Technical Report: TR-95-10, May 01, 1995

Solving Semidefinite Quadratic Problems Within Nonsmooth Optimization Algorithms

Frangioni Antonio

Dipartimento di Informatica, Università di Pisa
corso Italia 40, 56125 Pisa, Italy

Abstract

Bundle methods for Nondifferentiable Optimization are widely recognised as one of the best choices for the solution of Lagrangean Duals; one of their major drawbacks is that they require the solution of a Semidefinite Quadratic Programming subproblem at every iteration. We present an active-set method for the solution of such problems, that enhances upon the ones in the literature by distinguishing among bases with different properties and exploiting their structure in order to reduce the computational cost of the basic step. Furthermore, we show how the algorithm can be adapted to the several needs that arises in practice within Bundle algorithms; we describe how it is possible to allow constraints on the primal direction, how special (box) constraints can be more efficiently dealt with and how to accommodate changes in the number of variables of the nondifferentiable function. Finally, we describe the important implementation issues, and we report some computational experience to show that the algorithm is competitive with other QP codes when used within a Bundle code for the solution of Lagrangean Duals of large-scale (Integer) Linear Programs.

ACM–CR Subject Classification:

Keyword and phrases: Nondifferentiable Optimization, Bundle Methods, Quadratic Programming, Least Squares

0. Introduction

Nondifferentiable Optimization (NDO) is concerned with the solution of the generic problem

$$\min\{\ \varphi(\lambda) : \lambda \in \Lambda\ \}$$

where $\varphi()$ is a nondifferentiable function and $\Lambda \subseteq \mathcal{R}^n$: in the applications of interest, $\varphi()$ is a proper convex function and Λ is a polyhedral set (or even the whole $\mathcal{R}^n$). Usually, $\varphi()$ is known to the NDO algorithm by means of a "black box" solver that, for any given $\bar{\lambda}$, computes the value of $\varphi(\bar{\lambda})$ and a *subgradient* $\boldsymbol{g}(\bar{\lambda}) \in \mathcal{R}^n$, i.e. a vector that satisfies the well-known *subgradient inequality*

$$\varphi(\lambda) \geq \varphi(\bar{\lambda}) + g(\bar{\lambda})(\lambda - \bar{\lambda}) \qquad \forall\ \lambda \in \Lambda$$

NDO problems arises in many practical applications: among the most important ones, we mention the solution of Lagrangean Duals of large-scale (Integer) Linear Programs, where the computation of $\varphi(\bar{\lambda})$ and $\boldsymbol{g}(\bar{\lambda})$ corresponds to the solution of a (possibly hard) Lagrangean Relaxation of some difficult problem, $\varphi()$ is a polyhedral function and Λ is either the whole $\mathcal{R}^n$ or its nonnegative orthant.

Bundle methods [Ki85] [Le89] [SZ92] [CFN95] are a well-known class of NDO algorithms that are mainly characterised by keeping the first-order information (the subgradients) about the function $\varphi()$ in a "disaggregated" form, in contrast with *aggragated subgradient* algorithms. Visiting a (finite) sequence of points $\{\lambda_i\}$, the whole set (Bundle) of informations $\{< \lambda_i\ , \varphi(\lambda_i)\ , \boldsymbol{g}(\lambda_i) >\}$ that represents the "history" of the computations performed so far is used to compute the next trial point - actually a tentative descent direction $\boldsymbol{d}$ along which λ_{i+1} is choosen - by solving a Semidefinite Quadratic Programming subproblem. In principle, one may think to the current Bundle β as a (small) subset of an overall fixed (large) set of couples $(\boldsymbol{g}_i, \alpha_i)$, where $\boldsymbol{g}_i = \boldsymbol{g}(\lambda_i)$ and $\alpha_i = \alpha_i(\bar{\lambda}) = \varphi(\bar{\lambda}) - \boldsymbol{g}_i(\bar{\lambda} - \lambda_i) - \varphi(\lambda_i) \geq 0$ is the associated nonnegative *linearization error* w.r.t. the current point $\bar{\lambda}$, so that, when dealing with *unconstrained minimisation* of $\varphi()$ ($\Lambda = \mathcal{R}^n$), the problem

$$(\Pi_\beta) \qquad \min \left\{\ v + 1/2 \cdot \|\ \boldsymbol{d}\ \|^2 : v \geq \boldsymbol{g}_i\boldsymbol{d} - (1/t)\alpha_i\ \forall i \in \beta\ \right\}$$

gives a tentative descent direction $\boldsymbol{d}$ within the current "trust region" implicitly defined by the strictly positive *trust region parameter* t; alternatively, one can solve its Quadratic Dual

$$(\Delta_\beta) \qquad \min \left\{\ 1/2 \cdot \|\ \boldsymbol{G}_\beta \boldsymbol{x}\ \|^2 + (1/t)\alpha_\beta \boldsymbol{x} : \boldsymbol{x} \in \Theta\ \right\}$$

where $\boldsymbol{G}_\beta$ is the matrix having the vectors $\boldsymbol{g}_i$ ($i \in \beta$) as columns, and $\Theta = \{\ \boldsymbol{x} : \boldsymbol{ex} = 1, \boldsymbol{x} \geq \boldsymbol{0}\ \}$ (where $\boldsymbol{e} = [\ 1, 1, \ldots , 1\ ]^T$) is the unitary simplex. (Δ_β) and (Π_β) have several interpretations that cannot be discussed here (the interested reader can refer to [CFN95]): as a foretaste, we mention that from a "dual" point of view the $\boldsymbol{g}_i$s are α_i-subgradients of the current point $\bar{\lambda}$, so that (Δ_β) can be viewed as a generalisation of minimal partial derivative finding problem in some inner approximation of the $\max_i\{\alpha_i\}$-subdifferential of $\varphi()$ in $\bar{\lambda}$, while from a "primal" point of view (Π_β) is the minimisation of a polyhedral upper approximation of $\varphi()$ plus a "stabilising" quadratic term whose "weight" depends on t.

Both (Π_β) and (Δ_β) are Semidefinite Quadratic Problems, so they could be solved by standard SQP

codes [GMW81] [Po83] [CLM93], but the need for low solution times make it necessary to develop a specialized code in order to exploit their valuable structural properties: this is especially true because, within a Bundle algorithm, a (long) sequence of subproblems have to be solved that only differs for the addition/deletion of a few $\boldsymbol{g}_i$s from β, the modification of the vector α and/or the scalar t, so that a suitable algorithm can converge in very few steps if enough care is taken in conserving information between two subsequent calls. Furthermore, when applying Bundle methods to the solution of Lagrangean Duals, where the subgradients are obtained by solving a (possibly hard) optimization problem, the Bundle is usually small ($p = |\beta| \approx 100$) while the vectors $\boldsymbol{g}_i$ may be long ($n \approx 10000$), so that (Δ_β) has fewer variables and simpler constraints than (Π_β), even though the latter has a much simpler (separable) objective function: it is therefore not surprising that all the specialized algorithms proposed in the literature [Ki86] [Mo87] [Ki94] are “dual”, in the sense that they work on (Δ_β).
In this article, we propose an active-set method for the solution of (Δ_β), that enhances upon the specialized approaches proposed so far by distinguishing among bases with different properties and exploiting their structure in order to reduce the computational cost of the basic operations. The work is structured as follows: §1 an overview of the method is given, and simple proofs of convergence are presented, while in §2 the details of the matrix factorisation methods are discussed; in §3 the method is extended to a more general version of (Π_β) that allows (general) linear constraints on the primal solution $\boldsymbol{d}$, while in §4 a method for changing the number of variables of $\boldsymbol{d}$ is presented and in §5 this is used to develop a specialized algorithm for (Π_β) with box constraints on the primal space. Then, in §6 the relevant implementative details of our actual C++ code are discussed, and in §7 some computational comparisons are reported and conclusions are drawn.

1. An overview of the algorithm

Although the basic theory of our algorithm is almost the same as that of the other known specialized approaches, we present it here in a simple, self-contained form: by Quadratic Duality [Ge69], a feasible solution $\boldsymbol{x}$ of (Δ_β) is optimal if and only if

$$\boldsymbol{d} = - \boldsymbol{G}_\beta \boldsymbol{x} \qquad \text{and} \qquad v = - \boldsymbol{d}^T\boldsymbol{d} - (1/t)\alpha_\beta \boldsymbol{x}$$

are feasible for (Π_β) and vice-versa, since for any $\boldsymbol{x} \in \Theta$ ($\boldsymbol{x} \geq 0$, $\boldsymbol{ex} = 1$) and $(v, \boldsymbol{d})$ as above such that $\alpha_\beta \geq \boldsymbol{G}_\beta\boldsymbol{d} - v\boldsymbol{e}$, the *Complementary Slackness Conditions* $\boldsymbol{x}(\alpha_\beta - \boldsymbol{G}_\beta\boldsymbol{d} + v\boldsymbol{e}) = 0$ hold, so that $\boldsymbol{x}$ are feasible dual multipliers for (Π_β). Since during the solution of (Δ_β) t is fixed, for notational convenience in the following we will *drop the parameter t*, i.e. unless explicitly specified each α_i should be read as $(1/t)\alpha_i$, and v defined as $v = - \| \boldsymbol{d} \|^2 - \alpha_\beta\boldsymbol{x}$: it means viewing t as “embedded” in the vector α, while our implementation actually handles it explicitly.
We will address the solution of the “dual” problem

$$(\Delta_\beta) \qquad \min \left\{ 1/2 \cdot \boldsymbol{x}^T\boldsymbol{Q}_\beta\boldsymbol{x} + \alpha_\beta\boldsymbol{x} \ : \boldsymbol{x} \in \Theta \right\}$$

where $\boldsymbol{Q}_\beta = \boldsymbol{G}_\beta^T\boldsymbol{G}_\beta$ is a positive semidefinite matrix, i.e. the minimisation of the (non-strictly) convex function $f_\beta(\boldsymbol{x}) = 1/2 \cdot \boldsymbol{x}\boldsymbol{Q}_\beta\boldsymbol{x} + \alpha_\beta\boldsymbol{x}$ over the unitary simplex; our algorithm follows the well-known *active* *set* strategy [GMW81], i.e. at each step we attach a restriction of (Δ_β) in which some of the variables are fixed to zero:

Definition 1.1: *given any subset* $B \subseteq \beta$ *(a* ***base****), we set*

$$(\Delta_B) \qquad \min \left\{ 1/2 \cdot \boldsymbol{x} \boldsymbol{Q}_B \boldsymbol{x} + \alpha_B \boldsymbol{x} \;:\; \boldsymbol{x} \in \Theta \right\}$$

where $\boldsymbol{x} \in \mathcal{R}^{|B|}$ *and* $\boldsymbol{Q}_B$, α_B *are appropriate submatrices of* $\boldsymbol{Q}_\beta$, α_β.

By a little abuse of notation, we will consider bases as sets of indices as well as sets of subgradients (an element of the base will be generically called *item*): we will also make some other notational simplifications by avoiding to distinguish sets Θ with different dimensions and omitting transpose signs when clarity is not affected.

We deal with a sequence of subproblems $\{(\Delta_B)\}$, characterised by their *base B* (and the corresponding $\boldsymbol{Q}_B$, $\boldsymbol{G}_B$ ...) and the *current feasible point* $\boldsymbol{x} = [\, \boldsymbol{x}_B \; 0 \,] \in \Theta$: at each step, a descent direction $\boldsymbol{w}$ (for $f_\beta()$ at $\boldsymbol{x}$) is found, and B is revised until a solution that is optimal for both (Δ_B) and the original (Δ_β) is reached, as described below. At each iteration, we approach the *inequality constrained* subproblem (Δ_B) by considering its *equality constrained* relaxation

$$(R_B) \qquad \min \left\{ f_B(\boldsymbol{x}_B) : \boldsymbol{e}\boldsymbol{x}_B = 1 \right\}$$

where $\bar{\boldsymbol{x}}_B$ is an optimal solution if and only if the Kuhn-Tucker optimality conditions

$$\boldsymbol{e}\bar{\boldsymbol{x}}_B = 1 \qquad \text{and} \qquad \exists\, \rho \in \mathcal{R} \text{ s.t. } \rho \boldsymbol{e} = \boldsymbol{Q}_B \bar{\boldsymbol{x}}_B + \alpha_B$$

hold, i.e. if at least a solution in the form $[\, \bar{\boldsymbol{x}}_B \; \rho \,]$ of the linear system

$$(KT) \qquad \begin{bmatrix} \boldsymbol{Q}_B & -\boldsymbol{e} \\ \boldsymbol{e}^T & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_B \\ \rho \end{bmatrix} = \begin{bmatrix} -\alpha_B \\ 1 \end{bmatrix}$$

exists: finding a solution of the (KT) system is the basic step of our algorithm, and it will be shown in §2 how to (efficiently) construct either the unique $[\, \bar{\boldsymbol{x}}_B \; \rho \,]$, if uniqueness holds, or a direction $\boldsymbol{w}_B$ such that $\boldsymbol{w}_B \boldsymbol{Q}_B = 0$ and $\boldsymbol{e}\boldsymbol{w}_B = 0$ otherwise.

TT Algorithm:

```
$B = \{1\}$; $x = [\, 1, 0, \dots , 0 \,]$;                                  /* initialisation */
while( $\exists\, h \notin B$ such that $v < g_h d - \alpha_h$ )            /* main iteration */
   $B = B \cup \{h\}$;                                                     /* item insertion */
   do                                                                      /* inner iteration */
      if( the (KT) system admits unique solution $[\, \bar{x}_B \; \rho \,]$ )
         then if( $\bar{x}_B \geq 0$ )  then $x = [\, \bar{x}_B \; 0 \,]$; $w_B = 0$;
                                  else $w_B = \bar{x}_B - x_B$;
              else $\langle$ find a feasible (descent) direction $w_B$ such that $w_B Q_B = 0$ and $e w_B = 0$ $\rangle$;
      if( $w_B \neq 0$ )                                                   /* $\bar{x}_B$ is not optimal for $(\Delta_B)$ */
         then  $\eta = \min\{ -w_h / x_h : w_h < 0 , h \in B \}$               /* maximum feasible step $\eta > 0$ */
               $x_B = x_B + \eta w_B$;                                    /* take step $\eta$ along $w_B$ */
      forall( $h$ such that $x_h = 0$ )
         $B = B / \{h\}$;                                                  /* item(s) deletion */
   while( $w_B \neq 0$ );          /* end inner iteration */
endwhile                           /* end main iteration */
```

If (the unique) $\bar{\boldsymbol{x}}_B$ is feasible, i.e. $\bar{\boldsymbol{x}}_B \geq \mathbf{0}$, then it is optimal for (Δ_B) too, otherwise, by convexity of $f_B()$, $\boldsymbol{w}_B = \bar{\boldsymbol{x}}_B - \boldsymbol{x}_B$ is a direction of descent (and $\boldsymbol{e}\boldsymbol{w}_B = 0$): this is not enough, however, since we must also show that the direction is feasible, i.e. that for a sufficiently small $\eta > 0$ we still have $\boldsymbol{x}_B + \eta\boldsymbol{w}_B \geq \mathbf{0}$. If the current inner iteration is *not* the first of a main iteration, then $\boldsymbol{x}_B > \mathbf{0}$ (all the h such that $x_h = 0$ have been eliminated at the end of the previous inner iteration) and any $\boldsymbol{w}_B$ such that $\boldsymbol{e}\boldsymbol{w}_B = 0$ is feasible, otherwise we have $B = B' \cup \{h\}$, $\boldsymbol{x}_B = [\ \boldsymbol{x}_{B'}\ x_h\ ] = [\ \boldsymbol{x}_{B'}\ 0\ ]$, $\boldsymbol{x}_{B'} > \mathbf{0}$ is optimal for ($\Delta_{B'}$) and the h-th primal constraint is violated, i.e. $v < \boldsymbol{g}_h\boldsymbol{d} - \alpha_h$. In this case, by exploiting some basic relations it is possible to show that $\boldsymbol{w}_B = [\ \boldsymbol{w}_{B'}\ w_h\ ] = \bar{\boldsymbol{x}}_B - [\ \boldsymbol{x}_{B'}\ 0\ ]$ is feasible, i.e. that $w_h > 0$ (since no other entry can create problems): for $\boldsymbol{d} = -\boldsymbol{G}_{B'}\boldsymbol{x}_{B'}$ and $v = -\|\ \boldsymbol{d}\ \|^2 - \alpha_{B'}\boldsymbol{x}_{B'}$ (the optimal primal variables of ($\Delta_{B'}$)), the KT conditions have an obvious interpretation in terms of the “basic” primal constraints, since they can be rewritten

$$-\rho\boldsymbol{e} = \boldsymbol{G}_{B'}^T\boldsymbol{d} - \alpha_{B'}$$

where (left)multiplying both sides for $\boldsymbol{x}_{B'}^T$ and using $\boldsymbol{e}\boldsymbol{x}_{B'} = 1$ one has

$$-\rho = -\boldsymbol{d}^T\boldsymbol{d} - \alpha_{B'}\boldsymbol{x}_{B'} = v$$

so that finally

$$v = \boldsymbol{g}_h\boldsymbol{d} - \alpha_h \qquad \text{for each } h \in B'$$

i.e. all the primal constraints in B' are satisfied as strict equations and the optimal multiplier ρ associated with the constraint $\boldsymbol{e}\boldsymbol{x}_{B'} = 1$ is just $-v$. Then, take any direction $\boldsymbol{w} = [\ \boldsymbol{w}'\ w_h\ ]$ such that $\boldsymbol{e}\boldsymbol{w} = 0$ and note that

$$f(\boldsymbol{x}_B + \eta\boldsymbol{w}) = f(\boldsymbol{x}_{B'}) + \eta[\ \boldsymbol{w}'\ w_h\ ]\left(\begin{bmatrix} \boldsymbol{Q}_{B'} & \boldsymbol{Q}_{B'h} \\ \boldsymbol{Q}_{B'h}^T & \boldsymbol{q}_{hh} \end{bmatrix}\begin{bmatrix} \boldsymbol{x}_{B'} \\ 0 \end{bmatrix} + \begin{bmatrix} \alpha_{B'} \\ \alpha_h \end{bmatrix}\right) + \frac{1}{2}\eta^2\boldsymbol{w}^T\boldsymbol{Q}_B\boldsymbol{w}$$

where $\boldsymbol{Q}_{B'h} = \boldsymbol{G}_{B'}^T\boldsymbol{g}_h$: the first-order term can be rewritten as

$$\eta[\ \boldsymbol{w}'\ w_h\ ]\begin{bmatrix} \boldsymbol{Q}_{B'}\boldsymbol{x}_{B'} + \alpha_{B'} \\ \boldsymbol{g}_h^T\boldsymbol{G}_{B'}\boldsymbol{x}_{B'} + \alpha_h \end{bmatrix}$$

and since (from optimality of $\boldsymbol{x}_{B'}$) $\boldsymbol{Q}_{B'}\boldsymbol{x}_{B'} + \alpha_{B'} = -v\boldsymbol{e}$, we obtain

$$\eta[\ w_h(\alpha_h - \boldsymbol{g}_h\boldsymbol{d}) - v\boldsymbol{e}\boldsymbol{w}'\ ]$$

that, using the fact that $\boldsymbol{e}\boldsymbol{w} = 1$ ($\Rightarrow \boldsymbol{e}\boldsymbol{w}' = -w_h$), can be reduced to

$$\eta w_h[\ v + \alpha_h - \boldsymbol{g}_h\boldsymbol{d}\ ]$$

that is (strictly) negative if and only if w_h is (strictly) positive, since $v + \alpha_h - \boldsymbol{g}_h\boldsymbol{d} < 0$; therefore, a direction $\boldsymbol{w}$ is feasible if and only if it is (strictly) of descent, and we already know that (by convexity of $f_B()$) $\boldsymbol{w}_B$ is a direction of descent. If $\boldsymbol{w}_B\boldsymbol{Q}_B = 0$ (we call $\boldsymbol{w}_B$ an *infinite direction*) things are even simpler, since from the previous formulas

$$f\left(\begin{bmatrix} \boldsymbol{x}_{B'} \\ 0 \end{bmatrix} + \eta\begin{bmatrix} \boldsymbol{w}_{B'} \\ w_h \end{bmatrix}\right) = f(\boldsymbol{x}_{B'}) + \eta[\ \boldsymbol{w}_{B'}\ w_h\ ]\begin{bmatrix} \alpha_{B'} \\ \alpha_h \end{bmatrix}$$

and therefore one among $\pm\boldsymbol{w}_B$ (depending on the sign of the scalar product) is a direction of descent: if

$\boldsymbol{x}_B > \mathbf{0}$ there can be no problems, and when $\boldsymbol{x}_B = [\ \boldsymbol{x}_{B'}\ 0\ ]$ we have just shown that

$$\eta[\ \boldsymbol{w}_{B'}\ w_h\ ]\begin{bmatrix}\alpha_{B'}\\ \alpha_h\end{bmatrix} = \eta w_h[\ v + \alpha_h - \boldsymbol{g}_h\boldsymbol{d}\ ]$$

and therefore that by choosing among the directions $\pm\boldsymbol{w}_B$ the one with $w_h > 0$ we choose at the same time the descent one.

It is easy to show that the algorithm finitely terminates: only a finite number of inner iterations can be performed between two successive main iterations, since at any inner iteration (at least) one item is eliminated from the base; moreover, it is impossible for a base B to appear twice as the current base at the beginning of a main iteration, since there the current solution $\boldsymbol{x}_B$ is optimal for (Δ_B) and at least a strictly decreasing step has been performed since the latest main iteration (a *strictly violated constraint* has been inserted in base), but the number of different bases is finite (although exponential) and this ensures finite termination.

Although the basic structure of our algorithm is almost the same as that of [Ki86] and [Mo87], our particular view point makes the statement of the algorithm and the proof of convergence easier than the ones in the above mentioned papers: actually, all those methods differs essentially for the way in which the KT system is faced, and in the next paragraph we will show how this task is accomplished.

2. Solving the KT system

We solve of the (KT) system (or the determine an infinite direction) by essentially keeping a *lower trapezoidal factorisation* of the Hessian matrix $\boldsymbol{Q}_B$ of the current subproblem (Δ_B): that is, we will always have a matrix

$$\bar{\boldsymbol{L}}_B = \begin{bmatrix}\boldsymbol{L}_{B'} & \mathbf{0}\\ \boldsymbol{V}^T & 0\end{bmatrix} \qquad \text{such that} \qquad \bar{\boldsymbol{L}}_B\bar{\boldsymbol{L}}_B^T = \boldsymbol{Q}_B$$

and $\boldsymbol{L}_{B'}$ is a lower triangular matrix with all nonzero diagonal entries; actually, the submatrix $\boldsymbol{V}^T$ will always have only up to two rows, i.e. $\bar{\boldsymbol{L}}_B$ has only three possible configurations

0) $B = B'$ $\qquad \bar{\boldsymbol{L}}_B = \boldsymbol{L}_{B'}$

1) $B = B' \cup \{h\}$ $\qquad \bar{\boldsymbol{L}}_B = \begin{bmatrix}\boldsymbol{L}_{B'} & \mathbf{0}\\ \boldsymbol{l}_h^T & 0\end{bmatrix}$

2) $B = B' \cup \{h\} \cup \{j\}$ $\qquad \bar{\boldsymbol{L}}_B = \begin{bmatrix}\boldsymbol{L}_{B'} & \mathbf{0} & \mathbf{0}\\ \boldsymbol{l}_h^T & 0 & 0\\ \boldsymbol{l}_j^T & 0 & 0\end{bmatrix}$

The rationale is that B' is the subset of B containing *linearly independent items* $\boldsymbol{g}_i$, while $\boldsymbol{g}_h$ (and $\boldsymbol{g}_j$) are linearly dependent from the items in B': in fact, the (KT) system may have full row rank (hence unique optimal solution) even if $\boldsymbol{Q}_B$ is 1-rank deficitary, while if B contains two linearly dependent items then the (KT) system is underdetermined and a feasible infinite direction $\boldsymbol{w}_B$ (such that $\boldsymbol{e}\boldsymbol{w}_B = 0$) can be found. The basic relations are summarised in the following propositions:

Proposition 2.1: *in case (0), the unique solution of KT is*

$$\bar{\boldsymbol{x}}_B = \boldsymbol{Q}_B^{-1}[\,\rho\boldsymbol{e} - \alpha_B\,] \qquad \textit{where} \qquad \rho = \frac{1 + \boldsymbol{e}\boldsymbol{Q}_B^{-1}\alpha_B}{\boldsymbol{e}\boldsymbol{Q}_B^{-1}\boldsymbol{e}}$$

The proof of this proposition is just linear algebra applied to the (KT) system, and is omitted; rather, defining

$$\boldsymbol{z}_1 = \boldsymbol{L}_{B'}^{-1}\boldsymbol{e} \qquad \text{and} \qquad \boldsymbol{z}_2 = \boldsymbol{L}_{B'}^{-1}\alpha_B$$

(the *unique* solutions of the triangular systems $\boldsymbol{L}_B\boldsymbol{z}_1 = \boldsymbol{e}$, $\boldsymbol{L}_B\boldsymbol{z}_2 = \alpha_{B'}$), the above relations can be written

$$\bar{\boldsymbol{x}}_B = (\boldsymbol{L}_B^T)^{-1}[\,\rho\boldsymbol{z}_1 - \boldsymbol{z}_2\,] \qquad \text{where} \qquad \rho = \frac{1 + \boldsymbol{z}_1\boldsymbol{z}_2}{\boldsymbol{z}_1\boldsymbol{z}_1}$$

i.e. $\bar{\boldsymbol{x}}_B$ can be obtained at the cost of three backsolves and a few linear operations: actually we can do much better, and we will show later that we are always able to find $\bar{\boldsymbol{x}}_B$ and ρ (or $\boldsymbol{w}_B$) with just one backsolve on the triangular matrix $\boldsymbol{L}_{B'}$. We continue with other four propositions, whose validity can be checked by simple substitution:

Proposition 2.2: *in case (1), if* $\boldsymbol{l}_h z_1 \neq 1$ *then the unique solution of KT is*

$$\bar{\boldsymbol{x}}_h = \frac{1 + \boldsymbol{z}_1\boldsymbol{z}_2 - \rho\boldsymbol{z}_1\boldsymbol{z}_1}{1 - \boldsymbol{l}_h\boldsymbol{z}_1} \qquad \bar{\boldsymbol{x}}_{B'} = (\boldsymbol{L}_{B'}^T)^{-1}[\,\rho\boldsymbol{z}_1 - \boldsymbol{z}_2 - \bar{\boldsymbol{x}}_h\boldsymbol{l}_h\,]$$

$$\textit{where} \qquad \rho = \frac{\alpha_h - \boldsymbol{l}_h\boldsymbol{z}_2}{1 - \boldsymbol{l}_h\boldsymbol{z}_1}$$

Proposition 2.3: *in case (1), if* $\boldsymbol{l}_h z_1 = 1$ *then an* infinite direction *is*

$$\boldsymbol{w}_B = [\,\boldsymbol{w}_{B'}\ \ w_h\,] = [\,\boldsymbol{v}_1\ \ {-1}\,] \qquad \textit{where} \qquad \boldsymbol{v}_1 = (\boldsymbol{L}_{B'}^T)^{-1}\boldsymbol{l}_h$$

Proposition 2.4: *in case (2), if* $\boldsymbol{l}_j z_1 = 1$ *then an* infinite direction *is*

$$\boldsymbol{w}_B = [\,\boldsymbol{w}_{B'}\ \ w_h\ \ w_j\,] = [\,\boldsymbol{v}_2\ \ 0\ \ {-1}\,] \qquad \textit{where} \qquad \boldsymbol{v}_2 = (\boldsymbol{L}_{B'}^T)^{-1}\boldsymbol{l}_j$$

Proposition 2.5: *in case (2), if* $\boldsymbol{l}_j z_1 \neq 1$ *then an* infinite direction *is*

$$\boldsymbol{w}_B = [\,\boldsymbol{v}\ \ {-1}\ \ \mu\,] \qquad \textit{where} \qquad \mu = \frac{1 - \boldsymbol{l}_h\boldsymbol{z}_1}{1 - \boldsymbol{l}_j\boldsymbol{z}_1}$$

$$\textit{and} \qquad \boldsymbol{v} = \boldsymbol{v}_1 - \mu\boldsymbol{v}_2 = (\boldsymbol{L}_{B'}^T)^{-1}[\,\boldsymbol{l}_h - \mu\boldsymbol{l}_j\,]$$

Note that in case (2) we are always able to find an infinite direction $\boldsymbol{w}_B$: this immediately implies that *B* won’t ever contain more than 2 linearly dependent items, since at the very moment in which the second linearly dependent item is inserted in *B*, a sequence of inner iterations always using infinite directions starts, and at each iteration at least one item is deleted from *B*, until full rank of the (KT) system is restored. Actually, it may even happen that some of the infinite directions used by the algorithm be not *strictly descent* (if $\alpha_B\boldsymbol{w}_B = 0$), but this cannot happen in the first inner iteration of a main iteration (since the selected *h*-th constraint is *strictly* violated) and hence convergence is not affected.
In our code we calculate the objective function value only at the beginning of each main iteration, where

$$\boldsymbol{Q}_B\bar{\boldsymbol{x}}_B = \rho\boldsymbol{e} - \alpha_B \quad \Rightarrow \quad \bar{\boldsymbol{x}}_B\boldsymbol{Q}_B\bar{\boldsymbol{x}}_B = \rho - \bar{\boldsymbol{x}}_B\alpha_B \quad \Rightarrow \quad f_B(\bar{\boldsymbol{x}}_B) = 1/2\cdot\bar{\boldsymbol{x}}_B\boldsymbol{Q}_B\bar{\boldsymbol{x}}_B + \alpha_B\bar{\boldsymbol{x}}_B = 1/2\cdot(\rho - \bar{\boldsymbol{x}}_B\alpha_B)$$

so that the calculation can be performed in O(m) rather than O(m^2) ($m = |B|$): one might also cheaply calculate the value of $f_B()$ at any inner iteration, since $f()$ is linear when restricted along an infinite $\boldsymbol{w}_B$ (as shown in §1), and if $\boldsymbol{w}_B = \bar{\boldsymbol{x}}_B - \boldsymbol{x}_B$ one has

$$f(\boldsymbol{x}_B + \eta\boldsymbol{w}_B) = f(\boldsymbol{x}_B) + \eta w_h[\, v + \alpha_h - \boldsymbol{g}_h\boldsymbol{d}\,] + 1/2\cdot\eta^2[\, f(\boldsymbol{x}_B) - f(\bar{\boldsymbol{x}}_B)\,]$$

but this is essentially useless, and it is avoided in our implementation.

Given the lower trapezoidal matrix $\bar{\boldsymbol{L}}_B$ and the vectors $\boldsymbol{z}_1$, $\boldsymbol{z}_2$ for a certain base B, we can keep them updated at low cost whenever an item is either inserted into or deleted from B, therefore the above propositions show that $\bar{\boldsymbol{x}}_B$ or $\boldsymbol{w}_B$ can be obtained at each step at the cost of a single backsolve on $\boldsymbol{L}_{B'}$ (since at the first iteration $\bar{\boldsymbol{L}}_B = \boldsymbol{L}_{B'} = [\ 1 / \boldsymbol{Q}_{11}\ ]$ and $\boldsymbol{z}_1$, $\boldsymbol{z}_2$ can be obtained accordingly); the method is described by the following propositions:

Proposition 2.6: *for $B = B' \cup \{h\}$ and $\bar{L}_B = L_B$ nonsingular, then*

$$\boldsymbol{L}_B = \begin{bmatrix} \boldsymbol{L}_{B'} & \boldsymbol{0} \\ \boldsymbol{l}_h^T & \delta \end{bmatrix} \qquad \textit{where} \qquad \boldsymbol{l}_h = (\boldsymbol{L}_{B'}^{-1})[\boldsymbol{G}_{B'}^T\boldsymbol{g}_h] \qquad \textit{and} \qquad \delta = \sqrt{\|\,\boldsymbol{g}_h\,\|^2 - \|\,\boldsymbol{l}_h\,\|^2}$$

Hence, $\boldsymbol{L}_B$ is nonsingular $\Leftrightarrow \delta > 0 \Leftrightarrow \boldsymbol{g}_h$ is linearly independent from the items in B'.

This follows from the rules of Cholesky factorisation applied to $\boldsymbol{Q}_B = \boldsymbol{G}_B^T\boldsymbol{G}_B$: for a sketch, note that

$$\|\,\boldsymbol{l}_h\,\|^2 = (\boldsymbol{g}_h^T\boldsymbol{G}_B\boldsymbol{L}_{B'}^{-1T})(\boldsymbol{L}_{B'}^{-1}\boldsymbol{G}_B^T\boldsymbol{g}_h) = \boldsymbol{g}_h^T[\boldsymbol{G}_{B'}(\boldsymbol{G}_{B'}^T\boldsymbol{G}_{B'})^{-1}\boldsymbol{G}_{B'}^T]\boldsymbol{g}_h = [\boldsymbol{g}_h^T\boldsymbol{K}_{B'}]\boldsymbol{g}_h = \bar{\boldsymbol{g}}^T\boldsymbol{g}_h$$

where $\boldsymbol{K}_{B'}$ is the projection operator onto the subspace $\Gamma_{B'}$ spanned by the vectors in B', therefore $\bar{\boldsymbol{g}}$ is the projection of $\boldsymbol{g}_h$ onto $\Gamma_{B'}$, so that $\|\,\boldsymbol{l}_h\,\|^2 = \bar{\boldsymbol{g}}^T\boldsymbol{g}_h \leq \|\,\boldsymbol{g}_h\,\|^2$ and equality holds $\Leftrightarrow \boldsymbol{g}_h \in \Gamma_{B'}$ (i.e. $\boldsymbol{g}_h$ is linearly dependent from the items in B'). Actually, this is exactly one step of the *row-wise* LL^T factorisation algorithm applied to $\boldsymbol{Q}_\beta$, that we are just performing step by step each time we need; to extend the result to the case where $\bar{\boldsymbol{L}}_B$ is lower trapezoidal, consider a p-vector γ and its subvector $\gamma_{B'}$:

Proposition 2.7: *if $\boldsymbol{L}_{B'}\theta = \gamma_{B'}$, then*

$$\boldsymbol{L}_B\cdot\begin{bmatrix} \theta \\ \dfrac{\gamma_h - \boldsymbol{l}_h\theta}{\delta} \end{bmatrix} = \begin{bmatrix} \boldsymbol{L}_{B'} & \boldsymbol{0} \\ \boldsymbol{l}_h & \delta \end{bmatrix}\cdot\begin{bmatrix} \theta \\ \dfrac{\gamma_h - \boldsymbol{l}_h\theta}{\delta} \end{bmatrix} = \begin{bmatrix} \boldsymbol{L}_{B'}\theta \\ \boldsymbol{l}_h\theta + \delta\left(\dfrac{\gamma_h - \boldsymbol{l}_h\theta}{\delta}\right) \end{bmatrix} = \begin{bmatrix} \gamma_{B'} \\ \gamma_h \end{bmatrix} = \gamma_B$$

In other words, if we already know $\theta = \boldsymbol{L}_{B'}^{-1}\gamma_{B'}$, then we can calculate the solution to the “new” system $[\ \theta\ \ \theta_h\ ] = \boldsymbol{L}_B^{-1}\gamma_B$ (actually θ_h only) in O(m') (where $m' = |\,B'\,|$): the importance of this proposition lies in the fact that all the “relevant” vectors $\boldsymbol{l}_h$, $\boldsymbol{l}_j$, $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$ have the above form (for γ respectively equal to $\boldsymbol{Q}_h$, $\boldsymbol{Q}_j$, α and $\boldsymbol{e}$), and therefore can be cheaply updated when inserting a new item in B; moreover, also the scalar products among such vectors that are used within the algorithm (such as $\boldsymbol{l}_h\boldsymbol{z}_1$, $\boldsymbol{z}_1\boldsymbol{z}_2$ ...) can be updated in constant time when the new entries are calculated, saving some O(m) computations.

Using the above relations, we can also shed some light on Propositions 2.2 - 25: for example, the “critical” scalar product $\boldsymbol{l}_h\boldsymbol{z}_1$ of Proposition 2.2 can be rewritten as $\boldsymbol{l}_h^T[\boldsymbol{L}_{B'}^{-1}\boldsymbol{e}] = \boldsymbol{e}[\boldsymbol{L}_{B'}^{-1}\boldsymbol{l}_h] = \boldsymbol{e}\boldsymbol{v}_1$, so that the condition

$$l_h z_1 = 1 \qquad \text{is equivalent to} \qquad [\, v_1 \ {-1}\,]^T [\, e \ \ 1\,] = 0$$

i.e. to the feasibility of the (infinite) direction [v_1 -1] w.r.t. the constraint $ex_B = 1$; similar interpretations can be derived for all the other cases.

Now, we need a cheap updating procedure for the case $B = B' / \{ h \}$, where

$$L_{B'} = \begin{bmatrix} L_1 & \mathbf{0} & \mathbf{0} \\ v^T & \delta & \mathbf{0} \\ Z & w & L_2 \end{bmatrix} \qquad G_{B'}^T = \begin{bmatrix} G_1^T \\ g_h^T \\ G_2^T \end{bmatrix} Q_{B'} = G_{B'}^T G_{B'} = \begin{bmatrix} G_1^T G_1 & G_1^T g_h & G_1^T G_2 \\ g_h^T G_1 & \| g_h \|^2 & g_h^T G_2 \\ G_2^T G_1 & G_2^T g_h & G_2^T G_2 \end{bmatrix}$$

and
$$L_B = \begin{bmatrix} L_1 & \mathbf{0} \\ Z & L_2' \end{bmatrix} \qquad G_B^T = \begin{bmatrix} G_1^T \\ G_2^T \end{bmatrix} Q_B = \begin{bmatrix} G_1^T G_1 & G_1^T G_2 \\ G_2^T G_1 & G_2^T G_2 \end{bmatrix}$$

Obviously, the first k - 1 (where k is the position of g_h in $G_{B'}$) rows and columns of $L_{B'}$ are not altered by the operation, and only the submatrix L_2' changes: since $L_2' L_2'^T = L_2 L_2^T + ww^T$, all we need for retrieving L_2 (and hence L_B) is a *rank-one correction* of the *triangular matrix* L_2', that can be accomplished by a sweep of *Givens Rotations*.

Proposition 2.8: *there exists a m'-k orthogonal square matrix G_S (i.e. $G_S^{-1} = G_S^T$) such that*

$$\begin{bmatrix} L_1 & \mathbf{0} & \mathbf{0} \\ v^T & \delta & \mathbf{0} \\ Z & w & L_2 \end{bmatrix} \cdot \begin{bmatrix} I_{k-1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & & \\ \mathbf{0} & \multicolumn{2}{c}{G_S} \end{bmatrix} = \begin{bmatrix} L_1 & \mathbf{0} & \mathbf{0} \\ v^T & \beta & y^T \\ Z & \mathbf{0} & L_2' \end{bmatrix}$$

which implies $L_2' L_2'^T = L_2 L_2^T + ww^T$,

Proof: G_S is the product of m'-k Givens matrices (where $m' = m + 1$); the h-th matrix performs the transformation

$$\begin{array}{|c|c|c|c|l} \hline \omega_h & y_h & \mathbf{0} & \mathbf{0} & \\ \mathbf{0} & L_{h1} & \mathbf{0} & \mathbf{0} & \\ \hline w_h & x & d & \mathbf{0} & k+h \\ \hline w_{h2} & Z & v & L_{h2} & \\ \hline k & & k+h & & \end{array} \quad \longrightarrow \quad \begin{array}{|c|c|c|c|} \hline \omega_h' & y_h & y_h & \mathbf{0} \\ \mathbf{0} & L_{h1} & \mathbf{0} & \mathbf{0} \\ \hline 0 & x & d' & \mathbf{0} \\ \hline w_{h2}' & Z & v' & L_{h2} \\ \hline \end{array}$$

so that at the end all the entries of the k-th column of $L_{B'}$ (but L_{kk}) are zero, and eliminating the k-th row and column of the resulting matrix we obtain just L_B. ♦

As the h-th step requires 4(m - k - h) multiplications, the overall cost is O($2(m - k)^2$); furthermore, with the same method we can also update the vectors l_h, l_j, z_1 and z_2:

Proposition 2.9: *for G_S as in Proposition 2.8,* θ *such that*

$$L_{B'}\theta = \begin{bmatrix} L_1 & \mathbf{0} & \mathbf{0} \\ v^T & \delta & \mathbf{0} \\ Z & w & L_2 \end{bmatrix} \cdot \begin{bmatrix} \theta_1 \\ \theta_h \\ \theta_2 \end{bmatrix} = \begin{bmatrix} \gamma_1 \\ \gamma_h \\ \gamma_2 \end{bmatrix} = \gamma_{B'} \qquad \textit{and} \qquad \begin{bmatrix} \theta_1 \\ \theta_h' \\ \theta_2' \end{bmatrix} = \begin{bmatrix} I_{k-1} & \mathbf{0} \\ \mathbf{0} & G_S^T \end{bmatrix} \cdot \begin{bmatrix} \theta_1 \\ \theta_h \\ \theta_2 \end{bmatrix}$$

$$\textit{the vector } \bar{\theta} = \begin{bmatrix} \theta_1 \\ \theta_2' \end{bmatrix} \textit{ solves } L_{B'}\bar{\theta} = \begin{bmatrix} L_1 & \mathbf{0} \\ Z & L_2' \end{bmatrix} \cdot \begin{bmatrix} \theta_1 \\ \theta_2' \end{bmatrix} = \begin{bmatrix} \gamma_1 \\ \gamma_2 \end{bmatrix} = \gamma_B.$$

In other words, to calculate the new (last m'-k entries of) vector $\bar{\theta}$, we only have to apply the same Givens transformations of Proposition 2.8 to the old θ: it is also easy to show that

$$\bar{\boldsymbol{z}}_1\bar{\boldsymbol{z}}_2 = \boldsymbol{z}_1\boldsymbol{z}_2 - z'_{1h}z'_{2h}$$

(an analogous result holds for any scalar product of this kind), therefore $\boldsymbol{l}_h\boldsymbol{z}_1$, $\boldsymbol{z}_1\boldsymbol{z}_2$ ... can be kept updated in this case also. Some other details have to be taken into account: for instance, in case (1) the elimination of an item from the base may make $\boldsymbol{g}_h$ to "enter the linearly dependent part" of B, i.e.

$$\begin{bmatrix} \boldsymbol{L}_1 & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{v}^T & \beta & \boldsymbol{y}^T & 0 \\ \boldsymbol{Z} & \boldsymbol{0} & \boldsymbol{L}'_2 & \boldsymbol{0} \\ \boldsymbol{l}_1 & l'_h & \boldsymbol{l}'_2 & 0 \end{bmatrix} \qquad \Rightarrow \qquad \begin{bmatrix} \boldsymbol{L}_1 & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{Z} & \boldsymbol{L}'_2 & \boldsymbol{0} \\ \boldsymbol{l}_1 & \boldsymbol{l}'_2 & l'_h \end{bmatrix}$$

but obviously l'_h is all we need to check if this is the case; similarly, in case (2) $\boldsymbol{g}_j$ also may become independent, or $\boldsymbol{g}_h$ may be deleted from B.

Reoptimization is natural within our algorithm: when (Δ_β) has been solved, the optimal base B with the relative optimal solution $[\ \boldsymbol{x}_B\ 0\ ]$, the lower trapezoidal factorisation $\bar{\boldsymbol{L}}_B$ of $\boldsymbol{Q}_B$ and the vectors $\boldsymbol{z}_1, \boldsymbol{z}_2$ (with the relative scalar products) are kept, and they are updated as modifications occurs to the data of the problem; for instance, in our implementation the calling program is allowed to

- add a single new item $\boldsymbol{g}_h$ (a newly obtained subgradient) to β: hence, by skipping the initialisation phase the algorithm is restarted with the optimality test phase, in which all the stored data structures are used "as is" to test the optimality of the newly entered item
- delete a single new item $\boldsymbol{g}_h$ (an outdated subgradient) from β: if $h \notin B$ this has no effect on the problem since $\boldsymbol{x}_B$ obviously remains optimal, while if $h \in B$ the procedures described in Propositions 2.8 - 2.9 (actually, the same piece of code) are used to update all the data structures in such a way that the algorithm can restarted at the beginning of an inner iteration, just as if $\boldsymbol{g}_h$ had been deleted from B in the course of a normal iteration of the algorithm
- change the vector α (i.e. change the current point $\bar{\lambda}$): then, the vector $\boldsymbol{z}_2$ can be recalculated with just one backsolve on the (available) linearly independent matrix $\boldsymbol{L}_{B'}$ (plus the recalculation of $\boldsymbol{z}_1\boldsymbol{z}_2$), and again the algorithm can be simply restarted at the beginning of an inner iteration, since all the other data structures are still coherent

The main point here is that all the above modifications can be dealt with orthogonally, i.e., provided that memory of the changes is kept, any sequence of such operations leaves the data structures in a consistent state, so that in case of a warm start the (data structures of the) previous optimal solution can be fully exploited. Some minor details may also be noticeable: if the parameter t is explicitly handled, as in our implementation, changing t does not force the recalculation of $\boldsymbol{z}_2$, if $\boldsymbol{z} = \boldsymbol{L}_{B'}^{-1}\alpha$ is kept and $\boldsymbol{z}_2 = (1/t)\boldsymbol{z}$ is calculated only when required; furthermore, if α is changed "along" the latest optimal solution $\boldsymbol{d}$ (as in NDO solvers, where the current point is usually updated as $\bar{\lambda} := \bar{\lambda} + \tau\boldsymbol{d}$), i.e. the new vector is obtained according to the formula

$$\alpha' = \alpha - \tau[\boldsymbol{G}^T\boldsymbol{d}] + \Delta_o\boldsymbol{e}$$

(where $\Delta_o = \varphi(\bar{\lambda} + \tau\boldsymbol{d}) - \varphi(\bar{\lambda})$ is the *obtained increase* in the nondifferentiable function for a *step* τ along $\boldsymbol{d}$) we have that

$$\alpha'_B = \alpha_B + \tau[\boldsymbol{Q}_B\boldsymbol{x}_B] + \Delta_o\boldsymbol{e} = \alpha_B + \tau[\rho\boldsymbol{e} - (1/t)\alpha_B\,] + \Delta_o\boldsymbol{e} =$$

$$= \left(1 - \frac{\tau}{t}\right)\alpha_B + (\Delta_o - \tau v)\boldsymbol{e} = \left(1 - \frac{\tau}{t}\right)\alpha_B + \left(\Delta_o - \frac{\tau}{t}\Delta_e\right)\boldsymbol{e}$$

(where $\Delta_e = tv = -t\rho$ is the *expected increase* for a step of t along $\boldsymbol{d}$), so that $\boldsymbol{z}'_2 = \boldsymbol{L}_B^{-1}\alpha'$ can be obtained in linear time from $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$. The basic algorithm that we have just described can be extended in several ways, as we will show in the next three paragraphs.

3. Handling general primal constraints

When dealing with *unconstrained minimisation* of $\varphi()$ ($\Lambda \subseteq \Re^n$), it is necessary to impose general linear constraints on the primal solution $\boldsymbol{d}$, i.e. to solve the extended problem

$$(\Pi_{\beta\gamma}) \qquad \min\left\{\, v + 1/2\cdot\|\,\boldsymbol{d}\,\|^2 : v \geq \boldsymbol{g}_i\boldsymbol{d} - \alpha_i \;\; i \in \beta \;,\; 0 \geq \boldsymbol{g}_i\boldsymbol{d} - \alpha_i \;\; i \in \gamma \,\right\}$$

or, equivalently, its dual

$$(\Delta_{\beta\gamma}) \qquad \min\left\{\, 1/2\cdot\left\|\,[\,\boldsymbol{G}_\beta\;\boldsymbol{G}_\gamma\,]\cdot\begin{bmatrix}\boldsymbol{x}\\ \boldsymbol{y}\end{bmatrix}\right\|^2 + [\,\alpha_\beta\;\alpha_\gamma\,]\cdot\begin{bmatrix}\boldsymbol{x}\\ \boldsymbol{y}\end{bmatrix} : \boldsymbol{x} \in \Theta \;,\; \boldsymbol{y} \geq \boldsymbol{0} \,\right\}$$

where $(\boldsymbol{g}_i, \alpha_i)$ $i \in \gamma$ are (a subset of) the linear inequalities defining Λ, properly "translated" w.r.t. the current point $\bar{\lambda}$, that ensure feasibility of all the points $\bar{\lambda} + \tau\boldsymbol{d}$ with $\tau \leq t$.
By letting $\boldsymbol{i}$ = be the indicator vector of the set (of subgradients) β, i.e.

$$\boldsymbol{i}_h = \begin{cases} 1 & \text{if } h \in \beta \\ 0 & \text{if } h \in \gamma \end{cases}$$

the feasible set of $(\Delta_{\beta\gamma})$ can be written

$$\left\{\, [\,\boldsymbol{x}\;\;\boldsymbol{y}\,] : \boldsymbol{i}[\,\boldsymbol{x}\;\;\boldsymbol{y}\,] = 1 \;,\; [\,\boldsymbol{x}\;\;\boldsymbol{y}\,] \geq \boldsymbol{0} \,\right\}$$

and *the TT algorithm can still be applied,* provided that *the all-one vector* ***e*** *is replaced in every expression with* $\boldsymbol{i}_B$: in fact, $\boldsymbol{Q}_B\bar{\boldsymbol{x}}_B + \alpha_B = \rho\boldsymbol{i}_B$ and $\boldsymbol{i}_B\bar{\boldsymbol{x}}_B = 1$ are the KT conditions for (Δ_B), so that, letting $\boldsymbol{z}_1 = \boldsymbol{L}_B^{-1}\boldsymbol{i}_B$, the optimal solution to (R_B) in case (0) is

$$\bar{\boldsymbol{x}}_B = \boldsymbol{Q}_B^{-1}[\,\rho\boldsymbol{i}_B - \alpha_B\,] \qquad \text{where still} \qquad \rho = \frac{1 + \boldsymbol{z}_1\boldsymbol{z}_2}{\boldsymbol{z}_1\boldsymbol{z}_1}$$

In the same way, Propositions 2.2 - 2.3 hold true if the condition $\boldsymbol{l}_h\boldsymbol{z}_1 \neq 1$ is replaced by $\boldsymbol{l}_h\boldsymbol{z}_1 \neq \boldsymbol{i}_h$ and the infinite direction $[\;\boldsymbol{v}_1\;\; -1\;]$ is replaced by $[\;\boldsymbol{v}_1\;\; -\boldsymbol{i}_h\;]$, and a completely analogous treatment can be deserved to Propositions 2.4 - 2.5; moreover, the factorisation methods described in §2 are obviously still valid for the constrained case, since $\boldsymbol{z}_1$ (that solves $\boldsymbol{L}_B\boldsymbol{z}_1 = \boldsymbol{i}_{B'}$) has the form required by Propositions 2.7 and 2.9. However, minor modifications to the algorithm are in fact needed, since it is necessary that the base B always contains at least one subgradient $\boldsymbol{g}_i$ ($i \in \beta$), but it is easy to see that once this is ensured for the initial base, it will automatically hold true thereafter; every detail of the algorithm naturally extends to the new case, for instance movements "along" $\boldsymbol{d}$ as introduced at the end of the previous paragraphs now have the α updating formula

$$\alpha' = \alpha - \tau[\boldsymbol{G}^T\boldsymbol{d}] + \Delta_o\boldsymbol{i}$$

so that $\boldsymbol{z}_2'$ is obtained in the same way from $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$, changes of the set γ between two subsequent calls to the $(\Delta_{\beta\gamma})$-solver are completely analogous to changes of the set β (up to the point that the two things can be handled with the same piece of code) and so on.
Actually, these results can be generalised by recognising that $(\Delta_{\beta\gamma})$ is just a special case of the problem

$$\min \left\{ 1/2\cdot\| \boldsymbol{Gx} \|^2 + \alpha\boldsymbol{x} : \boldsymbol{cx} = 1 , \boldsymbol{x} \geq \boldsymbol{0} \right\}$$

with $\boldsymbol{c} \geq \boldsymbol{0}$ any vector of nonnegative coefficients: it is easy to show that all the above theory still holds valid if $\boldsymbol{e}$ is replaced everywhere with $\boldsymbol{c}$ in the formulas; this is not only a theoretical result, since Least-Square-like problems with this form frequently appear in very different fields (see [FV94] for an application to fractal compression of images): the TT algorithm can be easily extended to handle these problems, and, as a matter of fact, the changes to be made are so few and so localised that in our implementation it is possible to "switch out" all the extra code needed for the task by simply setting a compile time switch.

4. Changing the dimension of primal space

It is important to note that our algorithm works without ever accessing the vectors $\boldsymbol{g}_i$, since the scalar products $\boldsymbol{q}_{ij} = \boldsymbol{g}_i^T\boldsymbol{g}_j$ suffices to carry on (efficiently) all the computations: in fact, instead of checking the primal constraints $\boldsymbol{g}_h\boldsymbol{d} - v \leq \alpha_h$ directly, we use $\boldsymbol{d} = -\boldsymbol{G}_B\boldsymbol{x}_B$ to obtain

$$\boldsymbol{g}_h\boldsymbol{d} = -[\boldsymbol{g}_h^T\boldsymbol{G}_B]\boldsymbol{x}_B = -\boldsymbol{Q}_{Bh}\boldsymbol{x}_B$$

Since $\boldsymbol{Q}_{Bh}$, $\boldsymbol{x}_B$ are m-vectors while $\boldsymbol{g}_h$, $\boldsymbol{d}$ are n-vectors (and in our applications $m << n$), by exploiting this relation we save a relevant quantity of work; actually, in our implementation we never explicitly calculate $\boldsymbol{d}$, a O(nm) task that can easily be the time consuming operation of the whole algorithm, while we take care of making the scalar product $\boldsymbol{g}_h\boldsymbol{d}$ for each $h \in \beta$ available to the calling program: this information is readily available at no cost when the algorithm terminates, since if $h \in B$ then $\boldsymbol{g}_h\boldsymbol{d} = \rho\boldsymbol{i}_h - \alpha_h$, while if $h \notin B$ then $\boldsymbol{g}_h\boldsymbol{d} = -\boldsymbol{Q}_{Bh}\boldsymbol{x}_B$ has already been computed when checking the h-th (nonbasic) constraint, and it may be useful to NDO algorithms, since $\varphi(\boldsymbol{d}, \tau) = \min_{h \in \beta} \{ (\boldsymbol{g}_h\boldsymbol{d})\tau + \alpha_h \}$ is an upper approximation of the restriction of $\varphi()$ along the (tentative descent) direction $\boldsymbol{d}$.
Furthermore, the algorithm is also orthogonal to the sparsity of the vectors $\boldsymbol{g}_h$: in our implementation, subgradients (and constraints) are only known by means of "symbolic names" (indices), and our TT-solver is only allowed to call the function

```
GiTGj( Name_of_i , Name_of_j )
```

that returns the scalar product $\boldsymbol{g}_i^T\boldsymbol{g}_j$; this function must be supplied by the calling program, so that knowledge about the structural properties of the subgradients (and constraints) can be exploited without requiring a specialized code.

Since the TT algorithm has in fact no information on the “real” vectors $\boldsymbol{g}_h$, it is obviously also possible to change their length between two subsequent calls to the TT-solver, i.e. to add or remove entries from all the $\boldsymbol{g}_h$ ($h \in \beta \cup \gamma$) at the same time: this is important both for the extensions that will be discussed in §5 and for the fact that NDO solvers for huge problems (n very large) may benefit from *active set* (*column generation*) strategy in which one works with a (hopefully small) subset of the variables λ, that is dynamically revised.

Insertions and deletions of entries in $\boldsymbol{g}_h$ can be naturally viewed as manipulations on the data of the problem like the ones described §2; as usual, suppose that, after call to the TT-solver, a base B, a solution [$\boldsymbol{x}_B$ 0] and the related $\boldsymbol{Q}_\beta$, $\bar{\boldsymbol{L}}_B$, $\boldsymbol{z}_1$, $\boldsymbol{z}_2$ ... (that may no longer be optimal, since other changes may have already occurred) are kept in a consistent state: if a new entry has to be added, all we need is the p-vector [$\tilde{\boldsymbol{g}}$]$_h$ = new entry of $\boldsymbol{g}_h$, since the Hessian matrix $\boldsymbol{Q}_\beta$ just have to be updated as

$$\tilde{\boldsymbol{q}}_{ij} = \boldsymbol{q}_{ij} + \tilde{g}_i\tilde{g}_j \qquad \text{for each } i, j$$

(a relatively cheap O(p^2) task), while the update of $\bar{\boldsymbol{L}}_B$, $\boldsymbol{z}_1$, $\boldsymbol{z}_2$ can be performed with a sweep of p Givens rotations ($\boldsymbol{G}_S$), as

$$\tilde{\boldsymbol{Q}}_B = \boldsymbol{Q}_B + \tilde{\boldsymbol{g}}_B\tilde{\boldsymbol{g}}^T_B = [\ \tilde{\boldsymbol{g}}_B\ \bar{\boldsymbol{L}}_B\ ]\cdot\begin{bmatrix} \tilde{\boldsymbol{g}}^T_B \\ \bar{\boldsymbol{L}}^T_B \end{bmatrix}$$

i.e. we only need the rank-one correction of the lower trapezoidal matrix $\bar{\boldsymbol{L}}_B$

$$\begin{bmatrix} \tilde{\boldsymbol{g}}_{B'} & \boldsymbol{L}_{B'} \\ \tilde{g}_h & \boldsymbol{l}^T_h \\ \tilde{g}_j & \boldsymbol{l}^T_j \\ 0 & \boldsymbol{z}_1 \\ 0 & \boldsymbol{z}_2 \end{bmatrix}\cdot\boldsymbol{G}_S \qquad \Rightarrow \qquad \begin{bmatrix} \boldsymbol{0} & \tilde{\boldsymbol{L}}_{B'} \\ \delta_h & \tilde{\boldsymbol{l}}^T_h \\ \delta_j & \tilde{\boldsymbol{l}}^T_j \\ \sigma_1 & \tilde{\boldsymbol{z}}_1 \\ \sigma_2 & \tilde{\boldsymbol{z}}_2 \end{bmatrix}$$

Note that *the position of the new entry plays absolutely no role,* making it possible to insert/delete entries in any order whatsoever; once $\tilde{\boldsymbol{g}}_{B'}$ has been nullified (still an O(p^2) task), the new $\tilde{\boldsymbol{L}}_{B'}$ (and $\tilde{\boldsymbol{l}}_h$, $\tilde{\boldsymbol{l}}_j$, $\tilde{\boldsymbol{z}}_1$, $\tilde{\boldsymbol{z}}_2$) is just what we expect to, and the data we have at hand can be used to “normalise” the data structures. For instance, if $\delta_h > 0$ then the corresponding item is inserted into the linearly independent part of B ($\tilde{B}' = B' \cup \{h\}$), i.e.

$$\boldsymbol{L}_{\tilde{B}'} = \begin{bmatrix} \tilde{\boldsymbol{L}}_{B'} & \boldsymbol{0} \\ \tilde{\boldsymbol{l}}^T_h & \delta_h \end{bmatrix}$$

where σ_1, σ_2 are the new entries of $\tilde{\boldsymbol{z}}_1$, $\tilde{\boldsymbol{z}}_2$, so that $\tilde{\boldsymbol{z}}_1\tilde{\boldsymbol{z}}_2 = \boldsymbol{z}_1\boldsymbol{z}_2 + \sigma_1\sigma_2$: similar results holds for the other scalar products, and if $\delta_j > 0$.

When an entry (contained in $\tilde{\boldsymbol{g}}$) has to be deleted, we have instead

$$\tilde{\boldsymbol{q}}_{ij} = \boldsymbol{q}_{ij} - \tilde{g}_i\tilde{g}_j \qquad \text{i.e.} \qquad \tilde{\boldsymbol{Q}}_\beta = \boldsymbol{Q}_\beta - \tilde{\boldsymbol{g}}\,\tilde{\boldsymbol{g}}^T$$

that can be taken in the same form as above by considering that

$$\tilde{\boldsymbol{Q}}_B = [\ i\tilde{\boldsymbol{g}}_B\ \bar{\boldsymbol{L}}_B\ ]\cdot\begin{bmatrix} i\tilde{\boldsymbol{g}}^T_B \\ \bar{\boldsymbol{L}}^T_B \end{bmatrix}$$

where $i^2 = -1$: then, almost the same arguments as the above case hold, i.e. by using *complex* Givens matrices of the form

$$\boxed{\begin{matrix}1 & & & \\ & \ddots & & \\ \iddots & & \ddots & \\ \ddots & & & \\ & & & 1\end{matrix}}$$

where $c^2 - s^2 = 1$, we can obtain

$$\begin{bmatrix} \mathbf{0} & \tilde{\boldsymbol{L}}_{B'} \\ 0 & \tilde{\boldsymbol{l}}^T_h \\ 0 & \tilde{\boldsymbol{l}}^T_j \\ i\sigma_1 & \tilde{\boldsymbol{z}}_1 \\ i\sigma_2 & \tilde{\boldsymbol{z}}_2 \end{bmatrix}$$

In other words, it is easy to show that such matrices keeps the imaginary part on the $i\tilde{\boldsymbol{g}}_B$ column alone, thus leaving $\bar{\boldsymbol{L}}_B$ made of pure reals, that the newly obtained $\tilde{\boldsymbol{L}}_{B'}$ (and the relative vectors) is what we expect, that $\tilde{\boldsymbol{z}}_1\tilde{\boldsymbol{z}}_2 = \boldsymbol{z}_1\boldsymbol{z}_2 - \sigma_1\sigma_2$ (and the same for the other scalar products) and that $\delta_h = \delta_j = 0$, i.e. no "new linear independence" is created; conversely, in this case zeroes along the diagonal of $\tilde{\boldsymbol{L}}_{B'}$ may be generated. For ease of exposition, suppose that the current base B contains no linearly dependent items and that $B = [\ B'\ \ h\ ]$ (i.e. h is the last item in base - note that bases are actually ordered sets, the order being that of the rows of $\bar{\boldsymbol{L}}_B$): after applying m - 1 (complex) Givens rotations, one may find

$$\tilde{\boldsymbol{L}}_B = \begin{bmatrix} \mathbf{0} & \tilde{\boldsymbol{L}}_{B'} & \mathbf{0} \\ i\delta_h & \tilde{\boldsymbol{l}}^T_h & \tilde{l}_{hh} \end{bmatrix} \qquad \text{where} \qquad \tilde{l}^{\,2}_{hh} = \delta^2_h$$

and no complex Givens matrix nullifying $i\delta_h$ can be found; however, it is also true that

$$\tilde{\boldsymbol{L}}_B = \begin{bmatrix} \mathbf{0} & \tilde{\boldsymbol{L}}_{B'} & \mathbf{0} \\ 0 & \tilde{\boldsymbol{l}}^T_h & 0 \end{bmatrix}$$

i.e. by simply "marking" h as linearly dependent a consistent situation is naturally recovered. The above relation holds true even if B already contains linearly dependent items, even though if h is going to be the third of them one of the three must be removed from B (this can be done in O(1)): furthermore, if $\tilde{l}^{\,2}_{hh} = \delta^2_h$ for h some in position $k < m'$, then with a sweep of $m' - k - 1$ Givens rotations (exactly the same as in Proposition 2.8) and a permutation of B the h-th item can be put in the last position, and the method applied.

As in §2, we have a set of (cheap) operations that can be performed on the data of the problem, while keeping the whole set of data structures consistent and ready for a "warm" restart: this will be used in the next paragraph in order to efficiently cope with box constraints on the primal space.

5. Handling box constraints on the primal space

In many applications, typically when Λ is the nonnegative orthant of $\mathcal{R}^n$, (most of) the constraints $\boldsymbol{gd} \leq \alpha$ on $\boldsymbol{d}$ are just "boxes" $l_i \leq d_i \leq u_i$: in this paragraph, we will show how to efficiently handle lower bounds $l_i \leq d_i$ on a (sub)set γ of the variables, since the extension to the more general class is straightforward. Our (dual) method will still use an active-set strategy, in which only a subset $\chi \subseteq \gamma$ of the dual variables associated to the box constraints is allowed to be nonzero at a given time: since we won't deal with the part of the base concerning subgradients, to simplify the notation we drop the subscript β for columns and we consider the matrix $\boldsymbol{G}$ partitioned as

$$\boldsymbol{G} = \begin{bmatrix} \boldsymbol{G}_\chi \\ \boldsymbol{G}_\eta \end{bmatrix}$$

(i.e. now subscripts indicates rows), where $\eta = \{ 1 .. n \} / \chi$; the *restricted* problems are

$$(\Pi_\chi) \qquad \min \{ v + 1/2 \cdot \| \boldsymbol{d} \|^2 : v\boldsymbol{e} \geq \boldsymbol{Gd} - \alpha \,,\, l_\chi \leq d_\chi \}$$

$$(\Delta_\chi) \qquad \min \left\{ 1/2 \cdot \left\| \begin{bmatrix} \boldsymbol{G}_\chi & \boldsymbol{-I} \\ \boldsymbol{G}_\eta & \boldsymbol{0} \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y}_\chi \end{bmatrix} \right\|^2 + [\, \alpha \;\; \boldsymbol{-l}_\chi \,] \cdot \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y}_\chi \end{bmatrix} : \boldsymbol{x} \in \Theta \,, \boldsymbol{y}_\chi \geq \boldsymbol{0} \right\}$$

and therefore the "extended" Hessian matrix w.r.t. the "extended base" $\bar{B} = \{ 1 .. m \} \cup \chi$ is

$$\boldsymbol{Q}_{\bar{B}} = \begin{bmatrix} \boldsymbol{G}_\chi^T & \boldsymbol{G}_\eta^T \\ \boldsymbol{-I} & \boldsymbol{0} \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{G}_\chi & \boldsymbol{-I} \\ \boldsymbol{G}_\eta & \boldsymbol{0} \end{bmatrix} = \begin{bmatrix} \boldsymbol{G}^T\boldsymbol{G} & \boldsymbol{-G}_\chi^T \\ \boldsymbol{-G}_\chi & \boldsymbol{I} \end{bmatrix} = \begin{bmatrix} \boldsymbol{Q} & \boldsymbol{-G}_\chi^T \\ \boldsymbol{-G}_\chi & \boldsymbol{I} \end{bmatrix}$$

For feasible solution of (Δ_χ) to be optimal, it has to solve the system

$$\begin{bmatrix} \boldsymbol{Q} & \boldsymbol{-G}_\chi^T \\ \boldsymbol{-G}_\chi & \boldsymbol{I} \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y}_\chi \end{bmatrix} = \begin{bmatrix} \rho\boldsymbol{e} - \alpha \\ \boldsymbol{l}_\chi \end{bmatrix} \qquad \text{for some } \rho \in \mathcal{R}$$

so that one can immediately deduce that

$$\boldsymbol{y}_\chi = \boldsymbol{G}_\chi\boldsymbol{x} + \boldsymbol{l}_\chi \qquad \text{and therefore} \qquad [\, \boldsymbol{G}^T\boldsymbol{G} - \boldsymbol{G}_\chi^T\boldsymbol{G}_\chi \,]\boldsymbol{x} = \rho\boldsymbol{e} - [\, \alpha - \boldsymbol{G}_\chi^T\boldsymbol{l}_\chi \,]$$

But since $\boldsymbol{G}^T\boldsymbol{G} - \boldsymbol{G}_\chi^T\boldsymbol{G}_\chi = \boldsymbol{G}_\eta^T\boldsymbol{G}_\eta = \boldsymbol{Q}_\eta$, the *relaxed* problem

$$(R_\chi) \qquad \min \left\{ 1/2 \cdot \left\| \begin{bmatrix} \boldsymbol{G}_\chi & \boldsymbol{-I} \\ \boldsymbol{G}_\eta & \boldsymbol{0} \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y}_\chi \end{bmatrix} \right\|^2 + [\, \alpha \;\; \boldsymbol{-l}_\chi \,] \cdot \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y}_\chi \end{bmatrix} : \boldsymbol{x} \in \Theta \right\}$$

(in which the constraints $\boldsymbol{y}_\chi \geq \boldsymbol{0}$ are disregarded) is equivalent to the *unconstrained* problem

$$(\Delta^\eta) \qquad \min \{ 1/2 \cdot \boldsymbol{x}^T\boldsymbol{Q}_\eta\boldsymbol{x} + \bar{\alpha}\boldsymbol{x} : \boldsymbol{x} \in \Theta \}$$

where $\bar{\alpha} = \alpha - \boldsymbol{G}_\chi^T\boldsymbol{l}_\chi$ and all the entries of the $\boldsymbol{g}_h$s corresponding to active constraints (whose indices are in χ) have been deleted: therefore, we can solve the box-constrained problem (Π_γ) by using the TT algorithm for the unconstrained case as a "black box" to find a base B and a solution $\boldsymbol{x} = [\, \boldsymbol{x}_B \; 0 \,]$ such that $[\, \boldsymbol{x} \;\; \boldsymbol{y}_\chi \,]$ solves the (KT) system for (Δ_χ).

In other words, we have an (efficient) subroutine TT(χ) that reports and optimal solution $\boldsymbol{x}$ of (Δ_χ): building on this, we can construct a "two-level" solver for the box constrained, where the lower level (the TT algorithm) takes care of the $\boldsymbol{x}$ variables, while the upper level handles the $\boldsymbol{y}$ variables: it is immediately clear, just looking at the pseud-code below, that BTT can be viewed as TT applied to (Δ_γ), with just a different way of solving the (KT) system and a particular ordering in which constraints and subgradients are chosen for being inserted to and deleted from the "extended base" $\bar{B} = B \cup \chi$; in fact, convergence can be shown exactly in the same way (there cannot be infinitely many inner iterations, and every main iteration ensures a strict decrease) and all the data structures relative to a base B of the subproblem (Δ^η) have a straightforward interpretation in terms of the base $\bar{B}$ of the original problem (Δ_γ) - for instance, it is easy to see that

$$[\, \boldsymbol{x}^T \;\; \boldsymbol{y}_\chi^T \,] \cdot \begin{bmatrix} \boldsymbol{Q} & \boldsymbol{-G}_\chi^T \\ \boldsymbol{-G}_\chi & \boldsymbol{I} \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y}_\chi \end{bmatrix} = \boldsymbol{x}^T\boldsymbol{Q}_\eta\boldsymbol{x} + \| \boldsymbol{l} \|^2 \quad \text{and} \quad [\, \alpha \;\; \boldsymbol{-l}_\chi \,] \cdot \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y}_\chi \end{bmatrix} = \bar{\alpha}\boldsymbol{x} - \| \boldsymbol{l} \|^2$$

and that $\boldsymbol{d} = - [\, \boldsymbol{l}_\chi \;\; \boldsymbol{G}_\eta\boldsymbol{x} \,]$, therefore (by letting v', $\boldsymbol{d}'$ and $f'()$ be relative to (Δ^η))

$$f_{\bar{B}}(\,[\,\boldsymbol{x}_B\ \ \boldsymbol{y}_\chi\,]\,) = f'_{\bar{B}}(\,\boldsymbol{x}_B\,) \qquad v = v' \qquad \boldsymbol{d} = [\,\boldsymbol{l}_\chi\ \boldsymbol{d}'\,] \qquad \boldsymbol{G}^T\boldsymbol{d} = \boldsymbol{G}_\eta^T\boldsymbol{d}' - \boldsymbol{G}_\chi^T\boldsymbol{l}_\chi$$

BTT Algorithm:

```
$\chi = \emptyset$; $x$ = TT($\chi$); $y_\chi = G_\chi x + l_\chi$;                  /* initialisation */
    while( $\exists\ i \in (\gamma / \chi)$ such that $l_i > - G_i x$ )       /* main iteration */
        $\chi = \chi \cup \{i\}$;                                            /* item insertion - $y_i = G_i x + l_i > 0$ */
        do                                                             /* inner iteration */
            $x$ = TT($\chi$); $\bar{y}_\chi = G_\chi x + l_\chi$;                  /* KT solution - use TT as subroutine */
            if( $\exists\ i \in \chi$ such that $l_i < - G_i x$ )              /* $\bar{y}_i = G_i x + l_i < 0$ */
                then $w_\chi = \bar{y}_\chi - y_\chi$;                           /* $w_\chi$ is of descent in the $y_\chi$ space */
                    $\eta = \min\{ -w_i / y_i : w_i < 0 , i \in \chi \}$;     /* maximum feasible step $\eta > 0$ */
                    $y_\chi = y_\chi + \eta w_\chi$;                           /* take step $\eta$ along $w_\chi$ */
                else $y_\chi = \bar{y}_\chi$; $w_\chi = 0$;                        /* $\bar{y}_\chi$ is optimal for ($\Delta_\chi$) */
            forall( $i$ such that $y_i = 0$ )
                $\chi = \chi / \{i\}$;                                       /* item(s) deletion */
    while( $w_\chi \neq 0$ );                                            /* end inner iteration */
endwhile                                                               /* end main iteration */
```

This "hierarchic" view of the method greatly simplifies the design and the implementation of the algorithm, but due to our sophisticated reoptimization strategies discussed in §2 and §4 performances are even better than those of the "monolithic" approaches known in the literature: obviously, the above algorithm immediately extends to $u_i \geq d_i$ constraints (where $y_i = - u_i - \boldsymbol{G}_i\boldsymbol{x}$) and to general boxes $l_i \leq d_i \leq u_i$, since the two constraints $d_i \leq u_i$ and $l_i \leq d_i$ cannot be in the active set χ at the same time.

A point still needs clarifications: the (expensive) matrix-vector product $\boldsymbol{G}_\chi^T\boldsymbol{l}_\chi$ is not calculated from scratch at every iteration, since e.g. when $\chi' = \chi \cup \{i\}$ one has

$$\bar{\alpha}' = \alpha + \boldsymbol{G}_{\chi'}^T\boldsymbol{l}_{\chi'} = \alpha - [\ \boldsymbol{G}_\chi^T\ \ \tilde{\boldsymbol{g}}\ ]\cdot\begin{bmatrix}\boldsymbol{l}_\chi\\ l_i\end{bmatrix} = \bar{\alpha} - l_i\tilde{\boldsymbol{g}}$$

where $\tilde{\boldsymbol{g}} = \boldsymbol{G}_i$ is the i-th row of $\boldsymbol{G}$; however, this is not enough, since the vector $\boldsymbol{z}_2 = \boldsymbol{L}_{B'}^{-1}\bar{\alpha}$ also have to be updated without resorting to an expensive backsolve on $\boldsymbol{L}_{B'}$. In fact, in the above case we have

$$\tilde{\boldsymbol{z}}_2 = \tilde{\boldsymbol{L}}_{B'}^{-1}\bar{\alpha}' = \tilde{\boldsymbol{L}}_{B'}^{-1}\bar{\alpha} - l_i[\ \tilde{\boldsymbol{L}}_{B'}^{-1}\tilde{\boldsymbol{g}}\ \ ]$$

but the method discussed in §4 finds $\tilde{\boldsymbol{L}}_{B'}^{-1}\bar{\alpha}$ instead; however, it is easy to recover the "perturbation" vector $\tilde{\boldsymbol{L}}_{B'}^{-1}\tilde{\boldsymbol{g}}$, since the basic relation of the method is the existence of an orthogonal $(m + 1)\times(m + 1)$ matrix $\boldsymbol{G}_S$ such that

$$[\ \tilde{\boldsymbol{g}}\ \boldsymbol{L}_{B'}\ ]\boldsymbol{G}_S = [\ \boldsymbol{0}\ \tilde{\boldsymbol{L}}_{B'}\ ] \qquad \Rightarrow \qquad [\ \tilde{\boldsymbol{g}}\ \boldsymbol{L}_{B'}\ ] = [\ \boldsymbol{0}\ \tilde{\boldsymbol{L}}_{B'}\ ]\boldsymbol{G}_S^T$$

and therefore, by letting $[\ \upsilon_0\ \upsilon^T\ ] = [\ 1\ \ \boldsymbol{0}^T\ ]\boldsymbol{G}_S$ (this vector can be found in linear time) one has

$$\tilde{\boldsymbol{L}}_{B'}\upsilon = [\ \boldsymbol{0}\ \tilde{\boldsymbol{L}}_{B'}\ ]\cdot\begin{bmatrix}\upsilon_0\\ \upsilon\end{bmatrix} = [\ \boldsymbol{0}\ \tilde{\boldsymbol{L}}_{B'}\ ](\boldsymbol{G}_S^T)\cdot\begin{bmatrix}1\\ \boldsymbol{0}\end{bmatrix} = [\ \tilde{\boldsymbol{g}}\ \boldsymbol{L}_{B'}\ ]\cdot\begin{bmatrix}1\\ \boldsymbol{0}\end{bmatrix} = \tilde{\boldsymbol{g}}$$

i.e. $\upsilon = \tilde{\boldsymbol{L}}_{B'}^{-1}\tilde{\boldsymbol{g}}$ and $\tilde{\boldsymbol{z}}_2$ can be obtained in linear time from $\tilde{\boldsymbol{L}}_{B'}^{-1}\bar{\alpha}$. Since the case $\chi' = \chi / \{i\}$ is analogous, with $\bar{\alpha}' = \alpha + l_i\tilde{\boldsymbol{g}}$ and $[\ \upsilon_0\ \upsilon^T\ ] = [\ -i\ \ \boldsymbol{0}^T\ ]\boldsymbol{G}_S$, we have illustrated all the basic elements that are necessary to efficiently use TT as a subroutine for solving box constrained problems: a number of further details can be easily derived from all the above theory, for instance how to keep the scalar product updated ($\boldsymbol{z}_1\tilde{\boldsymbol{z}}_2 = \boldsymbol{z}_1\boldsymbol{z}_2 \pm \upsilon_0\sigma_1 l_i$ for σ_1 defined in §4), how to update $\boldsymbol{l}_\chi$ when moving "along" $\boldsymbol{d}$ of a step τ ($\boldsymbol{l}_\chi := \boldsymbol{l}_\chi \cdot (1 - \tau/t)$) or that all the above results holds true for the generalised problem of §3 with constraint $\boldsymbol{cx} = 1$, hence we now describe the major implementation issues of our actual C++ implementation of the [B]TT algorithm.

6. Implementative details

The [B]TT algorithm has been developed in the context of an experimental module for Nondifferentiable Optimization [CFN95], conceived to be mainly used for solving Lagrangean Duals of large-scale (Integer) Linear Problems: the code is written in C++, using its powerful inheritance and polymorphism capabilities in that the whole TT algorithm is embedded in one single *object*, `class MinQuad`, with an abstract interface containing the (only) *methods* allowing the NDO algorithm to modify and view the data structures (β, $\boldsymbol{Q}_\beta$, α_β, t ...) and to perform the actual calculations.
The BTT algorithm is contained in the *derived* `class BMinQuad`, *inheriting* from the *base class* most of its interface and code: in this way, modularity and efficiency are maximised at the same time, since the highly efficient TT algorithm can be used when no box constraints are required, while only minor modifications in the NDO solver are required in order to use the more general BTT algorithm; this also makes code maintenance easier, and allow for improvements in the TT code to be automatically exploited by BTT - actually, as long as the interface remains the same one might also replace TT with an entirely different algorithm, and we intend to exploit this possibility in the future.
Also, the code is an excellent base for developing NDO solvers, since it handles (uniformly for the TT and BTT case) most of the data structures and tasks that an NDO solver must manage: for example, ($\boldsymbol{g}_i$, α_i) is added [removed] from β by simply calling `AddSubGradient(`i, α_i`)` [`ResetBundle(`i`)`] and "movements" in the space of Λ are captured by simple calls such as `ChangeAlfa(`α`)` or `MoveAlongD(`Δ_o, τ`)` so that handling of the α vector can be entirely demanded to the class; furthermore, the calling codes have full and easy control on the variables of the "primal" problem (Π) (with functions such as `Add[Remove]Variable(`i`)`) and on the set γ of box constraints, that can be easily changed.
After that the a problem has been solved, the NDO solver can query some relevant informations that are calculated during the algorithm: examples are the vector $\boldsymbol{G}^T\boldsymbol{d}$ and three numbers $l_1 \geq 0$, l_2 and $l_3 \geq 0$ that describe the optimal solutions values of (Δ_β) and (Π_β) as a function of t in a neighbourhood of the current value, i.e.

$$\| \boldsymbol{Gx}(t) \|^2 = \| \boldsymbol{d}(t) \|^2 = l_1 + (1/t)^2 l_3$$

$$\alpha\boldsymbol{x}(t) = l_2 - (1/t)l_3$$

$$v(t) = - l_1 - (1/t)l_2$$

The above values can be obtained at essentially no cost, since it is easy to verify that

	Case (0)	Case (1)
l_1	$1 / \| \boldsymbol{z}_1 \|^2$	0
l_2	$\boldsymbol{z}_1\boldsymbol{z}_2 / \| \boldsymbol{z}_1 \|^2$	$\rho = (\alpha_h - \boldsymbol{l}_h\boldsymbol{z}_2) / (1 - \boldsymbol{l}_h\boldsymbol{z}_1)$
l_3	$[\| \boldsymbol{z}_1 \|^2 \| \boldsymbol{z}_2 \|^2 - (\boldsymbol{z}_1\boldsymbol{z}_2)^2] / \| \boldsymbol{z}_1 \|^2$	$\| \rho\boldsymbol{z}_1 - \boldsymbol{z}_2 \|^2 + \| \boldsymbol{l}_\chi \|^2$

and all the data necessary to the calculations is either already calculated during the algorithm, or can be kept updated in O(1) per iteration: if needed, the NDO solver can query other (a bit more costly) sensitivity analysis informations, such as the "stability interval" $t_m \leq t \leq t_M$ in which the currently optimal base B^* remains optimal (hence the above estimates on $\| \boldsymbol{d}(t) \|^2$, $\alpha\boldsymbol{x}(t)$ and $v(t)$ are exact) and two vectors $\boldsymbol{x}_1$, $\boldsymbol{x}_2$ such that, within $[t_m, t_M]$, $\boldsymbol{x}(t) = \boldsymbol{x}_1 + (1/t)\boldsymbol{x}_2$.

An important issue in any numerical algorithm is its resistance to ill-conditioned instances and to cumulation of rounding errors: in our code, the main problem is represented by some "$y = 0$" tests, that are crucial to determine the behaviour of the algorithm, i.e. the choice between cases (0), (1) and (2) and between "normal" and "infinite" steps. We addressed the problem by using the two dynamic tolerances adjustment methods: the "static" one consists in scaling the "dynamic" relative tolerance ε_R by implementing the tests as $|y| < \varepsilon_R \cdot \mu(\boldsymbol{L}_{B'}) \cdot m \cdot \delta$ (where $\mu()$ is the conditioning number and δ is related to the "size" of the numbers used in the calculation); in turn, erroneous conditions rising in different parts of the algorithm are interpreted as "increase ε_R" or "decrease ε_R" requests, and a custom exception mechanism is used to dynamically adjust ε_R in order to face "bad" instances without a need for hand-tuning of the parameters. From-scratch recalculation of the main data structures ($\boldsymbol{L}_{B'}$, $\boldsymbol{z}_1$ ...), that must always be implemented in any numerical code, is also performed as a part of the "exceptions" handling mechanism.

An apparently minor detail of our implementation is that it is parametric about the precision required: in particular, different formats (from `char` to `double`) can be specified for entries of the subgradients, for storing their scalar products (the matrix $\boldsymbol{Q}$) and also for the main data structures ($\boldsymbol{L}_{B'}$, $\boldsymbol{z}_1$ ...). This is useful since often the subgradients have only small integer entries, so that the scalar products may be computed with faster integer operations, and with small problems it is often possible to use single precision arithmetic (confirming the robustness of our approach): this cannot be done without care, since the conversion between different formats is not costless, however by only choosing the right data types performances can be improved of 30% in time and of almost 50% in memory.
Other important implementative details can only be sketched here: for instance, calculation of some data ($\boldsymbol{Q}_\beta$, $\boldsymbol{d}$) may or may not be performed "lazily", at least three different memory allocation strategies for matrix $\boldsymbol{Q}_\beta$ (with different price/performance ratios) exists, it might be sometimes worthwhile to recalculate the scalar products like $\boldsymbol{z}_1\boldsymbol{z}_2$ from scratch at every iteration to enhance numerical stability, different "pricing" strategies for selecting the entering variable might be used: for any of these characteristics, our code offers different choices, that can be easily selected (with compile time switches) to tailor the code to the particular class of instances to be solved.

7. Computational results and comparisons

Barring the “classical” QP algorithm in [Mi77], that solves a related but different problem, only the two specialized algorithms for the unconstrained version of (Δ_β) [Ki86], [Mo87] have been proposed in the past: our TT algorithm bears resemblance to both, but greatly enhances upon them in terms of cost per single iteration; in [Ki89], an algorithm for a slightly more general problem than the one solved by BTT (allowing more than one independent “bundle”) is described, but, using $n \times n$ matrices, it is admittedly not well-suited for the class of instances of interest here.
During the revision of this article, our attention was drawn upon the recent [Ki94], that solves the same generalised problem as [Ki89] with much better performances: it can be easily seen that any of the basic operations (inserting/deleting items or constraints in base, finding the descent direction ...) in this last algorithm has a correspondent in BTT with almost the same cost, but for a number of important details in which our algorithm improves on it.
For example, we calculate $f(\boldsymbol{x})$ and “downdate” the triangular factor after a constraint insertion much faster, we don’t need to perform row additions and subtractions when the “basic” variable x_h changes because we don’t use “projection” (the other method actually eliminates the constraint $\boldsymbol{ex} = 1$ by replacing $\boldsymbol{Q}$ with $\tilde{\boldsymbol{Q}} = \tilde{\boldsymbol{G}}^T\tilde{\boldsymbol{G}}$, where $\tilde{\boldsymbol{G}} = [\ \boldsymbol{g}_1\text{-}\boldsymbol{g}_h \ .. \ \boldsymbol{g}_m\text{-}\boldsymbol{g}_h\ ]$ for some h), and if more than one step of the TT algorithm is performed at each inner iteration of BTT, we save time by avoiding to recalculate $\boldsymbol{d}$ (that is an “heavy” task). Furthermore, the two-level nature of BTT also helps, since the same operation may have a different impact in the two cases: for instance, while in TT it is convenient to keep the base as small as possible, in BTT it may be wasteful to delete an item without need, so that implementing different strategies for each level can result in performance improvements.

Our implementations of the TT and BTT algorithms have already been used within several different codes, such as a custom implementations of Bundle algorithms for “nonexact” [EGM95] or “exact” [CGN95] Lagrangean decompositions, a general Bundle algorithm for unconstrained optimization of polyhedral functions [CFN95] (that has in turn been used to approach several different problems, as [MP95]) and a specialized Bundle code for Multicommodity Min Cost Flow Problems [FG96]: in the following, we will compare our code and two generic QP codes when used as (Δ_β)-solvers within the NDO solvers described in [CFN95] (for *unconstrained* problems) and [FG96] (for *constrained* ones).

Anticipating a bit the conclusions, our code has shown to be blatantly faster than the two benchmark QP solvers: we want to stress the fact that this result has some clear reasons, that will be illustrated in the following, and it is limited to the specific application of interest - in no way we are claiming that our code is any better than the others outside this framework, nor we are denying the value of the different technologies used therein, like Interior Point methods.

The first competitor is the well-known QL code described in [Po83]: despite its age, it is known to be an efficient public domain QP solver, and in fact it has already been used as (Δ_β)-solver within the Bundle code described in [SZ92] - actually, the version we tested was exactly the one used in that code, that had been somehow specialized for the task by the authors and J. Outrata, translated to C with the

f2c converter by R. Freling and further adapted to C++ by ourselves. Being a "primal" code, and being not able to reoptimize from previous calls, it was expected to be slower than our code: however, such a comparison give a measure of the need for specialized QP solver that originated our research.
The second competitor code is a beta release of the Interior Point QP solver that will be included in the next release of Cplex[‡] , one of the best commercial LP solver on the market: this code is based on the "modern" IP technology [CLM93] (opposed to the "old" active-set method of BTT), and it is a commercial code, hence it is supported by sophisticated sparse linear algebra routines and state-of-the-art coding. Since it is well-known that IP methods tend to behave better on large problems with sparse Hessians such as (Π_β) (that has in fact a separable quadratic objective function), we wanted to see if such a "primal" approach could be competitive with our "dual" approach to (Δ_β).

In our experiments, we paid great attention in making the tests fair: for example, the cost of data structures initialisations and problem scaling was not counted, so that the way in which problems were stored had no influence on the test (even though in our code we pay something for having an "easy" interface with the above NDO solver), we used double precision arithmetic (but our code might as well work in single precision) and fixed a set of "compile time" choices for all the problems (rather than using its flexibility to be faster on specific sub-classes of instances). The main disadvantage of the two competitor codes is that they are not able to reoptimize efficiently, or even to exploit "external" information such as a good starting point, so that when possible we tried to provide this information in an indirect way: for instance, we don't ask CQP to solve the "full" dual formulation (Δ_γ), but rather an easier restricted problem similar to (R_χ) w.r.t. the optimal set χ (that has been previously determined by BTT).

The first comparison involved the solution of Lagrangean Duals (LD), related to finding lower bounds for the Quadratic Semi-Assignment Problem [MP95], with a general unconstrained Bundle-based NDO solver [CFN95]: in each (LD), $n = |\boldsymbol{g}_i|$ was fixed between 10 and 1600, while m (the number of $\boldsymbol{g}_i$s) varied between 1 (at the very beginning) and up to 250; to estimate the efficiency of a (Δ_β)-solver, we have focused our analysis on the "total running time" of the QP codes, i.e. the sum of the running times of all the (Δ_β)s (up to 1800) needed to solve a single (LD).
For technical reasons, we tested CQP and QL against TT on two different (but analogous) sets of problems: the results are illustrated in the following tables and charts, where for ease of visualisation we aggregated the (LD) problems in groups with similar "total size", i.e. the product $n\times$(average value of m)$\times$(number of (Δ_β)s solved) that has shown to have the largest correlation with the running times, hence is presumably a good measure of the size (and difficulty) of a (LD) problem for our purposes. Here, N is the number of (LD)s in any group, the figures are averaged within the group, QPP and QPD stand for CQP solving respectively the primal and the dual formulation of the problem, times are in seconds on an HP9000-712/80 workstation with 64 Mb of RAM, the BTT and QL codes have been compiled with the HP C++ compiler (under HP-UX 9.05) invoking normal optimisations (`-O`), while CQP was available only as executable.

[‡] Cplex is a registered trademark of Cplex Inc.

N	n	m (avg)	Calls	TT time	QL time
6	**12**	**15**	**128**	**.02**	**.16**
12	**15**	**17**	**175**	**.03**	**.36**
17	**70**	**30**	**289**	**.16**	**2.58**
25	**75**	**43**	**502**	**.62**	**9.89**
20	**94**	**50**	**701**	**1.38**	**27.59**
19	**153**	**88**	**1296**	**12.72**	**277.80**
16	**284**	**89**	**1531**	**9.50**	**295.29**
11	**686**	**108**	**1842**	**10.72**	**721.98**
4	**1506**	**111**	**1885**	**10.12**	**751.90**

Table 1: comparison between TT and QL codes

N	n	m (avg)	Calls	TT time	QPD time	QPP time
6	12	**14**	**129**	**0.03**	**1.04**	**2.01**
8	62	**26**	**197**	**0.10**	**5.03**	**20.60**
9	68	**55**	**766**	**1.75**	**76.96**	**478.97**
6	152	**83**	**1269**	**8.91**	**346.91**	**2388.25**
8	275	**87**	**1327**	**11.60**	**590.91**	**2167.44**
7	679	**112**	**1821**	**10.37**	**2424.26**	**5022.14**
6	1385	**111**	**1874**	**10.00**	**2729.15**	**7607.03**

Table 2: comparison between TT and CQP solving Primal and Dual problems

The results are better visualised in the following charts, and clearly show that our code is by far superior to both QL and CQP, especially when dealing with large problems (note that the scale in charts 1 and 3 is logarithmic): we remark again that this is not necessarily true in general, but only within the particular class of problems that we are taking into account, and considering as measure the total running time over a whole execution of the NDO solver.

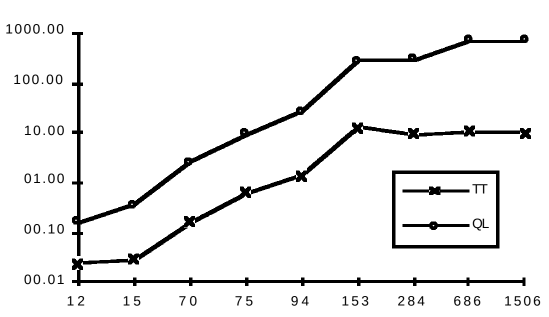

Chart 1: TT and QL running times

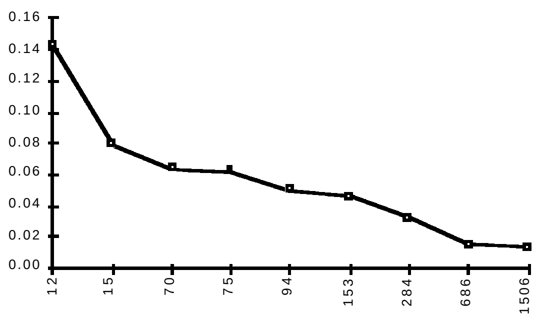

Chart 2: ratio between TT and QL running times

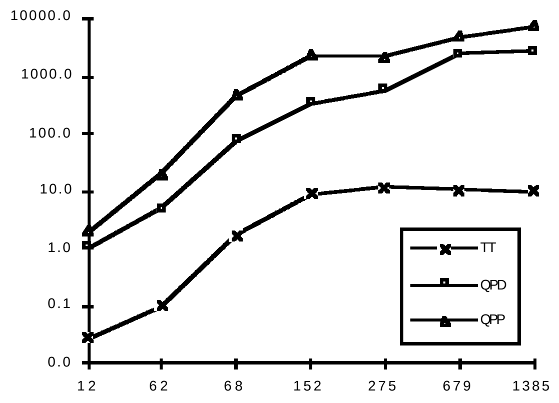

Chart 3: TT and CQP running times

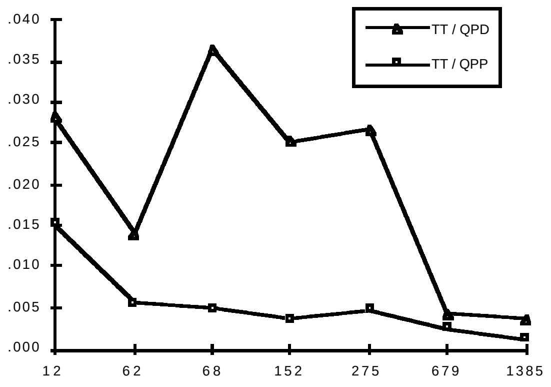

Chart 4: ratio between TT and CQP running times

The second comparison involved the solution of Multicommodity Min Cost Flow (MMCF) problems with a specialized (constrained) NDO solver [FG96]: in each problem, n (= number of edges in the graph) was fixed between 200 up to 700, while m varied between 1 and about 50; we remark that those problems are in some sense “easier” that the previous ones, since the overall NDO algorithm always terminated in less than 50 iterations (against up to 1000), so that the average value of m was much smaller, even though often more than one new item has been inserted at each iteration. Furthermore, from 94 to 99% of nonnegativity constraints were always “active”, and the average number of TT calls per BTT call were always smaller than 3 (and often almost 1 on small problems), so that the TT algorithm was never called more than 150 times (against up to 1800) during the solution of any single problem: hence, this data set is representative of a somewhat different class of problems, in which reoptimization is possibly less important than the ability of rapidly solving small-sized instances.
Due to limitations of the version of QL we had at hand, we only tested CQP in this setting: the results are reported in the following table and charts, and clearly confirm superiority of BTT, that is always more than 25 times faster than its competitor; note that for this class of problems QPP generally outperforms QPD, even though, extrapolating from the results we got, the largest instances we tried seem to represent the point in which the latter might become competitive with the former.

N	n	m (avg)	Calls	BTT time	QPD time	QPP time
7	257	2	4	.011	3.91	0.33
6	371	4	7	.013	5.46	0.96
10	487	5	10	.021	5.49	1.94
6	439	7	15	.040	7.80	2.98
5	366	9	22	.048	12.44	5.09
9	375	14	38	.149	20.39	16.09

Table 3: comparison between BTT and CQP solving Primal and Dual problems

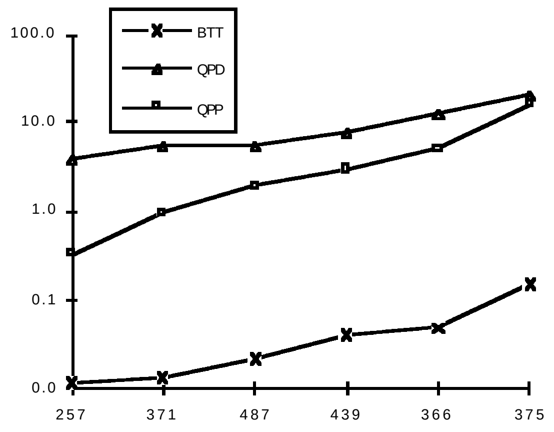

Chart 5: BTT and CQP running times

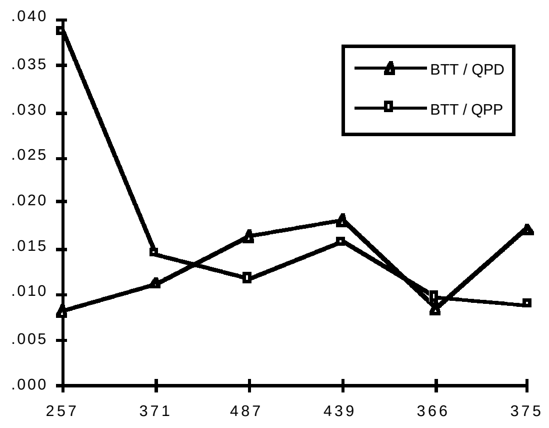

Chart 6: ratio between BTT and CQP running times

To conclude, a final remark might be worthwhile: in [FG96] it is shown how an Bundle-based approach to Multicommodity flow problems can be competitive with (and often better to) other methods; the algorithm described therein used our BTT code as (Δ_β)-solver, and, despite its efficiency, a significative fraction of the total solution time of a MMCF problem has to be imputed to the solution of the (Δ_β) subproblem. Clearly, the use of any other of the above algorithms would have dramatically changed such a positive result, and this a good example of how a specialized QP can be crucial to make Bundle algorithms attractive. In fact, Bundle methods have been since long widely recognised among experts as a very important class of NDO algorithms, but despite their theoretical and practical superiority w.r.t. simpler-to-program algorithms (such as subgradient methods), their success have been so far limited by both the cost of solving the quadratic (Δ_B) subproblems and the extra programming complexity they require, part of which is represented by the need of a suitable QP solver. Our personal experience shows that the BTT algorithm greatly reduces the overhead to be paid for the solution of (Δ_β), and our C++ implementation offers a simple an clear interface that greatly simplifies the task of developing a Bundle algorithm: we hope that this code will help researchers and practitioners interested in Nondifferentiable Optimization methods to move towards this interesting and powerful class of algorithms.

Acknowledgements

The author wish to acknowledge Prof. C. Lemaréchal for his precious advices, Prof. K.C. Kiwiel for pointing out his newest work, , Prof. K. Schittkowski and R. Freling for the FORTRAN and C version of the QL code respectively, and Prof. I. Lustig for his help with the beta release of Cplex QP solver.

References

[CFN95] P. Carraresi, A. Frangioni and M. Nonato **“Applying Bundle Methods to Optimization of Polyhedral Functions: An Applications-Oriented Development”** *Technical Report* **TR 17/95**, Dipartimento di Informatica, Università di Pisa, 1995

[CGN95] P. Carraresi, L. Girardi and M. Nonato **“Network Models, Lagrangean Relaxations and Subgradient Bundle Approach in Crew Scheduling Problems”** in *Computer Aided Scheduling of*

Public Transport (J. Paixao ed.), Lecture Notes in Economical and Mathematical Systems, Springer-Verlag, 1995

[CLM93] T.J CARPENTER, I.J. LUSTIG, J.M. MULVEY and D.F. SHANNO, **“Higher-order Predictor-Corrector Interior Point Methods with Application to Quadratic Objectives”**, *SIAM J. on Optimization*, **3**(4), p. 696-725, 1993

[EGM95] L. EQUI, G. GALLO, S. MARZIALE and A. WEINTRAUB **“A Combined Transportation and Scheduling Problem”** *Technical Report* **TR 5/95**, Dipartimento di Informatica, Università di Pisa, 1995

[FG96] A. FRANGIONI, G. GALLO “**A Bundle type Dual-ascent Approach to Linear MultiCommodity Min Cost Flow Problems**” *Technical Report* **TR 1/96**, Dipartimento di Informatica, Università di Pisa, 1996

[FV94] B. FORTE, E.R. VRSCAY “**Solving the Inverse Problem for Function/Image Approximation Using Iterated Function Systems - II. Algorithm and Computations**” *Fractals*, 2(3) p. 335-346, 1994

[Ge69] A.M. GEOFFRION “**Duality in Nonlinear Programming: a Simplified Applications-Oriented Development**” in *Perspective on Optimization: a collection of expository articles* (A.M. Geoffrion ed.), Addison-Welsey, 1972

[GMW81] P. E. GILL , W. MURRAY and M. H. WRIGHT “**Practical Optimization**” *Academic Press*, 1981

[Ki85] K.C. KIWIEL “**Methods of Descent for Nondifferentiable Optimization**” *Lecture Notes in Mathematics*, vol. **1133**, Springer-Verlag, 1985

[Ki86] K.C. KIWIEL “**A Method for Solving Certain Quadratic Programming Problems Arising in Nonsmooth Optimization**” *IMA J. Numer. Anal.* **6**, p. 137-152, 1986

[Ki89] K.C. KIWIEL “**A Dual Method for Certain Positive Semidefinite Quadratic Programming Problems**” *SIAM J. Sci. Statist. Comput.* **10**, p. 175-186, 1989

[Ki94] K.C. KIWIEL “**A Cholesky Dual Method for Proximal Piecewise Linear Programming**” *Numer. Mat.* **68**, p. 325-340, 1994

[Le89] C. LEMARÉCHAL “**Nondifferentiable Optimization**” *in Handbooks in Opns. Res. and Mgmt. Sci., vol. 1 : Optimization* (G.L. Nehmauser, A.H.G. Rinnoy Kan and M.J. Todd eds.), North-Holland, 1989

[Mi77] R. MIFFLIN “**A Feasible Descent Algorithm for Linearly Constrained Least Squares Problems**” *in Nonsmooth Optimization*, vol. **3** of IIASA Proceedings Series (C. Lemaréchal, R. Mifflin eds.), Pergamon Press

[MP95] F. MALUCELLI, D. PRETOLANI “**Lower Bounds for the Quadratic Semi-Assignment Problem**” *EJOR* **83**(2), 1995

[Mo87] M. F. MONACO “**An Algorithm for the Minimisation of a Convex Quadratic Function over a Simplex**” *Internal Report* **56**, Dipartimento di Sistemi, Università della Calabria, 1987

[Po83] M. POWELL “**ZQPCVX: A FORTRAN Subroutine for Convex Quadratic Programming**” *Report DAMTP* **1983/NA17**, University of Cambridge, 1983

[SZ92] H. SCHRAMM, J. ZOWE “**A Version of the Bundle Idea for Minimising a Nonsmooth Function: Conceptual Idea, Convergence Analysis, Numerical Results**” *SIAM J. Optimization*, **2**(1), p. 121-152, 1992