barrier on $\mathring{K}^*$ and $\nu_{f^*} \le (4\nu_f + 1)^2$. Moreover, if f is log-homogenous so is f^* and $\nu_f = \nu_{f^*}$.*

We use the analogous notations ν_{f^*} for the complexity of f^*, as well as g^* for its gradient and H^* for its Hessian.

Theorem 2.18 *If f is a barrier on $\mathring{K}$ then $-g$ is a bijection between $\mathring{K}$ and $\mathring{K}^*$. On the dual side, $-g^*$ is a bijection between $\mathring{K}^*$ and $\mathring{K}$. Moreover, if $x \in \mathring{K}$ and $s \in \mathring{K}^*$ are such that $s = -g(x)$ then*

$$-g^*(s) = x \text{ and } H(x)^{-1} = H^*(s)$$

Remark 2.3 *We now have a different way of looking at the properties of log-homogenous barriers from Theorem 2.16. $H(x)$ maps X to X and we have seen $-g$ is a bijection between $\mathring{K}$ and $\mathring{K}^*$. In Theorem 2.16, Property 3 says that x is mapped by $H(x)$ and $-g$ in the same point. Property 4 says that the duality gap at x and its image through $-g$ is the same for all x and is equal to ν_f. Same for Property 5.*

As every point $x \in \mathring{K}$ is creating an inner product through f, every point $s \in K^*$ is creating an inner product through f^*. Because any $s \in K^*$ is the image of some $x \in K$ through $-g$, we can say that any point $x \in K$ is creating an inner product through f^*.

Definition 2.5 *The inner product created by $x \in K$ through f^* is denoted with $\langle \cdot, \cdot \rangle_x^*$ and is defined as $\langle y, z \rangle_x^* = \langle y, H^*(-g(x))z \rangle_x^*$*

Remark 2.4 *It follows from Theorem 2.18 that $\langle y, z \rangle_x^* = \langle y, H(x)^{-1} z \rangle_x^*$*

The barrier f defines a central path $\{x_\mu : \mu > 0\}$, called the primal central path, where x_μ is the unique solution of

$$\begin{cases} \min_x \mu \langle c, x \rangle + f(x) \\ \text{s.t. } \mathcal{A}x = b \end{cases}$$

The barrier f^* defines a central path $\{(y_\mu, s_\mu) : \mu > 0\}$, called the dual central path, where (y_μ, s_μ) is the unique solution of

$$\begin{cases} \max_{y,s} \mu \langle b, y \rangle - f^*(s) \\ \text{s.t. } \mathcal{A}^* y + s = c \end{cases}$$

Remark 2.5 *1. Both primal and dual central paths exist because* (2) *and* (3) *are assumed to be strictly feasible.*
2. Sometimes we call dual central path the curve $\{s_\mu : \mu > 0\}$*. We can do this because* $\mathcal{A}^*$ *is injective and so* s_μ *uniquely determines* y_μ*.*

We will now see that, if f is log-homogenous, when following the primal central path one generates the dual central path as a by-product and vice-versa.

The optimality conditions that give x_μ are

$$\begin{cases} \mu c + g(x_\mu) \perp \{x : \mathcal{A}x = b\} \\ \mathcal{A}x_\mu = b \end{cases}$$

Therefore an unique y_μ exists such that

$$\begin{cases} \mu c + g(x_\mu) = \mathcal{A}^* y_\mu \\ \mathcal{A}x_\mu = b \end{cases}$$

Rearranging the terms we obtain

$$\begin{cases} c - \mathcal{A}^*(\frac{1}{\mu} y_\mu) = -\frac{1}{\mu} g(x_\mu) \\ \mathcal{A}x_\mu = b \end{cases}$$

Denoting $s_\mu := c - \mathcal{A}^*(\frac{1}{\mu}y_\mu)$ and reconsidering our notation for y_μ as $y_\mu := \frac{1}{\mu}y_\mu$ we get

$$\begin{cases} s_\mu = -\frac{1}{\mu}g(x_\mu) \\ \mathcal{A}x_\mu = b \\ \mathcal{A}^*(y_\mu) + s_\mu = c \end{cases}$$

Under the assumption that f is log-homogenous, it turns out that the pair (y_μ, s_μ) is the unique solution for the problem

$$\begin{cases} \max\limits_{y,s} \mu \langle b, y \rangle - f^*(s) \\ \text{s.t. } \mathcal{A}^*y + s = c \end{cases}$$

because the gradient of the objective at (y_μ, s_μ) is orthogonal on $\{(y,s) : \mathcal{A}^*y + s = c\}$:

$$\langle \mu b, -g^*(s_\mu) \rangle = \left\langle \mu \mathcal{A}x_\mu, -g^*(\tfrac{1}{\mu}(\mu s_\mu)) \right\rangle = \langle \mu \mathcal{A}x_\mu, -\mu g^*(\mu s_\mu) \rangle =$$
$$\langle \mu \mathcal{A}x_\mu, -\mu g^*(-g(x_\mu)) \rangle = \mu \langle \mathcal{A}x_\mu, x_\mu \rangle$$

Theorem 2.19 *If f is log-homogenous then its gradient is mapping the primal central path to the dual central path:*

$$s_\mu = -\frac{1}{\mu}g(x_\mu)$$

and conversely, the gradient of f^* is mapping the dual central path to the primal central path:

$$x_\mu = -\frac{1}{\mu}g^*(s_\mu)$$

Remark 2.6 *One can show that if f doesn't have the log-homogenuity property the path $\bar{s}_\mu$ in the dual space obtained by $\bar{s}_\mu := -\frac{1}{\mu}g(x_\mu)$ is still approaching a dual optimum when $\mu \to \infty$ i.e. $\langle x_\mu, \bar{s}_\mu \rangle \overset{\mu\to\infty}{\to} 0$.*

One can also show:

Theorem 2.20 *If f is a log-homogenous barrier than the following hold:*

1. *for any $x \in \mathring{K}$, $s \in \mathring{K}^*$: $f(x) + f^*(s) \geq -\nu_f - \nu_f \ln \frac{\langle s,x \rangle}{\nu_f}$*
2. *for any $\mu > 0$: $f(x_\mu) + f^*(s_\mu) = -\nu_f - \nu_f \ln \frac{1}{\mu}$.*

2.5 Predictor step in conic optimization

Symmetry and primal-dual algorithms

In Section 2.3 we have presented a predictor-corrector algorithm for the convex optimization problem (1). If the current point is on the central path the predictor direction is the tangent to the central path at that point. If it is off the central path the predictor direction is an approximation to the tangent to the central path. Let's consider the primal central path and look at the tangent to the primal central path. Recall that the system giving the primal central path is:

$$\begin{cases} s_\mu = -\frac{1}{\mu} g(x_\mu) \\ \mathcal{A} x_\mu = b \\ \mathcal{A}^*(y_\mu) + s_\mu = c \end{cases}$$

Taking the derivative with respect to μ we get:

$$\begin{cases} s'_\mu = \frac{1}{\mu^2} g(x_\mu) - \frac{1}{\mu} H(x_\mu) x'_\mu \\ \mathcal{A} x'_\mu = 0 \\ \mathcal{A}^*(y'_\mu) + s'_\mu = 0 \end{cases}$$

Using $-\frac{1}{\mu} g(x_\mu) = s_\mu$ (Theorem 2.19) and also $H(tx) = \frac{1}{t^2} H(x)$ (Theorem 2.16) and denoting the tangent directions s'_μ, x'_μ with Δs_μ and, respectively, Δx_μ we have:

$$\begin{cases} \Delta s_\mu + H(\sqrt{\mu} x_\mu)\Delta x_\mu = -\frac{1}{\mu} s_\mu \\ \mathcal{A}\Delta x_\mu^{'} = 0 \\ \mathcal{A}^*(\Delta y_\mu^{'}) + \Delta s_\mu^{'} = 0 \end{cases}$$

If we consider the dual central path and derive the tangent directions we obtain:

$$\begin{cases} \Delta x_\mu + H^*(\sqrt{\mu} s_\mu)\Delta s_\mu = -\frac{1}{\mu} x_\mu \\ \mathcal{A}\Delta x_\mu^{'} = 0 \\ \mathcal{A}^*(\Delta y_\mu^{'}) + \Delta s_\mu^{'} = 0 \end{cases}$$

Theorem 2.21 *The two systems are equivalent and therefore have the same solution* $(\Delta x_\mu, \Delta s_\mu, \Delta y_\mu)$.

Proof. Apply $H(\sqrt{\mu} x_\mu)$ to the first equation in the second system. All we need to check is that

$$H(\sqrt{\mu} x_\mu)(-\tfrac{1}{\mu} x_\mu) = -\tfrac{1}{\mu} s_\mu \text{ and } H(\sqrt{\mu} x_\mu)H^*(\sqrt{\mu} s_\mu)\Delta s_\mu = \Delta s_\mu.$$

We have:

$H(\sqrt{\mu} x_\mu)(-\frac{1}{\mu} x_\mu) = -\frac{1}{\mu} s_\mu \Longleftrightarrow H(\sqrt{\mu} x_\mu)(x_\mu) = s_\mu$

$\Longleftrightarrow \frac{1}{\mu} H(x_\mu)(x_\mu) = s_\mu \Leftrightarrow -\frac{1}{\mu} g(x_\mu) = s_\mu$

the last relation being true (Theorem 2.19).

Now rewrite the operator $H(\sqrt{\mu} x_\mu)H^*(\sqrt{\mu} s_\mu)$ as $H(x_\mu)H^*(\mu s_\mu)$. Because $\mu s_\mu = -g(x_\mu)$ it follows from Theorem 2.18 that $H(x_\mu)H^*(\mu s_\mu)$ is the identity. Therefore $H(\sqrt{\mu} x_\mu)H^*(\sqrt{\mu} s_\mu)\Delta s_\mu = \Delta s_\mu$ is true. ■

In Section 2.4 we saw that in conic programing with log-homogenous barriers when we set up the primal central path we obtain the dual central path and vice-versa. This result is analogous. It says that when we set up

the equations for the tangent to the primal central path we also obtain the tangent to the dual central path and vice-versa. Both are representative for what is referred in the literature as symmetry.

If the pair (x, s) is not on the central path, the first idea would be to keep the same systems to compute the directions. The system that is set-up from the primal perspective is

$$\begin{cases} \Delta s + H(\sqrt{\mu}x)\Delta x = -\frac{1}{\mu}s \\ \mathcal{A}\Delta x = 0 \\ \mathcal{A}^*(\Delta y) + \Delta s = 0 \end{cases} \tag{7}$$

while from the dual perspective we obtain

$$\begin{cases} \Delta x + H^*(\sqrt{\mu}s)\Delta s = -\frac{1}{\mu}x \\ \mathcal{A}\Delta x = 0 \\ \mathcal{A}^*(\Delta y) + \Delta s = 0 \end{cases} \tag{8}$$

For x and s off the central path, the μ used above has no meaning. One way to deal with this is to replace μ with a quantity that depends only on x and s. On the central path we have $\mu = \frac{\nu_f}{\langle x,s \rangle}$. Therefore we could replace μ with $\frac{\nu_f}{\langle x,s \rangle}$.

A more insightful remark is that when we are off the central path systems (7) and (8) don't give the same solutions. In other words, the symmetry is lost. When we are on the central path the symmetry follows from two facts:

$H(\sqrt{\mu}x)x = s$ and $H^*(\sqrt{\mu}s)^{-1} = H(\sqrt{\mu}x)$ (see the proof of Theorem 2.21).

To keep the symmetry one should replace (7) with:

$$\begin{cases} \Delta s + H(w)\Delta x = -s \\ \mathcal{A}\Delta x = 0 \\ \mathcal{A}^*(\Delta y) + \Delta s = 0 \end{cases} \quad \text{where } w \text{ is such that } H(w)x = s \tag{9}$$

and (8) with

$$\begin{cases} \Delta x + H^*(w^*)\Delta s = -x \\ \mathcal{A}\Delta x = 0 \\ \mathcal{A}^*(\Delta y) + \Delta s = 0 \end{cases} \quad \text{where } w^* \text{ is such that } H^*(w^*)s = x \tag{10}$$

Theorem 2.22 *If $H^*(w^*) = H(w)^{-1}$ then system (9) and system (10) are equivalent.*

Corollary 2.23 *1. Consider system (9) and construct (10) with $w^* = -g(w)$. Then (9) and (10) are equivalent.*

2. Consider system (10) and construct (9)) with $w = -g^*(w^*)$. Then (9) and (10) are equivalent.

Remark 2.7 *1. When we are on the central path, system* (9) *becomes*

$$\begin{cases} \Delta s + H(\sqrt{\mu}x)\Delta x = -s \\ \mathcal{A}\Delta x = 0 \\ \mathcal{A}^*(\Delta y) + \Delta s = 0 \end{cases} \tag{11}$$

which, strictly speaking, is not (7). The solution of (11) is the scaled solution of (9) by a factor of μ. Analogously, (10) becomes

$$\begin{cases} \Delta x + H^*(\sqrt{\mu}s)\Delta s = -x \\ \mathcal{A}\Delta x = 0 \\ \mathcal{A}^*(\Delta y) + \Delta s = 0 \end{cases} \tag{12}$$

and the solution of (10) is the scaled solution of (12) by a factor of μ.

2. Intuitively, the solution of (9) is an approximation to a tangent direction because when we are on the central path w becomes $\sqrt{\mu}x$. So the closest we are to the central path the more w approaches $\sqrt{\mu}x$. The same thing is valid for (10). Predictor-correct algorithms work with two neighborhoods of the central path. Roughly speaking, we need the pair to be in a smaller neighborhood in order to obtain reasonable approximation to a tangent direction. For the correction process we can afford to be in a larger neighborhood and still do it successfully.

3. The choice to find an approximation to the tangent by preserving the symmetry, makes this a primal-dual algorithm. It follows the primal and the dual central path at the same time and the directions to move are decided by considering both the primal and dual problem.

Definition 2.6 *The primal-dual central path is the curve* $\{(x_\mu, s_\mu) : \mu > 0\}$.

When setting up a system like (9) or (10) we must have w or w^*.

Definition 2.7 *For fixed* $x \in \mathring{K}$ *and* $s \in \mathring{K}^*$ *a point* w *such that* $H(w)x = s$ *is called a scaling point for the ordered pair* (x, s).

One can prove that scaling points exist for any pair in the most general case:

Theorem 2.24 *If* f *is a barrier, there exists at least one scaling point for any pair* $(x, s) \in \mathring{K} \times \mathring{K}^*$.

How to determine scaling points is a serious issue. In what follows we briefly present self-scaled barriers. Several properties make them attractive, but in this context the uniqueness of scaling points for a pair and the explicit formula for it are important. After self-scaled barriers, we present an algorithm by Nesterov [10] working with a log-homogenous barrier and where the correction process is not only finding a primal-dual pair in the small neighborhood but also a scaling point.

Definition 2.8 *A log-homogenous barrier* f *on* K *is called self-scaled if the following hold:*

1. For any $x \in \mathring{K}$ *we have* $K_x^* = K$ *where* K_x^* *is the dual of* K *w.r.t.* $\langle \cdot, \cdot \rangle_x$, *i.e.,* $K_x^* := \{z : \langle z, y \rangle_x \geq 0 \ \forall y \in K\}$

2. for any $x \in K$ *there exists a constant* C_x *such that* $f_x^* = f + C_x$ *where* f_x^* *is the conjugate of* f *with respect to* $\langle \cdot, \cdot \rangle_x$, *(see also Definition 2.4)*
$f_x^*(s) = -\inf_{y \in \mathring{K}} \langle y, s \rangle_x + f(y)$

Remark 2.8 *1. The gradient g_x and Hessian H_x for f are exactly the gradient and Hessian for f_x^*.*
2. One can show that C_x needs to be $C_x = -\nu_f - 2f(x)$.

We now state several other results concerning self-scaled barriers.

Theorem 2.25 *If f is a self-scaled barrier the following hold:*
1. $H(x)$ is a bijection between K and K for any $x \in \mathring{K}$.
2. for any ordered pair $x, s \in \mathring{K}$ there exists an unique scaling point $w \in \mathring{K}$, i.e., s.t. $H(w)x = s$.
3. if w is the scaling point for the pair x, s then $-g(w)$ is the scaling point for s, x, i.e., $H(-g(w))s = x$.
4. if $x, w \in \mathring{K}$ then $f(H_z(w)x) = f(x) + 2(f(w) - f(z))$ and $g(H_z(w)) = H_z(w)^{-1}g(x)$.

Güler [7] was the one to notice that self-scaled cones first introduced by Nesterov and Todd (see [8] and [9]) are the same as symmetric cones. Symmetric cones are Cartesian products of five basic symmetric cones: the cone of positive definite matrices, the second-order cone, the cone of positive definite Hermitian matrices, the cone of positive semidefinite Hermitian quaternion matrices and a 27-dimensional exceptional cone. In Section 4, we give the explicit formula for the scaling point for all self-scaled cones that we present.

2.6 A primal-dual predictor-corrector algorithm for nonsymmetric cones

In this part we present a primal-dual predictor-corrector algorithm for log-homogenous barriers from Nesterov [10]. We start with defining a functional that measures how far a pair $(x, s) \in K \times K^*$ is from the central path.

We define the following functional to measure how far away is the pair (x, s) from the primal-dual central path $\{(x_\mu, s_\mu) : \mu > 0\}$:

$$\Omega(x, s) := f(x) + f^*(s) + \nu_f \ln \frac{\langle x,s \rangle}{\nu_f} + \nu_f$$

Theorem 2.20 says that $\Omega(x, s) \geq 0$ for any pair (x, s) and is zero only for pairs on the primal-dual central path.

Remark 2.9 *Results showing neccesary and sufficient conditions for a pair (x, s) to be on the primal-dual central path create tools to measure the distance between a pair and the path. The distance functional Ω introduced above resulted from Theorem 2.20. For self-scaled barriers, we know that for any two points (x, s) there exists a unique w such that $H(w)x = s$. On the other hand, for any given $\mu > 0$ we have $H(\sqrt{\mu} x_\mu) x_\mu = s_\mu$. A measure of the distance of a pair (x, s) from the point (x_μ, s_μ) would be the quantity $\left|\left|w - \sqrt{\mu} x_\mu\right|\right|_w$. The latest measure is called "local" because it measures the distance **to** a point on the path. Ω is called "global" because it captures how far away is the point **from** the path.*

Another quantity associated with a pair $(x, s) \in \mathring{K} \times \mathring{K}^*$ is denoted with $\mu(x, s)$ and is called the penalty value. For a given pair (x, s) there exists an unique pair (x_μ, s_μ) on the primal-dual central path having the same

duality gap. We want $\mu(x,s)$ to be equal to μ. Because $\mu = \frac{\nu_f}{\langle x_\mu, s_\mu \rangle}$ we define $\mu(x,s) = \frac{\nu_f}{\langle x,s \rangle}$.

For barriers that are not self-scaled the existence of a scaling point for a pair $(x,s) \in \mathring{K} \times \mathring{K}^*$ is stated in Theorem 2.24. However how to determine a scaling point is not clear. One main ingredient in Nesterov [10] refers exactly to this issue. It gives a new interpretation to Newton's method applied to the problem

$$\begin{cases} \min_x f_\mu(x) := \mu \langle c, x \rangle + f(x) \\ \text{s.t. } \mathcal{A}x = b \end{cases}$$

Fix $\mu > 0$ and $u \in \mathring{K}$. The Newton step Δu in u is the solution of

$$\mu c + g(u) + H(u)\Delta u = \mathcal{A}^* y,\ \mathcal{A}\Delta u = 0$$

Rearranging and using $-g(u) = H(u)u$ (Theorem 2.16) we get

$$c - \frac{1}{\mu}\mathcal{A}^* y = \frac{1}{\mu} H(u)(u - \Delta u),\ \mathcal{A}\Delta u = 0$$

Denoting $x(\mu,u) := u - \Delta u$ and $s(\mu,u) := c - \frac{1}{\mu}\mathcal{A}^* y$ and using $\frac{1}{t^2}H(x) = H(tx)$ (Theorem 2.16) the above equality says $x(\mu,u) = H(\sqrt{\mu}u)s(\mu,u)$.

In other words, Newton's method gave a pair $(x(\mu,u), s(\mu,u))$ and a scaling point $\sqrt{\mu}u$ for it.

In Section 2.4 we have often seen that the size of the Newton step is an indicator of how far away we are from the central path. Intuitively, the closest u is to the minimizer x_μ the better are the properties of the pair $(x(\mu,u), s(\mu,u))$. All the results in the theorem bellow are already proved when $u = x_\mu$:

Theorem 2.26 *Assume $||\Delta u||_u \leq \beta < 1$. Then the pair $(x(\mu, u), s(\mu, u))$ is strictly feasible and the following relations hold:*

1. $s(\mu, u) = H(\sqrt{\mu}u)x(\mu, u)$
2. $||w||^2_{x(\mu,u)} \leq \frac{\mu}{(1-\beta)^2} ||w||^2_{\sqrt{\mu}u}$, $||w||^2_{s(\mu,u)} \leq \frac{\mu}{(1-\beta)^2}(||w||^*_{\sqrt{\mu}u})^2$ for any $w \in \mathring{K}$
3. $\Omega(x(\mu, u), s(\mu, u)) \leq 2(-\beta - \ln(1 - \beta)) + \beta^2$
4. $\left|\left|-H(\sqrt{\mu}u)g^*(s(\mu, u)) - (-g(x(\mu, u)))\right|\right|^*_{\sqrt{\mu}u} \leq \frac{2\beta^2}{1-\beta}\sqrt{\mu}$
5. $\mu e^{-\frac{2\beta}{\sqrt{\nu_f}}} \leq \frac{\mu}{(1+\frac{\beta}{\sqrt{\nu_f}})^2} \leq \mu(x(\mu, u), s(\mu, u)) \leq \frac{\mu}{(1-\frac{\beta}{\sqrt{\nu_f}})^2} \leq \mu e^{\frac{2\beta}{\sqrt{\nu_f}-\beta}}$

Remark 2.10 *1. Relation 1 is just repeating the fact that $\sqrt{\mu}u$ is a scaling point for the pair $(x(\mu, u), s(\mu, u))$.*

2. Relation 2 is about the difference between measuring with the norm induced by $x(\mu, u)$ or by $\sqrt{\mu}u$ and, respectively, the norm induced by $s(\mu, u)$ or by $\sqrt{\mu}u$.

3. Relation 3 is saying that the point is well centered.

4. We need an upper bound for the quantity in Relation 4 in order to upper bound the growth of the proximity measure along the predictor step that will be defined with the use of $\sqrt{\mu}u$, $x(\mu, u)$ and $s(\mu, u)$. In the case of self-scaled barriers, any strictly feasible pair (x, s) has an unique scaling point w which is also the scaling point for $(-g^*(s), -g(x))$ the pair obtained through the barriers. See Figure 1.

For log-homogenous barriers, although $H(\sqrt{\mu}u)$ scales $x(\mu, u)$ in $s(\mu, u)$, it doesn't scale $-g^*(s(\mu, u))$ into $-g(x(\mu, u))$. The upper bound in relation 4 controls exactly this difference. See Figure 2.

Also recall that if u is on the central path it follows from Theorem 2.16 that $\sqrt{\mu}u$ is a scaling point for both $(-g^*(s(\mu, u)), -g(x(\mu, u)))$ and

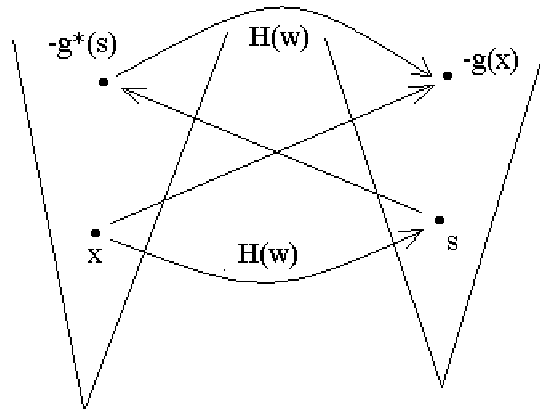

Figure 1: The case of symmetric cones.

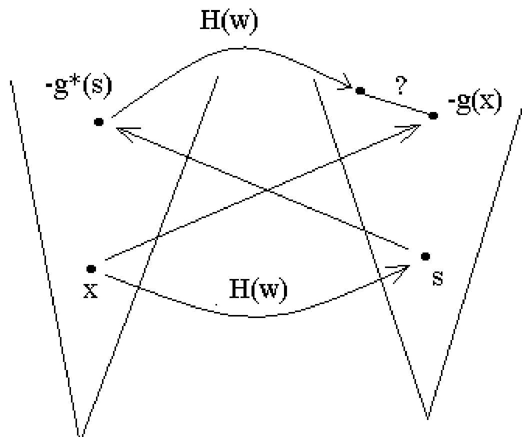

Figure 2: The case of nonsymmetric cones.

$(x(\mu,u),s(\mu,u))$. Intuitively, the closer a pair is to the central path the better the scaling behaves.

5. Relation 5 control the penalty value for the pair $(x(\mu,u),s(\mu,u))$.

We assume $u \in \mathring{K}$ and $\mu > 0$ satisfy the assumption of the previous theorem. To simplify notations, denote with $w := \sqrt{\mu}u$, $x := x(\mu,u)$ and $y := y(\mu,u)$. We define the predictor direction $(\Delta x, \Delta s)$ in the symmetric way (9):

$$\begin{cases} \Delta s + H(w)\Delta x = s \\ \mathcal{A}\Delta x = 0 \\ \mathcal{A}^*(\Delta y) + \Delta s = 0 \end{cases}$$

One can prove

Theorem 2.27 *The following relations hold:*

$$1.\ \langle \Delta x, s\rangle + \langle x, \Delta s\rangle = \langle x, s\rangle$$

$$2.\ \langle c, x - \Delta x\rangle - \langle b, y - \Delta y\rangle = 0$$

$$3.\ (||\Delta x||_w)^2 + (||\Delta s||_w^*)^2 = \langle x, s\rangle$$

$$\begin{aligned} 4.\ |\nu_f + \langle \Delta x, g(x)\rangle + \langle g^*(s), \Delta s\rangle| &\leq \tfrac{1}{2}\langle x, s\rangle^{1/2} ||g(x) - H(w)g^*(s)||_w^* \\ &\leq \tfrac{\beta^2}{1-\beta}(\beta + \sqrt{\nu_f}). \end{aligned}$$

We now look at how is Ω growing when we move along the direction $(\Delta x, \Delta s)$ that we have just defined.

Denote $\omega(t) = -t - \ln(1-t)$ and $\alpha \in (0, \frac{1-\beta}{\beta+\sqrt{\nu_f}})$.

$\Omega(x \pm \alpha\Delta x, s \pm \alpha\Delta s) - \Omega(x,s) = f(x \pm \alpha\Delta x) + f^*(s \pm \alpha\Delta s)$

$+\nu_f \ln \frac{\langle x \pm \alpha\Delta x, s \pm \alpha\Delta s\rangle}{\nu_f} - f(x) - f^*(s) - \nu_f \ln \frac{\langle x, s\rangle}{\nu_f}$

$\overset{\text{Theorem 2.27}}{=} f(x \pm \alpha\Delta x) + f^*(s \pm \alpha\Delta s) - f(x) - f^*(s) + \nu_f \ln(1 \pm \alpha)$

$$\overset{\text{Theorem 2.16 (6)}}{\leq} \pm\alpha[\langle \Delta x, g(x)\rangle + \langle g^*(s), \Delta s\rangle] + \nu_f \ln(1 \pm \alpha)+$$
$$+\omega(\alpha\,||\Delta x||_x) + \omega(\alpha\,||\Delta s||_s).$$
$$\overset{\text{Theorem 2.27 (4)}}{\leq} \alpha\tfrac{\beta^2}{1-\beta}(\beta + \sqrt{\nu_f}) + \omega(\alpha\,||\Delta x||_x) + \omega(\alpha\,||\Delta s||_s).$$

One can show that $t \longmapsto \omega(\sqrt{t})$ is convex and so

$\omega(\alpha\,||\Delta x||_x) + \omega(\alpha\,||\Delta s||_s) = \omega(\sqrt{\alpha^2\,||\Delta x||_x^2}) + \omega(\sqrt{\alpha^2\,||\Delta s||_s^2})$

$\leq \omega(\sqrt{\alpha^2\,||\Delta x||_x^2 + \alpha^2\,||\Delta s||_s^2}) = \omega(\alpha\sqrt{||\Delta x||_x^2 + ||\Delta s||_s^2}).$

From Theorem 2.16 (2) and Theorem 2.27 (3) follows

$\sqrt{||\Delta x||_x^2 + ||\Delta s||_s^2} \leq \frac{\sqrt{\mu}}{(1-\beta)^2}\sqrt{||\Delta x||_w^2 + ||\Delta s||_w^2}$

$\leq \frac{\langle \mu x, s\rangle^{1/2}}{(1-\beta)} \leq \frac{\beta+\sqrt{\nu_f}}{(1-\beta)}$

It follows:

Theorem 2.28 *If f is a log-homogenous barrier then:*

$$\Omega(x \pm \alpha\Delta x, s \pm \alpha\Delta s) - \Omega(x, s) \leq \beta^2\left(\alpha\frac{\beta+\sqrt{\nu_f}}{1-\beta}\right) + \omega\left(\alpha\frac{\beta+\sqrt{\nu_f}}{(1-\beta)}\right)$$

for all $\alpha \in (0, \frac{1-\beta}{\beta+\sqrt{\nu_f}})$. If f is a self-scaled barrier then

$$\Omega(x \pm \alpha\Delta x, s \pm \alpha\Delta s) - \Omega(x, s) \leq \omega\left(\alpha\frac{\beta+\sqrt{\nu_f}}{(1-\beta)}\right) \text{ for all } \alpha \in \left(0, \frac{1-\beta}{\beta+\sqrt{\nu_f}}\right).$$

Remark 2.11 *Note that because $\alpha \in \left(0, \frac{1-\beta}{\beta+\sqrt{\nu_f}}\right)$ we know that $x \pm \alpha\Delta x$, $s \pm \alpha\Delta s$ are feasible.*

Using this result, one can prove:

Theorem 2.29 *Fix $\beta, \gamma \in (0, 1)$. If $\alpha > 0$ is s.t. $\Omega(x - \alpha\Delta x, s - \alpha\Delta s) - \Omega(x, s) = \beta^2\gamma + \omega(\gamma)$ (see Figure 3b. for the behavior of the right-hand side) then*

1. the penalty level for $(x - \alpha\Delta x, s - \alpha\Delta s)$ is bounded by

$$\mu(x - \alpha\Delta x, s - \alpha\Delta s) \geq \mu \exp\left(\gamma\tfrac{1-\beta}{\beta+\sqrt{\nu_f}} - \tfrac{2\beta}{\sqrt{\nu_f}}\right)$$

see Figure 4; note how the increase in β is decreasing the lower bound on the new μ.

2. the centrality of $(x - \alpha\Delta x, s - \alpha\Delta s)$ is bounded by

$$\Omega(x - \alpha\Delta x, s - \alpha\Delta s) \leq 2\omega(\beta) + \beta^2(1 + \gamma) + \omega(\gamma)$$

see Figure 3a.; note how the right-hand side grows when β or γ approach 1 culminating with the case when both approach 1.

To summarize. Fix $\beta, \gamma \in (0, 1)$. If we have $\mu > 0$ and $u \in \mathring{K}$ close enough to x_μ, i.e., with the size of the Newton step less then β, then we can construct a pair of points $(x(\mu, u), s(\mu, u))$ and (symmetric) directions $(\Delta x(\mu, u), \Delta s(\mu, u))$ such that if we compute $\alpha > 0$ such that

$$\Omega(x(\mu, u) - \alpha\Delta x(\mu, u), s(\mu.u) - \alpha\Delta s(\mu, u)) = \Omega(x(\mu, u), s(\mu, u)) + \beta^2\gamma + \omega(\gamma)$$

then the penalty level and the centrality for the pair $(x - \alpha\Delta x, s - \alpha\Delta s)$ are bounded by

$$\mu(x - \alpha\Delta x, s - \alpha\Delta s) \geq \mu \exp\left(\gamma\frac{1-\beta}{\beta+\sqrt{\nu_f}} - \frac{2\beta}{\sqrt{\nu_f}}\right)$$

and, respectively,

$$\Omega(x - \alpha\Delta x, s - \alpha\Delta s) \leq 2\omega(\beta) + \beta^2(1 + \gamma) + \omega(\gamma)$$

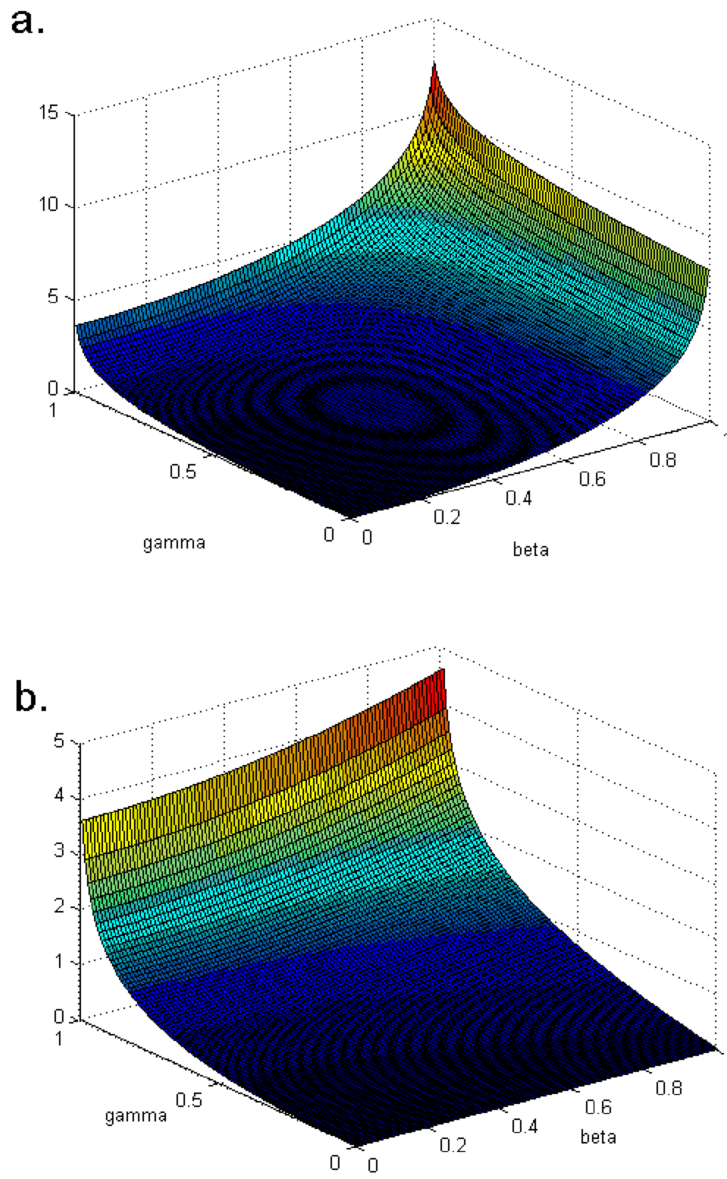

Figure 3: a.$2\omega(\beta) + \beta^2(1+\gamma) + \omega(\gamma)$ b.$\beta^2\gamma + \omega(\gamma)$

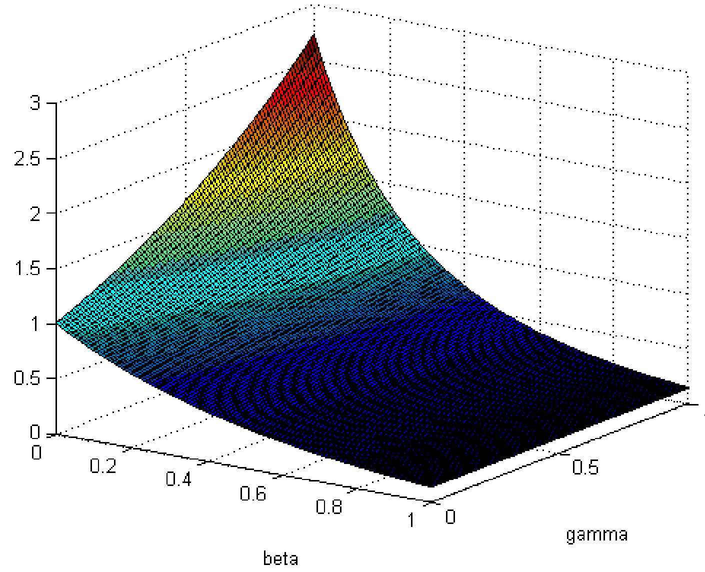

Figure 4: $\exp\left(\gamma\frac{1-\beta}{\beta+\sqrt{\nu_f}} - \frac{2\beta}{\sqrt{\nu_f}}\right)$

In order to repeat the previous procedure, having the point $(x-\alpha\Delta x, s-\alpha\Delta s)$ we need to be able to compute a new point $u \in \overset{\circ}{K}$ close enough to $x_{\mu(x-\alpha\Delta x, s-\alpha\Delta s)}$, i.e., with the size of the Newton step less then β. The proximity measure $\Omega(x,s)$ provides information about how far away is x from the minimizer $x_{\mu(x,s)}$ in terms of level curves of $f_{\mu(x,s)}$:

Theorem 2.30 *Given $(x,s) \in \overset{\circ}{K} \times \overset{\circ}{K}{}^*$ the following relation is true:*

$$0 \le f_{\mu(x,s)}(x) - f_{\mu(x,s)}(x_{\mu(x,s)}) \le \Omega(x,s)$$

But for a self-concordant functional one can guarantee a certain progress in the objective at every Newton step:

Theorem 2.31 *If f is self-concordant then for any $u \in \overset{\circ}{K}$ a Newton step for the problem*

$$\begin{cases} \inf_x f(x) \\ \text{s.t. } \mathcal{A}x = b \end{cases}$$

decreases the objective by at least $w_*(||\Delta u||_u)$ where $w_*(t) = t - \ln(1+t)$ and Δu is the Newton step in u.

If we start in $x - \alpha\Delta x$ and apply Newton's method to $f_{\mu(x-\alpha\Delta x, s-\alpha\Delta s)}$ as long as the size of the Newton step is bigger then β we can decrease the value of $f_{\mu(x-\alpha\Delta x, s-\alpha\Delta s)}$ by at least $\omega_*(\beta)$ at each iteration.

Because $\Omega(x - \alpha\Delta x, s - \alpha\Delta s) \leq 2\omega(\beta) + \beta^2(1+\gamma) + \omega(\gamma)$ we need at most $\frac{2\omega(\beta)+\beta^2(1+\gamma)+\omega(\gamma)}{\omega_*(\beta)}$ iterations to obtain a point $u \in K$ close enough to $x_{\mu(x-\alpha\Delta x, s-\alpha\Delta s)}$.

We have, therefore, described the following algorithm:

Input: $\beta, \gamma \in (0,1)$ (the parameter controlling the small neighborhood of the central path and, respectively, the large neighborhood)

$\epsilon > 0$ (the desired accuracy i.e. $\langle x, s\rangle \leq \epsilon$)

$\mu > 0$ and $u \in \mathring{K}$ such that $||\Delta u||_u \leq \beta$ where Δu is the Newton step applied in u for f_μ:

$$\begin{cases} \mu c + g(u) + H(u)\Delta u = \mathcal{A}^* y \\ \mathcal{A}\Delta u = 0 \end{cases}$$

Proceed:

$k := 0$

$u_1 := u$

$\mu_1 := \mu$

Repeat: $k = k + 1$.

Compute y and Δu the solutions of

$$\begin{cases} \mu_k c + g(u_k) + H(u_k)\Delta u = \mathcal{A}^* y \\ \mathcal{A}\Delta u = 0 \end{cases}$$

Define $x = u_k - \Delta u$, $s := c - \frac{1}{\mu_k}\mathcal{A}^* y$, $w := \sqrt{\mu_k} u_k$

Compute Δx and Δs from

$$\begin{cases} \Delta s + H(w)\Delta x = s \\ \mathcal{A}\Delta x = 0 \\ \mathcal{A}^*(\Delta y) + \Delta s = 0 \end{cases}$$

Compute $\alpha > 0$ be such that

$$\Omega(x - \alpha\Delta x, s - \alpha\Delta s) = \Omega(x, s) + \beta^2\gamma + \omega(\gamma)$$

Define $x^* := x - \alpha\Delta x$ and $s^* := s - \alpha\Delta s$

$\mu_{k+1} := \mu(x^*, s^*)$; $u_{k+1} := x^*$

Repeat

Compute y and Δu the solutions of

$$\begin{cases} \mu_{k+1} c + g(u_{k+1}) + H(u_{k+1})\Delta u = \mathcal{A}^* y \\ \mathcal{A}\Delta u = 0 \end{cases}$$

$u_{k+1} := u_{k+1} + \frac{1}{1+||\Delta u||_{u_{k+1}}}\Delta u$

Until $||\Delta u||_{u_{k+1}} \leq \beta$

Until: $\frac{\nu_f}{\mu_{k+1}} < \epsilon$

3 Semidefinite optimization in matrix variable

The natural formulation of problems in control theory is in matrix variable. One way to solve such problems is to vectorize the matrix variable, i.e., to write the matrix as a vector of appropriate size by stacking the lines or the columns. When vectorizing, one also needs to change the linear operators defining the constraints. We will illustrate these ideas on the following example:

$$\begin{cases} \sup -tr(P) \\ P \succeq 0 \\ F^T PF - P \preceq I \end{cases} \tag{13}$$

where P and F are 2×2 matrices with real elements, P symmetric.
Denote $P = \begin{bmatrix} y_1 & y_2 \\ y_2 & y_3 \end{bmatrix}$ and $F = \begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \end{bmatrix}$.
It follows that

$$\begin{aligned} F^T PF &= y_1 \begin{bmatrix} f_{11}^2 & f_{11}f_{12} \\ f_{12}f_{11} & f_{12}^2 \end{bmatrix} + y_2 \begin{bmatrix} 2f_{21}f_{11} & f_{11}f_{22} + f_{21}f_{12} \\ f_{21}f_{12} + f_{11}f_{22} & f_{12}f_{11} + f_{21}f_{22} \end{bmatrix} \\ &+ y_3 \begin{bmatrix} f_{21}^2 & f_{22}f_{21} \\ f_{21}f_{22} & f_{22}^2 \end{bmatrix} \end{aligned}$$

Therefore the equivalent problem is

$$\begin{cases} \sup -y_1 - y_3 \\ y_1 \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + y_2 \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} + y_3 \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \succeq 0 \\ y_1 \begin{bmatrix} f_{11}^2 - 1 & f_{11}f_{12} \\ f_{12}f_{11} & f_{12}^2 \end{bmatrix} + y_2 \begin{bmatrix} 2f_{21}f_{11} & f_{11}f_{22} + f_{21}f_{12} - 1 \\ f_{21}f_{12} + f_{11}f_{22} - 1 & f_{12}f_{11} + f_{21}f_{22} \end{bmatrix} + \\ +y_3 \begin{bmatrix} f_{21}^2 & f_{22}f_{21} - 1 \\ f_{21}f_{22} & f_{22}^2 \end{bmatrix} \succeq 0 \end{cases}$$

which is the same as

$$\begin{cases} \sup \quad - y_1 - y_3 \\ y_1A_1 + y_2A_2 + y_3A_3 \succeq 0 \end{cases} \tag{14}$$

where $A_1 := diag \left(\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} f_{11}^2 - 1 & f_{11}f_{12} \\ f_{12}f_{11} & f_{12}^2 \end{bmatrix} \right)$

$A_2 := diag \left(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 2f_{21}f_{11} & f_{11}f_{22} + f_{21}f_{12} - 1 \\ f_{21}f_{12} + f_{11}f_{22} - 1 & f_{12}f_{11} + f_{21}f_{22} \end{bmatrix} \right)$,

$A_3 := diag \left(\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} f_{21}^2 & f_{22}f_{21} - 1 \\ f_{21}f_{22} & f_{22}^2 \end{bmatrix} \right)$

In the general case when P and F are $n \times n$ matrices the equivalent problem is

$$\begin{cases} \sup \ \langle c, y \rangle \\ \sum_{i=1}^{n(n+1)/2} y_i A_i \succeq 0 \end{cases} \tag{15}$$

where c is a vector in $\mathbb{R}^{\frac{n(n+1)}{2}}$ and A_i's, $i = 1, \frac{n(n+1)}{2}$, are $2n \times 2n$ block-diagonal matrices. The evaluation of the linear constraint in (15) requires $\frac{n(n+1)}{2}(2n^2) = O(n^4)$ scalar-scalar multiplications and $\frac{n(n+1)}{2} - 1$ matrix additions. In total, $O(n^4)$ operations. The evaluation of the linear operator in (13) requires three matrix-matrix multiplications and one matrix additions. In total, $O(n^3)$ operations.

In implementations, the evaluation of the linear operator in matrix variable is cheaper than the evaluation of the linear operator in vectorized form. Not also in terms of number of operations but also because matrix-matrix multiplications are preferred to scalar-scalar multiplications.

Recall the convex cone optimization problem (2) and (3), from Chapter 1. Consider $X = S^{k\times k}$ with the inner product $\langle x, y \rangle := tr(xy)$ for $x, y \in S^{k\times k}$. Consider $Y = \mathbb{R}^{m\times n}$ with inner product $\langle x, y \rangle = tr(x^T y)$ for $x, y \in \mathbb{R}^{m\times n}$. Let $K = S_+^{k\times k}$,the cone of semidefinite positive matrices. One has $K^* = S_+^{k\times k}$. Let $c \in S^{k\times k}$, $b \in \mathbb{R}^{m\times n}$ and $\mathcal{A} : S^{k\times k} \to \mathbb{R}^{m\times n}$ be a linear operator with adjoint $\mathcal{A}^* : \mathbb{R}^{m\times n} \to S^{k\times k}$. Then the problems

$$\begin{cases} \inf\limits_{x} \langle c, x \rangle \\ \text{s.t. } \mathcal{A}x = b \\ x \in S_+^{k\times k} \end{cases} \tag{16}$$

and

$$\begin{cases} \sup\limits_{y \in \mathbb{R}^{m \times n}, s \in S_+^{k \times k}} \langle b, y \rangle \\ \text{s.t. } \mathcal{A}^* y + s = c \\ s \in S_+^{k \times k} \end{cases} \tag{17}$$

are referred as semidefinite optimization problems in matrix variable in the primal and, respectively, dual form. In applications, the usual input is a problem in the dual form. However, primal-dual algorithms require both $\mathcal{A}^*$ and $\mathcal{A}$. In what follows we present results that describe the set of linear operators $\mathcal{A}^* : \mathbb{R}^{m \times n} \to S^{k \times k}$ and $\mathcal{A} : S^{k \times k} \to \mathbb{R}^{m \times n}$ and show how to compute $\mathcal{A}$ if $\mathcal{A}^*$ is given and vice-versa.

Definition 3.1 *(Linear pencil in matrix variable). Assume $p \in \mathbb{N}$, $x_i \in \mathbb{R}^{k \times m}$, $y_i \in \mathbb{R}^{n \times k}$, $i = \overline{1,p}$. We call a linear pencil in matrix variable the linear operator $\mathcal{P} : \mathbb{R}^{m \times n} \to S^{k \times k}$*

$$\mathcal{P}(y) = \sum_{i=1}^{p} \left(x_i y y_i + y_i^T y^T x_i^T \right)$$

Definition 3.2 *(Linear function in matrix variable). Assume $p \in \mathbb{N}$, $x_i \in \mathbb{R}^{m \times n}$, $y_i \in S^{k \times k}$, $i = \overline{1,p}$. We call a linear function in matrix variable the linear operator $\mathcal{F} : \mathbb{R}^{m \times n} \to S^{k \times k}$*

$$\mathcal{F}(y) = \sum_{i=1}^{p} \langle x_i, y \rangle y_i$$

Theorem 3.1 *Assume $\mathcal{F} : \mathbb{R}^{m \times n} \to S^{k \times k}$, $\mathcal{F}(y) = \langle x_1, y \rangle y_1$ is a linear function in matrix variable, where $x_1 \in \mathbb{R}^{m \times n}$, $y_1 \in S^{k \times k}$. Then the adjoint $\mathcal{F}^* : S^{k \times k} \to \mathbb{R}^{m \times n}$ is given by*

$$\mathcal{F}^*(x) = \langle y_1, x\rangle\, x_1$$

Proof.

$\langle \mathcal{F}(y), x\rangle = tr(\mathcal{F}(y)x) = tr(\langle x_1, y\rangle\, y_1 x) = \langle x_1, y\rangle \langle y_1, x\rangle$

$\langle y, \mathcal{F}^*(x)\rangle = tr(y^{T\mathcal{F}*}(x)) = tr(y^T \langle y_1, x\rangle\, x_1)) = \langle y_1, x\rangle \langle y, x_1\rangle$

Therefore, for any $y \in S^{k\times k}$ any $x \in \mathbb{R}^{m\times n}$, we have showed

$$\langle \mathcal{F}(y), x\rangle = \langle y, \mathcal{F}^*(x)\rangle$$

■

Corollary 3.2 *Assume* $\mathcal{F} : \mathbb{R}^{m\times n} \to S^{k\times k}$, $\mathcal{F}(y) = \sum_{i=1}^{p} \langle x_i, y\rangle\, y_i$ *is a linear function in matrix variable, where* $p \in \mathbb{N}$, $x_i \in \mathbb{R}^{m\times n}$, $y_i \in S^{k\times k}$. *Then the adjoint* $\mathcal{F}^* : S^{k\times k} \to \mathbb{R}^{m\times n}$ *is given by*

$$\mathcal{F}^*(x) = \textstyle\sum_{i=1}^{p} \langle y_i, x\rangle\, x_i$$

Theorem 3.3 *Assume* $\mathcal{P} : \mathbb{R}^{m\times n} \to S^{k\times k}$, $\mathcal{P}(y) = x_1 y y_1 + y_1^T y^T x_1^T$ *is a linear pencil in matrix variable, where* $x_1 \in \mathbb{R}^{k\times m}$, $y_1 \in \mathbb{R}^{n\times k}$. *Then the adjoint* $\mathcal{P}^* : S^{k\times k} \to \mathbb{R}^{m\times n}$ *is given by*

$$\mathcal{P}^*(x) = 2x_1^T x y_1^T$$

Proof.

$\langle \mathcal{P}(y), x\rangle = \left\langle x_1 y y_1 + (x_1 y y_1)^T, x\right\rangle = 2tr(x_1 y y_1 x)$

$\langle y, \mathcal{P}^*(x)\rangle = \left\langle y, 2x_1^T x y_1^T\right\rangle = 2tr((x_1^T x y_1^T)^T y) = 2tr(y_1 x x_1 y) = 2tr(x_1 y y_1 x)$

Therefore, for any $y \in S^{k\times k}$ any $x \in \mathbb{R}^{m\times n}$, we have showed

$$\langle \mathcal{P}(y), x\rangle = \langle y, \mathcal{P}^*(x)\rangle$$

■

Corollary 3.4 *Assume* $\mathcal{P} : \mathbb{R}^{m\times n} \rightarrow S^{k\times k}$*,* $\mathcal{P}(y) = \sum_{i=1}^{p} x_i y y_i + y_i^T y^T x_i^T$ *is a linear pencil in matrix variable, where* $p \in \mathbb{N}$*,* $x_i \in \mathbb{R}^{k\times m}$*,* $y_i \in \mathbb{R}^{n\times k}$*. Then the adjoint* $\mathcal{P}^* : S^{k\times k} \rightarrow \mathbb{R}^{m\times n}$ *is given by*

$$\mathcal{P}^*(x) = 2\sum_{i=1}^{p} x_i^T x y_i^T$$

Theorem 3.5 *If* $\mathcal{A}^* : \mathbb{R}^{m\times n} \rightarrow S^{k\times k}$ *is a linear operator then there exists a linear pencil in matrix variable* $\mathcal{P}_{\mathcal{A}^*}$ *and a linear function in matrix variable* $\mathcal{F}_{\mathcal{A}^*}$ *such that*

$$\mathcal{A}^* = \mathcal{P}_{A^*} + \mathcal{F}_{\mathcal{A}^*}$$

4 Barriers for some well-structured sets

4.1 The non-negative orthant: logarithmic barrier

The logarithmic barrier on $\mathbb{R}^n_{++}$, the interior of $\mathbb{R}^n_+$, is defined by

$$f(x) = -\sum_{i=1}^n \ln x_i,\ x \in \mathbb{R}^n_{++}$$

Theorem 4.1 *If f is the logarithmic barrier on the non-negative orthant then*

1. f is a self-scaled barrier
2. $\nu_f = n$
3. $g(x) = (-\frac{1}{x_1}, ..., -\frac{1}{x_n})\,, \forall x \in \mathbb{R}^n_{++}$
4. $H(x)$ is the $n \times n$ matrix having the vector $(\frac{1}{x_1^2}, ..., \frac{1}{x_n^2})$ on the diagonal and zero otherwise
5. $H^{-1}(x)$ is the $n \times n$ matrix having the vector $(x_1^2, ..., x_n^2)$ on the diagonal and zero otherwise
6. The unique scaling point of a pair (x, s) is $w = (\sqrt{\frac{x_1}{s_1}}, ..., \sqrt{\frac{x_n}{s_n}})$. Equivalent, $H(w) = diag(\frac{s_1}{x_1}, ..., \frac{s_n}{x_n})$.

4.2 The second-order cone: logarithmic barrier

The logarithmic barrier on the interior of the second order cone

$$K = \{x \in \mathbb{R}^n : x_n \geq \sqrt{\sum_{i=1}^{n-1} x_i^2}\}$$

is defined by

$$f(x) = -\ln(x_n^2 - \textstyle\sum_{i=1}^{n-1} x_i^2)$$

For $x \in K$ denote $\det(x) := x_n^2 - \sum_{i=1}^{n-1} x_i^2$ and $\bar{x} = (x_1, ..., x_{n-1}, -x_n)$

Theorem 4.2 *If f is the logarithmic barrier on the second order cone then*

1. f is a self-scaled barrier

2. $\nu_f = 2$

3. $g(x) = \frac{2}{\det x}\bar{x}$

4. $H(x) = \frac{4}{\det(x)^2}\bar{x}\bar{x}^T + \frac{2}{\det(x)}diag(1, 1, ..., 1, -1)$,

   since $H(x) = \nabla g(x) = 2\nabla\left(\frac{1}{\det x}\right)\bar{x}^T + \frac{2}{\det x}\nabla(\bar{x}) =$

   $$= \frac{4}{(\det x)^2}\bar{x}\bar{x}^T + \frac{2}{\det x}diag(1, 1, ..., -1)$$

   For any vector $y \in \mathbb{R}^n$, from Sherman-Morisson formula we get

   $$(yy^T + A)^{-1} = A^{-1} - \frac{A^{-1}yy^TA^{-1}}{1+y^TA^{-1}y}.$$

   Letting $y = \frac{2}{\det x}\bar{x}$ and $A = \frac{2}{\det x}diag(1, 1, ..., -1)$ we have

   $$A^{-1} = \frac{\det x}{2}diag(1, 1, ..., -1)$$

   and also

   $$A^{-1}y = x$$

   Therefore

5. $H^{-1}(x) = \frac{\det x}{2}diag(1, 1, ..., -1) - \frac{xx^T}{1+\frac{2}{\det x}\bar{x}^Tx}$

   $= \frac{\det x}{2}diag(1, 1, ..., -1) + xx^T$

4.3 The cone of semidefinite positive matrices: logarithmic barrier

The logarithmic barrier on $S^{n\times n}_{++}$ the set of positive definite matrices and the interior of $S^{n\times n}_{+}$ is defined by

$$f(x) = -\ln\det(x),\ x \in S^{n\times n}_{++}$$

Theorem 4.3 *If f is the logarithmic barrier on $S^{n\times n}_{+}$ then*

1. f is a self-scaled barrier
2. $\nu_f = n$
3. $g(x) = -x^{-1},\ x \in S^{n\times n}_{++}$
4. $H(x)(y) = x^{-1}yx^{-1},\ x \in S^{n\times n}_{++},\ y \in \mathbb{R}^{n\times n}$
5. $H^{-1}(x)(y) = xyx,\ x \in S^{n\times n}_{++}, y \in \mathbb{R}^{n\times n}$
6. Although there is an explicit formula for computing the scaling point w, it is not efficient in practice. See [18] for one more efficient procedure.

4.4 The p-cone

Consider the following set $K_p = \{(\tau, z) \in \mathbb{R}\times\mathbb{R}^n : \tau \geq ||z||_p\}$ where $||z||_p \stackrel{def}{=} \left(\sum_{i=1}^n |z_i|^p\right)^{1/p}$

K_p is a closed convex cone referred in the literature as the $p-$cone.

In what follows, we present a barrier with complexity $4n$ on K_p due to Nesterov. We only present the end results; for all details see [10].

Let $Q_\alpha \stackrel{def}{=} \{(x,y,z) \in \mathbb{R}^2_+ \times \mathbb{R} : x^\alpha \cdot y^{1-\alpha} \geq |z|\}$ with $\alpha \in [0,1]$. One can show that Q_α is a closed, convex and self-dual cone.

The following result

Theorem 4.4 *The point (τ, x) is in K_p if and only if there exists $x \in \mathbb{R}^n_+$ satisfying the conditions:*

$$x_i^\alpha \tau^{1-\alpha} \geq |z_i|$$
$$\textstyle\sum_{i=1}^n x_i = \tau$$

Remark 4.1 *An equivalent statement is as follows. The point $(\tau, x) \in K_p$ if and only if there exist $x, y \in \mathbb{R}^n$ such that*

$$\begin{aligned} (x_i, y_i, z_i) &\in Q_{1/p} \\ y_i &= \tau \\ \sum_{i=1}^n x_i &= \tau \end{aligned}$$

allows us to change a problem whose variable $(\tau, z) \in \mathbb{R}^{n+1}$ is constrained to belong to K_p into a problem whose variable $(\tau, z, x, y) \in \mathbb{R}^{n+1} \times \mathbb{R}^n \times \mathbb{R}^n$ belong to the closed convex cone

$$Q := \{(\tau, z, x, y) \in \mathbb{R}^{n+1} \times \mathbb{R}^n \times \mathbb{R}^n : (x_i, y_i, z_i) \in Q_{1/p}\}$$

and also satisfy two more linear constraints

$$y_i - \tau = 0$$
$$\textstyle\sum_{i=1}^n x_i - \tau = 0.$$

Having a barrier $f_{1/p}$ on $Q_{1/p}$, the functional defined on Q by $f(x,y,z) = \sum_{i=1}^{n} f_{1/p}(x_i, y_i, z_i)$ is a barrier on Q. The primal-dual interior point algorithm that was described can only be used if we can compute the values of f, its gradient, its Hessian and the values of f^*, the conjugate of f (see Definition 2.4).

In what follows, we present a barrier on Q_α as presented in [10]. We anticipate by saying that the complexity of the barrier on Q_α is 4 which implies that the complexity of f is $4n$.

Theorem 4.5 *For any $\alpha \in [0,1]$ the function $f_\alpha(x,y,z) = -\ln(x^{2\alpha}y^{2(1-\alpha)} - z^2) - \ln x - \ln y$ is a self-concordant barrier for the closed convex self-dual cone $Q_\alpha = \{(x,y,z) \in \mathbb{R}^2_+ \times \mathbb{R} : x^\alpha \cdot y^{1-\alpha} \geq |z|\}$*

Unfortunately, the conjugate functional of f_α cannot be written in closed form. One can compute the values of f^*_α, g^*_α, H^*_α using the following result:

Theorem 4.6 *Let $f^*_\alpha(s) = -\min_x[\langle s,x\rangle + f_\alpha(x)]$ be the conjugate functional of f_α. Denote with $x^s = (x^s_1, x^s_2, x^s_3)$ the unique solution of $\min_x[\langle s,x\rangle + f_\alpha(x)]$. Then*

1. $g^*_\alpha(s) = -x^s$
2. $H^*_\alpha(s) = [H^*_\alpha(x^s)]^{-1}$
3. $s_1 x^s_1 = 1 + 2\alpha - \alpha s_3 x^s_3$ and $s_2 x^s_2 = 3 - 2\alpha - (1-\alpha)s_3 x^s_3$
4. $f^*_\alpha(s) = -\min_{x_3}[-\ln\left(\left(\frac{1+2\alpha-\alpha s_3 x_3}{s_1}\right)^{2\alpha}\left(\frac{1+2(1-\alpha)-(1-\alpha)s_3 x_3}{s_2}\right)^{2(1-\alpha)} - x_3^2\right) - -\ln\frac{1+2\alpha-\alpha s_3 x_3}{s_1} - \ln\frac{1+2(1-\alpha)-(1-\alpha)s_3 x_3}{s_2}] - 4$

Note that the function to be minimized in 4. can be written as
$-\ln\left(\left(s_3\frac{\alpha}{s_1}\right)^{2\alpha}\left(\frac{1}{\alpha}+2-x_3\right)^{2\alpha}\left(s_3\frac{1-\alpha}{s_2}\right)^{2(1-\alpha)}\left(\frac{1}{1-\alpha}+2-x_3\right)^{2(1-\alpha)}-x_3^2\right)-$
$-\ln s_3\frac{\alpha}{s_1}\left(\frac{1}{\alpha}+2-x_3\right)-\ln s_3\frac{1-\alpha}{s_2}\left(\frac{1}{1-\alpha}+2-x_3\right).$

Denoting $q := |s_3|\left(\frac{\alpha}{s_1}\right)^{\alpha}\left(\frac{1-\alpha}{s_2}\right)^{1-\alpha} \in [0,1]$, after removing the constants, we are interested in the following problem:

$$\min_{\tau}\left[\begin{array}{c} h(\tau) = -\ln\left(q^2\left(\frac{1}{\alpha}+2-\tau\right)^{2\alpha}\left(\frac{1}{1-\alpha}+2-\tau\right)^{2(1-\alpha)}-\tau^2\right)- \\ -\ln\left(\frac{1}{\alpha}+2-\tau\right)-\ln\left(\frac{1}{1-\alpha}+2-\tau\right)\end{array}\right] \quad (18)$$

The objective here is a self-concordant functional with complexity 4. Denoting $r=\frac{q}{1-q}$, Nesterov shows:

Theorem 4.7 *The solution of (18) is in $[-4r,0]$. Moreover, after applying at most seven times the bisection method to f' with the starting interval being $[-4r,0]$, any point in the new interval used as a starting point for Newton method is guarantee to provide quadratic convergence.*

Part II

The design of YAS

5 Overview

In this chapter we want to

- familiarize the user with the design of the solver and describe the advantages resulting from this design
- make use of a terminology that anticipates the names and the purposes of the classes and routines to be described later. We use bold font to signal such terms.

The routines and classes of YAS are grouped in two layers:

- the Linear Algebra (LA) layer
- the Interior Point Methods (IPM) layer

5.1 LA Layer

This layer's end goal is to provide the user classes that substitute matrices and allow linear algebra operations. They will be used in the implementation of the IPM layer to operate with and store matrices.

These classes' design allows to:

- efficiently store a matrix by splitting it into **blocks** of different **types** (such as dense, sparse, symmetric, upper-diagonal, lower-diagonal, sparse symmetric etc)

- efficiently add/multiply such **matrices of blocks** by exploiting the type of the consisting **blocks** (in particular taking advantage of Sparse BLAS routines)

- factorize a **matrix of blocks** (LU, QR or Cholesky factorization) with algorithms that use the factorization of the building blocks

- efficiently add/multiply, invert, factorize or solve linear systems associated with a **block** by exploiting the block's type (such as using ScaLAPACK and Sparse BLAS routines for sparse blocks)

- invert or factorize **update matrices** without ignoring the available information (use Sherman-Morrison like formulas)

- use different data precision to store the elements of a **block** (such as double or float)

The above mentioned linear algebra operations on **matrices of blocks** or just on **blocks** make use of certain **low-level routines** (such as multiplication of two dense blocks). The LA Layer interfaces these routines. For example, YAS_gemm is a function that multiplies two dense blocks and it is the only function used for this purpose in the rest of the code. The user can choose to modify its implementation without worrying about the rest of the code. In particular, one can use linear algebra packages specially designed for a certain computing platform (such as the MKL Library for Intel processors, or the ACML library for AMD processors). For most **low-level routines**, current implementation allows the switch between MKL and ACML.

The LA Layer consists of

- the low-level routines, further grouped into:
  - BLAS routines (matrix-matrix multiplications and additions for different types of matrices)
  - LAPACK routines (used for inverting, factorizing, solving linear systems with different types of matrices)
- the YAS_K_block class (allows the storage of one or more blocks of the same type and dimension and provides methods to do linear algebra operations)
- the YAS_K_mb class (allows the storage of one or more matrices of blocks of the same type and dimension and provides methods to do linear algebra operations)

We conclude this introduction of the LA Layer by remarking that the classes YAS_K_block and YAS_K_mb have the same interface (i.e. the same methods). Because of this, any code that works with YAS_K_block objects also works with YAS_K_mb objects and vice-versa, in the same way as the instruction $a+b$ works when the type of a and b is *double* as well as when it is $float$. Therefore, the user might want to start by modelling the data using **blocks** which is, of course, simpler. If the situation requires, i.e. the algorithm is too slow or requires too much memory, the data could be accommodated with **matrices of blocks**.

5.2 IPM layer

This layer provides a framework to implement interior point algorithms. To get a sense of the motivation behind the design and the resulting advantages we briefly describe some of its classes:

- YAS_k_EVS is a class that is a container of a k-tuple of elements of a vector space. The class provides methods such as adding, scaling or computing the norm of objects of this type.

  An example, motivated by semidefinite programming, of a class derived from YAS_k_EVS is as follows. An object from this class stores a pair of matrices (X, Z) in a scaled spaces i.e. it stores a matrix of blocks U and a pair of matrices of blocks $(\bar{X}, \bar{Z})$ such that $X = U\bar{X}U^T$ and $Z = U^{-1}\bar{Z}U^{-T}$. The addition method in this derived class is overloaded so that if the user is asking to add the object containing $(U, \bar{X}_1, \bar{Z}_1)$ with an object containing $(U, \bar{X}_2, \bar{Z}_2)$ the result will be an object containing $(U, \bar{X}_1 + \bar{X}_2, \bar{Z}_1 + \bar{Z}_2)$ where by $\bar{X}_1 + \bar{X}_2$ and $\bar{Z}_1 + \bar{Z}_2$ we mean the usual addition of matrices. This vector space has the following property. If $X := UVU^T$, $Z := U^{-1}VU^{-T}$, $X^* := U\overline{X^*}U^T$ and $Z^* = U^{-1}\overline{Z^*}U^{-T}$ are all positive definite then there exists a unique U^+ and a unique V^+ such that $X + X^* = U^+V^+(U^+)^T$ and $Z + Z^* = (U^+)^{-1}V^+(U^+)^{-T}$. In other words, if the pair (X, Z) is represented by the triplet (U, V, V) and (X^*, Z^*) by the triplet $(U, \overline{X^*}, \overline{Z^*})$ then there exists U^+, V^+ such that the pair $(X + X^*, Z + Z^*)$ is represented by the triplet (U^+, V^+, V^+). For this special situation the addition is implemented such that the resulting object contains

(U^+, V^+, V^+) instead of the usual $(U, V + \overline{X^*}, V + \overline{Z^*})$. How to compute U^+ and V^+ and what are the numerical advantages for keeping the variable in this form see [15]. This technique is currently implemented in SeDuMi.

Note that this trick, motivated by numerical issues, is happening in backstage. The algorithm is only working with objects of type YAS_k_EVS and is just asking for two such objects to be added. This results in very clean-looking algorithms.

Our goal is to provide a library of such derived classes that the user might want to use for his problem.

- $YAS_barrier$ is a class which is of obvious importance for interior point methods, as seen in Chapter 2.

  For a given derived class YAS_k_EVS, we provide derived classes of $YAS_barrier$ that will represent the feasible region. For the example above, the only thing that makes sense is a barrier for the second order cone. Given an object of type YAS_k_EVS and a barrier associated with this class, the algorithm might ask, through the methods of $YAS_barrier$, to evaluate the barrier, its gradient or its Hessian on the element. Note that the gradient will be an object from the same class YAS_k_EVS.

  Again, our goal is make available a library of such barriers. Besides the usual barriers for the positive orthant, the second order cone and the semidefinite cone, the algorithm by Nesterov [20] for nonsymmetric cones shows that it makes sense to have in the library the barrier for

the $p-$cone, see Chapter 4.

- YAS_LO is a class used to replicate a linear operator. Evaluation of the linear operator on an element of type YAS_k_EVS can be done in the natural formulation. As in the case of YAS_k_EVS, through derived classes we can make put in backstage and efficient implementation of which the algorithm doesn't have to know. In particular, this has practical relevance in the case of semidefinite programming in matrix variable, see Chapter 3.

- YAS_norm_eq is a class that is derived from YAS_LO allowing the user to form a compressed and expanded versions of normal equations and to solve the latter.

We conclude this section by stressing the clean-looking algorithms that can be implemented by using objects from this classes. The algorithm tricks, including heuristics, aiming for faster convergence and the numerical tricks aiming for speed/efficiency in memory storing/accuracy are completely separated.

Finishing up all details is "work in progress"

6 The Linear Algebra Layer

This part describes the routines and classes in the LA Layer. The layers' end goal is to provide classes that substitute matrices.

This layer consists of the low-level routines, the YAS_K_block class and the YAS_K_mb class. We start with some important conventions regarding matrix storage.

Dense matrix storage convention. $In\ YAS$, dense blocks are stored columnwise. For example, the dense block

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

is stored as $\{1, 4, 2, 5, 3, 6\}$.

CSC (Compressed Sparse Column) sparse storage convention. In YAS, storing a matrix (it is desired that the matrix is sparse, but the scheme works with any matrix) with m lines and n columns in the CSC format requires:

nnz	an integer variable storing the number of nonzero elements in the matrix
$pVals$	an array of length nnz with real elements to store the nonzero entries in the matrix
$pRowIndx$	an array of length nnz with integer elements such that the nonzero entry stored in $pVals[i]$ sits on the row $pRowIndx[i]$

$pColIndxB$	an array of length n with integer elements
$pColsIndxE$	an array of length n with integer elements the nonzero entries in the $i-th$ column of the matrix are stored in $pVals$ starting with position $pColIndxB[i]$ and ending with position $pColIndxE[i]-1$ Note: B stands for Beginning while E for End

We use the $C/C++$ convention that the positive integers start from 0 not from 1.

As an example, assume we want to use CSC format to store the matrix

$$\begin{bmatrix} 0.1 & 0 & 0 & 0.2 \\ 0 & 0.4 & 0 & 0 \\ 0 & 0.3 & 0.7 & 0.9 \end{bmatrix}$$

Then $nnz = 6$, $m = 3$ and $n = 4$;

$pVals$ has 6 elements and they are $pVals = \{0.1, 0.4, 0.3, 0.7, 0.2, 0.9\}$.

$pRowIndx$ has 6 elements and they are $pRowIndx = \{0, 1, 2, 2, 0, 2\}$ because 0.1 stands in row 0, 0.4 stands in row 1, 0.3 stands in row 2 etc.

$pColIndxB[0] = 0$ and $pColIndxE[1] = 1$ because $pVals[0] = 0.1$ is the first nonzero element in column 0 and it is also the last element.

$pColIndxB[1] = 1$ and $pColIndxE[1] = 3$ because $pVals[1] = 0.4$ is the first nonzero element in column 1 and $pVals[3-1] = 0.3$ is the last nonzero element in the column 1.

$pColIndxB[2] = 3$ and $pColIndxE[2] = 4$ because $pVals[3] = 0.7$ is the first nonzero element in column 2 and it is also the last element.

$pColIndxB[3] = 4$ and $pColIndxE[3] = 6$ because $pVals[4] = 0.2$ is the first nonzero element in column 3 is $pVals[6-1] = 0.9$ is the last nonzero element in column 3.

6.1 Low-level routines

The low-level routines are available through $YAS_blas.h$, $YAS_lapack.h$ and implemented in $YAS_blas.cpp$ and $YAS_lapack.cpp$.

Implementing routines as templates. *Template* is a C++ feature that we will use to, eventually, allow the switch between different data precision to be really simple. The first step to achieve this is to implement all low-level routines as templates.

We illustrate this concept on a simple example. Assume we need a function *max* that takes two arguments, x and y, of the same type and returns 0 if they can not be compared, 1 if the first one is the biggest and 2 otherwise. The type of the arguments can be a standard C++ type or user defined. If x and y are of type *double* it is enough to use $x < y$ to find out what to return. If x and y are strings and we use the "dictionary order" than the decision requires more work. By defining *max* as a template function, one can make the function's behavior to vary with the type of the arguments. The following code defines the function *max* as a template function. It implements its standard behavior and specialize its behavior if the argument's type is *char** and YAS_K_Block*, a user defined class.

$template < class\ type > int$ $\max(type *x, type *y)$

$\{if\ \ *x > *y\ return$ 1;

$else\ return$ 2;

```
}
int max(char *x, char *y)
{
/* different implementation */
}
int max(YAS_K_Block *x, YAS_K_Block *y)
{
/* another implementation */
}
```

Defining YAS low-level routines as templates, allows us to use implementations that depend on the type of the parameters.

For example, *YAS_gemm*, used to multiply two dense blocks, has different implementations for data stored as *double* and for data stored as *float*. This is conformal with the standard linear algebra libraries, such as BLAS, that use different routines for multiplication of matrices stored in *double* precision and for matrices stored in *float* precision, *dgemm* and *sgemm* respectively. This choice is motivated by both memory and speed concerns. The compiler checks the type of the parameters and use the corresponding code. If *YAS_gemm* will be called with data other than *double* or *float* the error routine will be called saying that nothing was done since the function is not overloaded for this type of data. It is possible to extend the *YAS_gemm* with implementations for other types of data precision such as *quad* precision. Such extensions require no modifications to the other parts of the solver as the user always calls *YAS_gemm*.

Switching between different implementations for the same rou-

tine. Each routine has one or more implementations. The user can switch between one or another using *macros*. This C++ feature is an elegant way to exclude or include parts of the code. We illustrate it on a simple example. If the following code is compiled

$\#define\ use_MKL$

$\#ifdef\ use_MKL$

instruction 1;

$\#endif$

then *instruction* 1 will be included. If we remove the line $\#define$ use_MKL the compiler will not include in the code *instruction* 1.

A macro is associated with each implementation of a given routine. These macros are collected in the first part of $YAS_blas.h$ signaled with a commented line as "Part A. Implementation to be used for each routine".

For example, the routine YAS_gemm, with data stored as *double*, has two implementations. One uses the MKL library and the other uses the ACML library. The user will see in Part A of $YAS_blas.h$, after the commented line /*—Implementation to be used by YAS_gemm. Choose one—*/, the following instructions:

$\#define_YAS_gemm_use_MKL_dbl$

and

$\#define_YAS_gemm_use_ACML_dbl$

The user will have to leave only one of these two macros active by commenting the other one. To add and use a different implementation, one has to define another macro in Part A of $YAS_blas.h$ and then add the desired implementation in $YAS_blas.c$.

The macros associated with different implementations for each routine are signaled in Part A of $YAS_blas.h$ by a commented line /*—Implementation to be used by <routine_name>. Choose one—*/. By convention, the name of each macro is <routine_name>_use_<implementation_name>.

Processors manufactures develop and support linear algebra libraries tuned for their products. For example, Intel develops the MKL package while AMD develops the ACML package. Any such library works on most processors but the claim is that the difference in the performance might be significant. Using the technique described above, most low-level routines have two implementations: one using MKL and the other using ACML.

Structure of the file YAS_blas.h and YAS_lapack.h. $YAS_blas.h$ and $YAS_lapack.h$ are both divided in three parts that are signaled with commented lines as Part A, Part B and Part C.

Part A. Implementation to be used for each routine was mostly discussed in Switching between different implementations for the same routine. In addition, this part is used to link to the libraries needed by each implementation. This is done again through macros.

As discussed, YAS_gemm has two implementations, one using the MKL library and the other the ACML library. Therefore, the following code can be seen

```
#ifdef _YAS_gemm_use_MKL_dbl
    #include <mkl.h>
#endif
#ifdef _YAS_gemm_use_ACML_dbl
    #include <acml.h>
```

$\#endif$

If the user creates a new implementation for YAS_gemm where he is using some library he needs to add

$\#ifdef\ _YAS_gemm_use_NewImplementation$

$\#include < userlibrary >$

$\#endif$

Part B. Debug settings for each routine. Each routine has parts of its code that is not changing its end goal and is needed only in some situations. For example, YAS_gemm has a part of the code that is checking whether the dimensions of the blocks are compatible and activates the error routine if something is wrong. This part of the code is contained under the macro $_YAS_gemm_SAFETY_ON$. The code is compiled only if this macro is defined.

The macros that contains such special parts of the code for each routines are defined here, in Part B.

Part C. Syntax for each routine. In this part, each routine is defined. The user can see not just the declaration of the template routine but also the declaration of the available overloads.

6.1.1 BLAS routines

These routines are available through $YAS_blas.h$ and implemented

in $YAS_blas.cpp$.

YAS_gemm

Source

Declared in $YAS_blas.h$. Implemented in $YAS_blas.cpp$.

Description

Performs matrix-matrix multiplication. If A, B and C are matrices with real entries, the operation is defined as $C := \alpha \cdot op(A) \cdot op(B) + \beta \cdot C$, given that $op(A)$, $op(B)$ and C have compatible dimensions and $\alpha, \beta \in \mathbb{R}$.

A, B and C are dense matrices.

$op(X)$ is either X or X^T

Syntax

YAS_gemm(*char* transa, *char* transb, $yasInt$ m, $yasInt$ n, $yasInt$ k, $templateT$ alpha, $templateT$ *a, $yasInt$ lda, $templateT$ *b, $yasInt$ ldb, $templateT$ beta, $templateT$ *c, $yasInt$ ldc)

Implementations

This template routine is overloaded for $templateT = double$ and $templateT = float$. The following table shows existing implementations for each type. For details about different implementations for one routine see Low-level routines. Introduction.

templateT	Impl. name	Macro	Details
double	MKL_dbl	$\#define_YAS_gemm_use_MKL_dbl$	this implementation uses *dgemm* routine from the MKL package
	$ACML_dbl$	$\#define_YAS_gemm_use_ACML_dbl$	this implementation uses *dgemm* routine from the ACML package
float	MKL_flt	$\#define_YAS_gemm_use_MKL_flt$	this implementation uses *sgemm* routine from the MKL package
	$ACML_flt$	$\#define_YAS_gemm_use_ACML_flt$	this implementation uses *sgemm* routine from the ACML package

Input parameters

transa — *char*; determines $op(A)$

if $transa =$'N' or 'n' then $op(A) = A$

if $transa =$'T' or 't' then $op(A) = A^T$

transb — *char*; determines $op(B)$

if $transb =$'N' or 'n' then $op(B) = B$

if $transb =$'T' or 't' then $op(B) = B^T$

m	$yasInt$ (see Appendix A)
	specifies the number of rows of $op(A)$, which is the same as the number of rows of C
n	$yasInt$ (see Appendix A)
	specifies the number of columns of $op(B)$ which is the same as the number of columns of C
k	$yasInt$ (see Appendix A)
	specifies the number of columns of $op(A)$ which is the same as the number of rows of $op(B)$
$alpha$	$templateT$
	specifies the scalar α
a	$templateT*$
	an array representing the columnwise storage of a dense block with lda rows and ka columns where $ka = k$ if $op(A) = A$ $ka = m$ if $op(A) = A^T$
lda	$yasInt$ (see Appendix A)
	specifies the leading dimension of A if $op(A) = A$ then lda must be at least $max(1, m)$ if $op(A) = A^T$ then lda must be at least $max(1, k)$

b	$templateT*$
	an array representing the columnwise storage of a dense block
	with ldb rows and kb columns where
	$kb = n$ if $op(B) = B$
	$kb = k$ if $op(B) = B^T$
ldb	$yasInt$ (see Appendix A)
	specifies the leading dimension of B
	if $op(B) = B$ then ldb must be at least $max(1, k)$
	if $op(B) = B^T$ then ldb must be at least $max(1, n)$
$beta$	$templateT$
	specifies the scalar β
c	$templateT*$
	an array representing the columnwise storage of a dense block
	with ldc rows and n columns
ldc	$yasInt$ (see Appendix A)
	specifies the leading dimension of C
	its value must be at least $max(1, m)$

Output parameters

c	overwritten by the m by n matrix $alpha \cdot op(A) \cdot op(B) + beta \cdot C$

YAS_CSC_gemm

Source

Declared in $YAS_blas.h$. Implemented in $YAS_blas.cpp$.

Description

Performs matrix-matrix multiplication. If A, B and C are matrices with real entries, the operation is defined as $C := \alpha \cdot op(A) \cdot B + \beta \cdot C$, given that $op(A)$, B and C have compatible dimensions and $\alpha, \beta \in \mathbb{R}$.

A is a sparse matrix stored in the CSC format; B, C are dense matrices.

$op(X)$ is either X or X^T

Syntax

*template < classT > int YAS_CSC_gemm(char transa, yasInt m, yasInt n, yasInt k, templateT alpha, templateT *pVals, yasInt *pRowIndx, yasInt *pColIndxB, yasInt* ColIndxE, yasInt nnz, templateT *b, yasInt ldb, templateT beta, templateT *c, yasInt ldc)*

Implementations

This template routine is overloaded for *templateT* = *double*. The following table shows existing implementations for each type. For details about different implementations for one routine see Low-level routines. Introduction.

templateT	Impl. name	Macro	Details
double	*MKL_dbl*	*#define_YAS _CSC_gemm _use_MKL_dbl*	this implementation uses *mkl_dcscmm* routine from the MKL package

Input parameters

transa *char*; determines $op(A)$

if $transa$ ='N' or 'n' then $op(A) = A$

if $transa$ ='T' or 't' then $op(A) = A^T$

m	$yasInt$ (see Appendix A)
	specifies the number of rows of $op(A)$, which is the same as the number of rows of C
n	$yasInt$ (see Appendix A)
	specifies the number of columns of B which is the same as the number of columns of C
k	$yasInt$ (see Appendix A)
	specifies the number of columns of $op(A)$ which is the same as the number of rows of B
$alpha$	$templateT$
	specifies the scalar α
$pVals$	$templateT*$
	see CSC sparse storage convention
$pRowIndx$	$yasInt*$
	see CSC sparse storage convention
$pColIndxB$	$yasInt*$
	see CSC sparse storage convention
$pColIndxE$	$yasInt*$
	see CSC sparse storage convention
nnz	$yasInt$
	see CSC sparse storage convention
b	$templateT*$
	an array representing the columnwise storage of a dense block with ldb rows and n columns where

ldb	$yasInt$ (see Appendix A)
	specifies the leading dimension of B
	ldb must be at least $max(1, k)$
$beta$	$templateT$
	specifies the scalar β
c	$double*$ or $float*$.
	an array representing the columnwise storage of a dense block
	with ldc rows and n columns
ldc	$yasInt$ (see Appendix A)
	specifies the leading dimension of C
	its value must be at least $max(1, m)$

Output parameters

c	overwritten by the m by n matrix $alpha \cdot op(A) \cdot op(B) + beta \cdot C$

6.1.2 LAPACK routines

These routines are available through $YAS_lapack.h$ and implemented in $YAS_lapack.cpp$.

YAS_ge_solv

Source

Declared in $YAS_lapack.h$. Implemented in $YAS_lapack.cpp$

Description

Solves a system of equations with a general square matrix and with multiple right hand side, $AX = B$.

A, B and X are stored as dense matrices.

Syntax

$template <class\, templateT> int\, YAS_ge_solv(yasInt\, m, yasInt\, nRHS, templateT^*\ a, yasInt\ lda, templateT^*\ b, yasInt\ ldb)$

Implementations

This template routine is overloaded for $templateT = double$ and $templateT = float$. The following table shows existing implementations for each type. For details about different implementations for one routine see Low-level routines. Introduction.

templateT	Impl. name	Macro	Details
$double$	MKL_dbl	$\#define_YAS_gemm_use_MKL_dbl$	this implementation uses $dgesv$ routine from the MKL package
$float$	MKL_flt	$\#define_YAS_gemm_use_MKL_flt$	this implementation uses $sgesv$ routine from the MKL package

Input parameters

m $yasInt$ (see Appendix A)
the dimension of the square matrix A

$nRHS$ $yasInt$ (see Appendix A)
the number of vectors on the right hand side; equals the number of columns of B

a	*templateT**
	array with lda rows and m columns
	A is stored in a as a dense matrix
lda	$yasInt$ (see Appendix A)
	specifies the leading dimension of A
	lda is at least max(1,m)
b	*templateT**
	array with ldb rows and n columns
	B is stored in b as a dense matrix
ldb	$yasInt$ (see Appendix A)
	specifies the leading dimension of B
	ldb is at least max(1,m)

Output parameters

returns 1 for success and 0 for failure

a	the data in a is lost
b	is overwritten with the solution

YAS_symm_PD_solv

Source

Declared in $YAS_lapack.h$. Implemented in $YAS_lapack.cpp$

Description

Solves a system of equations with a square symmetric and positive-definite matrix and with multiple right hand side, $AX = B$.

A, B and X are dense matrices.

Syntax

template <class templateT> int YAS_symm_PD_solv(yasInt m, yasInt nRHS, templateT a, yasInt lda, templateT* b, yasInt ldb)*

Implementations

This template routine is overloaded for *templateT* = *double* and *templateT* = *float*. The following table shows existing implementations for each type. For details about different implementations for one routine see Low-level routines. Introduction.

templateT	Impl. name	Macro	Details
double	*MKL_dbl*	*#define_YAS _symm_PD _use_MKL_dbl*	this implementation uses *dposv* routine from the MKL package
float	*MKL_flt*	*#define_YAS _symm_PD _use_MKL_flt*	this implementation uses *sposv* routine from the MKL package

Input parameters

m — *yasInt* (see Appendix A)
the dimension of the square matrix A

nRHS — *yasInt* (see Appendix A)
the number of vectors on the right hand side; equals the number of columns of B

a	*templateT**
	array with *lda* rows and *m* columns
	A is stored in *a* as a dense matrix
lda	*yasInt* (see Appendix A)
	specifies the leading dimension of *A*
	lda is at least max(1,m)
b	*templateT**
	array with *ldb* rows and *n* columns
	B is stored in *b* as a dense matrix
ldb	*yasInt* (see Appendix A)
	specifies the leading dimension of *B*
	ldb is at least max(1,m)

Output parameters

returns 1 for success and 0 for failure

a	the data in *a* is lost
b	is overwritten with the solution

YAS_triang_solv

Source

Declared in *YAS_lapack.h*. Implemented in *YAS_lapack.cpp*

Description

Solves a system of equations with a triangular matrix and with multiple right hand side, $AX = B$.

A, B and X are dense matrices.

Syntax

template <class templateT> int YAS_triang_solv(yasInt m, yasInt nRHS, char type, char unit, templateT a, yasInt lda, templateT* b, yasInt ldb)*

Implementations

This template routine is overloaded for *templateT* = *double* and *templateT* = *float*. The following table shows existing implementations for each type. For details about different implementations for one routine see Low-level routines. Introduction.

templateT	Impl. name	Macro	Details
double	*MKL_dbl*	*#define_YAS_triang_solv_use_MKL_dbl*	this implementation uses *dtrsm* routine from the MKL package
float	*MKL_flt*	*#define_YAS_triang_solv_use_MKL_flt*	this implementation uses *strsm* routine from the MKL package

Input parameters

m — *yasInt* (see Appendix A)
the dimension of the square matrix A

nRHS — *yasInt* (see Appendix A)
the number of vectors on the right hand side; equals the number of columns of B

type $char$; specifies if A is lower-diagonal or upper-diagonal

if $type$ ='U' or 'u' then A is upper-diagonal

if $transa$ ='L' or 'l' then A is lower-diagonal

unit $char$; specify if A is a unit matrix i.e. has only ones on the diagonal

if $unit$ ='u' or 'U' then A is unit

if $unit$ ='n' or 'N' then A is not unit

a $templateT*$

array with lda rows and m columns

a is a dense block (see Dense matrix storage convetion)

Note: A is completely stored in a even though A is triangular and some data is unnecesarry and, in fact, ignored

lda $yasInt$ (see Appendix A)

specifies the leading dimension of A

lda is at least max(1,m)

b $templateT*$

array with ldb rows and n columns

B is a dense block (see Dense matrix storage convetion)

ldb $yasInt$ (see Appendix A)

specifies the leading dimension of B

ldb is at least max(1,m)

Output parameters

returns 1 for success and 0 for failure

b	is overwritten with the solution

6.2 YAS_K_block

In what follows, by a block we refer to a matrix of a special type such as dense, sparse, symmetric, sparse symmetric, upper-diagonal, lower-diagonal etc.

An object from the class YAS_K_block acts like a container for a block, hiding its type but providing the user the methods for all linear algebra operations needed in the implementation of interior point algorithms. Once an object of type YAS_K_block is associated to a given block, the user doesn't have to worry about the actual type of the block. At the same time, in the backstage, both the storage and the linear algebra operations exploit the type of the block. For example, a n by n symmetric block requires the storage of $\frac{n(n+1)}{2}$ elements instead of n^2. Also, multiplying a sparse block with a dense block is usually faster than multiplying two dense blocks assuming the dimensions are the same.

Storing more blocks of the same type and dimension in one object from class YAS_K_block. The design of the class YAS_K_block is such that an object from this class can contain one or more blocks of a certain type and of the same dimension.

This can be done in several ways. Assume we start with k blocks of dimension n by m, all of the same type. The first way is to stack the blocks vertically and store the resulting big block. It has the same type but its dimension is m by $k \cdot n$. We refer to this as vertical stacking. The second way is to stack them horizontally and store the resulting big block. It has

the same type and its dimension is n by $k \cdot m$. We refer to this as horizontal stacking. Finally, the third way is to store each block independently. We refer to this as sequential stacking.

This feature, that apparently complicates the usage, pays off in different situations. For example, when a block is a vector and we need to multiply a matrix with several vectors. It is worth to have the vectors stored horizontally because then one can use one matrix-matrix multiplication which is faster than doing several matrix-vector multiplications.

Nevertheless, for simplicity, the user might decide that YAS_K_block object should contain only one block.

Template data structures defined to store blocks. These structures are available through $YAS_simple_defs.h$.

In YAS, a structure is defined for each (currently supported) type of block. The structure contains all the entries needed to store a block from that type. For example, for a dense block whose elements are of type *double*, the structure only requires a pointer to *double*.

Note that the structure do not contain the dimension of the block. This information will be available through the class YAS_K_block. Therefore, a dense matrix is not completely determined by the structure associate to it. The true container of a dense block is an object from the YAS_K_block class.

It is important that these structures are defined as template structures. Together with the low-level template routines they will allow the YAS_K_block and YAS_K_mb to be defined as template classes.

For convenience, we illustrate the concept of a template structure on the

yasDense structure. This structure is defined as

template <class Tfloat> struct yasDense $\{Tfloat^*\ pVals;\}$;

If the user is defining a variable as

$yasDense < double > p;$

then the type of $p.pVals$ is $double*$.

The currently available structures, all defined as templates, are listed and explained in Properties of YAS_K_Block.

YAS_K_Block implemented as a template class The following code is an example of a class that is implemented as a template. It is a container for a dense block. It can be seen as a simplified version of YAS_K_Block.

$template < class\ Tfloat > class\ dense_block$

$\{public:$

$int\ m; //\ number\ of\ lines$

$int\ n; //\ number\ of\ columns$

$Tfloat\ {*}data; //an\ array$ that can store $m \cdot n\ elements$

....

$\}$

The user can instantiate an object of type $dense_block < double >$ in which case the type of the property *data* is *double** and so the elements of the dense block are stored in *double* precision. If an object of type $dense_block <$ float $>$ is instantiate, the elements will be stored in float precision.

Template classes become really powerful when they make use of template routines and template structures. This is the case of the class YAS_K_block.

The implementation of YAS_K_Block as a template wouldn't be possible without template routines and structures.

The YAS_K_block class is declared as

$template \ < class\ Tfloat >$

$class\ YAS_K_block$

{

....

}

The reader will note that some properties and methods of the class involve the type $Tfloat$.

Objects of type $YAS_K_block < double >$ store the data in double precision while objects of type $YAS_K_block <$ float $>$ store the data in single precision. Currently these these two types are available. To allow objects of type $YAS_K_block < MyClass >$ the user has to make sure the low-level routines are overloaded for the type $MyClass$. In particular, extension to *quad* precision is subject of future work.

6.2.1 Properties of YAS_K_Block

In what follows, we list the the properties of YAS_K_Block. We specify the type of the property and describe its scope. In some cases, the description also contains an exhaustive list of values the properties can take as well as the name of the constants that were created especially to be used for this property.

type

Property type. *yasType* (see Appendix A)

Description. This property is uniquely determined by the type of the block(s) (dense, sparse etc) that is (are) contained in the object and the stacking method being used.

It can only take the values in the table below. Moreover, a constant is defined for each value in $YAS_simple_defs.h$.

Constant	Value	Description
$_yas_zero_block$	0	zero block
$_yas_meye_block$	1	multiple of identity block
$_yas_dense_block_ver$	100	vertically stacked dense block
$_yas_dense_block_hor$	101	horizontally stacked dense bloc
$_yas_dense_block_seq$	102	sequentially stacked dense block
$_yas_sparse_CSC_block_ver$	200	vertically stacked CSC sparse block
$_yas_sparse_CSC_block_hor$	201	horizontally stacked CSC sparse block
$_yas_sparse_CSC_block_seq$	202	horizontally stacked CSC sparse block

K,m,n

Property type. *yasInt*, *yasInt*, *yasInt* (see Appendix A)

Description. The number of stored blocks, the number of lines for each block and, respectively, the number of columns of each block. If the formulation "each block" is not clear see Storing more blocks of the same type and dimensions in one object from class YAS_K_block

pData,pDataDescription

Property type. $void*$, $void*$

Description. $pData$ and $pDataDescriptor$ are two pointers to a structure and, respectively,a type that are uniquely determined by the value stored in the *type* property (see Template data structures defined to store blocks). In what follows, for every possible value of the property $Type$, we show the value of $pData$ and $pDataDescriptor$ together with more details on how to use them.

$_yas_zero_block$	$pData = NULL$ $pDataDescription = NULL$
$_yas_meye_block$	$pData = *Tfloat$ (see YAS_K_Block implemented as a template class) $pDataDescription = NULL$

	Usage.If *pData=NULL* then the block is simply identity Otherwise, the scalar multiplying the identity is stored in a variable of type $Tfloat$ and $pData$ should point to it
$_yas_dense_block_???$	$pData = *yasDense < Tfloat >$ $pDataDescription = NULL$ **Recall**. $template < classTfloat >$ $struct\ yasDense\{Tfloat\ *pVals;\}$

Usage. If ??? $= ver$ or ??? $= hor$ then the blocks are stacked vertically and, respectively, horizontally into one big dense block which is stored into a $yasDense < Tfloat >$structure and $pData$ points to this structure.

(to see how to store a dense block into a $yasDense$ structure see Appendix A. YAS defined types.)

if ??? $= seq$ then there exists a pointer $yasDense < T\text{float} >{*}p = new\ yasDense < T\text{float} > [K]$ and $pData = p$;

The first block is stored in $*pData$, the second in $*(pData + 1)$ etc

$_yas_sparse_CSC_block_???$ $pData = *yasSparseCSC < Tfloat >$ $pDataDescription = *yasSparseCSCDescriptor$

Recall. 1. $struct\ yasSparseCSC\{Tfloat\ *pVals; yasInt\ *pRowIndx; yasInt\ *pColIndxB; yasInt\ *pColIndxE; yasInt\ nnz;\};$

2. $typedef\ yasInt\ yasSparseCSCDescr;$

Usage. If ??? $= ver$ or ??? $= hor$ then the blocks are stacked vertically and, respectively, horizontally into one big sparse block which is stored into a $yasSparseCSC < Tfloat >$ structure to which $pData$ points.
Usually the length of $pVals$ is nnz but the user can choose to allocate more memory then required by nnz. This available length is stored in a variable of type $yasSparseCSCDescr$ to which $pDataDescription$ points.

(to see how to store a sparse block into a $yasSparseCSC$ structure see Appendix A)

if ??? $= seq$ then there exists a pointer $yasSparseCSC < Tfloat > *p = new$ $yasSparseCSC < Tfloat > [K]$ and $pData = p$;

also, a pointer $yasSparseCSCDescr$ $*q = new\ yasSparseCSCDescr[K]$ exists and $pDataDescription = q$;

The first block is stored in $*pData$, the second in $*(pData + 1)$ etc

For the first block $(*pData).pVals$ might refer to a memory bigger than the one required by $(*pData).nnz$. This bigger value is stored in $*pDataDescription$

Similar, for the second block the total available memory is stored in $*(pDataDescription + 1)$

errCode

Property type. int

Description. the methods of the YAS_K_block are using this vari-

able to store the error code in case some error is encountered during the execution.

storeTranspose

Property type. char

Description. it is set to 1 then the object contains the transpose of the blocked effectively stored in *pData*.

6.2.2 Methods of YAS_K_block

For convenience, the methods are grouped into: member access methods, interface unification methods, data managing methods and linear algebra methods.

Member access methods

The following methods should be used to read the properties of the class.

Syntax	**Property returned**
yasType GetType()	*type*
yasInt GetK()	*K*
yasInt GetM()	*N*
void GetPData*()	*pData*
void GetPDataDescriptor*()	*pDataDescriptor*
char GetStoreTranspose()	*storeTranspose*
int GetErrCode	*errCode*

The following methods should be used to write the properties of the class.

Syntax	**Property set**
void SetType(yasType typeB)	*type*
void SetK(yasInt KB)	*K*
void SetN(yasInt nB)	*N*
void SetData(void pDataB)*	*pData*
void SetPDataDescriptor(void pDataDescriptorB)*	*pDataDescriptor*
void SetStoreTranspose(char storeTransposeB)	*storeTranspose*

Interface unification methods

The reader might want to skip these methods for now and come back to them after looking at the *YAS_K_mb*. Except for *SetCDEUpdate*, these methods have no practical importance for *YAS_K_block*, the user doesn't need them when working with blocks. They are implemented only to make *YAS_K_mb*, *YAS_K_block* and the classes derived from them have the same interface. The reason for this is explained in Overview. LA Layer.

Syntax	**Description**
yasInt GetBlockM()	returns 1
void SetBlockM(yasInt MB)	doesn't do anything
yasInt GetBlockN()	returns 1
void SetBlockN(yasInt NB)	doesn't do anything
yasInt GetActiveBlockI()	returns 1
void SetActiveBlockI(yasInt IB)	doesn't do anything
yasInt GetActiveBlockJ()	returns 1
void SetActiveBlockJ(yasInt JB)	doesn't do anything

In additions to the methods in the table above, we have

GetCDEUpdate

Syntax. $void\, GetCDEUpdate(yasInt\&\, rU, Tfloat\, {*}pAlpha, YAS_K_block$ $*pBlockC$, YAS_K_block $*pBlockD$, YAS_K_block $*pBlockE)$

Description. this method doesn't do anything in this class. See the derived class of YAS_K_block

SetCDEUpdate

Syntax. $void\, SetCDEUpdate(yasInt\, U, Tfloat\, {*}pAlpha, YAS_K_block$ $*pBlockC$, YAS_K_block $*pBlockD$, YAS_K_block $*pBlockE)$

Description. This method does $*this :=*this + \sum_{i=1}^{U} \alpha_i \cdot C_i \cdot D_i \cdot E_i$

U stores the number of updates

$\alpha_i =*(pAlpha + i)$

$C_i =*(pBlockC + i)$

$D_i =*(pBlockD + i)$

$E_i =*(pBlockE + i)$

Data managing methods

Before using an object of type YAS_K_block, the user needs to load it with real data, i.e. matrices.

Using the data managing methods, the user can allocate memory, delete allocated memory and copy data.

Note that the methods do not work with the data contained in the block. In particular, the type of the block is not affecting the way these methods work.

NewData

Syntax. *void* NewData*($yasType$& $rTypeB$, $yasInt\ KB$, $yasInt\ mB$, $yasInt\ nB$, *void** $pDataDescriptorB = NULL$)

Description. Allocates data storage and returns its pointer. Use descriptor if additional information is needed. If operations fails, $rTypeB$ is set to *_yas_no_type*.

DeleteData

Syntax. *int DeleteData*(*void** $pDataB$, $yasType\ TypeB$, $yasInt\ KB = 0$, $yasIntmB = 0$, $yasIntnB = 0$, *void** $pDataDescriptorB = NULL$);

Description. DeleteData frees data storage pointed to by pDataB.

NewDataDescriptor

Syntax. *void* NewDataDescriptor*($yasType$& $rTypeB$, $yasInt\ KB$, $yasInt\ mB$, $yasInt\ nB$);

Description. Allocates data descriptor and returns its pointer.

DeleteDataDescriptor

Syntax. *int DeleteDataDescriptor*(*void** $pDataDescriptorB$, $yasType$ $typeB$, $yasInt\ KB$, $yasInt\ mB = 0$, $yasInt\ nB = 0$);

Description. DeleteDataDescriptor frees data descriptor pointed to by pDataDescriptorB.

CloneData

Syntax.*virtual void* CloneData*($yasType$& $rTypeB$, $yasInt\ kStart$, $yasInt\ kEnd$,$yasInt\ iStart$, $yasInt\ iEnd$, $yasInt\ jStart$, $yasInt\ jEnd$,

void pDataDescriptorB = NULL*);

Description. CloneData attempts to clone (part of) data from *pData* with a suggested type conversion to rTypeB. If the cloning is unsuccessful, the function returns a *NULL* pointer and the rTypeB is changed to *_yas_no_type*; the error code is set accordingly. If the cloning is successful, the function returns a valid pointer to a new copy of the (portion of) the data.

CloneDataDescriptor

Syntax.*virtual void* CloneDataDescriptor*(*yasType& rTypeB*, *yasInt kStart*, *yasInt kEnd*, *yasInt iStart*, *yasInt iEnd*, *yasInt jStart*, *yasInt jEnd*);

Description. *CloneDataDescriptor* attempts to clone (part of) data descriptor with a suggested type conversion to rTypeB. The function behavior is consistent with CloneData.

CopyData

Syntax.*virtual int CopyData*(*void *pDataB*, *yasType typeB*, *yasInt kStart*, *yasInt kEnd*, *yasInt iStart*, *yasInt iEnd*, *yasInt jStart*, *yasInt jEnd*, *void* pDataDescriptorB = NULL*);

Description. CopyData copies (part of) data from *pData* to *pDataB* and performs type conversion to typeB. The difference between, copy and clone is that the function assumes *pDataB* is already allocated.

CopyDataDescriptor

Syntax. *virtual int CopyDataDescriptor*(*void *pDataDescriptor*, *yasType*

$typeB$, $yasInt\ kStart$, $yasInt\ kEnd$, $yasInt\ iStart$, $yasInt\ iEnd$, $yasInt\ jStart$, $yasInt\ jEnd$);

Description. CopyDataDescriptor copies (part of) data descriptor and performs type conversion to typeB.

Linear algebra methods

GetBkij

Syntax. $virtual\ Tfloat\ GetBkij(yasInt\ kB, yasInt\ iB, yasInt\ jB)$;

Description. Returns the (i, j)-element of the k-th block.

SetBkij

Syntax.$virtual\ int\ SetBkij(Tfloat\ val, yasInt\ kB, yasInt\ iB, yasInt\ jB)$

Description. Sets $(i, j) - element$ of the k-th block.

Transpose

Syntax. $virtual\ int\ Transpose(YAS_K_block\&\ rBlockB)$

Description. The method transposes the block data $rBlockB$, without changing block stacking structure.

Sum

Syntax. $virtual\ int\ Sum(YAS_K_block\&\ rBlockA, YAS_K_block\&\ rBlockB, Tfloat\ alpha = (Tfloat)1)$

Description. Computes the block-Kronecker sum $*this := alpha \cdot rBlockA \oplus rBlockB$.

For example, if $rBlockA$ contains k_A blocks $A_1, ..., A_{k_A}$ and $rBlockB$ contains k_B blocks $B_1, ...B_{k_B}$ then $*this$ will contain the $k_A \cdot k_B$ blocks $alpha \cdot A_1 + B_1$, $alpha \cdot A_1 + B_2, ...$, $alpha \cdot A_1 + B_{k_B}$, $alpha \cdot A_2 + B_1, ...$, $alpha \cdot A_2 + B_{k_B}, ...$, $alpha \cdot A_{k_A} + B_1, ...$, $alpha \cdot A_{k_A} + B_{k_B}$.

If $this- > type = _yas_no_type$, the resulting type is assigned automatically, otherwise the result is converted to the prescribed type.

Prod

Syntax.$virtual\ int\ Prod(YAS_K_block\&\ rBlockA,\ YAS_K_block\&\ rBlockB,\ Tfloat\ alpha = (Tfloat)1,\ Tfloat\ beta = (Tfloat)0)$

Description. Computes the block-Kronecker product $*this := alpha \cdot rBlockA \odot rBlockB + beta \cdot (*this)$

For example, if $rBlockA$ contains k_A blocks $A_1, ..., A_{k_A}$ and $rBlockB$ contains k_B blocks $B_1, ...B_{k_B}$ then $*this$ will contain the $k_A \cdot k_B$ blocks : $A_1 \cdot B_1 + beta \cdot (*this)_{11}$, $A_1 \cdot B_2 + beta \cdot (*this)_{12}$, ..., $A_1 \cdot B_{k_B} + beta.(*this)_{1k_B}$, $A_2 \cdot B_1 + beta \cdot (*this)_{21}, ...$, $A_2 \cdot B_{k_B} + beta \cdot (*this)_{2k_B}, ...$, $A_{k_A} \cdot B_1 + beta \cdot (*this)_{k_A 1}$, ... , $A_{k_A} \cdot B_{k_B} + beta \cdot (*this)_{k_A k_B}$.

If $this- > type = _yas_no_type$, the resulting type is assigned automatically, otherwise the result is converted to the prescribed type.

Note that if $beta \neq 0$ then $*this$ must contain $k_A \cdot k_B$ blocks.

DotSum

Syntax.$virtual\ int\ DotSum(YAS_K_block\&\ rBlockA, YAS_K_block\&\ rBlockB,\ Tfloat\ alpha = (Tfloat)1)$

Description. Computes the block–dot sum $*this := alpha \cdot rBlockA$.+ $rBlockB$

Note that $rBlockA$ and $rBlockB$ must contain the same number of blocks.

For example, if each of $rBlockA$ and $rBlockB$ contains k blocks $A_1, ..., A_k$ and, respectively, $B_1, ...B_k$ then $*this$ will contain the k blocks $alpha \cdot A_1 + B_1, alpha \cdot A_2 + B_2, ..., alpha \cdot A_k + B_k$.

If $this- > type = _yas_no_type$, the resulting type is assigned automatically, otherwise the result is converted to the prescribed type.

DotProd

Syntax.$virtual\ int\ DotProd(YAS_K_block\&\ rBlockA, YAS_K_block\&\ rBlockB,\ Tfloat\ alpha = (Tfloat)1,\ Tfloat\ beta = (Tfloat)0)$

Description. Computes the block–dot product $*this := alpha{\cdot}rBlockA\ .\cdot\ rBlockB + beta \cdot (*this)$

Note that $rBlockA$ and $rBlockB$ must contain the same number of blocks.

Note that if $beta \neq 0$ then $*this$ must contain the same number of blocks as $rBlockA, rBlockB$.

For example, if each of $rBlockA$ and $rBlockB$ contains k blocks $A_1, ..., A_k$ and, respectively, $B_1, ...B_k$ then $*this$ will contain the k blocks $alpha \cdot A_1 \cdot B_1 + beta \cdot (*this)_1, alpha \cdot A_2 \cdot B_2 + beta \cdot (*this)_2, ..., alpha \cdot A_k \cdot B_k + beta \cdot (*this)_k$.

If $this- > type = _yas_no_type$, the resulting type is assigned automatically, otherwise the result is converted to the prescribed type.

ProdBBc

Syntax. $virtual\ int\ ProdBBc(YAS_K_block\&\ rBlockB, Tfloat\ alpha =$

$(Tfloat)1, Tfloat\ beta = (Tfloat)0)$

Description. Computes block-dot product $*this := alpha \cdot rBlockB\ .\cdot (rBlockB)^C + beta \cdot (*this)$.

Note that if $beta \neq 0$ then $this$ must contain the same number of blocks as $rBlockB$.

If $this- > type = _yas_no_type$, the resulting type is assigned automatically, otherwise the result is converted to the prescribed type.

ProdBcB

Syntax.$virtual\ int\ ProdBcB(YAS_K_block\&\ rBlockB, Tfloat\ alpha = (Tfloat)1, Tfloat\ beta = (Tfloat)0)$

Description. Computes block-dot product $*this := alpha \cdot (rBlockB)^C .\cdot (rBlockB) + beta \cdot (*this)$.

Note that if $beta \neq 0$ then $this$ must contain the same number of blocks as $rBlockB$.

If $this- > type = _yas_no_type$, the resulting type is assigned automatically, otherwise the result is converted to the prescribed type.

LU

Syntax.$virtual\ int\ LU(YAS_K_block\&\ rBlockA, YAS_K_block\&\ rBlockB, yasType\ method)$

Description. Performs LU-factorization of $*this$ with a method of choice.

The results are stored in $rBlockA := L$ and $rBlockB := U$

If a LU decomposition update is to be computed, according to a method of choice, one would pass the necessary data via $rBlockA$ and $rBlockB$.

QR

Syntax. *virtual int QR(YAS_K_block& rBlockA, YAS_K_block& rBlockB, yasType method)*

Description. Performs QR-factorization of $*this$.

The results are stored in $rBlockA := Q$ and $rBlockB := R$.

If a QR decomposition update is to be computed, according to a method of choice, one would pass the necessary data via $rBlockA$ and $rBlockB$.

Cholesky

Syntax. *virtual int Cholesky(YAS_K_block& rBlockA, yasType method)*

Description. Performs Cholesky-factorization of $*this$.

The result is stored in $rBlockA := L$.

If a Cholesky decomposition update is to be computed, according to a method of choice, one would pass the necessary data via $rBlockA$.

Inverse

Syntax. *virtual int Inverse(YAS_K_block& rBlockB, yasType method)*

Description. Attempts to compute the block inverse, if block is square. The result is put into $*this$.

Solve

Syntax. *int Solve(YAS_K_block& rBlockA, YAS_K_block& rBlockB, yasType method)*

Description. Solves a system of linear equations with a chosen method.

The result of $rBlockA \backslash rBlockB$ is stored in $*this$.

6.3 YAS_K_mb

Just as in the case of YAS_K_block, an object from the class YAS_K_mb acts like a container for a matrix. Recall that objects of type YAS_K_block are containers for matrices of special types such as dense, sparse, symmetric, upper-diagonal, lower-diagonal etc. The class YAS_K_mb is more sofisticated then YAS_K_block allowing the user to store a matrix which doesn't have a special type as a whole but can be spitted into blocks of special types. For example, the following matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

can be considered to be a sparse or dense matrix but, at the same time, can be seen as a matrix of blocks, the first block of type diagonal and the second block of type dense. This viewpoint is particularly helpful when the size of each block is big.

Note that in the name of the class mb stands for matrix of blocks.

YAS_K_block and YAS_K_mb have the same interface. Recall that our goal is to have the same interface for YAS_K_mb and YAS_K_block, i.e. have the same properties and methods. The benefits of this feature are explained in Overview.

Active block in an object of type YAS_K_mb. Among all blocks that make a matrix stored in an object of type YAS_K_mb, at a given time just one block is considered active. The member access methods are acting on the active block.

Recall that to guarantee the same interface for YAS_K_block and YAS_K_mb we had to add to YAS_K_block the group of Interface unification methods.

The action of those methods are changing as follows:

Syntax	**Description**
$yasInt\ GetBlockM()$	returns the number of blocks in a row
$void\ SetBlockM(yasInt\ MB)$	set the number of blocks in a row
$yasInt\ GetBlockN()$	returns the number of blocks in a column
$void\ SetBlockN(yasInt\ MB)$	set the number of blocks in a column
$yasInt\ GetActiveBlockI()$	returns the row coordinate of the active block
$void\ SetActiveBlockI(yasInt\ IB)$	sets the row coordinate of the active block
$yasInt\ GetActiveBlockJ()$	returns the column coordinate of the active block
$void\ SetActiveBlockJ(yasInt\ JB)$	sets the column coordinate of the active block

7 Appendix A. YAS defined types

In this part, we describe the YAS defined types. They are available through $YAS_simple_defs.h$ and are used throughout the solver.

The first column of the table contains the name of the type. The second

column contains its the declaration, when is it used and some remarks.

yasType	$typedef\ int\ yasType$ holds positive integer values used to specify the type of a block see the property *type* in the class YAS_K_block by default is set to be *int* but can be changed to any other C++ integer types
yasInt	$typedef\ int\ yasInt$ holds integer values; used throughout the solver for variables such as the number of lines/columns of a block/matrix of blocks. by default is set to be *int* but can be changed to any other C++ integer types
yasSparseCSC	$template\ < class\ Tfloat > struct\ yasSparseCSC\ \{Tfloat$ $*pVals; yasInt\ *pRowIndx; yasInt\ *pColIndxB;$ $yasInt\ *pColIndxE; yasInt\ nnz; \};$ This structure is a template structure (see YAS_K_block → Introduction for motivations and use of template classes) Its variables are enough for storing a matrix in CSC sparse format. (see CSC sparse storage convention for more details)

yasDense $template < classTfloat > struct\ yasDense\{Tfloat$

$*pVals;\}$

This structure is a template structure

(see YAS_K_block → Introduction for motivations and

use

of template classes)

$pVals$ is used to store, columnwise, the elements of the

matrix.

(see Dense matrix storage convention)

Note that $yasDense$ is not used to store a dense block.

It only stores its elements. Use YAS_K_block to store

a dense block.

Part III

Optimization in IMRT

8 Problem formulation

The goal of this part is to formulate an optimization problem that is relevant for radiation therapy and to get an insight on the computational difficulties that result. A main concern is how to model the problem. We present the model from [3].

Radiation therapy (RT) is used in cancer treatment to control the development of malignant tumors by exposing them to ionizing radiation beams. As all tissues are affected during the process, the main issue in RT is how to spare the healthy ones while doing as much damage as possible to the tumor.

Intensity modulated radiation therapy (IMRT) is an advanced type of radiation therapy that allows high-precision control of the angle and intensity of the beams. Treatment planning is the process during which these angles and intensities are determined such that a prescribed quantity of radiation is delivered to the tumor and, at the same time, the radiation delivered to the vital organs is kept below a critical level. Also, the fact that healthy tissues recover faster than tumor tissues is exploited by fractionating the treatment.

Damage to vital organs is unavoidable for most types of cancer. For example, for prostate cancer, although vital organs such as the heart or the liver can easily be avoided, the blader is in the immediate vicinity of the prostate. These organs are the main challenges in treatment planning. Sparing them is made especially difficult due to inevitable movements inside the body which introduce uncertainties. For example, periodic breathing, cardiac motion, changes in intra-abdominal pressure as well as weight changes over the course of treatment are controllable to a small extend.

Different ways to deal with the uncertainties were suggested and compared in the literature. In what follows, we present the model from [3] and the resulting optimization problem. We will mention the assumptions of this model but not the inconveniences, some not obvious, they cause. One such inconvenience is that some assumptions are simplifying the real situation. In many cases, they are justified by the need to produce a computationaly tractable model. For all details, see [3].

With the use of a computed tomography (CT), the planner can visualize the location and the size of the tumor as well as the surrounding tissues. Assume there are K types of healthy tissues. The region on the CT where the tumor can be seen is caller the gross tumor volume (GTV). Next, the planner is identifying the clinical targeted volume (CTV) which is a region consisting of GTV and additional areas suspected to be affected and requiring treatment. The CT is discretized into voxels and so each voxel is either in the tumor or in one of the K healthy tissues. Denote with N the total number of fractions when radiation is applied.

Assumption 1. The beam angles have been preselected by an experi-

enced planner, so the task is to assign the intensity of every beam i.e. decide the vector x where each component of x is the intensity of one beam.

Assumption 2. The uncertainty is modeled by assuming that on a single fraction, one of the n possible scenarios $s_1, ..., s_n$ can occur with probability $p_1, ..., p_n$, respectively.

Let a_{ij} be a column vector that denotes the deterministic dose delivered to voxel i in scenario j when all beams have intensity 1. Denote

$$A_i := \begin{pmatrix} a_{i,1}^T \\ ... \\ a_{i,n}^T \end{pmatrix} \tag{19}$$

Denote with $D_i(x)$ the total dose delivered to voxel i during all N fractions for the given intensity vector x. Denote with $D_{il}(x)$, $l = 1, .., N$, the dose delivered to voxel i in the lth fraction.

Assumption 3. The cumulative dose $D_i(x)$ delivered to a voxel during the treatment is linearly additive i.e. $D_i(x) = \sum_{k=1}^{N} D_{il}(x)$.

$D_i(x)$ and $D_{ij}(x)$ are random variables since the position of the voxel i is uncertain. In great generality, a sum of random variables is normally distributed and so:

Assumption 4. $D_i(x)$ is normally distributed with mean $\mu(x, i)$ and variance $\sigma^2(x, i)$.

Assumption 5. For fixed i, the N random variables $D_{i1}(x), ..., D_{iN}(x)$ are independent and identically distributed.

It follows that

$$\begin{aligned} \mu(x,i) &= N \cdot E[D_{i1}(x)] \\ &= N \cdot \sum_{j=1}^{n} p_j a_{ij}^T x \\ &= N \cdot (p^T A_i x) \end{aligned} \tag{20}$$

and

$$\begin{aligned} \sigma^2(x,i) &= N \cdot Var[D_{i1}(x)] \\ &= N \cdot [A_i x - e(p^T A_i x)]^T P [A_i x - e(p^T A_i x)] \\ &= N \cdot [(I - ep^T) A_i x]^T P [(I - ep^T) A_i x] \\ &= N \cdot ||R A_i x||^2 \end{aligned} \tag{21}$$

where, in the formula above, $e = (1, 1, ..., 1)$, $P = diag(p_1, ..., p_n)$ and $R = P^{1/2}(I - ep^T)$.

$D_{il}(x) = a_{i,S(l)}^T x$ where $S(l)$ is the index of the scenario that occurs in fraction l.

Constraints controling the bounds on the total dose per voxel. Suppose that voxel i belongs to a healthy structure H_k. To protect the H_k, physicians require that the dose $D_i(x)$ does not exceed some level m_k. If the voxel belongs to the tumor, it is also required that the dose is above a certain constant.

For this we fix a δ and require that $P(D_i(x) > m_k) \leq \delta$. Equivalently,

$$P\left(\frac{D_i(x) - \mu(x,i)}{\sigma(x,i)} > \frac{m_k - \mu(x,i)}{\sigma(x,i)}\right) \leq \delta$$

By Assumption 4 we get

$$\frac{m_k - \mu(x,i)}{\sigma(x,i)} \geq z_{1-\delta}$$

where $z_{1-\delta}$ is uniquely defined from $P(Z > z_{1-\delta}) = \delta$ with Z being normally distributed with mean 0 and variance 1. Using (20) and (21) we obtain the following second-order constraint:

$$||RA_i x|| \leq \frac{m_k - Np^T A_i x}{z_{1-\delta}\sqrt{N}} \tag{22}$$

In a completely analogous way, one can deal with constraints that require $D_i(x)$ to be above a minimum level m_k. We obtain that $P(D_i(x) < m_k) \leq \delta$ is equivalent with

$$||RA_i x|| \leq \frac{Np^T A_i x - m_k}{z_{1-\delta}\sqrt{N}} \tag{23}$$

Remark 8.1 *In [3] it is showed that the same constraints (22) and (23) are arising if one is modeling the uncertainty using robust linear programming. We briefly describe this process in what follows.*

In a pure deterministic model i.e. where $D_i(x)$ are not random variables, asking the total dose on voxel i to be below a critical value m_k is equivalent with asking that $D_i(x) := a_i^T x \leq m_k$. Here a_i is the vector whose component j is the dose received by voxel i if the $j-th$ beam is used with intensity 1.

In the robust linear model, the constraint $a_i^T x \leq m_k$ is replaced by a

family of constraints $a^T x \leq m_k$ *where* a *belongs to some uncertainty set* U*. In other words, we don't know the precise value of* a *and so we require* $a^T x \leq m_k$ *to hold for all* a*'s in some set* U*. If the set* U *is the ellipsoid* $\{a_i^* + W_i u_i : ||u_i|| \leq 1\}$ *and* $W_i = z_{1-\delta} A_i^T R^T / \sqrt{N}$ *then the following is true:*

$$\{x : a^T x \leq m_k \ \forall a \in U\} = \left\{ x : ||RA_i|| \leq \frac{m_k - Np^T A_i x}{z_{1-\delta}\sqrt{N}} \right\}$$

In other words, we have replaced a family of linear constraints with one second-order constraint.

DV constraints. Another type of constraints required by physicians are the so called dose-volume (DV) constraints which are of the form "no more than $100v_k$% of healthy structure H_k may receive more than d_k units of radiation (Gy)". An exact way to model this is to introduce for each voxel a binary variable but this makes the problem computationally intractable. A approximate way to model this is to add the constraint

$$\sum_{i \in H_k} (Np^T A_i x - d_k)_+ \leq g_k \tag{24}$$

where $(\cdot)_+$ is the positive part and g_k is a parameter chosen by the planner. Such a constraint can be reformulated as an equality constraint.

DVH constraints. Assume a DV constraint combined with a constraint on the upper bound on the total dose per voxel says that for the healthy tissue H_k no voxel can receive more than 70 Gy and not more than 40% can receive more than 50Gy. The inconvenience with this formulation is that it allows a treatment where 39% of all voxels receive 69 Gy and the rest receive 49 Gy. Such situations should be avoided. For this purpose one could use

more than one DV constraint. For example, no voxel can receive more than 70 Gy, no more than 40% can receive more than 50 Gy, no more than 30% can receive more than 55 Gy, no more than 20% can receive more than 60 Gy. In fact, there exists the so called dose-volume histograms (DVH) that specify for each real number between 0 and 100 the maximum allowed dose. A DVH is determined for each healthy organ. Accomodating all constraints in a DVH is equivalent with adding an infinite number of constraints. In practice, DVH constraints are replaced by the so-called $gEUD$ constraints, introduced in [12]. For every $a \in \mathbb{R}$, the $gEUD_a$ constraint for the healthy organ H_k is defined as

$$\begin{cases} \frac{1}{|H_k|} \sum\limits_{i \in H_k} [D_i(x)]^a \leq m_k^a \text{ if } a \in (-\infty, 0] \cup [1, \infty) \\ \frac{1}{|H_k|} \sum\limits_{i \in H_k} [D_i(x)]^a \geq m_k^a \text{ if } a \in [0, 1] \end{cases}$$

where $|H_k|$ is the number of elements in H_k. Note that these are convex constraints. For good results, such constraints are introduced for different values of a. We recognize here the p-cone type constraints, see Chapter 4. Another way to model the DVH constraints is described in ([3]) and give rise to second-order cone constraints.

Problem formulation. We are ready to state an optimization problem that is incorporating the DV and the total dose per voxel constraints presented above. For simplicity, we start with a problem that incorporates only the bounds on the total dose received by each voxel.

$$
\begin{cases}
\min w_{T\min} r_{T\min} + w_{T\max} r_{T\max} + \sum_{j=1}^{n} w_{T_j} r_{T_j} + \sum_{k=1}^{K} w_k r_k \\
||RA_i x|| \leq \frac{Np^T A_i x - u_{T\min}}{z_{1-\delta}\sqrt{N}}, \forall i \in CTV \\
m_{T\min} - u_{T\min} \leq r_{T\min} \\
||RA_i x|| \leq \frac{u_{T\max} - Np^T A_i x}{z_{1-\delta}\sqrt{N}}, \forall i \in CTV \\
u_{T\max} - m_{T\max} \leq r_{T\max} \\
a_{ij}^T x \geq u_{T_j}, \forall i \in CTV,\ j = 1, .., n \\
m_T - u_{T_j} \leq r_{T_j}\ ,\ j = 1, ..., n \\
||RA_i x|| \leq \frac{u_k - Np^T A_i x}{z_{1-\delta}\sqrt{N}}, \forall i \in H_k, k = 1, .., K \\
u_k - m_k \leq r_k, k = 1, .., K \\
r_{T\min}, r_{T\max} \geq 0 \\
r_{T_j} \geq 0,\ j = 1, ..., n \\
r_k \geq 0,\ k = 1, ..., K \\
x \geq 0
\end{cases}
\tag{25}
$$

Note that in (25) $w_{T\min}, w_{T\max}, w_{T_j}, w_k$ are weights that penalize the failure to reach the minimum total dose for CTV, to exceed the maximum total dose $m_{T\max}$ for CTV, to reach the minimum dose m_T in scenario j for CTV and to exceed the maximum dose m_k for structure H_k, respectively.

The DV constraint (24) for the healthy tissue H_k is introduced in the problem by adding to the objective $\overline{w_k} q_k$, with $\overline{w_k}$ a penalty weight and q_k a variable, and the following constraints:

$$\begin{cases} \sum_{i \in H_k} (Np^T A_i x - d_k)_+ \leq f_k \\ f_k - g_k \leq q_k \\ q_k \geq 0 \end{cases}$$

Conclusions. Regarding the computational difficulty of the problems above we make the following remarks.

1. We have seen that when the DVH constraint is modeled through $gEUD$ the resulting problem has $p-$cone constraints. This motivates our interest in algorithms for conic optimization problems for nonsymmetric cones like the one by Nesterov that was described in Chapter 2. In particular, we are interested in efficient barriers for the p-cones like the one due to Nesterov from Chapter 4.

2. The A_i matrices in (19) are almost fully dense matrices since all the beams together reach almost all voxels. When they are combined, the big matrix is nearly dense with aproximately 40% nonzero elements. Moreover, it is made of completely dense blocks and totally sparse blocks such as diagonal blocks.

3. The size of the problem is determined by the number of beams and on how fine is the discretization. The two lead to large scale optimization problems. For example, 1000 beams, approximately 4000 voxels that are resulting from a very coarse discretization, where the side of each voxel is 1 cm, and 8 different scenarios result in a problem with $4000 \cdot 9 = 36.000$ variables and 4000 cones.

4. Due to Remarks 2 and 3, the optimization problem that we have presented can not be solved by existing solvers such as SeDuMi, SDPT3,

DSDP etc. A solver that allows tuning the linear algebra and is capable to handle nonsymmetric cones is essential.

9 A MATLAB prototype

In the previous section we have described an optimization problem relevant to IMRT treament planning and we have emphasized that the matrix defining the linear constraints is nearly dense with approximately 40% nonzero elements. Also, a coarse discretization ussualy creates instances with approximately 5000 positive variables and 4000 second order cones of dimension 9.

We have created an instance with 5458 positive variables, 3395 second order cones of dimension 9 and a constraint matrix with 38% nonzero elements and 1245 rows. Both SeDuMi [16] and SDPT3 [19] crash during the first iteration printing "Out of Memory". The test was run on an Intel Core (TM) 2 CPU 6600 @2.4 GHz computer with 2GB of RAM memory.

Therefore, we have written a MATLAB implementation of Nesterov's primal-dual algorithm for nonsymmetric cones that was described in Section 2.6. We have followed two objectives. First, we wanted a prototype to be used later when all the tools required to implement this algorithm in YAS are available and we can proceed to implementation. As we have mentioned, for IMRT, $gEUD$ constraints motivate nonsymmetric conic optimization. Second, we wanted to explore the consequences of treating the constraint matrix as a dense matrix when this is the case. We expected a decrease of the time required at each iteration when compared with exist-

ing solvers. Moreover, dense matrix operations can benefit from multiple processors platform. Therefore, it is desirable that a solver allows the user to model the data at least with dense and sparse matrices.

We tried to replicate the shape of YAS by using functions that return the value, gradient or Hessian of a barrier at a given point. As expected, for a product cone the barriers for each member cone are used. We have implemented barriers for the positive orthant, the second order cone and the cone that is a product of second order cones with the same dimension. Note our choice of including the barrier for the product of second order cones in the set of basic barriers. Doing this caused a major improvement in the speed of our code as loops in MATLAB are extremely slow. Alternatively, we could have implemented the loops in C. However, we suspect that, like in our case, a barrier written for a family of cones of the same type and dimension will be much faster than the barrier obtained for the product cone. This issue, which we haven't anticipated, can be accommodated with the design of YAS and can imply speed improvements for such problems.

10 Benchmarking

In what follows we present the results of some experiments. We are only concerned with the time required by one iteration and not with the number of iterations, which depends on the algorithm that is used, or with the precision that we can solve the problem. The main computational effort in one iteration is to form the normal equations.

As mentioned before, we are concerned with a conic problem with 5458

positive variables, 3395 second order cones of dimension 9 and a constraint matrix with 1245 rows and 38% nonzero elements.

Ideally, IMRT optimization problems should not be solved on supercomputers bur rather on desktop machines. Therefore, we have started our testing on an Intel Core (TM) 2 CPU 6600 @2.4 GHz computer with 2GB of RAM memory.

As mentioned, SeDuMi and SDPT3 have run out of memory at the first iteration when 3395 cones are considered. We were interested to see what is the maximum number of cones that can be supported. Their upper bound is 1500 cones in which case SeDuMi requires 141 seconds/iteration and SDPT3 requires 22 seconds/iteration while our implementation requires 12 seconds/iteration. The upper limit for our implementation is 2000 cones and we require 33 seconds/iteration.

Next, we have moved to a multiprocessors workstation and compared our implementation with SeDuMi. The workstation has 16 Dual Core AMD Athlon Opteron (885) processors at 2.4Ghz and 64Gb of RAM. Further, MATLAB for 64 bits is used together with SeDuMi 64. Using the MKL library through MATLAB we have speed up the computations in our implementation by using more processors. We have recorded the following:

SO cones	Processors	Sec/Iter SeDuMi	Sec/Iteration Prototype
1500	1	90	17
	4	90	7.5
	8	90	6
	16	90	5.7
3395	1	143	33
	4	143	14
	8	143	10
	16	143	10
6000	1	262	56
	4	262	26
	8	262	20
	16	262	20

From this table one can see that, when only one processor is used, the time per iteration required by the prototype code is significantly less than the time required by SeDuMi. This is due to the fact that SeDuMi treats data as sparse while these instances have dense data. The prototype code treats data as dense. Furthermore, multiplication of dense matrices can be parallelized and so the prototype code is faster when more processors are used.

11 Conclusions

In this thesis we have set the basis of a new IPM-based solver, YAS, with a modular design implemented using the object-oriented paradigm in C++. Through this design we aim to

- allow easy development of IPM algorithms for optimization problems beyond standard symmetric cone optimization problems;
- allow easy switch between different linear algebra packages that supply routines required by IPM algorithms; in particular, one can use platform-tuned linear algebra packages;
- allow the user to exploit the type/structure of the matrices involved in the linear algebra by introducing a notion of k-tuple block-structured matrix with (low-rank) multiplicative updates;
- allow easy switch between different numerical precisions of data to speed up computations;
- allow transparent implementation of techniques motivated by numerical accuracy such as storing the iterates of the algorithms in a scaled space (see YAS_k_EVS in Section 5.2);
- allow modelling of optimization problems in their natural formulation (see Semidefinite optimization in matrix variable in Chapter 3) including supporting both primal and dual problem formulations;
- provide a framework for the development of an open-source library of derived classes that are tuned for optimization problems with specific

data structure and problem classes, such as optimization over p-cones.

YAS's design is split in two levels:

- basic linear algebra layer;
- interior point methods layer.

Basic linear algebra layer's goal is to provide a transparent access to hardware-tuned linear algebra routines. This layer consists of:

- the $low - level\ routines$, further grouped into:
  - BLAS routines (matrix-matrix multiplications and additions for different types of matrices);
  - LAPACK routines (used for inverting, factorizing, solving linear systems with different types of matrices);
- the YAS_K_block class (allows the storage of one or more blocks of the same type and dimension and provides methods to do linear algebra operations. By a block of a certain type we refer a matrix with an exploitable structure such as a matrix that is sparse, symmetric, diagonal etc);
- the YAS_K_mb class (allows the storage of one or more matrices of blocks of the same type and dimension and provides methods to do linear algebra operations. By a matrix of blocks we refer a matrix that can be splited into blocks).

The main contribution of this thesis is the design and implementation of the linear algebra layer for the new object-oriented platform. In particular, the following tasks are accomplished:

- a detailed design of the linear algebra layer. The consisting classes and routines are described in all details. Resulting advantages are discussed;
- implementation of most low-level routines;
- implementation of YAS_K_block;
- a rough design of the IPM layer. The main classes needed in this layer are identified. Resulting advantages are discussed.

In addition, we present a "proof of concept". The problem considered is a large scale, dense optimization problems arrising in radiation therapy treatment planning. Time per iteration is compared with two of the state-of-the-art IPM solvers SeDuMi and SDPT3 on a multi-processor computing server comprised of 16 AMD computing cores.

References

[1] Benson, S.J, Ye, Y., Zhang, X.: Solving Large-Scale Sparse Semidefinite Programs for Combinatorial Optimization, SIAM J. Optim, 10(2), 443-461 (2000)

[2] Borchers, B.: CSDP User's Guide, Optim. Meth. Soft. 11, 597-611 (1999)

[3] Chu, M., Zinchenko, Y., Henderson, S.G., Sharpe, M.B.: Robust Optimization for Intensity Modulated Radiation Therapy Treatment Planning under Uncertainty. Phys. Med. Biol., 50, 5463-5477 (2005)

[4] de Oliveira, M., Helton, B.: Numerical Optimization Assisted by Symbolic Noncommutative Algebra. Slides of Talk given in the AdvOL Seminar, McMaster University, (2007)

[5] Fiacco, A.V., McCormick, G.P.: Nonlinear Programming: Sequential Unconstrained Minimization Techniques, John Wiley & Sons, New York (1968). Republished, SIAM, Classics in Applied Mathematics, Philadelphia (1990)

[6] Fujisawa, K., Kojima, M., Nakata, K.: SDPA User's Manual, Research Reports on Mathematical and Computing Sciences, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, (2000)

[7] Güler, O.: Barrier Functions in Interior Point Methods, Math. of Oper. Res., 21, 860-885 (1996)

[8] Nesterov, Y., Todd, M.J.: Self-Scaled Barriers and Interior-Point Methods for Convex Programming. Math. of Oper. Res., 22 , 1-42 (1997)

[9] Nesterov, Y., Todd, M.J.: Primal-Dual Interior-Point Methods for Self-Scaled Cones. SIAM J. Optim., 8 (2), 324-364 (1998)

[10] Nesterov, Y.: Towards Nonsymmetric Conic Optimization. CORE Discussion Paper, Catholic University of Louvain, Louvain-la-Neuve, Belgium (2006)

[11] Nesterov, Y., Nemirovskii, A.: Interior-Point Polynomial Algorithms in Convex Programming. SIAM, Philadelphia (1994)

[12] Niemierko, A.: A Generalized Concept of Equivalent Uniform Dose (EUD). Med. Pshy., 26, 1100 (abstract) (1999)

[13] Renegar, J.: A Mathematical View of Interior-Point Methods in Convex Optimization. SIAM, Philadelphia (2001)

[14] Roos, C., Terlaky, T., Vial, J.-P.: Interior Point Methods for Linear Optimization, Springer, New York (2006)

[15] Sturm, J.F.: Avoiding Numerical Cancellation in the Interior Point Method for Solving Semidefinite Programs. Math. Progr., Ser. B 95, 219-247 (2003)

[16] Sturm, J.F.: Using SeDuMi 1.02, a MATLAB Toolbox for Optimization over Symmetric Cones. Optim. Meth. Soft., 11–12, Special Issue on Interior Point Methods, 625–653 (1999)

[17] Tapia, R.A.: The Kantorovich Theorem for Newton's Method. Amer. Math. Monthly. 78(4), 389-392 (1971)

[18] Todd, M.J., Toh, K.C., Tütüncü, R.H.: On the Nesterov-Todd Direction in Semidefinite Programming. SIAM J. Optim., 8 (3), 769-796 (1998)

[19] Toh, K.C., Todd, M. J., Tütüncü, R.H.: SDPT3-A MATLAB Software Package for Semidefinite Programming, Optim. Meth. Soft., 11-12, 545-581 (1999)