Quick user guide for a MatLab implementation of DSA-BD

Camilo Ortiz

September 20, 2011

1 Introduction

This document is a quick reference for a MATLAB package developed by C. Ortiz, R.D.C. Monteiro and B.F. Svaiter for solving general semidefinite programs (SDPs) using a block-decomposition first-order method, which is a special case of the hybrid proximal extragradient (HPE) method, a framework of inexact proximal point methods introduced by B.F. Svaiter and M. Solodov in [2, 3]. The details of the algorithm used by DSA-BD can be found in [1].

The primal and dual problems of interest are

$$\min\{\langle c, x\rangle | \mathcal{A}x = b \ , \ x \succeq_K 0\}, \qquad \max\{\langle b, y\rangle | \mathcal{A}^T y + z = c \ , \ z \succeq_K 0\}, \quad (1)$$

where $\mathcal{A} : \mathbb{R}^{n_u} \times \mathbb{R}^{n_l} \times S^{n_s} \to \mathbb{R}^m$ is linear, $c = (c^u, c^l, C^s) \in \mathbb{R}^{n_u} \times \mathbb{R}^{n_l} \times S^{n_s}$, $b \in \mathbb{R}^m$ and $K = \mathbb{R}^{n_u} \times \mathbb{R}^{n_l}_+ \times S^{n_s}_+$.

The basic description of DSA-BD is given in Algorithm 4 in [1]:

Algorithm 1 : Scaled adaptive block-decomposition (SA-BD) method for solving (1) .

0) Let $x_0 \in \mathcal{X}$, $y_0 \in \mathcal{Y}$, $0 < \sigma \leq 1$ and $\theta > 0$ be given, and set $k = 1$ and

$$\tilde{\lambda}_x = \frac{\sigma\sqrt{\theta}}{\|\mathcal{U}^{-1/2}\mathcal{A}\|}, \qquad \tilde{\lambda}_y = \frac{\sigma}{\|\mathcal{U}^{-1/2}\mathcal{A}\|\sqrt{\theta}}; \tag{2}$$

1) compute

$$\tilde{y}_k = y_{k-1} - \tilde{\lambda}_y(\mathcal{A}x_{k-1} - b), \qquad \tilde{x}_k = \Pi_K\left[x_{k-1} - \tilde{\lambda}_x\left(c - \mathcal{A}^*\mathcal{U}^{-1}\tilde{y}_k\right)\right]; \tag{3}$$

2) define

$$\tilde{v}_k = \begin{pmatrix} (x_{k-1} - \tilde{x}_k)/\tilde{\lambda}_y \\ \mathcal{A}\tilde{x}_k - b \end{pmatrix}, \tag{4}$$

choose λ_k to be the largest $\lambda > 0$ such that

$$\left\|\lambda\tilde{v}_k + \begin{pmatrix} \tilde{x}_k \\ \tilde{y}_k \end{pmatrix} - \begin{pmatrix} x_{k-1} \\ y_{k-1} \end{pmatrix}\right\|_{[\theta,\mathcal{U}]} \leq \sigma\left\|\begin{pmatrix} \tilde{x}_k \\ \tilde{y}_k \end{pmatrix} - \begin{pmatrix} x_{k-1} \\ y_{k-1} \end{pmatrix}\right\|_{[\theta,\mathcal{U}]}; \tag{5}$$

3) set

$$\begin{pmatrix} x_k \\ y_k \end{pmatrix} = \begin{pmatrix} x_{k-1} \\ y_{k-1} \end{pmatrix} - \lambda_k\tilde{v}_k, \tag{6}$$

and $k \leftarrow k + 1$, and go to step 1.

Several ingredients are introduced to speed-up the method in its pure form such as: an aggressive choice of stepsize for performing the extragradient step; use of scaled inner products in the primal and dual spaces; dynamic update of the scaled inner product in the primal space for properly balancing the primal and dual relative residuals, and; proper choices of the initial primal and dual iterates, as well as the initial parameter for the primal scaled inner product. The subroutines for most of the algebraic operations (SDV, matrix-vector multiplication) use a LAPACK implementation coded in C, using the SDP data structure implemented in [4] and [5]. We describe how to use each of the ingredients in this implementation by explaining in detail each component of the main routine in the following section.

2 Main routine

The main routine `DSA_BD.m` has the following calling syntax:

```
[obj,X,y,Z,runhist]=DSA_BD(blk,At,C,b,par,X0,y0,Z0)
```
(7)

2.1 Output

`obj(1)`: value of the primal objective function $\langle c, x\rangle$ at the last iteration of DSA-BD.

`obj(2)`: value of the dual objective function $\langle b, y\rangle$ at the last iteration of DSA-BD.

`X`: value of the primal variable, x, at the last iteration of DSA-BD.

`y`: value of the dual variable, y, at the last iteration of DSA-BD.

`Z`: value of the slack variable, z, in the dual problem at the last iteration of DSA-BD.

`runhist`: structure containing information that measures the progress of the algorithm:

`runhist.pobj`: array with the value of the primal objective function at each iteration.

`runhist.dobj`: array with the value of the dual objective function at each iteration.

`runhist.pfeas`: array with the primal relative residual (infeasibility) at each iteration: $\epsilon_P = \frac{\|b-\mathcal{A}x\|}{1+\|b\|}$.

`runhist.dfeas`: array with the dual relative residual (infeasibility) at each iteration: $\epsilon_D = \frac{\|\mathcal{A}^* y+z-c\|}{1+\|c\|}$.

`runhist.cputime`: array with the time taken to perform each iteration.

`runhist.epsilon`: array with the value of $\langle x, z\rangle$ at each iteration.

`runhist.lambdaratio`: array with the value of $\lambda_k/\tilde{\lambda}_y$ at each iteration.

`runhist.plambda`: array with the value of $\theta^{-1} = \sqrt{\tilde{\lambda}_y/\tilde{\lambda}_x}$ at each iteration.

2.2 Input

`par`: structure of the input problem as well as the modifications or speed-up ingredients to be used by DSA-BD.

`par.adaptivelambda`: flag that specifies the use of the aggressive choice of stepsize for performing the extragradient step:

- If `par.adaptivelambda=0`, then $\lambda_k = \tilde{\lambda}_y$ and step 2 is not performed in Algorithm 1.
- If `par.adaptivelambda=1`, then λ_k is obtained as in step 2 in Algorithm 1.

`par.balancedinit`: defines the initialization of θ, x_0 and y_0 in Algorithm 1:

- If `par.balancedinit=0`, then x_0 and y_0 are given by `X0` and `y0` in (7), respectively, and $\theta = 1$.
- If `par.balancedinit=1`, then $x_0 = 0$, $y_0 = \arg\min \left\|\mathcal{A}^*\mathcal{U}^{-1}y - c\right\|$ and $\theta = 1$.

* If `par.balancedinit=2`, then x_0 and y_0 are given by `X0` and `y0` in (7), respectively, and $\theta > 0$ is chosen such that $\epsilon_P, \epsilon_D = \mathcal{O}(1)$.
* If `par.balancedinit=3`, then $x_0 = 0$, $y_0 = \arg\min \left\| \mathcal{A}^* \mathcal{U}^{-1} y - c \right\|$ and $\theta > 0$ is chosen such that $\epsilon_P, \epsilon_D = \mathcal{O}(1)$.

`par.dynamicscaling`: flag that specifies the use of the dynamic scaling of the primal inner product, θ, in Algorithm 1. If `par.dynamicscaling=1`, $\bar{k} :=$ `par.dyn_scale_updateiteration`, $\tau :=$ `par.scalecorrection` and $\gamma :=$ `par.scaleratio`, then, if θ_k denotes the dynamic value of θ at the kth iteration of the algorithm, we use the rule for updating θ_k as in equation (59) in Section 5 of [1]:

$$\theta_k = \begin{cases} \theta_{k-1}, & k \not\equiv 0 \bmod \bar{k} \text{ or } \gamma^{-1} \leq \epsilon_{P,k-1}/\epsilon_{D,k-1} \leq \gamma \\ \theta_{k-1} \cdot \tau, & k \equiv 0 \bmod \bar{k} \text{ and } \epsilon_{P,k-1}/\epsilon_{D,k-1} > \gamma \\ \theta_{k-1}/\tau, & k \equiv 0 \bmod \bar{k} \text{ and } \epsilon_{D,k-1}/\epsilon_{P,k-1} > \gamma \end{cases}, \qquad \forall k \geq 2. \tag{8}$$

The update rule in (8) requires $\bar{k} \geq 1$ integer, and scalars $\gamma > 1$ and $0 < \tau < 1$. If the flag `par.scalerelnorm` is set to `1`, then the normalized residuals

$$\tilde{\epsilon}_P = \frac{\|\mathcal{U}^{-1/2}(b - \mathcal{A}x)\|}{1 + \|b\|}, \qquad \tilde{\epsilon}_D = \frac{\|\mathcal{A}^* \mathcal{U}^{-1} y + z - c\|}{1 + \|c\|}$$

are used instead of ϵ_P and ϵ_D, respectively, in (8).

`par.maxit`: maximum number of iterations.

`par.normalize`: defines the normalization used by the problem:

* If `par.normalize=0`, then $\mathcal{U} = I$ in Algorithm 1.
* If `par.normalize=1`, then $\mathcal{U} = I$ in Algorithm 1 and the input

$$\tilde{\mathcal{A}} = \frac{\mathcal{A}}{\|b\| + 1}, \ \tilde{b} = \frac{b}{\|b\| + 1} \ \textit{and} \ \tilde{C} = \frac{C}{\|C\|}$$

  is used instead of $\mathcal{A}$, b and C, respectively.
* If `par.normalize=2,3`, then $\mathcal{U} = AA^T$ in Algorithm 1. `par.normalize=3` exploits the sparsity of t $\mathcal{A}\mathcal{A}^T$.

`par.sigma`: value of σ in Algorithm 1.

`par.tol`: value of the desired accuracy $\bar{\epsilon} > 0$. The algorithm is stopped whenever $\max\{\epsilon_P, \epsilon_D\} < \bar{\epsilon}$.

`blk`: defines the block structure of the SDP as in [4], allowing only one sparse block for the positive definite part of the cone K, i.e, the part of the variable in $S_+^{n_s}$ of (1) are stored in only one block.

`At`: stores the value of $\mathcal{A}^T$ in (1).

`C`: stores the value of c in (1).

`b`: stores the value of b in (1).

`X0`: stores the value of x_0 in Algorithm 1 if `par.balancedinit`=0.

`y0`: stores the value of $y0$ in Algorithm 1 if `par.balancedinit`=0.

`Z0`: stores the initial value of z in (1).

References

[1] Renato D. C. Monteiro, Camilo Ortiz, and Benar F. Svaiter. Implementation of a block-decomposition algorithm for solving large-scale conic semidefinite programming problems. *Optimization-online preprint 3032*, pages 1–32, 2011.

[2] M. V. Solodov and B. F. Svaiter. A hybrid approximate extragradient – proximal point algorithm using the enlargement of a maximal monotone operator. *SetValued Analysis*, 7(4):323–345, 1999.

[3] M. V. Solodov and B. F. Svaiter. A hybrid projection-proximal point algorithm. *Journal of Convex Analysis*, 6(1):59–70, 1999.

[4] K. C. Toh, M.J. Todd, and R. H. Tütüncü. Sdpt3 - a matlab software package for semidefinite programming. *Optimization Methods and Software*, 11:545–581, 1999.

[5] Xin-Yuan Zhao, Defeng Sun, and Kim-Chuan Toh. A Newton-CG augmented lagrangian method for semidefinite programming. *SIAM Journal on Optimization*, 20(4):1737–1765, 2010.