I.3 The (QP) Subproblem

While the previous Chapters have discussed the *theoretical* meaning of various direction finding subproblems, the present one is entirely devoted to their *efficient numerical solution*: the attention will be focused on the "standard" ($\Delta_{\beta t}$) and ($\Pi_{\beta t}$) problems of §I.1.1 and some of their generalizations. For notational convenience, a slightly different formulation will be considered throughout this Chapter: in the problems arising from *minimization* of a *convex* function, by multiplying the objective function of ($\Delta_{\beta t}$) by $1 / t$ one obtains

($\Delta_{\beta t}$) $\quad \min_{\theta} \{ \frac{1}{2} \| \boldsymbol{G}_{\beta} \theta \|^2 + (\frac{1}{t}) \alpha_{\beta} \theta : \theta \in \Theta \}$

($\Pi_{\beta t}$) $\quad \min_{d,v} \{ v + \frac{1}{2} \| \boldsymbol{d} \|^2 : v \geq \boldsymbol{dg}_i - (\frac{1}{t}) \alpha_i \;\; i \in \beta \}$

(note the changes in the signs), where each occurrence of the linearization errors is multiplied by $1 / t$. Hence, t can be "embedded" in the vector α: in this Chapter, each occurrence of α_i should be read (unless explicitly stated) as $(1/t)\alpha_i$, and the problems become

(Δ_{β}) $\quad \min_{\theta} \{ \frac{1}{2} \| \boldsymbol{G}_{\beta} \theta \|^2 + \alpha_{\beta} \theta : \theta \in \Theta \}$

(Π_{β}) $\quad \min_{d,v} \{ v + \frac{1}{2} \| \boldsymbol{d} \|^2 : v \geq \boldsymbol{dg}_i - \alpha_i \;\; i \in \beta \}$.

Both (Π_{β}) and (Δ_{β}) are Semidefinite Quadratic Problems and could be solved by standard SQP codes [GMW81] [Po83] [CLM93], but the need for low solution times calls for the development of a specialized code that exploits 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 α and/or of 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, in the solution of large-scale Lagrangean Duals the "long" subgradients ($m \approx 10^3$ - 10^5) are obtained by solving a (possibly hard) optimization problem, so that β is usually "small" ($p = | \beta | \approx 10^2$): hence, ($\Delta_{\beta}$) has much fewer variables and simpler constraints than (Π_{β}), even though the latter has a much simpler (separable) objective function - in fact, most of the specialized algorithms proposed in the literature [Ki86] [Mo87] [Ki94] work primarily on (Δ_{β}).

In this Chapter, a new active-set method for the solution of (Δ_{β}) is proposed, that enhances upon the known specialized approaches by distinguishing among bases with different properties and exploiting their structure in order to reduce the computational cost of the basic operations: in §I.3.1 an overview of the method is given, and simple proofs of convergence are presented, while in §I.3.2 the details of

the matrix factorisation methods are discussed. In §I.3.3 the method is extended to a more general version of (Π_β) with (general) linear constraints on the dual solution $\boldsymbol{d}$, while in §I.3.4 it is shown how to reoptimize after changes in the number of variables of $\boldsymbol{d}$: this is used in §I.3.5 to develop a specialized algorithm for (Π_β) with box constraints on the dual space. Then, in §I.3.6 the relevant implementative details of the actual C++ code are discussed, and in §I.3.7 some computational comparisons are reported to show the effectiveness of the approach with respect to other non-specialized (QP) codes and the importance of an ad-hoc approach to the solution of (Δ_β) for the overall efficiency of a Bundle method.

I.3.1 An overview of the algorithm

The basic theory of the TT algorithm (whose name comes from the typical "shape" of the $\boldsymbol{G}_\beta$ matrix for this class of applications, i.e. Tall and Thin) is presented here in a simple, self-contained form: by Quadratic Duality [Ge69], a feasible solution θ of (Δ_β) is optimal if and only if

$$\boldsymbol{d} = -\boldsymbol{G}_\beta\theta \qquad \text{and} \qquad v = -\boldsymbol{d}^T\boldsymbol{d} - \alpha_\beta\theta$$

are feasible for (Π_β) and vice-versa, since for any $\theta \in \Theta$ ($\theta \geq \boldsymbol{0}$, $\boldsymbol{e}\theta = 1$) and (v, $\boldsymbol{d}$) as above such that $v\boldsymbol{e} \geq \boldsymbol{d}\boldsymbol{G}_\beta - \alpha_\beta$, the *Complementary Slackness Conditions* $(\alpha_\beta - \boldsymbol{d}\boldsymbol{G}_\beta + v\boldsymbol{e})\theta = 0$ hold, so that θ are feasible dual multipliers for (Π_β). The "primal" problem can be rewritten as

(Δ_β) $\qquad \min_\theta \{ f_\beta(\theta) = {}^1/_2\theta^T\boldsymbol{Q}_\beta\theta + \alpha_\beta\theta : \theta \in \Theta \}$

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 $f_\beta()$ over the unitary simplex: the algorithm is of the feasible descent type, generating a finite sequence $\{ \bar{\theta}_i \}$ of feasible points with strictly decreasing objective function values, and uses the well-known *active set* strategy [GMW81] - attaching at each step a restriction of (Δ_β) in which some of the variables are fixed to zero - to handle the constraints.

Definition I.3.1.1: *given any subset* $B \subseteq \beta$ *(a* ***base****), we set*

(Δ_B) $\qquad \min_\theta \{ f_B(\theta) = {}^1/_2\theta\boldsymbol{Q}_B\theta + \alpha_B\theta \ : \theta \in \Theta \}$

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

As for the bundle β, bases will be considered sets of indices as well as sets of subgradients (an element of the base will be called an *item*): some other notational simplifications, such as avoiding to distinguish sets Θ with different dimensions and omitting transpose signs, will also be made when clarity is not affected. Each step is characterised by the *base* B, the corresponding $\boldsymbol{Q}_B$, $\boldsymbol{G}_B$ ... of subproblem (Δ_B) and the *current feasible point* $\theta = [\ \theta_B\ \boldsymbol{0}\ ] \in \Theta$: at each step, a *descent direction* $\boldsymbol{w}$ (for $f_\beta()$ at θ) is found, and B is revised until a solution that is optimal for both (Δ_B) and the whole (Δ_β) is reached. The *inequality constrained* subproblem (Δ_B) is tackled by considering its *equality constrained* relaxation

(R_B) $\qquad \min_\theta \{ f_B(\theta_B) : \boldsymbol{e}\theta_B = 1 \}$

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

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

hold, i.e. iff at least a solution $[\ \bar{\theta}_B\ \rho\ ]$ of the linear system

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

exists: finding a solution of the (KT) system is the basic step of the algorithm, described in Figure 14, and it will be shown in §I.3.2 how to (efficiently) construct either the unique $[\ \bar{\theta}_B\ \rho\ ]$, if uniqueness holds, or a direction $\boldsymbol{w}_B$ such that $\boldsymbol{w}_B\boldsymbol{Q}_B = \boldsymbol{0}$ and $\boldsymbol{e}\boldsymbol{w}_B = 0$ otherwise.

TT Algorithm

```
B = { 1 }; θ = [ 1, 0, … , 0 ];                          /* initialisation */
while( ∃ h ∉ B such that v < g_h d − α_h )                /* main iteration */
    B = B ∪ { h };                                       /* item insertion */
    do                                                   /* inner iteration */
        if( the (KT) system admits unique solution [ θ̄_B ρ ] )
            then if( θ̄_B ≥ 0 )   then θ = [ θ̄_B 0 ]; w_B = 0;
                                 else w_B = θ̄_B − θ_B;
          else ⟨ find a feasible (descent) direction w_B s.t. w_B Q_B = 0 and e w_B = 0 ⟩;
        if( w_B ≠ 0 )                                    /* θ̄_B is not optimal for (Δ_B) */
            then  η = min{ −w_h / θ_h : w_h < 0 , h ∈ B  };   /* maximum feasible step η > 0 */
                  θ_B = θ_B + η w_B;                     /* take step η along w_B */
        forall( h such that θ_h = 0 )
            B = B / { h };                               /* item(s) deletion */
    while( w_B ≠ 0 );                                    /* end inner iteration */
endwhile                                                 /* end main iteration */
```

Figure 14: the TT algorithm

If (the unique) $\bar{\theta}_B$ is feasible, i.e. $\bar{\theta}_B \geq \boldsymbol{0}$, then it is optimal for (Δ_B) too, otherwise, by convexity of $f_B()$, $\boldsymbol{w}_B = \bar{\theta}_B - \theta_B$ is a direction of descent (and $\boldsymbol{e}\boldsymbol{w}_B = 0$): furthermore, it is also feasible, i.e. $\theta_B + \eta\boldsymbol{w}_B \geq \boldsymbol{0}$ for a sufficiently small $\eta > 0$. If the current inner iteration is *not* the first of a main iteration, then $\theta_B > \boldsymbol{0}$ (all the h such that $\theta_h = 0$ 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 $B = B' \cup \{\ h\ \}$, $\theta_B = [\ \theta_{B'}\ \theta_h\ ] = [\ \theta_{B'}\ 0\ ]$, $\theta_{B'} > \boldsymbol{0}$ is optimal for $(\Delta_{B'})$ and the h-th dual constraint is violated, i.e. $v < \boldsymbol{g}_h\boldsymbol{d} - \alpha_h$. In this case, it is possible to show that $\boldsymbol{w}_B = [\ \boldsymbol{w}_{B'}\ w_h\ ] = \bar{\theta}_B - [\ \theta_{B'}\ 0\ ]$ is feasible, i.e. that $w_h > 0$ (no other entry can create problems): for $\boldsymbol{d} = -\boldsymbol{G}_{B'}\theta_{B'}$ and $v = -\left\|\boldsymbol{d}\right\|^2 - \alpha_{B'}\theta_{B'}$ (the optimal solutions of $(\Pi_{B'})$), the KT conditions can be rewritten

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

that, (right)multiplying both sides for $\theta_{B'}^T$ and using $\boldsymbol{e}\theta_{B'} = 1$, transforms into

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

so that $\quad v = \boldsymbol{d}\boldsymbol{g}_h - \alpha_h \quad$ for each $h \in B'$

i.e. all the (dual) constraints in B' are satisfied as equations and the optimal multiplier ρ associated with the (primal) constraint $\boldsymbol{e}\theta_{B'} = 1$ is just $-v$. Then, for any $\boldsymbol{w} = [\ \boldsymbol{w}'\ w_h\ ]$ such that $\boldsymbol{e}\boldsymbol{w} = 0$

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

where $\boldsymbol{Q}_{B'h} = \boldsymbol{G}_B^T \boldsymbol{g}_h$: using the optimality of $\theta_{B'}$ ($\Rightarrow \boldsymbol{Q}_{B'} \theta_{B'} + \alpha_{B'} = -v\boldsymbol{e}$) and $\boldsymbol{e}\boldsymbol{w} = 1$ ($\Rightarrow \boldsymbol{e}\boldsymbol{w}' = -w_h$) the first-order term

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

can be rewritten as

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

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 of descent, and, by convexity of $f_B()$, $\boldsymbol{w}_B$ was already know to be a direction of descent. If $\boldsymbol{w}_B \boldsymbol{Q}_B = \boldsymbol{0}$ (an *infinite direction*), from the previous formulas

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

and therefore one among $\pm \boldsymbol{w}_B$ (depending on the sign of the scalar product) is a direction of descent: if $\theta_B > \boldsymbol{0}$ there can be no problems, while $\theta_B = [\, \theta_{B'} \; 0 \,]$ one has

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

and therefore by choosing the feasible one among $\pm \boldsymbol{w}_B$ (the one with $\pm w_h > 0$), a feasible direction is also obtained.

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 θ_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 the TT algorithm is almost the same as that of [Ki86] and [Mo87], the above proof of convergence is much simpler than the ones available in the literature: actually, all these methods differ essentially for how the (KT) system is solved, and the next paragraph will show how this task is accomplished in our algorithm.

I.3.2 Solving the (KT) system

At each step, the (KT) system is solved (or an infinite direction is determined) by essentially keeping a *lower trapezoidal factorisation* of the Hessian matrix $\boldsymbol{Q}_B$ of the current subproblem (Δ_B), that is

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

where $\boldsymbol{L}_{B'}$ is a lower triangular matrix with all nonzero diagonal entries: actually, $\boldsymbol{V}$ always has only

up to two columns, i.e. $\bar{\boldsymbol{L}}_B$ has only the following three possible configurations

(0) $B = B'$	(1) $B = B' \cup \{ h \}$	(2) $B = B' \cup \{ h \} \cup \{ j \}$
$\bar{\boldsymbol{L}}_B = \boldsymbol{L}_{B'}$	$\bar{\boldsymbol{L}}_B = \begin{bmatrix} \boldsymbol{L}_{B'} & \boldsymbol{0} \\ \boldsymbol{l}_h^T & 0 \end{bmatrix}$	$\bar{\boldsymbol{L}}_B = \begin{bmatrix} \boldsymbol{L}_{B'} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{l}_h^T & 0 & 0 \\ \boldsymbol{l}_j^T & 0 & 0 \end{bmatrix}$

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 $\boldsymbol{w}_B$ can be found. The basic relations are summarised in the following propositions, whose proofs are just linear algebra, and can be omitted.

Proposition I.3.2.1: *in case (0), the unique solution of (KT) is*

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

Note that, 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 $\boldsymbol{L}_B \boldsymbol{z}_1 = \boldsymbol{e}$, $\boldsymbol{L}_B \boldsymbol{z}_2 = \alpha_{B'}$), one has

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

i.e. $\bar{\theta}_B$ can be obtained at the cost of three backsolves on the triangular matrix $\boldsymbol{L}_{B'}$ and a few linear operations: actually, by exploiting the base of the previous step, this will be reduced to only one backsolve.

Proposition I.3.2.2: *in case (1), if* $\boldsymbol{l}_h \boldsymbol{z}_1 \neq 1$ *then the unique solution of (KT) is*

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

Proposition I.3.2.3: *in case (1), if* $\boldsymbol{l}_h \boldsymbol{z}_1 = 1$ *then an infinite direction is*

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

Proposition I.3.2.4: *in case (2), if* $\boldsymbol{l}_j \boldsymbol{z}_1 = 1$ *then an infinite direction is*

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

Proposition I.3.2.5: *in case (2), if* $\boldsymbol{l}_j z_1 \neq 1$ *then an infinite direction is*

$$\boldsymbol{w}_B = [\ \nu\ \ -1\ \ \mu\ ] \qquad \textit{where} \quad \mu = \frac{1 - \boldsymbol{l}_h \boldsymbol{z}_1}{1 - \boldsymbol{l}_j \boldsymbol{z}_1} \ \textit{and} \qquad \boldsymbol{\nu} = \boldsymbol{\nu}_1 - \mu \boldsymbol{\nu}_2 = (\boldsymbol{L}_{B'}^T)^{-1}[\ \boldsymbol{l}_h - \mu \boldsymbol{l}_j\ ].$$

Note that in case (2) an infinite direction $\boldsymbol{w}_B$ can always be found: this immediately implies that B will never contain more than two linearly dependent items, since when the second one is inserted in B, a sequence of inner iterations (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 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. Since at the beginning of each main iteration

$$\boldsymbol{Q}_B \bar{\theta}_B = \rho \boldsymbol{e} - \alpha_B \quad \Rightarrow \quad \bar{\theta}_B \boldsymbol{Q}_B \bar{\theta}_B = \rho - \bar{\boldsymbol{x}}_B \alpha_B \ \Rightarrow\ f_B(\ \bar{\theta}_B\ ) = {}^1\!/_2 \bar{\theta}_B \boldsymbol{Q}_B \bar{\theta}_B + \alpha_B \bar{\theta}_B = {}^1\!/_2(\ \rho - \bar{\theta}_B \alpha_B\ )$$

the cost of the $f_B(\ \bar{\theta}_B\ )$ calculation can be reduced from $O(\ b^2\ )$ to $O(\ b\ )$ ($b = |\ B\ |$): this could be done at any inner iteration also, since $f_B()$ is linear when restricted along an infinite $\boldsymbol{w}_B$ (as shown in §I.3.1), and if $\boldsymbol{w}_B = \bar{\theta}_B - \theta_B$

$$f_B(\ \theta_B + \eta \boldsymbol{w}_B\ ) = f_B(\ \theta_B\ ) + \eta w_h[\ \nu + \alpha_h - \boldsymbol{g}_h \boldsymbol{d}\ ] + {}^1\!/_2 \eta^2 [\ f_B(\ \theta_B\ ) - f_B(\ \bar{\theta}_B\ )\ ]$$

but this is not useful (and probably error-prone), and it is avoided in the current implementation.

Once the lower trapezoidal matrix $\bar{\boldsymbol{L}}_B$ and the vectors z_1, z_2 have been found, they can be keet updated at low cost whenever an item is inserted into, or deleted from, B: therefore, the above propositions show that $\bar{\theta}_B$ or $\boldsymbol{w}_B$ can be obtained at each step at the cost of a single backsolve on $\boldsymbol{L}_{B'}$ (at the first iteration $\bar{\boldsymbol{L}}_B = \boldsymbol{L}_{B'} = [\ 1\ /\ \boldsymbol{Q}_{11}\ ]$ and z_1, z_2 can be obtained accordingly). The method is described by the following propositions:

Proposition I.3.2.6: *for* $B = B' \cup \{\ h\ \}$ *and* $\bar{L}_B = L_B$ *nonsingular*

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

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'. Hence, $\bar{\boldsymbol{g}}$ is the projection of $\boldsymbol{g}_h$ onto $\Gamma_{B'}$, so that $\|\ \boldsymbol{l}_h\ \|^2 = \bar{\boldsymbol{g}}^T \boldsymbol{g}_h \le \|\ \boldsymbol{g}_h\ \|^2$ and equality holds $\Leftrightarrow \boldsymbol{g}_h \in \Gamma_{B'}$ i.e. $\boldsymbol{g}_h$ is linearly dependent from the items in B'. ♦

This is exactly one step of the *row-wise* LL^T factorization algorithm applied to $\boldsymbol{Q}_\beta$: 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 I.3.2.7: *if* $\boldsymbol{L}_B \boldsymbol{r}_{B'} = \gamma_{B'}$, *then*

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

In other words, $\boldsymbol{r}_B = \boldsymbol{L}_B^{-1} \gamma_B = [\, \boldsymbol{r}_{B'} \;\; r_h \,]$ where $\boldsymbol{r}_{B'}$ is known and r_h can be calculated in O($b' = \left| B' \right|$): 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. Analogously, 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 also be updated in constant time when the new entries are calculated, saving some O(b) computations.

The above relations also shed some light on Propositions I.3.2.2 - I.3.2.5: for example, the "critical" scalar product $\boldsymbol{l}_h \boldsymbol{z}_1$ of Proposition I.3.2.2 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 $\boldsymbol{l}_h \boldsymbol{z}_1 = 1$ is equivalent to $[\, \boldsymbol{v}_1 \; -1 \,]^T [\, \boldsymbol{e} \;\; 1 \,] = 0$, i.e. to the feasibility of the (infinite) direction $[\, \boldsymbol{v}_1 \; -1 \,]$ w.r.t. the constraint $\boldsymbol{e} \theta_B = 1$; similar interpretations can be derived for all the other cases.

Obviously, a cheap updating procedure is also needed for the case $B = B' / \{ h \}$, where

$$\boldsymbol{L}_{B'} = \begin{bmatrix} \boldsymbol{L}_1 & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{v}^T & \delta & \boldsymbol{0} \\ \boldsymbol{Z} & \boldsymbol{w} & \boldsymbol{L}_2 \end{bmatrix}, \quad \boldsymbol{G}_{B'}^T = \begin{bmatrix} \boldsymbol{G}_1^T \\ \boldsymbol{g}_h^T \\ \boldsymbol{G}_2^T \end{bmatrix}, \quad \boldsymbol{Q}_{B'} = \boldsymbol{G}_{B'}^T \boldsymbol{G}_{B'} = \begin{bmatrix} \boldsymbol{G}_1^T \boldsymbol{G}_1 & \boldsymbol{G}_1^T \boldsymbol{g}_h & \boldsymbol{G}_1^T \boldsymbol{G}_2 \\ \boldsymbol{g}_h^T \boldsymbol{G}_1 & \| \boldsymbol{g}_h \|^2 & \boldsymbol{g}_h^T \boldsymbol{G}_2 \\ \boldsymbol{G}_2^T \boldsymbol{G}_1 & \boldsymbol{G}_2^T \boldsymbol{g}_h & \boldsymbol{G}_2^T \boldsymbol{G}_2 \end{bmatrix}$$

and $\boldsymbol{L}_B = \begin{bmatrix} \boldsymbol{L}_1 & \boldsymbol{0} \\ \boldsymbol{Z} & \boldsymbol{L}_2' \end{bmatrix}, \quad \boldsymbol{G}_B^T = \begin{bmatrix} \boldsymbol{G}_1^T \\ \boldsymbol{G}_2^T \end{bmatrix}, \quad \boldsymbol{Q}_B = \begin{bmatrix} \boldsymbol{G}_1^T \boldsymbol{G}_1 & \boldsymbol{G}_1^T \boldsymbol{G}_2 \\ \boldsymbol{G}_2^T \boldsymbol{G}_1 & \boldsymbol{G}_2^T \boldsymbol{G}_2 \end{bmatrix};$

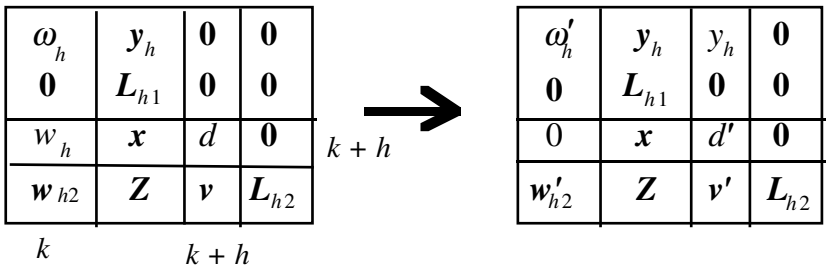
since the first $k - 1$ (where k is the position of $\boldsymbol{g}_h$ in $\boldsymbol{G}_{B'}$) rows and columns of $\boldsymbol{L}_{B'}$ are not altered by the operation, only $\boldsymbol{L}_2$ changes as $\boldsymbol{L}_2' \boldsymbol{L}_2'^T = \boldsymbol{L}_2 \boldsymbol{L}_2^T + \boldsymbol{w} \boldsymbol{w}^T$, so that it can be retrieved by a *rank-one correction* of the *triangular matrix* $\boldsymbol{L}_2'$ performed with a sweep of *Givens Rotations*.

Proposition I.3.2.8: *there exists a* $b' - k$ *orthogonal square matrix* $\boldsymbol{G}_S$ (*i.e.* $\boldsymbol{G}_S^{-1} = \boldsymbol{G}_S^T$) *such that*

$$\begin{bmatrix} \boldsymbol{L}_1 & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{v}^T & \delta & \boldsymbol{0} \\ \boldsymbol{Z} & \boldsymbol{w} & \boldsymbol{L}_2 \end{bmatrix} \begin{bmatrix} \boldsymbol{I}_{k-1} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & & \\ \boldsymbol{0} & & \boldsymbol{G}_S \end{bmatrix} = \begin{bmatrix} \boldsymbol{L}_1 & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{v}^T & \beta & \boldsymbol{y}^T \\ \boldsymbol{Z} & \boldsymbol{0} & \boldsymbol{L}_2' \end{bmatrix}$$

which implies $\boldsymbol{L}_2' \boldsymbol{L}_2'^T = \boldsymbol{L}_2 \boldsymbol{L}_2^T + \boldsymbol{w} \boldsymbol{w}^T$.

Proof: $\boldsymbol{G}_S$ is the product of $b' - k$ Givens matrices: the h-th matrix performs the transformation



so that at the end all the entries of the k-th column of $\boldsymbol{L}_{B'}$ (but $\boldsymbol{L}_{kk}$) are zero, and eliminating the k-th

row and column we obtain $\boldsymbol{L}_B$. ♦

As the h-th step requires $4(\, b - k - h \,)$ multiplications, the overall cost is $O(\, 2(\, b - k \,)^2 \,)$; furthermore, the same method can also be used to update the vectors $\boldsymbol{l}_h$, $\boldsymbol{l}_j$, $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$:

Proposition I.3.2.9: *for $\boldsymbol{G}_S$ as in Proposition I.3.2.8, $\boldsymbol{r}_{B'}$ such that*

$$\boldsymbol{L}_B \boldsymbol{r}_{B'} = \begin{bmatrix} \boldsymbol{L}_1 & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{v}^T & \delta & \boldsymbol{0} \\ \boldsymbol{Z} & \boldsymbol{w} & \boldsymbol{L}_2 \end{bmatrix} \begin{bmatrix} \boldsymbol{r}_1 \\ r_h \\ \boldsymbol{r}_2 \end{bmatrix} = \begin{bmatrix} \gamma_1 \\ \gamma_h \\ \gamma_2 \end{bmatrix} = \gamma_B \quad and \quad \begin{bmatrix} \boldsymbol{r}_1 \\ r'_h \\ \boldsymbol{r}'_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{I}_{k-1} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{G}_S^T \end{bmatrix} \begin{bmatrix} \boldsymbol{r}_1 \\ r_h \\ \boldsymbol{r}_2 \end{bmatrix}$$

the vector $\boldsymbol{r}_B = \begin{bmatrix} \boldsymbol{r}_1 \\ \boldsymbol{r}'_2 \end{bmatrix}$ *solves* $\boldsymbol{L}_B \boldsymbol{r}_B = \begin{bmatrix} \boldsymbol{L}_1 & \boldsymbol{0} \\ \boldsymbol{Z} & \boldsymbol{L}'_2 \end{bmatrix} \begin{bmatrix} \theta_1 \\ \theta'_2 \end{bmatrix} = \begin{bmatrix} \gamma_1 \\ \gamma_2 \end{bmatrix} = \gamma_B$.

In other words, by applying the same Givens rotations to the old $\boldsymbol{r}_{B'}$, the new last $b' - k$ entries of $\boldsymbol{r}_B$ are obtained; it is also easy to show that, for instance,

$$\bar{z}_1 \bar{z}_2 = z_1 z_2 - z'_{1h} z'_{2h}$$

therefore $z_1 z_2$ can be kept updated in $O(\, 1 \,)$ in this case also - analogous results hold for any scalar product of this kind. 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.

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

but this is faced by a simple $l'_h \neq 0$ and the few logical operations required to adjust the indices sets; similarly, in case (2) $\boldsymbol{g}_j$ may become independent, or $\boldsymbol{g}_h$ may be deleted from B.

Reoptimization is natural within the TT algorithm: after that (Δ_β) has been solved, the optimal base B, the optimal solution $[\, \theta_B \, \boldsymbol{0} \,]$, 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. 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 data structures are used "as is" to test the violation of the corresponding dual constraint;
- delete a single item $\boldsymbol{g}_h$ (an outdated subgradient) from β: if $h \notin B$ this has no effect on the problem, since θ_B obviously remains optimal, while if $h \in B$ the procedures described in Propositions I.3.2.8 - I.3.2.9 (actually, the same piece of code) are used to update all the data structures in such a way that the algorithm can be 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. the current point $\bar{\boldsymbol{y}}$): then, the vector $\boldsymbol{z}_2$ can be recalculated with just one backsolve on the linearly independent matrix $\boldsymbol{L}_{B'}$, and again the algorithm can be simply restarted at

the beginning of an inner iteration, since all the other data structures are still coherent.

Any sequence of such operations leaves the data structures in a *consistent state*: 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 $\boldsymbol{d}$, i.e. $\boldsymbol{y} = \bar{\boldsymbol{y}} + \tau\boldsymbol{d}$, then the new vector is obtained according to the formula

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

where $\Delta_o = \varphi(\bar{\boldsymbol{y}} + \tau\boldsymbol{d}) - \varphi(\bar{\boldsymbol{y}})$ is the *obtained increase* in the nondifferentiable function for a *step of* τ along $\boldsymbol{d}$, and

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

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

(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 $\boldsymbol{z}_2$. This ends the description of the "basic" TT algorithm for the unconstrained minimization case: several extensions to more general cases are possible, as the next three paragraphs show.

I.3.3 Handling general dual constraints

As discussed in more details in §I.4.3, Bundle algorithms can be extended to the *constrained* minimisation of $\varphi()$ over a polyhedron $Y = \{ \boldsymbol{y} : \boldsymbol{h} \geq \boldsymbol{y}\boldsymbol{H} \} \subseteq \mathcal{R}^m$, provided that the constraints are inserted into the (Π_β) subproblem: therefore, the (QP) solver must be capable of efficiently deal with general linear constraints on the "dual" space. As for the subgradients, only a subset γ of the constraint set (i.e. a subsystem $\boldsymbol{h}_\gamma \geq \boldsymbol{y}\boldsymbol{H}_\gamma$) may be known at any given point of the algorithm, either because not all the constraints are known a priori, or because some form of constraint generation is used to keep their number small: hence, the problem to be solved is

$$(\Pi_{\beta\gamma}) \qquad \min_{\boldsymbol{d},v} \{ v + 1/2 \| \boldsymbol{d} \|^2 : v\boldsymbol{e} \geq \boldsymbol{d}\boldsymbol{G}_\beta - \alpha_\beta \ , \boldsymbol{h}_\gamma \geq (\bar{\boldsymbol{y}} + \boldsymbol{d})\boldsymbol{H}_\gamma \}$$

or, equivalently, its dual

$$(\Pi_{\beta\gamma}) \qquad \min_\theta \{ 1/2 \| \boldsymbol{G}_\beta\theta + \boldsymbol{H}_\gamma\mu \|^2 + \alpha_\beta\theta + \bar{\boldsymbol{h}}_\gamma\mu : \theta \in \Theta \ , \mu \geq \boldsymbol{0} \}$$

where $\bar{\boldsymbol{h}}_\gamma = \bar{\boldsymbol{y}}\boldsymbol{H}_\gamma - \boldsymbol{h}_\gamma$, i.e. the inequalities are properly "translated" w.r.t. the current point to ensure feasibility of all the points $\bar{\boldsymbol{y}} + \tau\boldsymbol{d}$ with $\tau \leq t$.
Let $\boldsymbol{i}$ be the characteristic vector of the set (of "true" subgradients) β, i.e.

$$i_h = \begin{cases} 1 & \text{if } h \in \beta \\ 0 & \text{if } h \in \gamma \end{cases} :$$

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

$$\{\ [\ \theta\ \ \mu\ ] : \boldsymbol{i}[\ \theta\ \ \mu\ ] = 1\ ,\ [\ \theta\ \ \mu\ ] \geq \mathbf{0}\ \}$$

and *the TT algorithm can still be applied*, provided that *the all-one vector* $\boldsymbol{e}$ *is replaced in every expression with* $\boldsymbol{i}_B$ (where $B \subseteq \beta \cup \gamma$). In fact, $\boldsymbol{Q}_B\bar{\theta}_B + \alpha_B = \rho\boldsymbol{i}_B$ and $\boldsymbol{i}_B\bar{\theta}_B = 1$ are the (KT) conditions for (Δ_B), so that, letting $z_1 = \boldsymbol{L}_{B'}^{-1}\boldsymbol{i}_B$, the optimal solution to (R_B) in case (0) is

$$\bar{\theta}_B = \boldsymbol{Q}_B^{-1}[\ \rho\boldsymbol{i}_B - \alpha_B\ ] \qquad \text{where still} \qquad \rho = \frac{1 + z_1 z_2}{z_1 z_1};$$

Propositions I.3.2.2 - I.3.2.3 hold true if the condition $\boldsymbol{l}_h z_1 \neq 1$ is replaced by $\boldsymbol{l}_h z_1 \neq \boldsymbol{i}_h$ and the infinite direction $[\ \boldsymbol{v}_1\ -1\ ]$ is replaced by $[\ \boldsymbol{v}_1\ -\boldsymbol{i}_h\ ]$, and an analogous treatment can be deserved to Propositions I.3.2.4 - I.3.2.5. Moreover, the factorisation methods described in §I.3.2 are obviously still valid for the constrained case, since z_1 (that solves $\boldsymbol{L}_B z_1 = \boldsymbol{i}_{B'}$) has the form required by Propositions I.3.2.7 and I.3.2.9. Minor modifications to the algorithm are 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 §I.3.2 now have the α updating formula

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

so that z_2' is obtained in the same way from z_1 and 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.

These results can be further generalised by recognising that $(\Delta_{\beta\gamma})$ is just a special case of

$$\min_\theta\ \{\ {}^1\!/_2 \| \boldsymbol{G}\theta \|^2 + \alpha\theta : \boldsymbol{c}\theta = 1\ ,\ \theta \geq \mathbf{0}\ \}$$

where $\boldsymbol{c}$ is any vector of nonnegative coefficients: it is easy to show that all the above theory is valid if $\boldsymbol{e}$ (or $\boldsymbol{i}$) is replaced everywhere with $\boldsymbol{c}$ in the formulas. This is interesting since Least-Square-like problems frequently appear in various fields (see e.g. [FV94] for an application to fractal compression of images), and the TT algorithm can be easily extended to this case: the changes to be made are so few and so localised that this possibility is already provided by our implementation, via a "compile time switch" (see §I.3.6).

I.3.4 Changing the dimension of the dual space

The TT algorithm can work without accessing the vectors $\boldsymbol{g}_i$, since all the computations can be (efficiently) performed by means of the scalar products $\boldsymbol{q}_{ij} = \boldsymbol{g}_i^T\boldsymbol{g}_j$: by $\boldsymbol{d} = -\boldsymbol{G}_B\theta_B$, one obtains

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

that can be used for checking the dual constraints $\boldsymbol{d}\boldsymbol{g}_h - v \leq \alpha_h$ without explicitly forming $\boldsymbol{d}$ (if $b < m$ this is also computationally convenient, since it reduces the complexity of the feasibility check from $O(\ bm\ )$ to $O(\ b^2\ )$). Yet, when the algorithm terminates the scalar product $\boldsymbol{d}\boldsymbol{g}_h$ is known for each $h \in \beta$ (it has been computed during the feasibility check if $h \notin B$, and $\boldsymbol{d}\boldsymbol{g}_h = \rho\boldsymbol{i}_h - \alpha_h$ if $h \in B$), so that it can be used, for instance, for changing α_h "along" $\boldsymbol{d}$; furthermore, the NDO algorithm should have granted access to it, since $\varphi_{\beta,\boldsymbol{d}}(\ \tau\ ) = \min_{h \in \beta}\ \{\ \tau(\ \boldsymbol{d}\boldsymbol{g}_h\ ) + \alpha_h\ \}$ is the restriction of $\varphi_\beta()$ along the

(tentative descent) direction $\boldsymbol{d}$.

Since the TT algorithm has in fact no information on the "real" vectors $\boldsymbol{g}_h$, it is possible to design its implementation (see §I.3.6) to be "orthogonal" to thire structure by simply demanding the task to a user-provided subroutine that computes $\boldsymbol{q}_{ij}$: furthermore, it is also possible to change their length between two subsequent calls to the TT-solver, i.e. to add or remove entries from all the $\boldsymbol{g}_h$ (and $\boldsymbol{h}_h$) at the same time. This is really useful, both for the extensions that will be discussed in §I.3.5 and for the fact that NDO solvers may benefit from *active set* strategies, where only an (hopefully small) subset of the variables $\boldsymbol{y}$ - that is dynamically revised - is considered at any given step: as shown in §I.3.7, this approach can be crucial for the effectiveness of a NDO code.

Insertions and deletions of entries in (all) the subgradients $\boldsymbol{g}_h$ (treatment of constraints $\boldsymbol{h}_h$ is identical) can be "naturally" performed with cheap data structures updates, analogous to the ones described §I.3.2, provided that a - not necessarily optimal, if other changes have already occurred - base B and the related $[\ \theta_B\ \mathbf{0}\ ]$, $\boldsymbol{Q}_\beta$, $\bar{\boldsymbol{L}}_B$, z_1, z_2 ... are kept in a consistent state. When adding a new entry, the new Hessian matrix $\tilde{\boldsymbol{Q}}_\beta = \boldsymbol{Q}_\beta + \tilde{\boldsymbol{g}}\ \tilde{\boldsymbol{g}}^T$ (where $\tilde{g}_h = \langle$new entry of $\boldsymbol{g}_h\rangle$) can be obtained in O(p^2), and the update of $\bar{\boldsymbol{L}}_B$, z_1, z_2 can be performed with a sweep of p Givens rotations ($\boldsymbol{G}_S$), as

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

so that the *rank-one correction* of the lower trapezoidal matrix $\bar{\boldsymbol{L}}_B$

$$\begin{bmatrix} \tilde{\boldsymbol{g}}_{B'}\ \boldsymbol{L}_{B'} \\ \tilde{g}_h\ \ \boldsymbol{l}_h^T \\ \tilde{g}_j\ \ \boldsymbol{l}_j^T \\ 0\ \ z_1 \\ 0\ \ z_2 \end{bmatrix} \boldsymbol{G}_S \qquad \Longrightarrow \qquad \begin{bmatrix} \mathbf{0}\ \ \tilde{\boldsymbol{L}}_{B'} \\ \delta_h\ \tilde{\boldsymbol{l}}_h^T \\ \delta_j\ \tilde{\boldsymbol{l}}_j^T \\ \sigma_1\ \tilde{z}_1 \\ \sigma_2\ \tilde{z}_2 \end{bmatrix}$$

suffices; note that the *position* of the *new entry* plays absolutely *no role* in the formulas, making it possible to insert (and delete) entries in any order whatsoever. Once $\tilde{\boldsymbol{g}}_{B'}$ has been nullified, again an O(p^2) task, the new $\tilde{\boldsymbol{L}}_{B'}$ and the related $\tilde{\boldsymbol{l}}_h$, $\tilde{\boldsymbol{l}}_j$, $\tilde{z}_1$, $\tilde{z}_2$ can be made "consistent" - if they are not already - with the data at hand: if $\delta_h > 0$, then the corresponding item must be inserted into the "linearly independent part" of B ($\tilde{B}' = B' \cup \{\ h\ \}$), i.e.

$$\boldsymbol{L}_{\tilde{B}'} = \begin{bmatrix} \tilde{\boldsymbol{L}}_{B'}\ \ \mathbf{0} \\ \tilde{\boldsymbol{l}}_h^T\ \delta_h \end{bmatrix}$$

where σ_1, σ_2 are the new entries of $\tilde{z}_1$, $\tilde{z}_2$ (so that also $\tilde{z}_1\tilde{z}_2 = z_1z_2 + \sigma_1\sigma_2$); the case $\delta_j > 0$ is analogous.

When an entry (contained in $\tilde{\boldsymbol{g}}$) has to be deleted, one has $\tilde{\boldsymbol{Q}}_\beta = \boldsymbol{Q}_\beta - \tilde{\boldsymbol{g}}\ \tilde{\boldsymbol{g}}^T$, i.e.

$$\tilde{\boldsymbol{Q}}_B = [\ i\tilde{\boldsymbol{g}}_B\ \bar{\boldsymbol{L}}_B\ ] \begin{bmatrix} i\tilde{\boldsymbol{g}}_B^T \\ \bar{\boldsymbol{L}}_B^T \end{bmatrix} \qquad \text{where } i^2 = -1;$$

then, almost the same arguments as before hold, i.e. by a sweep of *complex* Givens matrices

$$\begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & c & \cdots & -is \\ & & \vdots & & \vdots \\ & & is & \cdots & c \\ & & & & & \ddots \\ & & & & & & 1 \end{bmatrix}$$

where $c^2 - s^2 = 1$, it is possible to obtain

$$\begin{bmatrix} \mathbf{0} & \tilde{\boldsymbol{L}}_{B'} \\ 0 & \tilde{\boldsymbol{l}}_h^T \\ 0 & \tilde{\boldsymbol{l}}_j^T \\ i\sigma_1 & \tilde{\boldsymbol{z}}_1 \\ i\sigma_2 & \tilde{\boldsymbol{z}}_2 \end{bmatrix}.$$

Note that, while $\tilde{\boldsymbol{g}}_B$ is nullified, all the "imaginary parts" are kept on the leftmost column alone, so that the resulting $\bar{\boldsymbol{L}}_B$ and $\tilde{\boldsymbol{l}}_h$, $\tilde{\boldsymbol{l}}_j$, $\tilde{\boldsymbol{z}}_1$, $\tilde{\boldsymbol{z}}_2$ are real matrix/vectors which bear many of the expected properties, such as $\tilde{\boldsymbol{z}}_1\tilde{\boldsymbol{z}}_2 = \boldsymbol{z}_1\boldsymbol{z}_2 - \sigma_1\sigma_2$ (and the same for the other scalar products): making the data structures "consistent", however, is somewhat more complicated that in the previous case, since zeroes along the diagonal of $\tilde{\boldsymbol{L}}_{B'}$ may be generated (but $\delta_h = \delta_j = 0$, i.e. no "new linear independence" is created). Dealing with such a situation is easy if it involves the last row of $\bar{\boldsymbol{L}}_B$, i.e. if after applying $b - 1$ (complex) Givens rotations the result is

$$\tilde{\boldsymbol{L}}_B = \begin{bmatrix} \mathbf{0} & \tilde{\boldsymbol{L}}_{B'} & \mathbf{0} \\ i\delta_h & \tilde{\boldsymbol{l}}_h^T & \tilde{l}_{hh} \end{bmatrix} \qquad \text{where} \qquad \tilde{l}_{hh}^{\,2} = \delta_h^2$$

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

$$\tilde{\boldsymbol{L}}_B = \begin{bmatrix} \mathbf{0} & \tilde{\boldsymbol{L}}_{B'} & \mathbf{0} \\ 0 & \tilde{\boldsymbol{l}}_h^T & 0 \end{bmatrix}$$

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, regardless of h being or not one of them, even though if the third dependent item is created then one of them must be removed from B (an O(1) task): if $\tilde{l}_{hh}^{\,2} = \delta_h^2$ for some h in position $k < b$, then with a sweep of (at most) $b - k - 1$ Givens rotations - those of Proposition I.3.2.8 - and a permutation of B the h-th item can be put in the last position, and the method applied.

As in §I.32, a set of cheap operations are available to modify the data of the problem, while keeping the data structures consistent and ready for a "warm" restart: this will be used in the next paragraph to efficiently cope with box constraints on the dual space.

I.3.5 Handling box constraints on the dual space

In some applications, (most of) the constraints $\boldsymbol{h} \geq \boldsymbol{yH}$ are just "boxes" $l_i \leq d_i \leq u_i$: in this paragraph, a version of TT that efficiently handles lower bounds $l_i \leq d_i$ on a subset γ of the variables is presented, and the extension to the more general class is straightforward. The BTT method still uses an *active set* strategy, in which at each step only a subset $\chi \subseteq \gamma$ of the primal multipliers associated to the box constraints are allowed to be nonzero: since the part of the base concerning subgradients (or general

linear constraints) will be "implicitly" taken into account, to simplify the notation the subscript β for the columns of $\boldsymbol{G}$ will be dropped, and it will be considered partitioned as

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

where $\eta = \{ 1 .. n \} / \chi$ - that is, subscripts will hereafter indicate rows instead.
Given a "partial" base χ, the *restricted* problems to be solved at each step are

$$(\Pi_\chi) \qquad \min_{d,v} \left\{ v + 1/2 \left\| \boldsymbol{d} \right\|^2 : v\boldsymbol{e} \geq \boldsymbol{d}\boldsymbol{G} - \alpha \, , \, \boldsymbol{l}_\chi \leq \boldsymbol{d}_\chi \right\}$$

$$(\Delta_\chi) \qquad \min_{\theta,\mu} \left\{ \frac{1}{2} \left\| \begin{bmatrix} \boldsymbol{G}_\chi & -\boldsymbol{I} \\ \boldsymbol{G}_\eta & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \theta \\ \mu_\chi \end{bmatrix} \right\|^2 + \begin{bmatrix} \alpha & -\boldsymbol{l}_\chi \end{bmatrix} \begin{bmatrix} \theta \\ \mu_\chi \end{bmatrix} : \theta \in \Theta \, , \, \mu_\chi \geq \boldsymbol{0} \right\}$$

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

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

a feasible solution of (Δ_χ) is optimal if and only if it solves the (KT) system

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

so that $\mu_\chi = \boldsymbol{G}_\chi\theta + \boldsymbol{l}_\chi$, and therefore $[\boldsymbol{G}^T\boldsymbol{G} - \boldsymbol{G}_\chi^T\boldsymbol{G}_\chi]\theta = \rho\boldsymbol{e} - [\alpha - \boldsymbol{G}_\chi^T\boldsymbol{l}_\chi]$. 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

$$(R_\chi) \qquad \min_{\theta,\mu} \left\{ \frac{1}{2} \left\| \begin{bmatrix} \boldsymbol{G}_\chi & -\boldsymbol{I} \\ \boldsymbol{G}_\eta & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \theta \\ \mu_\chi \end{bmatrix} \right\|^2 + \begin{bmatrix} \alpha & -\boldsymbol{l}_\chi \end{bmatrix} \begin{bmatrix} \theta \\ \mu_\chi \end{bmatrix} : \theta \in \Theta \right\}$$

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

$$(\Delta^\eta) \qquad \min_\theta \left\{ 1/2\theta^T\boldsymbol{Q}_\eta\theta + \bar{\alpha}\theta : \theta \in \Theta \right\}$$

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, the box-constrained problem (Π_γ) could be solved by using *any* algorithm for the unconstrained case as a "black box" to find a base B and a solution $\theta = [\theta_B \; \boldsymbol{0}]$ such that $[\theta \; \mu_\chi]$ solves the (KT) system for (Δ_χ): clearly, the efficient TT algorithm will be used as the subroutine TT(χ) that reports an optimal solution θ of (Δ_χ). Building on this, a "two-level" solver for the box constrained case can be constructed, where the lower level handles the θ variables while the upper level handles the μ variables: it should be clear, by just looking at the pseudo-code of Figure 15, 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

$$[\ \theta^T\ \ \mu_\chi^T\ ]\begin{bmatrix} \boldsymbol{Q} & -\boldsymbol{G}_\chi^T \\ -\boldsymbol{G}_\chi & \boldsymbol{I} \end{bmatrix}\begin{bmatrix} \theta \\ \mu_\chi \end{bmatrix} = \theta^T \boldsymbol{Q}_\eta \theta + \| \boldsymbol{l} \|^2 \quad \text{and} \quad [\ \alpha\ -\boldsymbol{l}_\chi\ ]\begin{bmatrix} \theta \\ \mu_\chi \end{bmatrix} = \tilde{\alpha}\theta - \| \boldsymbol{l} \|^2$$

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

$$f_B(\ [\ \theta_B\ \ \mu_\chi\ ]\ ) = f'_B(\ \theta_B\ ) \qquad v = v' \qquad \boldsymbol{d} = [\ \boldsymbol{l}_\chi\ \boldsymbol{d}'\ ] \qquad \boldsymbol{G}^T\boldsymbol{d} = \boldsymbol{G}_\eta^T\boldsymbol{d}' - \boldsymbol{G}_\chi^T\boldsymbol{l}_\chi.$$

This "hierarchic" view of the method greatly simplifies the design and the implementation of the algorithm but, due to the sophisticated reoptimization strategies discussed in §I.3.2 and §I.3.4, performances can be expected to be even better than those of the "monolithic" approaches known in the literature: obviously, the method immediately extends to $u_i \geq d_i$ constraints (where $\mu_i = -u_i - \boldsymbol{G}_i\theta$) 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.

BTT Algorithm

```
χ = ∅; θ = TT( χ ); μ_χ = G_χ θ + l_χ;                 /* initialization */
while( ∃ i ∈ ( γ / χ ) such that l_i > − G_i θ )        /* main iteration */
    χ = χ ∪ { i };                                     /* item insertion − μ_i = G_i θ + l_i > 0 */
    do                                                 /* inner iteration */
        θ = TT( χ ); μ̄_χ = G_χ θ + l_χ;                /* KT solution - use TT as subroutine */
        if( ∃ i ∈ χ such that l_i < − G_i θ )          /* μ̄_i = G_i θ + l_i < 0 */
            then  w_χ = μ̄_χ − μ_χ;                     /* w_χ is of descent in the μ_χ space */
                  η = min_{i∈χ} { −w_i / μ_i : w_i < 0 }; /* maximum feasible step η > 0 */
                  μ_χ = μ_χ + η w_χ;                   /* take step η along w_χ */
            else  μ_χ = μ̄_χ; w_χ = 0;                  /* μ̄_χ is optimal for (Δ_χ) */
        forall( i such that μ_i = 0 )
            χ = χ / { i };                             /* item(s) deletion */
    while( w_χ ≠ 0 );                                  /* end inner iteration */
endwhile                                               /* end main iteration */
```

Figure 15: the BTT algorithm

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

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

where $\tilde{\boldsymbol{g}} = \boldsymbol{G}_i$ is the i-th row of $\boldsymbol{G}$; furthermore, the vector $z_2 = \boldsymbol{L}_{B'}^{-1}\bar{\alpha}$ can be updated without resorting to an expensive backsolve on $\boldsymbol{L}_{B'}$, as

$$\tilde{z_2} = \tilde{\boldsymbol{L}}_{B'}^{-1}\bar{\alpha}' = \tilde{\boldsymbol{L}}_{B'}^{-1}\bar{\alpha} - l_i[\ \tilde{\boldsymbol{L}}_{B'}^{-1}\tilde{\boldsymbol{g}}\ ].$$

Actually, the method discussed in §I.3.4 finds $\tilde{\boldsymbol{L}}_{B'}^{-1}\bar{\alpha}$ instead, but it is easy to recover the "perturbation" vector $\tilde{\boldsymbol{L}}_{B'}^{-1}\tilde{\boldsymbol{g}}$, since the orthogonal $(b+1)\times(b+1)$ matrix $\boldsymbol{G}_S$ such that

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

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

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

i.e. $\upsilon = \tilde{\boldsymbol{L}}_{B'}^{-1}\tilde{\boldsymbol{g}}$ and $\tilde{z}_2$ can be cheapily obtained from $\tilde{\boldsymbol{L}}_{B'}^{-1}\bar{\alpha}$. The case $\chi' = \chi / \{ i \}$ is analogous, with $\bar{\alpha}' = \alpha + l_i\tilde{\boldsymbol{g}}$ and $[\ \upsilon_0\ \upsilon^T\ ] = [\ -i\ \mathbf{0}^T\ ]\boldsymbol{G}_S$, and all the elements that are necessary to use TT as a subroutine for BTT have been introduced: a number of further details can be easily derived from the above theory, such as how to keep the scalar product updated ($z_1\tilde{z}_2 = z_1 z_2 \pm \upsilon_0\sigma_1 l_i$ for σ_1 defined in §I.3.4), how to update $\boldsymbol{l}_\chi$ when moving "along" $\boldsymbol{d}$ of a step τ ($\boldsymbol{l}_\chi := (1 - \tau / t)\boldsymbol{l}_\chi$) or that the above results hold true for the generalised problem of §I.3.3 with constraint $\boldsymbol{c}\theta = 1$. The two-level structure is also important in the design of an efficient implementation of the BTT algorithm, that will be discussed in the next paragraph.

I.3.6 Implementation details

The implementation of the TT algorithm was the initial step of the larger project aimed at developing efficient Bundle codes for solving Lagrangean Duals of large-scale (Integer) Linear Problems: hence, an effort was made from the beginning to obtain a code that was not only efficient and easily usable, but also customizable to applications that might not have been foreseen at the developing time. This approach was very useful for the development of the different Bundle codes that have been produced so far, but it was also the basis for the ideation of the BTT algorithm - that was later imposed by the need of solving constrained NDO problems - and hence played a considerable role in all the research. Like all the other codes, the code is written in C++ and makes extensive use of its *encapsulation* 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. Moreover, the *inheritance* and *polymorphism* capabilities of the language are exploited since the BTT algorithm is contained in the *derived* `class BMinQuad`, inheriting from the *base class* most of its data and code, and only redefining the necessary methods through the mechanism of *virtual functions*.

Although a full description of the classes is not possible here, a brief review of the main methods of their interface may help a better understanding of the structure: the methods can be divided into

– a set of *public* methods for handling the data of (Δ_β), such as `AddSubGradient(`i, α_i`)` to add ($\boldsymbol{g}_i$, α_i) to β (and an analogous `AddConstraint()`), `ResetBundle(`i`)` to remove it, `ChangeAlfa(`α`)` or `MoveAlongD(`Δ_o, τ`)` for performing "movements" in the dual space and `Add[Remove]Variable(`$\tilde{\boldsymbol{g}}$`)` to control the variables of (Π_β); only a few of them need to be redefined by `BMinQuad`, that adds methods such as `Add[/Remove]ConstrVar(`i, l_i`)` or `MakeVar[Un]Constr(`i`)` to control the set of constrained variables. Although apparently simple, these methods implement the mechanisms discussed in §I.3.2 and §I.3.4 for allowing a fast reoptimization after a data change: furthermore, they isolate the NDO code from the details of the data structure implementation in the `MinQuad` and `BMinQuad` classes, that can therefore change (and they did) without requiring any modification in the NDO code. As an example of the usefulness of such an approach, the computational comparisons reported in the next paragraph have been obtained by simply "disguising" the two other (completely different) (QP) codes under

the same interface, and keeping the two different NDO solvers untouched;

– the "main" method `MQFrontEnd(t)` for solving the current instance of (Δ_β) (with the given t): far from being only a call to the actual core solver code (the *protected* `CalculateOptDir()`), this method implements a "dynamic" relative tolerances adjustment method that greatly improves resistance of the algorithm to ill-conditioned instances and to cumulation of rounding errors. In the TT algorithm, some "$y = 0$" tests that are crucial to determine the behaviour of the algorithm - such as the choice between cases (0), (1) and (2) and between "normal" and "infinite" steps - are implemented as $| y | < m\sigma\mu(\boldsymbol{L}_{B'})\varepsilon_R$, where $\mu(\cdot)$ is the condition number, σ is related to the "size" of the numbers used in the calculation and ε_R is the "dynamic" relative precision required: erroneous conditions (such as a $\delta << 0$ in I.3.2.6) forcing `CalculateOptDir()` to quit are interpreted as "increase ε_R" or "decrease ε_R" requests, so that ε_R is dynamically adjusted in order to face "bad" instances without a need for hand-tuning of the parameters. From-scratch recalculation of the main data structures ($\boldsymbol{L}_{B'}$, z_1 ...), that must always be implemented in any numerical code, is performed by the *protected* method `RebuildBase()` within the above ε_R tuning mechanism;

– methods for retrieving the solution informations like (B, θ_B) = `ReadMult()`, $\boldsymbol{dG}$ = `ReadGTd()` and - since the handling of the α vector can be entirely demanded to the class - α = `ReadAlfa()`; the `BMinQuad` class, where $\boldsymbol{d}_\chi$ has to be computed at each step, adds `ReadD()`, `ReadZ()` and related methods. As shown in Chapter I.2, sensitivity analysis information may also be useful to NDO solvers, and the method (l_1, l_2, l_3) = `SensitAnals1()` provides a full description of the optimal solutions values of (Δ_β) and (Π_β) as a function of t

$$\left\| \boldsymbol{G}\theta(t) \right\|^2 = \left\| \boldsymbol{d}(t) \right\|^2 = l_1 + (1/t)^2 l_3$$

$$\sigma(t) = \alpha\theta(t) = l_2 - (1/t) l_3$$

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

at essentially no cost, since it is easy to verify that

	Case (0)	Case (1)
l_1 (≥ 0)	$1 / \left\| z_1 \right\|^2$	0
l_2	$z_1 z_2 / \left\| z_1 \right\|^2$	$\rho = (\alpha_h - \boldsymbol{l}_h z_2) / (1 - \boldsymbol{l}_h z_1)$
l_3 (≥ 0)	$[\left\| z_1 \right\|^2 \left\| z_2 \right\|^2 - (z_1 z_2)^2] / \left\| z_1 \right\|^2$	$\left\| \rho z_1 - z_2 \right\|^2 + \left\| \boldsymbol{l}_\chi \right\|^2$

and all the necessary data is either already calculated during the algorithm, or can be kept updated in O(1) per iteration. Other more costly information, such as the "stability interval" $t_m \leq t \leq t_M$ in which the current base B remains optimal (hence the above estimates on $\left\| \boldsymbol{d}(t) \right\|^2$, $\sigma(t)$ and $v(t)$ are "exact") and the two vectors θ_1, θ_2 such that $\theta(t) = \theta_1 + (1/t)\theta_2$ can be queried with (t_m, t_M) = `SensitAnals2()` and (θ_1, θ_2) = `SensitAnals3()`;

– the *protected* interface with the NDO solver, that needs to be a derived class: in fact, the *pure virtual* method q_{ij} = `GiTGj(i, j)` is the only way in which `MinQuad` obtains information about the Hessian $\boldsymbol{Q}$, and it has to be implemented by the derived class. This makes `MinQuad` independent from how subgradients are stored in the NDO solver, allowing easy exploitation of

sparsity or known structural properties of the vectors $\boldsymbol{g}_h$ (and $\boldsymbol{h}_h$) and a great degree of freedom in the design of Bundle codes: since subgradients (and constraints) are only known to `MinQuad` by means of their "symbolic names" (indices), the NDO solver can maintain the matrix $\boldsymbol{G}$ in a row-wise or column-wise fashion (or both), with a sparse or dense (or mixed) implementation and also have a complete control on details such as which entry in the vector represents which variable, with the possibility of even "restructuring" the data on line. The `BMinQuad` class requires more information, hence adds to its protected interface other pure virtual methods such as $\boldsymbol{G}_h$ = `GiTilde(`h`)` or z_h = `CalculateZh(`h`)`: these methods also must be implemented by the derived class, and since variables are yet referenced through "symbolic names" - that need not have any connection with the actual data layout - the NDO code have a complete freedom in the choice of the data structures implementation.

A graphical resume of the structure of the `MinQuad` and `BMinQuad` classes is presented in Figure 16, where arrows represent functional interactions and/or data exchange with the "outside world", thick lines inheritance relationships and dashed lines the use of the *virtual* mechanism to redefine a method in a drived class.

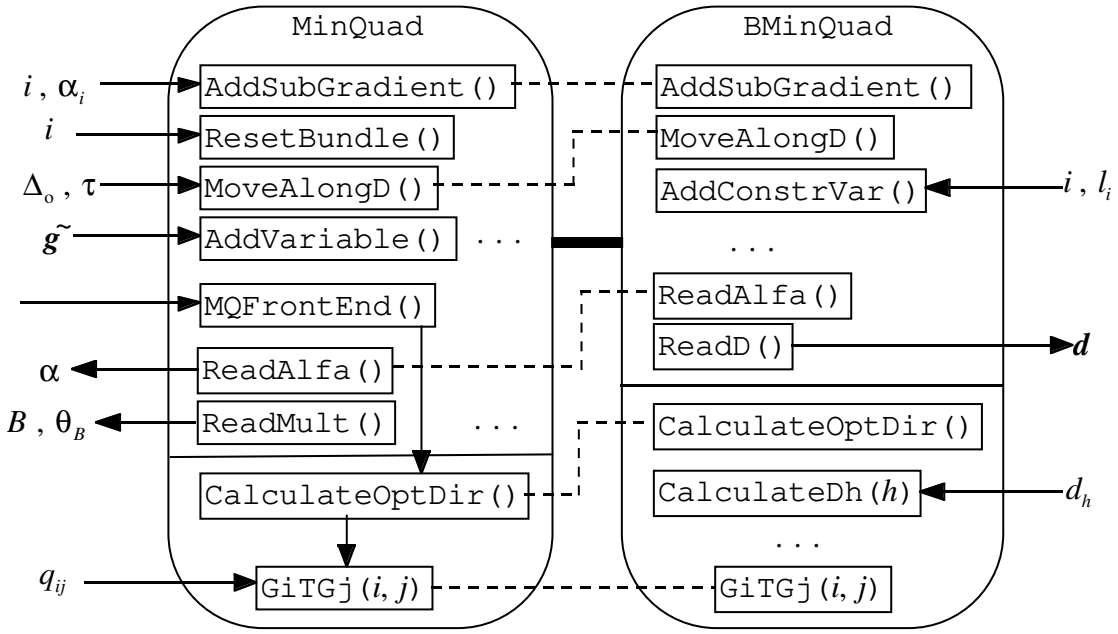


Figure 16: basic structure of the `MinQuad` and `BMinQuad` classes

A number of important advantages have been obtained with this structure: for instance, the `MinQuad` class alone can be used when no box constraints are required, avoiding any extra overhead, while only minor modifications in the NDO solver are required in order to use the more general `BMinQuad` instead; furthermore, code maintenance is much easier than if two separate codes had been written, and improvements in the TT code are automatically exploited by BTT - actually, as long as the interface remains the same TT, might also be replaced by an entirely different algorithm, and this possibility could be exploited in the future.

In the implementation of both classes, many different algorithmic options were available, among which the choice should have been dictated by the - unknown at developing time - characteristics of

the "final" NDO code and, worse, of the particular set of instances: some of them are

- whether calculation of the new row $\boldsymbol{Q}_h$ have to be performed immediately after $\boldsymbol{g}_h$ is inserted, or each scalar product $\boldsymbol{g}_i^T\boldsymbol{g}_j$ have to be performed "on demand" the first time that the corresponding element $\boldsymbol{Q}_{ij}$ is required;
- whether calculation of the whole direction $\boldsymbol{d}$ have to be entirely performed at the beginning of any inner iteration of the BTT algorithm, or each entry d_h have to be separately calculated "on demand";
- whether general box constraints $\boldsymbol{l} \le \boldsymbol{d} \le \boldsymbol{u}$ are required, or simple lower bounds $\boldsymbol{l} \le \boldsymbol{d}$ suffice;
- whether or not general linear constraints are required;
- whether the box bounds ($\boldsymbol{l}$, $\boldsymbol{u}$) will ever change during the lifetime of a `BMinQuad` object (but for the use of `MoveAlongD()`);
- which implementation of the $\boldsymbol{Q}$ matrix have to be chosen;
- whether the scalar products like $z_1 z_2$ have to be kept updated in O(1), or recalculated from scratch at every iteration to enhance numerical stability.

Since it was not practical (if ever possible) to accommodate all these different alternatives by means of the standard inheritance and polymorphism concepts, but a reasonable choice could not be done a priori, we resorted to the - much more C-ish - mechanism of "compile time switches": for each of these alternatives, a preprocessor variable is `#define`(d) in the include file of the class and appropriate pieces of code are compiled or not according to the value of the switch. Although being less elegant (once an alternative is chosen, it is fixed for all the instances of the class in that object code), this method have shown to be somewhat complementary to the use of different classes: in several cases, the correct choice of some of these implementative details was of great importance in obtaining good computational results.

The final noticeable detail of the [B]TT implementation is the use of a set of "abstract" basic data types that makes it "parametric" about the precision required: in particular, for all the different kinds of basic data algorithm (indices, entries of $\boldsymbol{L}$, entries of $\boldsymbol{g}_h$, entries of $\boldsymbol{Q}$, entries of $\boldsymbol{d}$ ...) a data type is defined (`Index`, `HpNum`, `SgNum`, `QuNum`, `LMNum` ...) and all the variables pertaining to that type are declared of this "abstract" data type rather than, say, `int` or `double`. This can be useful in several ways: for instance, the subgradients have often only small integer entries, so that their (long) scalar products may be computed with faster integer operations; furthermore, especially with small problems, it is possible to use single precision floating point numbers (confirming the robustness of our approach) rather than the usual double precision ones, and the integers corresponding to indices can be kept as small as the expected maximum size of the problems require. In a set of preliminary experiences, the correct choice of the right data types improved performances of 30% in time and of almost 50% in memory: some care is required, however, since the conversion between different formats is not costless, and the relative benefits of different choices can be very different on different target architectures. Furthermore, this mechanism made the porting of the codes easier, since the implementation of the basic C types can be very different on different machines: although portability is rarely a concern when implementing experimental codes, a little care in the design phase can greatly help to make the code widely usable on different situations and architectures.

I.3.7 Computational results and comparisons

Barring the "classical" (QP) algorithm in [Mi77], that solves the related but different (Δ_β^ε) problem, only two specialized algorithms for the unconstrained version of (Δ_β) [Ki86] [Mo87] have been proposed in the literature: the TT algorithm bears resemblance to both, but greatly enhances upon them in terms of cost per iteration. An algorithm for the more general (Δ_β) problem of §I.4.3 - for "separable" functions - is proposed in [Ki89]: using $m \times m$ matrices, however, it is admittedly not well-suited for the class of instances of interest here. Independently from our development, the recent [Ki94] proposes an algorithm that solves the same problem as [Ki89] avoiding the use of $m \times m$ matrices, so that its basic operations (inserting/deleting items in base, finding the descent direction ...) have almost the same cost than in BTT: however, the latter improves on it on a number of important details, such as calculation of $f(\theta)$ in linear rather than quadratic time, lighter operations corresponding to the "downdate" the triangular factor after a constraint insertion (moving a newly created linear dependent item in last position, as described at the end of §I.3.4) and no row additions and subtractions needed at "basic variable changes" (since the constraint $\boldsymbol{e}\theta = 1$ is explicitly handled rather than "projected away" by replacing $\boldsymbol{Q}$ with $\hat{\boldsymbol{Q}} = \hat{\boldsymbol{G}}^T\hat{\boldsymbol{G}}$, where $\hat{\boldsymbol{G}} = [\, \boldsymbol{g}_1 - \boldsymbol{g}_h \ \ldots \ \boldsymbol{g}_m - \boldsymbol{g}_h \,]$ for some h). Furthermore, the two-level nature of BTT also helps, since whenever more than one step of the TT algorithm is performed at each inner iteration of BTT, recalculation of $\boldsymbol{d}$ (a "heavy" task) can be avoided; in general, performances can be improved by using different strategies at the two levels, since the same operation may have a different impact in the two cases: for instance, in TT the base must always be kept as small as possible, while this is not necessary in BTT. However, no implementation of the algorithm of [Ki94] was made available to us, and no computational experiences are reported (but it is considered efficient among the experts), so that no measure of relative efficiency could be obtained: therefore, [B]TT have been compared with two non specialized codes, to demonstrate at least the necessity of an ad-hoc approach. Several Bundle codes based on (early) implementations of the TT and the BTT algorithms have been developed in different contexts, such as Crew Scheduling problems [CGN95], Forestry problems [EGM95], Quadratic Semi-Assignment problems [MP95] and Multicommodity Flow problems [FG96]: hence, it was possible to compare the (QP) codes in a very "realistic" environment, by using them as (Δ_β)-solvers within some the above Bundle codes on "real" instances.

The first benchmark code is a "specialized" version of the QL code [Po83], that was modified to be used as (Δ_β)-solver within the Bundle code of [OSZ91]: the Fortran code was translated to C with the `f2c` converter by R. Freling and further adapted to C++ for the current experiments. Despite its age, it is well-known to be an efficient public domain (QP) solver: however, working on the "dual" (Π_β), and not being able to reoptimize from previous calls, it was expected to be slower than BTT. The second benchmark code is a beta release of the Interior Point (QP) solver of Cplex‡ 4.0, hereafter called CQP: it is based on the "modern" IP technology [CLM93], as opposed to the "old" active-set method of BTT, and exploits the Interior Point routines of one of the best commercial (LP) solvers on the market, using sophisticated sparse linear algebra routines and state-of-the-art coding. Hence, in this case it was not clear whether or not BTT would have really had a decisive advantage on it:

‡ Cplex is a registered trademark of Cplex Inc.

furthermore, since IP methods behave better on large problems with sparse Hessians such as (Π_β), that has a separable objective function, both the "primal" and "dual" formulation were solved.

Anticipating a bit the conclusions, [B]TT has shown to be blatantly faster than the other (QP) solvers: such a result is however limited to the specific application, and in no way has to be intended as a claim of superior efficiency outside this framework. Yet, a considerable care has been taken in order to make the tests fair: for instance, the cost of data structures initialisations and problem scaling was not counted, so that no advantage is given to BTT for its the problem-specific data structures; also, double precision arithmetic was used and the "compile time" options were kept fixed for all instances, even though savings could be obtained by exploiting its superior flexibility. Furthermore, the other codes were provided with all the "external" information that they could use: for instance, CQP was not asked to solve the "full" primal formulation (Δ_γ), but rather an easier restricted problem similar to (R_χ) w.r.t. the optimal set χ that had been previously determined by BTT.

The first set of experiences was performed with the Bundle code of [MP95] for finding Lagrangean-based lower bounds for the Quadratic Semi-Assignment Problem: the number of (unconstrained) Lagrangean multipliers varied, depending on the instance, from 10 up to 1600, and the maximal Bundle dimension reached 250 subgradients in the larger instances. For technical reasons, CQP and QL were tested against TT on two different (but analogous) sets of 50 and 130 instances respectively: the results are reported in Table 1 and Table 2, where *m* indicates the number of Lagrangean multipliers, *c* the number of calls to the (QP) solver and *b* the average Bundle dimension troughout the calls. Problems with similar "size" *mbc* are aggregated in groups (*n* indicates the number of instances), and the figures are averaged within the group: the chosen measure of "size" of an instance, actually corresponding to a large number of (Δ_β) instances, has shown to have the largest correlation with the running times. The performances of the (Δ_β)-solvers are measured in terms of the "total running time", i.e. the sum of the running times of all the (Δ_β)s - up to 1800 - needed to solve a single instance: 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 only available as executable code.

n	*m*	*b*	*c*	*TT*	*QP*
6	12	15	128	0.02	0.16
12	15	17	175	0.03	0.36
17	70	30	289	0.16	2.58
25	75	43	502	0.62	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	*m*	*b*	*c*	*TT*	*QPP*	*QPD*
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 both the Primal and Dual formulation)

The results are better visualised in Chart 1 - Chart 4, and clearly show that to TT code is by far superior to both QL and CQP, especially when dealing with large problems (note that the scale in Chart 1 and Chart 3 is logarithmic): again, we remark that this is true only for this particular application, and with respect to the total running time over a whole execution of the NDO solver.

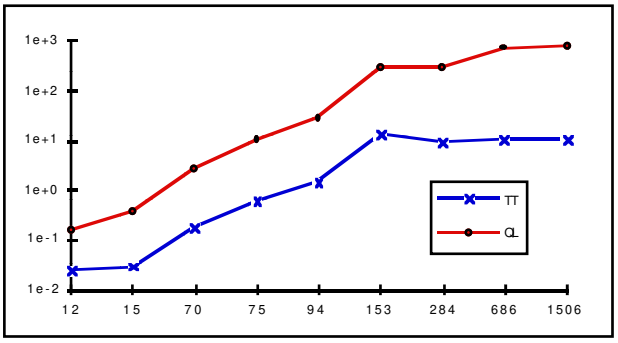


Chart 1: TT and QL running times

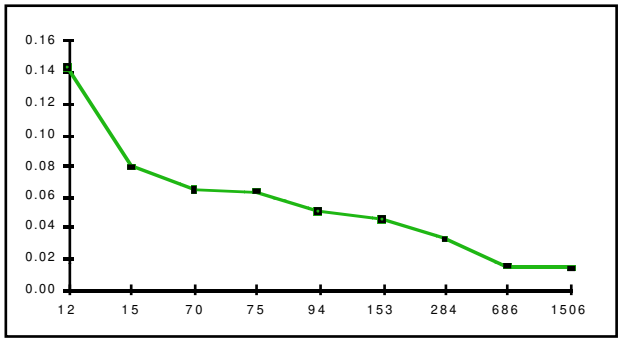


Chart 2: ratio between TT and QL running times

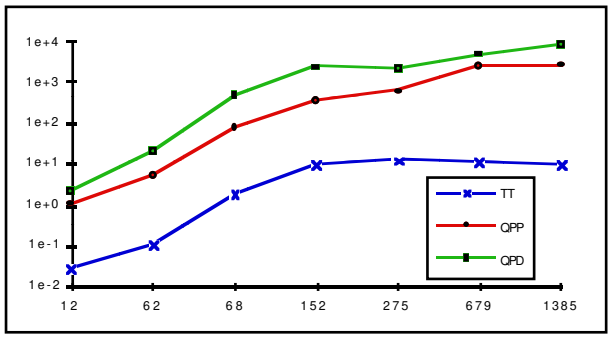


Chart 3: TT and CQP running times

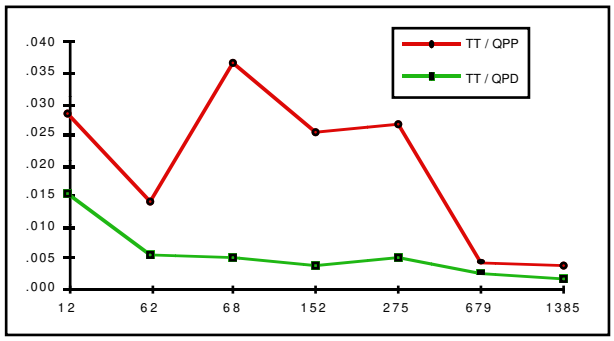


Chart 4: ratio between TT and CQP running times

The second comparison involved an early version of the MMCFB code, described in §II.2.1, ran on 43 Multicommodity Min Cost Flow (MMCF) problems (of the data set described in II.2.3.1): these instances are "easier" than the previous ones, since m (= number of edges in the graph) is between 200 and 700, and the Bundle code always terminated in less than 50 iterations - against up to 1000 - so that the average value of b was much smaller. Furthermore, from 94 to 99% of the nonnegativity constraints were always "active", and the average number of TT calls per BTT call were always smaller than 3 (and often ≈ 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 QL code at hand, only CQP was tested in this setting: the results are reported in Table 3, and show that the generic code is more than 25 times slower than BTT. On these problems, QPD outperforms QPP, even though, as shown in Chart 5 and Chart 6, for the largest instances the latter starts becoming competitive with the former. The results clearly show that the availability of an ad-hoc (QP) solver is crucial for a Bundle approach to be efficient in practice, at least in terms of total running time: other evidences supporting this assertion will be presented in Part II, but it should be already clear that not using a specialized code, i.e. paying 1-2 orders of magnitude more for the solution of (Δ_β), may well make a(n otherwise efficient) Bundle code unattractive w.r.t. alternative approaches.

n	*m*	*b*	*c*	*TT*	*QPP*	*QPD*
7	257	2	4	0.01	3.91	0.33
6	371	4	7	0.01	5.46	0.96
10	487	5	10	0.02	5.49	1.94
6	439	7	15	0.04	7.80	2.98
5	366	9	22	0.05	12.44	5.09
9	375	14	38	0.15	20.39	16.09

Table 3: comparison between BTT and CQP (solving both the Primal and Dual formulation)

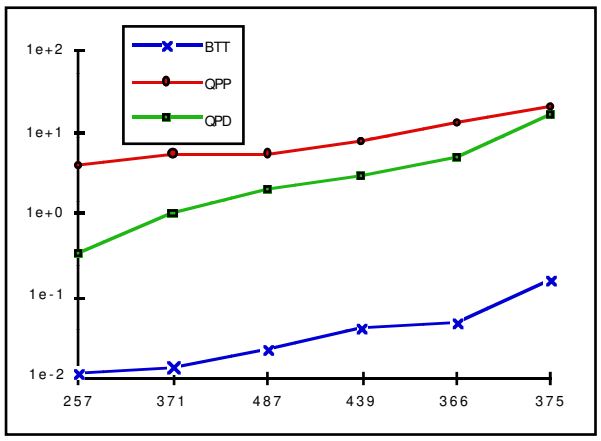


Chart 5: BTT and CQP running times

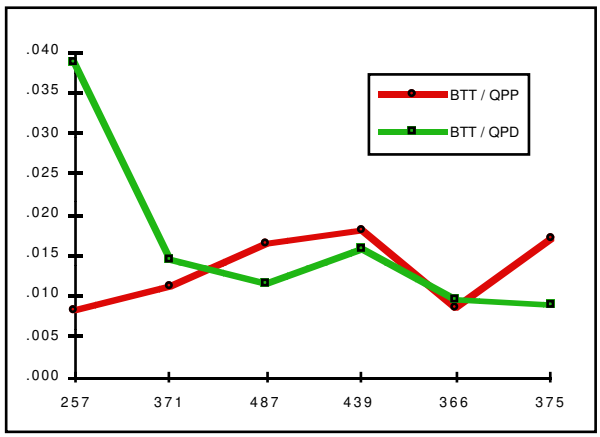


Chart 6: ratio between BTT and CQP running times

Another important feature of [B]TT, that makes it particularly well-suited as a (Δ_β)-solver, is that it allows the NDO solver to modify the dimension of the "dual" space at a low cost, allowing the implementation of *active-set* strategies that can further reduce the computational burden of the (Δ_β) solution. The importance of such strategies, that are clearly related to the multilevel pricing methods used in most simplex implementations, will be extensively proved in Part II: as a foretaste, we anticipate here some results obtained in the context of finding lower bounds to the Fixed Charge Multicommodity problem (Cf. §II.5.2). A sample of 12 medium-to-large size problems is presented in Table 4: each problem was solved twice, using (*Active Set*) or not (*Plain*) an active-set technique, but for the rest with exactly the same code and parameters. In the Table, the number of Lagrangean multipliers is reported in the *Variables* column: for both the variants, the number of (Δ_β) solved is reported in the *Calls* column (note that the Active Set strategy may change the number of steps performed by the Bundle algorithm), the total running time is reported in the *Total* column, and the time spent in the $\varphi(\boldsymbol{y})$ calculation is reported in the *Fi()* column.

	Active Set				Plain			
Variables	Calls	Total	Fi()	QP / Iter	Calls	Total	Fi()	QP / Iter
4400	178	24.24	9.17	0.08	186	67.04	7.56	0.32
4400	209	25.68	11.80	0.07	215	52.47	9.61	0.20
4400	100	13.05	5.38	0.08	98	37.76	4.13	0.34
4400	114	14.28	6.68	0.07	113	31.77	5.11	0.24
40400	501	2065.60	261.28	3.60	501	11688.60	257.08	22.82
40400	501	1041.37	244.90	1.59	501	4581.39	234.91	8.68
40400	501	1231.60	258.17	1.94	501	6062.63	239.96	11.62
40400	501	926.59	234.66	1.38	501	4137.74	228.24	7.80
6875	501	86.18	33.99	0.10	501	266.83	33.32	0.47
6875	347	68.24	26.60	0.12	347	161.64	25.75	0.39
6875	133	22.53	8.84	0.10	142	91.37	9.28	0.58
6875	129	22.24	9.25	0.10	134	73.93	9.32	0.48
63125	501	2410.37	456.92	3.90	501	10507.60	414.92	20.15
63125	501	2537.95	432.06	4.20	501	8291.95	370.12	15.81

Table 4: influence of the Active Set strategy on the overall efficiency

Wihout the Active Set strategy, the (Δ_β) solution takes from the 50% to the 98% of the total running time, especially on large problems: the Active Set strategy reduces the (Δ_β) cost per iteration of a 3 - 5

factor (Cf. the *QP/Iter* column), yielding a dramatic overall performance improvement. This is almost entirely due to the two-level structure of BTT, i.e. to its ability of fully exploiting both the active set of subgradients and the active set of variables from the previous call: although none of the sets is necessarily very "stable" among the iterations, they are stable enough to provide valuable information for helping in the (QP) solution.

Finally, a specific benefit of our C++ implementation is its simple and clear interface discussed in §I.3.6, that directly supports the basic operations of a Bundle code (insertion and deletion of subgradients and constraints, modification of α ...): it greatly simplified the task of developing and modifying the several specialized Bundle variants that have been realized so far, and isolated them from the (sometimes dramatic) changes that the implementation itself went through over the years.