Blended Conditional Gradients: The Unconditioning of Conditional Gradients

Gábor Braun [1] Sebastian Pokutta [1] Dan Tu [1] Stephen Wright [2]

Abstract

We present a *blended conditional gradient* approach for minimizing a smooth convex function over a polytope P, combining the Frank–Wolfe algorithm (also called conditional gradient) with gradient-based steps, different from away steps and pairwise steps, but still achieving linear convergence for strongly convex functions, along with good practical performance. Our approach retains all favorable properties of conditional gradient algorithms, notably avoidance of projections onto P and maintenance of iterates as sparse convex combinations of a limited number of extreme points of P. The algorithm is *lazy*, making use of inexpensive inexact solutions of the linear programming subproblem that characterizes the conditional gradient approach. It decreases measures of optimality rapidly, both in the number of iterations and in wall-clock time, outperforming even the lazy conditional gradient algorithms of (Braun et al., 2017). We also present a streamlined version of the algorithm that applies when P is the probability simplex.

1. Introduction

A common paradigm in convex optimization is minimization of a smooth convex function f over a polytope P. The conditional gradient (CG) algorithm, also known as "Frank–Wolfe" (Frank & Wolfe, 1956), (Levitin & Polyak, 1966) is enjoying renewed popularity because it can be implemented efficiently to solve important problems in data analysis. It is a first-order method, requiring access to gradients $\nabla f(x)$ and function values $f(x)$. In its original form, CG employs a linear programming (LP) oracle to minimize a linear function over the polytope P at each iteration. The cost of this operation depends on the complexity of P.

In this work, we describe a *blended conditional gradient (BCG)* approach, which takes one of several types of steps on the basis of the gradient ∇f at the current point. Our approach maintains an "active vertex set," consisting of some solutions from previous iterations. Building on (Braun et al., 2017), BCG uses a "weak-separation oracle" to find a vertex of P for which the linear objective attains some fraction of the reduction in f promised by the LP oracle, typically by searching among the current set of active vertices. If no vertex yielding acceptable reduction can be found, the LP oracle used in the original FW algorithm may be deployed. On other iterations, BCG employs a "simplex descent oracle," which takes a step within the convex hull of the active vertices, yielding progress either via reduction in function value (a "descent step") or via culling of the active vertex set (a "drop step"). The size of the active vertex set typically remains small, which benefits both the efficiency of the method and the "sparsity" of the final solution (i.e., its representation as a convex combination of a relatively small number of vertices).

BCG has similar theoretical convergence rates to several other variants of CG that have been studied recently, including pairwise-step and away-step variants and the lazy variants of (Braun et al., 2017). In several cases, we observe better empirical convergence for BCG than for these other variants. While the lazy variant of (Braun et al., 2017) has an advantage over baseline CG when the LP oracle is expensive, our BCG approach consistently outperforms the other variants in more general circumstances, both in per-iteration progress and in wall-clock time.

Related work

There has been an extensive body of work on conditional gradient algorithms; see the excellent overview of (Jaggi, 2013). Here we review only those papers most closely related to our work.

Our main inspiration comes from (Braun et al., 2017; Lan et al., 2017), which introduces the weak-separation oracle

[1]ISyE, Georgia Institute of Technology, Atlanta, GA [2]Computer Sciences Department, University of Wisconsin, Madison, WI. Correspondence to: Gábor Braun <gabor.braun@isye.gatech.edu>, Sebastian Pokutta <sebastian.pokutta@isye.gatech.edu>, Dan Tu <dan.tu@gatech.edu>, Stephen Wright <swright@cs.wisc.edu>.

Proceedings of the 36th International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019. Copyright 2019 by the author(s).

as a lazy alternative to calling the LP oracle in every iteration. It is influenced too by the method of (Rao et al., 2015), which maintains an active vertex set, using projected descent steps to improve the objective over the convex hull of this set, and culling the set on some steps to keep its size under control. While the latter method is a heuristic with no proven convergence bounds beyond those inherited from the standard Frank–Wolfe method, our BCG algorithm employs a criterion for *optimal trade-off between the various steps*, with a *proven* convergence rate equal to state-of-the-art Frank–Wolfe variants up to a constant factor.

Our main result shows linear convergence of BCG for strongly convex functions. Linearly convergent variants of CG were studied as early as (Guélat & Marcotte, 1986) for special cases and (Garber & Hazan, 2013) for the general case (though the latter work involves very large constants). More recently, linear convergence has been established for various pairwise-step and away-step variants of CG in (Lacoste-Julien & Jaggi, 2015), where the concept of an active vertex set is used to improve performance. Other memory-efficient decomposition-invariant variants were described in (Garber & Meshi, 2016) and (Bashiri & Zhang, 2017). Modification of descent directions and step sizes, reminiscent of the drop steps used in BCG, have been considered by (Freund & Grigas, 2016; Freund et al., 2017). The use of an inexpensive oracle based on a subset of the vertices of P, as an alternative to the full LP oracle, has been considered in (Kerdreux et al., 2018b). (Garber et al., 2018) proposes a fast variant of conditional gradients for matrix recovery problems.

BCG is quite distinct from the fully-corrective Frank–Wolfe algorithm (FCFW) (see, for example, (Holloway, 1974; Lacoste-Julien & Jaggi, 2015)). Both approaches maintain active vertex sets, generate iterates that lie in the convex hulls of these sets, and alternate between Frank–Wolfe steps generating new vertices and correction steps optimizing within the current active vertex set. However, convergence analyses of the FCFW algorithm assume that the correction steps have unit cost, though they can be quite expensive in practice, requiring multiple evaluations of the gradient ∇f. For BCG, by contrast, we assume only a *single* step of gradient descent type having unit cost (disregarding cost of line search). For further explanation of the differences between BCG and FCFW, see computational results in Figure 12 and discussion in Appendix D.

Contribution

Our contribution can be summarized as follows:

Blended Conditional Gradients (BCG). The BCG approach blends different types of descent steps: the traditional CG steps of (Frank & Wolfe, 1956), the lazified CG steps of (Braun et al., 2017), and gradient descent steps over the convex hull of the current active vertex set. It avoids projections onto P or onto the convex hull of the active vertices, and does not use away steps and pairwise steps, which are elements of other popular variants of CG. It achieves linear convergence for strongly convex functions (see Theorem 3.1), and $O(1/t)$ convergence after t iterations for general smooth functions. While the linear convergence proof of the Away-step Frank–Wolfe Algorithm (Lacoste-Julien & Jaggi, 2015, Theorem 1, Footnote 4) requires the objective function f to be defined on the Minkowski sum $P - P + P$, BCG does not need f to be defined outside the polytope P. The algorithm has complexity comparable to pairwise-step or away-step variants of conditional gradients, both in per-iteration running time and in the space required to store vertices and iterates. It is affine-invariant and parameter-free; estimates of such parameters as smoothness, strong convexity, or the diameter of P are not required. It maintains iterates as (often sparse) convex combinations of vertices, typically much sparser than the baseline CG methods, a property that is important for some applications. Such sparsity is due to the aggressive reuse of active vertices, and the fact that new vertices are added only as a kind of last resort. In wall-clock time as well as per-iteration progress, our computational results show that BCG can be orders of magnitude faster than competimg CG methods on some problems.

Simplex Gradient Descent (SiGD). In Section 4, we describe a new projection-free gradient descent procedure for minimizing a smooth function over the probability simplex, which can be used to implement the "simplex descent oracle" required by BCG.

Computational Experiments. We demonstrate the excellent computational behavior of BCG compared to other CG algorithms on standard problems, including video colocalization, sparse regression, structured SVM training, and structured regression. We observe significant computational speedups and in several cases empirically better convergence rates.

Outline

We summarize preliminary material in Section 2, including the two oracles that are the foundation of our BCG procedure. BCG is described and analyzed in Section 3, establishing linear convergence rates. The simplex gradient descent routine, which implements the simplex descent oracle, is described in Section 4. Our computational experiments are summarized in Section 5; more extensive experiments appear in Appendix D. Variants on the analysis and other auxiliary materials are relegated to the appendix. We mention in particular a variant of BCG that applies when P is the probability simplex, a special case that admits several simplifications and improvements to the analysis.

2. Preliminaries

We use the following notation: e_i is the i-th coordinate vector, $\mathbb{1} := (1, \ldots, 1) = e_1 + e_2 + \cdots$ is the all-ones vector, $\|\cdot\|$ denotes the Euclidean norm (ℓ_2-norm), $D = \operatorname{diam}(P) = \sup_{u,v \in P} \|u - v\|_2$ is the ℓ_2-diameter of P, and $\operatorname{conv} S$ denotes the convex hull of a set S of points. The *probability simplex* $\Delta^k := \operatorname{conv}\{e_1, \ldots, e_k\}$ is the convex hull of the coordinate vectors in dimension k.

Let f be a differentiable convex function. Recall that f is L-*smooth* if $f(y) - f(x) - \nabla f(x)(y - x) \leq L\|y - x\|^2/2$ for all $x, y \in P$. The function f has *curvature* C if $f(\gamma y + (1 - \gamma)x) \leq f(x) + \gamma \nabla f(x)(y - x) + C\gamma^2/2$, for all $x, y \in P$ and $0 \leq \gamma \leq 1$. (Note that an L-smooth function always has curvature $C \leq LD^2$.) Finally, f is *strongly convex* if for some $\alpha > 0$ we have $f(y) - f(x) - \nabla f(x)(y - x) \geq \alpha\|y - x\|^2/2$, for all $x, y \in P$. We use the following fact about strongly convex function when optimizing over P.

Fact 2.1 (Geometric strong convexity guarantee)**.** *(Lacoste-Julien & Jaggi, 2015, Theorem 6 and Eq. (28)) Given a strongly convex function f, there is a value $\mu > 0$ called the* geometric strong convexity *such that*

$$f(x) - \min_{y \in P} f(y) \leq \frac{\left(\max_{y \in S, z \in P} \nabla f(x)(y - z)\right)^2}{2\mu}$$

for any $x \in P$ and for any subset S of the vertices of P for which x lies in the convex hull of S.

The value of μ depends both on f and the geometry of P.

2.1. Simplex Descent Oracle

Given a convex objective function f and an ordered finite set $S = \{v_1, \ldots, v_k\}$ of points, we define $f_S \colon \Delta^k \to \mathbb{R}$ as follows:

$$f_S(\lambda) := f\left(\sum_{i=1}^{k} \lambda_i v_i\right). \tag{1}$$

When f_S is L_{f_S}-smooth, Oracle 1 returns an improving point x' in $\operatorname{conv} S$ together with a vertex set $S' \subseteq S$ such that $x' \in \operatorname{conv} S'$.

Oracle 1 Simplex Descent Oracle $\operatorname{SiDO}(x, S, f)$

Input: finite set $S \subseteq \mathbb{R}^n$, point $x \in \operatorname{conv} S$, convex smooth function $f \colon \operatorname{conv} S \to \mathbb{R}^n$;

Output: finite set $S' \subseteq S$, point $x' \in \operatorname{conv} S'$ satisfying either

drop step: $f(x') \leq f(x)$ and $S' \neq S$

descent step:

$$f(x) - f(x') \geq [\max_{u,v \in S} \nabla f(x)(u - v)]^2/(4L_{f_S})$$

In Section 4 we provide an implementation (Algorithm 2) of this oracle via a *single descent step*, which avoids projection and does not require knowledge of the smoothness parameter L_{f_S}.

2.2. Weak-Separation Oracle

Oracle 2 Weak-Separation Oracle $\operatorname{LPsep}_P(c, x, \Phi, K)$

Input: linear objective $c \in \mathbb{R}^n$, point $x \in P$, accuracy $K \geq 1$, gap estimate $\Phi > 0$;

Output: Either (1) vertex $y \in P$ with $c(x - y) \geq \Phi/K$, or (2) **false**: $c(x - z) \leq \Phi$ for all $z \in P$.

The weak-separation oracle Oracle 2 was introduced in (Braun et al., 2017) to replace the LP oracle traditionally used in the CG method. Provided with a point $x \in P$, a linear objective c, a target reduction value $\Phi > 0$, and an inexactness factor $K \geq 1$, it decides whether there exists $y \in P$ with $cx - cy \geq \Phi/K$, or else certifies that $cx - cz \leq \Phi$ for all $z \in P$. In our applications, $c = \nabla f(x)$ is the gradient of the objective at the current iterate x. Oracle 2 could be implemented simply by the standard LP oracle of minimizing cz over $z \in P$. However, it allows more efficient implementations, including the following. (1) *Caching*: testing previously obtained vertices $y \in P$ (specifically, vertices in the current active vertex set) to see if one of them satisfies $cx - cy \geq \Phi/K$. If not, the traditional LP oracle could be called to either find a new vertex of P satisfying this bound, or else to certify that $cx - cz \leq \Phi$ for all $z \in P$, and (2) *Early Termination*: Terminating the LP procedure as soon as a vertex of P has been discovered that satisfies $cx - cy \geq \Phi/K$. (This technique requires an LP implementation that generates vertices as iterates.) If the LP procedure runs to termination without finding such a point, it has certified that $cx - cz \leq \Phi$ for all $z \in P$. In (Braun et al., 2017) these techniques resulted in orders-of-magnitude speedups in wall-clock time in the computational tests, as well as sparse convex combinations of vertices for the iterates x_t, a desirable property in many contexts.

3. Blended Conditional Gradients

Our BCG approach is specified as Algorithm 1. We discuss the algorithm in this section and establish its convergence rate. The algorithm expresses each iterate x_t, $t = 0, 1, 2, \ldots$ as a convex combination of the elements of the active vertex set, denoted by S_t, as in the Pairwise and Away-step variants of CG. At each iteration, the algorithm calls either Oracle 1 or Oracle 2 in search of the next iterate, whichever promises the smaller function value, using a test in Line 6 based on an estimate of the dual gap. The same greedy principle is used in the Away-step CG approach, and its lazy variants. A critical role in the algorithm (and particularly in the test of Line 6) is played by the value

Φ_t, which is a current estimate of the primal gap — the difference between the current function value $f(x_t)$ and the optimal function value over P. When Oracle 2 returns **false**, the curent value of Φ_t is discovered to be an overestimate of the dual gap, so it is halved (Line 13) and we proceed to the next iteration. In subsequent discussion, we refer to Φ_t as the "gap estimate."

Algorithm 1 Blended Conditional Gradients (BCG)

```
Input: smooth convex function $f$, start vertex $x_0 \in P$,
    weak-separation oracle $\mathrm{LPsep}_P$, accuracy $K \geq 1$
Output: points $x_t$ in $P$ for $t = 1, \dots, T$
 1: $\Phi_0 \leftarrow \max_{v \in P} \nabla f(x_0)(x_0 - v)/2$   {Initial gap estimate}
 2: $S_0 \leftarrow \{x_0\}$
 3: for $t = 0$ to $T - 1$ do
 4:   $v_t^A \leftarrow \mathrm{argmax}_{v \in S_t} \nabla f(x_t) v$
 5:   $v_t^{FW-S} \leftarrow \mathrm{argmin}_{v \in S_t} \nabla f(x_t) v$
 6:   if $\nabla f(x_t)(v_t^A - v_t^{FW-S}) \geq \Phi_t$ then
 7:     $x_{t+1}, S_{t+1} \leftarrow \mathrm{SiDO}(x_t, S_t)$   {either a drop step or a descent step}
 8:     $\Phi_{t+1} \leftarrow \Phi_t$
 9:   else
10:     $v_t \leftarrow \mathrm{LPsep}_P(\nabla f(x_t), x_t, \Phi_t, K)$
11:     if $v_t =$ false then
12:       $x_{t+1} \leftarrow x_t$
13:       $\Phi_{t+1} \leftarrow \Phi_t/2$   {gap step}
14:       $S_{t+1} \leftarrow S_t$
15:     else
16:       $x_{t+1} \leftarrow \mathrm{argmin}_{x \in [x_t, v_t]} f(x)$   {FW step, with line search}
17:       Choose $S_{t+1} \subseteq S_t \cup \{v_t\}$ minimal such that
          $x_{t+1} \in \mathrm{conv}\, S_{t+1}$.
18:       $\Phi_{t+1} \leftarrow \Phi_t$
19:     end if
20:   end if
21: end for
```

In Line 17, the active set S_{t+1} is required to be minimal. By Caratheodory's theorem, this requirement ensures that $|S_{t+1}| \leq \dim P + 1$. In practice, the S_t are invariably small and no explicit reduction in size is necessary. The key requirement, in theory and practice, is that if after a call to Oracle SiDO the new iterate x_{t+1} lies on a face of the convex hull of the vertices in S_t, then at least one element of S_t is dropped to form S_{t+1}. This requirement ensures that the local pairwise gap in Line 6 is not too large due to stale vertices in S_t, which can block progress. Small size of the sets S_t is crucial to the efficiency of the algorithm, in rapidly determining the maximizer and minimizer of $\nabla f(x_t)$ over the active set S_t in Lines 4 and 5.

The constants in the convergence rate described in our main theorem (Theorem 3.1 below) depend on a modified curvature-like parameter of the function f. Given a vertex set S of P, recall from Section 2.1 the smoothness parameter L_{f_S} of the function $f_S \colon \Delta^k \to \mathbb{R}$ defined by (1). Define the *simplicial curvature* C^Δ to be

$$C^\Delta := \max_{S \colon |S| \leq 2 \dim P} L_{f_S} \tag{2}$$

to be the maximum of the L_{f_S} over all possible active sets. This affine-invariant parameter depends both on the shape of P and the function f. This is the relative smoothness constant $L_{f,A}$ from the predecessor of (Gutman & Peña, 2019), namely (Gutman & Peña, 2018, Definiton 2a), with an additional restriction: the simplex is restricted to faces of dimension at most $2 \dim P$, which appears as a bound on the size of S in our formulation. This restriction improves the constant by removing dependence on the number of vertices of the polytope, and can probably replace the original constant in convergence bounds. We can immediately see the effect in the common case of L-smooth functions, that the simplicial curvature is of reasonable magnitude, specifically,

$$C^\Delta \leq \frac{LD^2(\dim P)}{2},$$

where D is the diameter of P. This result follows from (2) and the bound on L_{f_S} from Lemma A.1 in the appendix. This bound is not directly comparable with the upper bound $L_{f,A} \leq LD^2/4$ in (Gutman & Peña, 2018, Corollary 2), because the latter uses the 1-norm on the standard simplex, while we use the 2-norm, the norm used by projected gradients and our simplex gradient descent. The additional factor $\dim P$ is explained by the n-dimensional standard simplex having constant minimum width 2 in 1-norm, but having minimum width dependent on the dimension n (specifically, $\Theta(1/\sqrt{n})$) in the 2-norm. Recall that the minimum width of a convex body $P \subseteq \mathbb{R}^n$ in norm $\|\cdot\|$ is $\min_\phi \max_{u,v \in P} \phi(u - v)$, where ϕ runs over all linear maps $\mathbb{R}^n \to \mathbb{R}$ having dual norm $\|\phi\|_* = 1$. For the 2-norm, this is just the minimum distance between parallel hyperplanes such that P lies between the two hyperplanes.

For another comparison, recall the curvature bound $C \leq LD^2$. Note, however, that the algorithm and convergence rate below are affine invariant, and the only restriction on the function f is that it has finite simplicial curvature. This restriction readily provides the curvature bound

$$C \leq 2C^\Delta, \tag{3}$$

where the factor 2 arises as the square of the diameter of the standard simplex Δ^k. (See Lemma A.2 in the appendix for details.) Note that S is allowed to be large enough so that every point of P is in the convex hull of some vertex subset S, by Caratheodory's theorem, and that the simplicial curvature provides an upper bound on the curvature

We describe the convergence of BCG (Algorithm 1) in the following theorem.

Theorem 3.1. *Let f be a strongly convex, smooth function over the polytope P with simplicial curvature C^Δ and geometric strong convexity μ. Then Algorithm 1 ensures $f(x_T) - f(x^*) \leq \varepsilon$, where x^* is an optimal solution to f in P for some iteration index T that satisfies*

$$T \leq \left\lceil \log \frac{2\Phi_0}{\varepsilon} \right\rceil + 8K \left\lceil \log \frac{\Phi_0}{2KC^\Delta} \right\rceil + \frac{64K^2C^\Delta}{\mu} \left\lceil \log \frac{4KC^\Delta}{\varepsilon} \right\rceil = O\left(\frac{C^\Delta}{\mu} \log \frac{\Phi_0}{\varepsilon} \right), \quad (4)$$

where $\log$ *denotes logarithms to the base* 2.

For smooth but not necessarily strongly convex functions f, the algorithm ensures $f(x_T) - f(x^*) \leq \varepsilon$ after $T = O(\max\{C^\Delta, \Phi_0\}/\varepsilon)$ iterations by a similar argument, which is omitted.

Proof. The proof tracks that of (Braun et al., 2017). We divide the iteration sequence into epochs that are demarcated by the *gap steps*, that is, the iterations for which the weak-separation oracle (Oracle 2) returns the value **false**, which results in Φ_t being halved for the next iteration. We then bound the number of iterates within each epoch. The result is obtained by aggregating across epochs.

We start by a well-known bound on the function value using the Frank–Wolfe point $v_t^{FW} := \operatorname{argmin}_{v \in P} \nabla f(x_t) v$ at iteration t, which follows from convexity:

$$f(x_t) - f(x^*) \leq \nabla f(x_t)(x_t - x^*) \leq \nabla f(x_t)(x_t - v_t^{FW}).$$

If iteration $t-1$ is a gap step, we have using $x_t = x_{t-1}$ and $\Phi_t = \Phi_{t-1}/2$ that

$$f(x_t) - f(x^*) \leq \nabla f(x_t)(x_t - v_t^{FW}) \leq 2\Phi_t. \quad (5)$$

This bound also holds at $t = 0$, by definition of Φ_0. Thus Algorithm 1 is guaranteed to satisfy $f(x_T) - f(x^*) \leq \varepsilon$ at some iterate T such that $T-1$ is a gap step and $2\Phi_T \leq \varepsilon$. Therefore, the total number of gap steps N_Φ required to reach this point satisfies

$$N_\Phi \leq \left\lceil \log \frac{2\Phi_0}{\varepsilon} \right\rceil, \quad (6)$$

which is also a bound on the total number of epochs. The next stage of the proof finds bounds on the number of iterations of each type within an individual epoch.

If iteration $t-1$ is a gap step, we have $x_t = x_{t-1}$ and $\Phi_t = \Phi_{t-1}/2$, and because the condition is false at Line 6 of Algorithm 1, we have

$$\nabla f(x_t)(v_t^A - x_t) \leq \nabla f(x_t)(v_t^A - v_t^{FW-S}) \leq 2\Phi_t. \quad (7)$$

This condition also holds trivially at $t = 0$, since $v_0^A = v_0^{FW-S} = x_0$. By summing (5) and (7), we obtain $\nabla f(x_t)(v_t^A - v_t^{FW}) \leq 4\Phi_t$, so it follows from Fact 2.1 that $f(x_t) - f(x^*) \leq \frac{[\nabla f(x_t)(v_t^A - v_t^{FW})]^2}{2\mu} \leq \frac{8\Phi_t^2}{\mu}$. By combining this inequality with (5), we obtain

$$f(x_t) - f(x^*) \leq \min\left\{8\Phi_t^2/\mu, 2\Phi_t\right\}, \quad (8)$$

for all t such that either $t = 0$ or else $t-1$ is a gap step. In fact, (8) holds for *all* t, because (1) the sequence of function values $\{f(x_s)\}_s$ is non-increasing; and (2) $\Phi_s = \Phi_t$ for all s in the epoch that starts at iteration t.

We now consider the epoch that starts at iteration t, and use s to index the iterations within this epoch. Note that $\Phi_s = \Phi_t$ for all s in this epoch.

We distinguish three types of iterations besides gap step. The first type is a *Frank–Wolfe* step, taken when the weak-separation oracle returns an improving vertex $v_s \in P$ such that $\nabla f(x_s)(x_s - v_s) \geq \Phi_s/K = \Phi_t/K$ (Line 16). Using the definition of curvature C, we have by standard Frank–Wolfe arguments that (c.f., (Braun et al., 2017)).

$$\begin{aligned} f(x_s) - f(x_{s+1}) &\geq \frac{\Phi_s}{2K} \min\left\{1, \frac{\Phi_s}{KC}\right\} \\ &\geq \frac{\Phi_t}{2K} \min\left\{1, \frac{\Phi_t}{2KC^\Delta}\right\}, \end{aligned} \quad (9)$$

where we used $\Phi_s = \Phi_t$ and $C \leq 2C^\Delta$ (from (3)). We denote by N_{FW}^t the number of Frank–Wolfe iterations in the epoch starting at iteration t.

The second type of iteration is a *descent step*, in which Oracle SiDO (Line 7) returns a point x_{s+1} that lies in the relative interior of $\operatorname{conv} S_s$ and with strictly smaller function value. We thus have $S_{s+1} = S_s$ and, by the definition of Oracle SiDO, together with (2), it follows that

$$\begin{aligned} f(x_s) - f(x_{s+1}) &\geq \frac{[\nabla f(x_s)(v_s^A - v_s^{FW-S})]^2}{4C^\Delta} \\ &\geq \frac{\Phi_s^2}{4C^\Delta} = \frac{\Phi_t^2}{4C^\Delta}. \end{aligned} \quad (10)$$

We denote by N_{desc}^t the number of descent steps that take place in the epoch that starts at iteration t.

The third type of iteration is one in which Oracle 1 returns a point x_{s+1} lying on a face of the convex hull of S_s, so that S_{s+1} is strictly smaller than S_s. Similarly to the Away-step Frank–Wolfe algorithm of (Lacoste-Julien & Jaggi, 2015), we call these steps *drop steps*, and denote by N_{drop}^t the number of such steps that take place in the epoch that starts at iteration t. Note that since S_s is expanded only at Frank–Wolfe steps, and then only by at most one element, the *total* number of drop steps across the whole algorithm cannot exceed the total number of Frank–Wolfe steps. We use this fact and (6) in bounding the total number of iterations T required for $f(x_T) - f(x^*) \le \varepsilon$:

$$\begin{aligned} T &\le N_\Phi + N_{\text{desc}} + N_{\text{FW}} + N_{\text{drop}} \\ &\le \left\lceil \log \frac{2\Phi_0}{\varepsilon} \right\rceil + N_{\text{desc}} + 2N_{\text{FW}} \\ &= \left\lceil \log \frac{2\Phi_0}{\varepsilon} \right\rceil + \sum_{t:\text{epoch start}} (N^t_{\text{desc}} + 2N^t_{\text{FW}}). \end{aligned} \tag{11}$$

Here N_{desc} denotes the total number of descent steps, N_{FW} the total number of Frank–Wolfe steps, and N_{drop} the total number of drop steps, which is bounded by N_{FW}, as just discussed.

Next, we seek bounds on the iteration counts N^t_{desc} and N^t_{FW} within the epoch starting with iteration t. For the total decrease in function value during the epoch, Equations (9) and (10) provide a lower bound, while $f(x_t) - f(x^*)$ is an obvious upper bound, leading to the following estimate using (8).

- If $\Phi_t \ge 2KC^\Delta$ then

  $$\begin{aligned} 2\Phi_t &\ge f(x_t) - f(x^*) \ge N^t_{\text{desc}} \frac{\Phi_t^2}{4C^\Delta} + N^t_{\text{FW}} \frac{\Phi_t}{2K} \\ &\ge N^t_{\text{desc}} \frac{\Phi_t K}{2} + N^t_{\text{FW}} \frac{\Phi_t}{2K} \ge (N^t_{\text{desc}} + 2N^t_{\text{FW}}) \frac{\Phi_t}{4K}, \end{aligned}$$

  hence

  $$N^t_{\text{desc}} + 2N^t_{\text{FW}} \le 8K. \tag{12}$$

- If $\Phi_t < 2KC^\Delta$, a similar argument provides

  $$\begin{aligned} \frac{8\Phi_t^2}{\mu} &\ge f(x_t) - f(x^*) \ge N^t_{\text{desc}} \frac{\Phi_t^2}{4C^\Delta} + N^t_{\text{FW}} \frac{\Phi_t^2}{4K^2C^\Delta} \\ &\ge (N^t_{\text{desc}} + 2N^t_{\text{FW}}) \frac{\Phi_t^2}{8K^2C^\Delta}, \end{aligned}$$

  leading to

  $$N^t_{\text{desc}} + 2N^t_{\text{FW}} \le \frac{64K^2C^\Delta}{\mu}. \tag{13}$$

There are at most

$$\left\lceil \log \frac{\Phi_0}{2KC^\Delta} \right\rceil \text{ epochs in the regime with } \Phi_t \ge 2KC^\Delta,$$
$$\left\lceil \log \frac{2KC^\Delta}{\varepsilon/2} \right\rceil \text{ epochs in the regime with } \Phi_t < 2KC^\Delta.$$

Combining (11) with the bounds (12) and (13) on N^t_{FW} and N^t_{desc}, we obtain (4). □

4. Simplex Gradient Descent

Here we describe the Simplex Gradient Descent approach (Algorithm 2), an implementation of the SiDO oracle (Oracle 1). Algorithm 2 requires only $O(|S|)$ operations beyond the evaluation of $\nabla f(x)$ and the cost of line search. (It is assumed that x is represented as a convex combination of vertices of P, which is updated during Oracle 1.) Apart from the (trivial) computation of the projection of $\nabla f(x)$ onto the linear space spanned by Δ^k, no projections are computed. Thus, Algorithm 2 is typically faster even than a step of Frank–Wolfe, for typical small sets S.

Alternative implementations of Oracle 1 are described in Section C.1. Section C.2 describes the special case in which P itself is a probability simplex. Here, BCG and its oracles are combined into a single, simple method with better constants in the convergence bounds.

In the algorithm, the form $c\mathbb{1}$ denotes the scalar product of c and $\mathbb{1}$, i.e., the sum of entries of c.

Algorithm 2 Simplex Gradient Descent Step (SiGD)

```
Input: polyhedron $P$, smooth convex function $f\colon P \to \mathbb{R}$,
    subset $S = \{v_1, v_2, \dots, v_k\}$ of vertices of $P$, point
    $x \in \operatorname{conv} S$
Output: set $S' \subseteq S$, point $x' \in \operatorname{conv} S'$
 1: Decompose $x$ as a convex combination $x = \sum_{i=1}^k \lambda_i v_i$,
    with $\sum_{i=1}^k \lambda_i = 1$ and $\lambda_i \ge 0$, $i = 1, 2, \dots, k$
 2: $c \leftarrow [\nabla f(x) v_1, \dots, \nabla f(x) v_k]$ {$c = \nabla f_S(\lambda)$; see (1)}
 3: $d \leftarrow c - (c\mathbb{1})\mathbb{1}/k$  {Projection onto the hyperplane of
    $\Delta^k$}
 4: if $d = 0$ then
 5:     return $x' = v_1$, $S' = \{v_1\}$        {Arbitrary vertex}
 6: end if
 7: $\eta \leftarrow \max\{\eta \ge 0 : \lambda - \eta d \ge 0\}$
 8: $y \leftarrow x - \eta \sum_i d_i v_i$
 9: if $f(x) \ge f(y)$ then
10:     $x' \leftarrow y$
11:     Choose $S' \subseteq S$, $S' \neq S$ with $x' \in \operatorname{conv} S'$.
12: else
13:     $x' \leftarrow \operatorname{argmin}_{z \in [x,y]} f(z)$
14:     $S' \leftarrow S$
15: end if
16: return $x'$, $S'$
```

To verify the validity of Algorithm 2 as an implementation of Oracle 1, note first that since y lies on a face of $\operatorname{conv} S$ by definition, it is always possible to choose a proper subset $S' \subseteq S$ in Line 11, for example, $S' := \{v_i : \lambda_i > \eta d_i\}$. The following lemma shows that with the choice $h := f_S$, Algorithm 2 correctly implements Oracle 1.

Lemma 4.1. *Let Δ^k be the probability simplex in k dimensions and suppose that $h\colon \Delta^k \to \mathbb{R}$ is an L_h-smooth* *function. Given some $\lambda \in \Delta^k$, define $d := \nabla h(\lambda) - (\nabla h(\lambda)\mathbb{1}/k)\mathbb{1}$ and let $\eta \geq 0$ be the largest value for which $\tau := \lambda - \eta d \geq 0$. Let $\lambda' := \operatorname{argmin}_{z \in [\lambda, \tau]} h(z)$. Then either $h(\lambda) \geq h(\tau)$ or*

$$h(\lambda) - h(\lambda') \geq \frac{[\max_{1 \leq i,j \leq k} \nabla h(\lambda)(e_i - e_j)]^2}{4L_h}.$$

Proof. Let $g(\zeta) := h(\zeta - (\zeta\mathbb{1})\mathbb{1}/k)$, then $\nabla g(\zeta) = \nabla h(\zeta - (\zeta\mathbb{1})\mathbb{1}/k) - (\nabla h(\zeta - (\zeta\mathbb{1})\mathbb{1}/k)\mathbb{1})\mathbb{1}/k$, and g is clearly L_h-smooth, too. In particular, $\nabla g(\lambda) = d$.

From standard gradient descent bounds, not repeated here, we have the following inequalities, for $\gamma \leq \min\{\eta, 1/L_h\}$:

$$\begin{aligned} h(\lambda) - h(\lambda - \gamma d) &= g(\lambda) - g(\lambda - \gamma \nabla g(\lambda)) \\ &\geq \gamma \frac{\|\nabla g(\lambda)\|_2^2}{2} \geq \gamma \frac{[\max_{1 \leq i,j \leq k} \nabla g(\lambda)(e_i - e_j)]^2}{4} \\ &= \gamma \frac{[\max_{1 \leq i,j \leq k} \nabla h(\lambda)(e_i - e_j)]^2}{4}, \end{aligned} \tag{14}$$

where the second inequality uses that the ℓ_2-diameter of the Δ^k is 2, and the last equality follows from $\nabla g(\lambda)(e_i - e_j) = \nabla h(\lambda)(e_i - e_j)$.

When $\eta \geq 1/L_h$, we conclude that $h(\lambda') \leq h(\lambda - (1/L_h)d) \leq h(\lambda)$, hence

$$h(\lambda) - h(\lambda') \geq \frac{[\max_{i,j \in \{1,2,\ldots,k\}} \nabla h(\lambda)(e_i - e_j)]^2}{4L_h},$$

which is the second case of the lemma. When $\eta < 1/L_h$, then setting $\gamma = \eta$ in (14) clearly provides $h(\lambda) - h(\tau) \geq 0$, which is the first case of the lemma. □

5. Computational Experiments (Summary)

To compare our experiments to previous work, we used problems and instances similar to those in (Lacoste-Julien & Jaggi, 2015; Garber & Meshi, 2016; Rao et al., 2015; Braun et al., 2017; Lan et al., 2017). These include structured regression, sparse regression, video co-localization, sparse signal recovery, matrix completion, and Lasso. We compared various algorithms denoted by the following acronyms: our algorithm (BCG), the Away-step Frank–Wolfe algorithm (ACG) and the Pairwise Frank–Wolfe algorithm (PCG) from (Lacoste-Julien & Jaggi, 2015; Garber & Meshi, 2016), the vanilla Frank–Wolfe algorithm (CG), as well as their lazified versions from (Braun et al., 2017). We add a prefix 'L' for the lazified versions. Figure 1 summarizes our results on four test problems. Further details and more extensive computational results are reported in Appendix D.

Performance Comparison

We implemented Algorithm 1 as outlined above and used SiGD (Algorithm 2) for the descent steps as described in Section 4. For line search in Line 13 of Algorithm 2, we perform standard backtracking, and for Line 16 of Algorithm 1, we do ternary search. In Figure 1, each of the four plots itself contains four subplots depicting results of four variants of CG on a single instance. The two subplots in each upper row measure progress in the logarithm (to base 2) of the function value, while the two subplots in each lower row report the logarithm of the gap estimate Φ_t from Algorithm 1. The subplots in the left column of each plot report performance in terms of number of iterations, while the subplots in the right column report wall-clock time.

As discussed earlier, $2\Phi_t$ upper bounds the primal gap (the difference between the function value at the current iterate and the optimal function value). The lazified algorithms (including BCG) halve Φ_t occasionally, which provides a stair-like appearance in the graphs. In implementations, if a stronger bound on the primal gap is available (e.g., from an LP oracle call), we reset Φ_t to half of that value, thus removing unnecessary successive halving steps. For the non-lazified algorithms, we plot the dual gap $\max_{v \in P} \nabla f(x_t)(x_t - v)$ as a gap estimate. The dual gap does not necessarily decrease in a monotone fashion (though of course the primal gap is monotone decreasing), so the plots have a zigzag appearance in some instances.

6. Final Remarks

In (Lan et al., 2017), an accelerated method based on weak separation and conditional gradient sliding was described. This method provided optimal tradeoffs between (stochastic) first-order oracle calls and weak-separation oracle calls. An open question is whether the same tradeoffs and acceleration could be realized by replacing SiGD (Algorithm 2) by an accelerated method.

After an earlier version of our work appeared online, (Kerdreux et al., 2018a) introduced the *Hölder Error Bound condition* (also known as *sharpness* or the *Łojasiewicz growth condition*). This is a family of conditions parameterized by $0 < p \leq 1$, interpolating between strongly convex ($p = 0$) and convex functions ($p = 1$). For such functions, convergence rate $O(1/\varepsilon^p)$ has been shown for Away-step Frank–Wolfe algorithms, among others. Our analysis can be similarly extended to objective functions satisfying this condition, leading to similar convergence rates.

Acknowledgements

We are indebted to Swati Gupta for the helpful discussions. Research reported in this paper was partially supported by NSF CAREER award CMMI-1452463, and also NSF Awards 1628384, 1634597, and 1740707; Subcontract 8F-30039 from Argonne National Laboratory; and Award N660011824020 from the DARPA Lagrange Program.

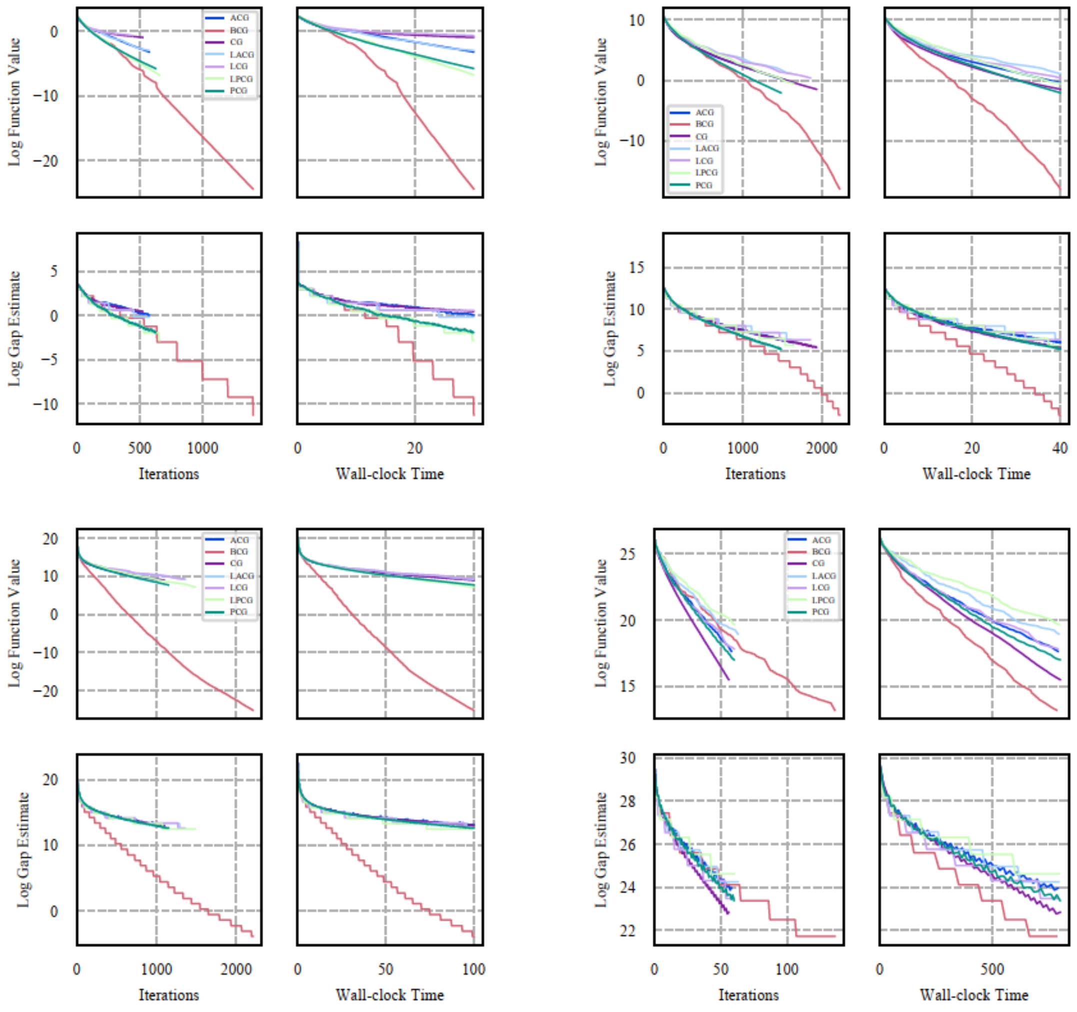

Figure 1. Four representative examples. (Upper-left) Sparse signal recovery: $\min_{x \in \mathbb{R}^n : \|x\|_1 \leq \tau} \|y - \Phi x\|_2^2$, where Φ is of size 1000×3000 with density 0.05. BCG made 1402 iterations with 155 calls to the weak-separation oracle LPsep_P. The final solution is a convex combination of 152 vertices. (Upper-right) Lasso. We solve $\min_{x \in P} \|Ax - b\|^2$ with P being the (scaled) ℓ_1-ball. A is a 400×2000 matrix with 100 non-zeros. BCG made 2130 iterations, calling LPsep_P 477 times, with the final solution being a convex combination of 462 vertices. (Lower-left) Structured regression over the Birkhoff polytope of dimension 50. BCG made 2057 iterations with 524 calls to LPsep_P. The final solution is a convex combination of 524 vertices. (Lower-right) Video co-localization over `netgen_12b` polytope with an underlying 5000-vertex graph. BCG made 140 iterations, with 36 calls to LPsep_P. The final solution is a convex combination of 35 vertices.

References

Bashiri, M. A. and Zhang, X. Decomposition-invariant conditional gradient for general polytopes with line search. In *Advances in Neural Information Processing Systems*, pp. 2687–2697, 2017.

Braun, G., Pokutta, S., and Zink, D. Lazifying conditional gradient algorithms. *Proceedings of ICML*, 2017.

Frank, M. and Wolfe, P. An algorithm for quadratic programming. *Naval research logistics quarterly*, 3(1-2): 95–110, 1956.

Freund, R. M. and Grigas, P. New analysis and results for the Frank–Wolfe method. *Mathematical Programming*, 155(1):199–230, 2016. ISSN 1436-4646. doi: 10.1007/s10107-014-0841-6. URL http://dx.doi.org/10.1007/s10107-014-0841-6.

Freund, R. M., Grigas, P., and Mazumder, R. An extended Frank–Wolfe method with "in-face" directions, and its application to low-rank matrix completion. *SIAM Journal on Optimization*, 27(1):319–346, 2017.

Garber, D. and Hazan, E. A linearly convergent conditional gradient algorithm with applications to online and stochastic optimization. *arXiv preprint arXiv:1301.4666*, 2013.

Garber, D. and Meshi, O. Linear-memory and decomposition-invariant linearly convergent conditional gradient algorithm for structured polytopes. *arXiv preprint, arXiv:1605.06492v1*, May 2016.

Garber, D., Sabach, S., and Kaplan, A. Fast generalized conditional gradient method with applications to matrix recovery problems. *arXiv preprint arXiv:1802.05581*, 2018.

Guélat, J. and Marcotte, P. Some comments on wolfe's 'away step'. *Mathematical Programming*, 35(1):110–119, 1986.

Gurobi Optimization. Gurobi optimizer reference manual version 6.5, 2016. URL https://www.gurobi.com/documentation/6.5/refman/.

Gutman, D. H. and Peña, J. F. The condition of a function relative to a polytope. *arXiv preprint arXiv:1802.00271*, February 2018.

Gutman, D. H. and Peña, J. F. The condition of a function relative to a set. *arXiv preprint arXiv:1901.08359*, January 2019.

Holloway, C. A. An extension of the Frank and Wolfe method of feasible directions. *Mathematical Programming*, 6(1):14–27, 1974.

Jaggi, M. Revisiting Frank–Wolfe: Projection-free sparse convex optimization. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pp. 427–435, 2013.

Joulin, A., Tang, K., and Fei-Fei, L. Efficient image and video co-localization with frank-wolfe algorithm. In *European Conference on Computer Vision*, pp. 253–268. Springer, 2014.

Kerdreux, T., d'Aspremont, A., and Pokutta, S. Restarting Frank–Wolfe. *arXiv preprint arXiv:1810.02429*, 2018a.

Kerdreux, T., Pedregosa, F., and d'Aspremont, A. Frank–Wolfe with subsampling oracle. *arXiv preprint arXiv:1803.07348*, 2018b.

Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R. E., Danna, E., Gamrath, G., Gleixner, A. M., Heinz, S., Lodi, A., Mittelmann, H., Ralphs, T., Salvagnin, D., Steffy, D. E., and Wolter, K. MIPLIB 2010. *Mathematical Programming Computation*, 3(2):103–163, 2011. doi: 10.1007/s12532-011-0025-9. URL http://mpc.zib.de/index.php/MPC/article/view/56/28.

Lacoste-Julien, S. and Jaggi, M. On the global linear convergence of Frank–Wolfe optimization variants. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 28, pp. 496–504. Curran Associates, Inc., 2015. URL http://papers.nips.cc/paper/5925-on-the-global-linear-convergence-of-frank-wolfe-optimization-variants.pdf.

Lan, G., Pokutta, S., Zhou, Y., and Zink, D. Conditional accelerated lazy stochastic gradient descent. *Proceedings of ICML*, 2017.

Lan, G. G. *Lectures on Optimization for Machine Learning*. ISyE, April 2017.

Levitin, E. S. and Polyak, B. T. Constrained minimization methods. *USSR Computational mathematics and mathematical physics*, 6(5):1–50, 1966.

Nemirovski, A. and Yudin, D. *Problem complexity and method efficiency in optimization*. Wiley, 1983. ISBN 0-471-10345-4.

Rao, N., Shah, P., and Wright, S. Forward–backward greedy algorithms for atomic norm regularization. *IEEE Transactions on Signal Processing*, 63(21):5798–5811, 2015.

A. Upper bound on simplicial curvature

Lemma A.1. *Let $f\colon P \to \mathbb{R}$ be an L-smooth function over a polytope P with diameter D in some norm $\|\cdot\|$. Let S be a set of vertices of P. Then the function f_S from Section 2.1 is smooth with smoothness parameter at most*

$$L_{f_S} \leq \frac{LD^2|S|}{4}.$$

Proof. Let $S = \{v_1, \dots, v_k\}$. Recall that $f_S\colon \Delta^k \to \mathbb{R}$ is defined on the probability simplex via $f_S(\alpha) := f(A\alpha)$, where A is the linear operator defined via $A\alpha := \sum_{i=1}^k \alpha_i v_i$. We need to show

$$f_S(\alpha) - f_S(\beta) - \nabla f_S(\beta)(\alpha - \beta) \leq \frac{LD^2|S|}{8} \cdot \|\alpha - \beta\|_2^2, \qquad \alpha, \beta \in \Delta^k. \quad (15)$$

We start by expressing the left-hand side in terms of f and applying the smoothness of f:

$$\begin{aligned} & f_S(\alpha) - f_S(\beta) - \nabla f_S(\beta)(\alpha - \beta) \\ & \quad = f(A\alpha) - f(A\beta) - \nabla f(A\beta) \cdot (A\alpha - A\beta) \\ & \quad \leq \frac{L}{2} \cdot \|A\alpha - A\beta\|^2. \end{aligned} \quad (16)$$

Let $\gamma_+ := \max\{\alpha - \beta, 0\}$ and $\gamma_- := \max\{\beta - \alpha, 0\}$ with the maximum taken coordinatewise. Then $\alpha - \beta = \gamma_+ - \gamma_-$ with γ_+ and γ_- nonnegative vectors with disjoint support. In particular,

$$\|\alpha - \beta\|_2^2 = \|\gamma_+ - \gamma_-\|_2^2 = \|\gamma_+\|_2^2 + \|\gamma_-\|_2^2. \quad (17)$$

Let $\mathbb{1}$ denote the vector of length k with all its coordinates 1. Since $\mathbb{1}\alpha = \mathbb{1}\beta = 1$, we have $\mathbb{1}\gamma_+ = \mathbb{1}\gamma_-$. Let t denote this last quantity, which is clearly nonnegative. If $t = 0$ then $\gamma_+ = \gamma_- = 0$ and $\alpha = \beta$, hence the claimed (15) is obvious. If $t > 0$ then γ_+/t and γ_-/t are points of the simplex Δ^k, therefore

$$D \geq \|A(\gamma_+/t) - A(\gamma_-/t)\| = \frac{\|A\alpha - A\beta\|}{t}. \quad (18)$$

Using (17) with k_+ and k_- denoting the number of non-zero coordinates of γ_+ and γ_-, respectively, we obtain

$$\begin{aligned} \|\alpha - \beta\|_2^2 = \|\gamma_+\|_2^2 + \|\gamma_-\|_2^2 &\geq t^2 \left(\frac{1}{k_+} + \frac{1}{k_-}\right) \\ &\geq t^2 \cdot \frac{4}{k_+ + k_-} \geq \frac{4t^2}{k}. \end{aligned} \quad (19)$$

By (18) and (19) we conclude that $\|A\alpha - A\beta\|^2 \leq kD^2\|\alpha - \beta\|_2^2/4$, which together with (16) proves the claim (15). □

Lemma A.2. *Let $f\colon P \to \mathbb{R}$ be a convex function over a polytope P with finite simplicial curvature C^Δ. Then f has curvature at most*

$$C \leq 2C^\Delta.$$

Proof. Let $x, y \in P$ be two distinct points of P. The line through x and y intersects P in a segment $[w, z]$, where w and z are points on the *boundary* of P, i.e., contained in facets of P, which have dimension $\dim P - 1$. Therefore by Caratheodory's theorem there are vertex sets S_w, S_z of P of size at most $\dim P$ with $w \in \operatorname{conv} S_w$ and $z \in \operatorname{conv} S_z$. As such $x, y \in \operatorname{conv} S$ with $S := S_w \cup S_z$ and $|S| \leq 2 \dim P$.

Reusing the notation from the proof of Lemma A.1, let $k := |S|$ and A be a linear transformation with $S = \{Ae_1, \dots, Ae_k\}$ and $f_S(\zeta) = f(A\zeta)$ for all $\zeta \in \Delta^k$. Since $x, y \in \operatorname{conv} S$, there are $\alpha, \beta \in \Delta^k$ with $x = A\alpha$ and $y = A\beta$. Therefore by smoothness of f_S together with $L_{f_S} \leq C^\Delta$ and $\|\beta - \alpha\| \leq \sqrt{2}$:

$$\begin{aligned} & f(\gamma y + (1-\gamma)x) - f(x) - \gamma \nabla f(x)(y - x) \\ & = f(\gamma A\beta + (1-\gamma)A\alpha) - f(A\alpha) - \gamma \nabla f(A\alpha) \cdot (A\beta - A\alpha) \\ & \quad = f_S(\gamma\beta + (1-\gamma)\alpha) - f_S(\alpha) - \gamma \nabla f_S(\alpha)(\beta - \alpha) \\ & \quad \leq \frac{L_{f_S}\|\gamma(\beta - \alpha)\|^2}{2} = \frac{L_{f_S}\|\beta - \alpha\|^2}{2} \cdot \gamma^2 \leq C^\Delta \gamma^2 \end{aligned}$$

showing that $C \leq 2C^\Delta$ as claimed. □

B. Algorithmic enhancements

We describe various enhancements that can be made to the BCG algorithm, to improve its practical performance while staying broadly within the framework above. Computational testing with these enhancements is reported in Section D.

B.1. Sparsity and culling of active sets

Sparse solutions (which in the current context means "solutions that are a convex combination of a small number of vertices of P") are desirable for many applications. Techniques for promoting sparse solutions in conditional gradients were considered in (Rao et al., 2015). In many situations, a sparse approximate solution can be identified at the cost of some increase in the value of the objective function.

We explored two sparsification approaches, which can be applied separately or together, and performed preliminary computational tests for a few of our experiments in Section D.

(i) *Promoting drop steps.* Here we relax Line 9 in Algorithm 2 from testing $f(y) \geq f(x)$ to $f(y) \geq f(x) - \varepsilon$, where $\varepsilon := \min\{\frac{\max\{p,0\}}{2}, \varepsilon_0\}$ with $\varepsilon_0 \in \mathbb{R}$ some upper bound on the accepted potential increase in objective function value and p being the amount of reduction in f achieved on the latest iteration. This technique allows a controlled increase of the objective function value in return for additional sparsity. The same convergence analysis will apply, with an additional factor of 2 in the estimates of the total number of iterations.

(ii) *Post-optimization.* Once the considered algorithm has stopped with active set S_0, solution x_0, and dual gap d_0, we re-run the algorithm with the same objective function f over the facet $\operatorname{conv} S_0$, i.e., we solve $\min_{x \in \operatorname{conv} S_0} f(x)$ terminating when the dual gap reaches d_0.

These approaches can sparsify the solutions of the baseline algorithms Away-step Frank–Wolfe, Pairwise Frank–Wolfe, and lazy Pairwise Frank–Wolfe; see (Rao et al., 2015). We observed, however, that the iterates generated by BCG are often quite sparse. In fact, the solutions produced by BCG are sparser than those produced by the baseline algorithms even when sparsification is used in the benchmarks but *not* in BCG! This effect is not surprising, as BCG adds new vertices to the active vertex set only when really necessary for ensuring further progress in the optimization.

Two representative examples are shown in Table 1, where we report the effect of sparsification in the size of the active set as well as the increase in objective function value.

We also compared evolution of the function value and size of the active set. BCG decreases function value much more for the same number of vertices because, by design, it performs more descent on a given active set; see Figure 2.

B.2. Blending with pairwise steps

Algorithm 1 mixes descent steps with Frank–Wolfe steps. One might be tempted to replace the Frank–Wolfe steps with (seemingly stronger) pairwise steps, as the information needed for the latter steps is computed in any case. In our tests, however, this variant did not substantially differ in practical performance from the one that uses the standard Frank–Wolfe step (see Figure 9). The explanation is that BCG uses descent steps that typically provide better directions than either Frank–Wolfe steps or pairwise steps. When the pairwise gap over the active set is small, the Frank–Wolfe and pairwise directions typically offer a similar amount of reduction in f.

C. Algorithmic Variations

C.1. Alternative implementations of Oracle 1

Algorithm 2 is probably the least expensive possible implementation of Oracle 1, in general. We may consider other implementations, based on projected gradient descent, that aim to decrease f by a greater amount in each step and possibly make more extensive reductions to the set S. *Projected gradient descent* would seek to minimize f_S along the piecewise-linear path $\{\operatorname{proj}_{\Delta^k}(\lambda - \gamma \nabla f_S(\lambda)) \mid \gamma \geq 0\}$. Such a search is more expensive, but may result in a new active set S' that is significantly smaller than the current set S and, since the reduction in f_S is at least as great as the reduction on the interval $\gamma \in [0, \eta]$ alone, it also satisfies the requirements of Oracle 1.

More advanced methods for optimizing over the simplex could also be considered, for example, mirror descent (see (Nemirovski & Yudin, 1983)) and accelerated versions of mirror descent and projected gradient descent; see (Lan, 2017) for a good overview. The effects of these alternatives on the overall convergence rate of Algorithm 1 has not been studied; the analysis is complicated significantly by the lack of guaranteed improvement in each (inner) iteration.

The accelerated versions are considered in the computational tests in Section D, but on the examples we tried, the inexpensive implementation of Algorithm 2 usually gave the fastest overall performance. We have not tested mirror descent versions.

C.2. Simplex Gradient Descent as a stand-alone algorithm

We describe a variant of Algorithm 1 for the special case in which P is the probability simplex Δ^k. Since optimization of a linear function over Δ^k is trivial, we use the standard LP oracle in place of the weak-separation oracle (Oracle 2), resulting in the non-lazy variant Algorithm 3. Observe that the per-iteration cost is only $O(k)$. In cases of k very large, we could also formulate a version of Algorithm 3 that uses a weak-separation oracle (Oracle 2) to evaluate only a subset of the coordinates of the gradient, as in coordinate descent. The resulting algorithm would be an interpolation of Algorithm 3 below and Algorithm 1; details are left to the reader.

When line search is too expensive, one might replace Line 14 by $x_{t+1} = (1 - 1/L_f)x_t + y/L_f$, and Line 17 by $x_{t+1} = (1 - 2/(t+2))x_t + (2/(t+2))e_w$. These employ the standard step sizes for (projected) gradient descent and the Frank–Wolfe algorithm, and yield the required descent guarantees.

We now describe convergence rates for Algorithm 3, noting that better constants are available in the convergence rate expression than those obtained from a direct application of Theorem 3.1.

Corollary C.1. *Let f be an α-strongly convex and L_f-smooth function over the probability simplex Δ^k with $k \geq 2$. Let x^* be a minimum point of f in Δ^k. Then Algorithm 3*

Table 1. Size of active set and percentage increase in function value after sparsification. (No sparsification performed for BCG.) Left: Video Co-localization over `netgen_08a`. Since we use LPCG and PCG as benchmarks, we report (i) separately as well. Right: Matrix Completion over `movielens100k` instance. BCG without sparsification provides sparser solutions than the baseline methods with sparsification. In the last column, we report the percentage increase in objective function value due to sparsification. (Because this quantity is not affine invariant, this value should serve only to rank the quality of solutions.)

	vanilla	(i)	(i), (ii)	$\Delta f(x)$
PCG	112	62	60	2.6%
LPCG	94	70	64	0.1%
BCG	60	59	40	0.0%

	vanilla	(i), (ii)	$\Delta f(x)$
ACG	300	298	7.4%
PCG	358	255	8.2%
BCG	211	211	0.0%

Algorithm 3 Stand-Alone Simplex Gradient Descent

```
Input: convex function $f$
Output: points $x_t$ in $\Delta^k$ for $t = 1, \dots, T$
1: $x_0 = e_1$
2: for $t = 0$ to $T - 1$ do
3:   $S_t \leftarrow \{i : x_{t,i} > 0\}$
4:   $a_t \leftarrow \operatorname{argmax}_{i \in S_t} \nabla f(x_t)_i$
5:   $s_t \leftarrow \operatorname{argmin}_{i \in S_t} \nabla f(x_t)_i$
6:   $w_t \leftarrow \operatorname{argmin}_{1 \leq i \leq k} \nabla f(x_t)_i$
7:   if $\nabla f(x_t)_{a_t} - \nabla f(x_t)_{s_t} > \nabla f(x_t) x_t - \nabla f(x_t)_{w_t}$ then
8:     $d_i = \begin{cases} \nabla f(x_t)_i - \sum_{j \in S} \nabla f(x_t)_j / |S_t| & i \in S_t \\ 0 & i \notin S_t \end{cases}$
       for $i = 1, 2, \dots, k$
9:     $\eta = \max\{\gamma : x_t - \gamma d \geq 0\}$          {ratio test}
10:    $y = x_t - \eta d$
11:    if $f(x_t) \geq f(y)$ then
12:      $x_{t+1} \leftarrow y$                                   {drop step}
13:    else
14:      $x_{t+1} \leftarrow \operatorname{argmin}_{x \in [x_t, y]} f(x)$   {descent step}
15:    end if
16:  else
17:    $x_{t+1} \leftarrow \operatorname{argmin}_{x \in [x, e_{w_t}]} f(x)$       {FW step}
18:  end if
19: end for
```

converges with rate

$$f(x_T) - f(x^*) \leq \left(1 - \frac{\alpha}{4L_f k}\right)^T \cdot (f(x_0) - f(x^*)), \quad T = 1, 2, \dots .$$

If f is not strongly convex (that is, $\alpha = 0$), we have

$$f(x_T) - f(x^*) \leq \frac{8L_f}{T}, \quad T = 1, 2, \dots .$$

Proof. The structure of the proof is similar to that of (Lacoste-Julien & Jaggi, 2015, Theorem 8). Recall from (Lacoste-Julien & Jaggi, 2015, §B.1) that the pyramidal width of the probability simplex is $W \geq 2/\sqrt{k}$, so that the geometric strong convexity of f is $\mu \geq 4\alpha/k$. The diameter of Δ^k is $D = \sqrt{2}$, and it is easily seen that $C^\Delta = L_f$ and $C \leq L_f D^2/2 = L_f$.

To maintain the same notation as in the proof of Theorem 3.1, we define $v_t^A = e_{a_t}$, $v_t^{FW-S} = e_{s_t}$ and $v_t^{FW} = e_{w_t}$. In particular, we have $\nabla f(x_t)_{w_t} = \nabla f(x_t) v_t^{FW}$, $\nabla f(x_t)_{s_t} = \nabla f(x_t) v_t^{FW-S}$, and $\nabla f(x_t)_{a_t} = \nabla f(x_t) v_t^A$. Let $h_t := f(x_t) - f(x^*)$.

In the proof, we use several elementary estimates. First, by convexity of f and the definition of the Frank–Wolfe step, we have

$$h_t = f(x_t) - f(x^*) \leq \nabla f(x_t)(x_t - v_t^{FW}). \tag{20}$$

Second, by Fact 2.1 and the estimate $\mu \geq 4\alpha/k$ for geometric strong convexity, we obtain

$$h_t \leq \frac{[\nabla f(x_t)(v_t^A - v_t^{FW})]^2}{8\alpha/k}. \tag{21}$$

Let us consider a fixed iteration t. Suppose first that we take a descent step (Line 14), in particular, $\nabla f(x_t)(v_t^A - v_t^{FW-S}) \geq \nabla f(x_t)(x_t - v_t^{FW})$ from Line 7 which, together with $\nabla f(x_t) x_t \geq \nabla f(x_t) v^{FW-S}$, yields

$$2\nabla f(x_t)(v_t^A - v_t^{FW-S}) \geq \nabla f(x_t)(v_t^A - v_t^{FW}). \tag{22}$$

By Lemma 4.1, we have

$$\begin{aligned} f(x_t) - f(x_{t+1}) &\geq \frac{\left[\nabla f(x_t)(v^A - v^{FW-S})\right]^2}{4L_f} \\ &\geq \frac{\left[\nabla f(x_t)(v^A - v^{FW})\right]^2}{16L_f} \geq \frac{\alpha}{2L_f k} \cdot h_t, \end{aligned}$$

where the second inequality follows from (22) and the third inequality follows from (21).

If a Frank–Wolfe step is taken (Line 17), we have similarly to (9) that

$$f(x_t) - f(x_{t+1}) \geq \frac{\nabla f(x_t)(x_t - v^{FW})}{2} \cdot \min\left\{1, \frac{\nabla f(x_t)(x_t - v^{FW})}{2L_f}\right\}.$$

Combining with (20), we have either $f(x_t) - f(x_{t+1}) \geq h_t/2$ or

$$\begin{aligned} f(x_t) - f(x_{t+1}) &\geq \frac{[\nabla f(x_t)(x_t - v^{FW})]^2}{4L_f} \\ &\geq \frac{\left[\nabla f(x_t)(v^A - v^{FW})\right]^2}{16L_f} \geq \frac{\alpha}{2L_f k} \cdot h_t. \end{aligned}$$

Since $\alpha \leq L_f$, the latter is always smaller than the former, and hence is a lower bound that holds for all Frank–Wolfe steps.

Since $f(x_t) - f(x_{t+1}) = h_t - h_{t+1}$, we have $h_{t+1} \leq (1 - \alpha/(2L_f k))h_t$ for descent steps and Frank–Wolfe steps, while obviously $h_{t+1} \leq h_t$ for drop steps (Line 12). For any given iteration counter T, let T_{desc} be the number of descent steps taken before iteration T, T_{FW} be the number of Frank–Wolfe steps taken before iteration T, and T_{drop} be the number of drop steps taken before iteration T. We have $T_{\text{drop}} \leq T_{\text{FW}}$, so that similarly to (11)

$$T = T_{\text{desc}} + T_{\text{FW}} + T_{\text{drop}} \leq T_{\text{desc}} + 2T_{\text{FW}}. \tag{23}$$

By compounding the decrease at each iteration, and using (23) together with the identity $(1 - \epsilon/2)^2 \geq (1 - \epsilon)$ for any $\epsilon \in (0, 1)$, we have

$$\begin{aligned} h_T &\leq \left(1 - \frac{\alpha}{2L_f k}\right)^{T_{\text{desc}}+T_{\text{FW}}} h_0 \leq \left(1 - \frac{\alpha}{2L_f k}\right)^{T/2} h_0 \\ &\leq \left(1 - \frac{\alpha}{4L_f k}\right)^{T} \cdot h_0. \end{aligned}$$

The case for the smooth but not strongly convex functions is similar: we obtain for descent steps

$$\begin{aligned} h_t - h_{t+1} &= f(x_t) - f(x_{t+1}) \\ &\geq \frac{\left[\nabla f(x_t)(v^A - v^{FW-S})\right]^2}{4L_f} \\ &\geq \frac{\left[\nabla f(x_t)(x - v^{FW})\right]^2}{4L_f} \geq \frac{h_t^2}{4L_f}, \end{aligned} \tag{24}$$

where the second inequality follows from (20).

For Frank–Wolfe steps, we have by standard estimations

$$h_{t+1} \leq \begin{cases} h_t - h_t^2/(4L_f) & \text{if } h_t \leq 2L_f, \\ L_f \leq h_t/2 & \text{otherwise.} \end{cases} \tag{25}$$

Given an iteration T, we define T_{drop}, T_{FW} and T_{desc} as above, and show by induction that

$$h_T \leq \frac{4L_f}{T_{\text{desc}} + T_{\text{FW}}}, \quad \text{for } T \geq 1. \tag{26}$$

Equation (26), i.e., $h_T \leq 8L_f/T$ easily follows from this via $T_{\text{drop}} \leq T_{\text{FW}}$. Note that the first step is necessarily a Frank–Wolfe step, hence the denominator is never 0.

If iteration T is a drop step, then $T > 1$, and the claim is obvious by induction from $h_T \geq h_{T-1}$. Hence we assume that iteration T is either a descent step or a Frank–Wolfe step. If $T_{\text{desc}} + T_{\text{FW}} \leq 2$ then by (24) or (25) we obtain either $h_T \leq L_f < 2L_f$ or $h_T \leq h_{T-1} - h_{T-1}^2/(4L_f) \leq 2L_f$, without using any upper bound on h_{T-1}, proving (26) in this case. Note that this includes the case $T = 1$, the start of the induction.

Finally, if $T_{\text{desc}} + T_{\text{FW}} \geq 3$, then $h_{T-1} \leq 4L_f/(T_{\text{desc}} + T_{\text{FW}} - 1) \leq 2L_f$ by induction, therefore a familiar argument using (24) or (25) provides

$$\begin{aligned} h_T &\leq \frac{4L_f}{T_{\text{desc}} + T_{\text{FW}} - 1} - \frac{4L_f}{(T_{\text{desc}} + T_{\text{FW}} - 1)^2} \\ &\leq \frac{4L_f}{T_{\text{desc}} + T_{\text{FW}}}, \end{aligned}$$

proving (26) in this case, too, finishing the proof. □

D. Computational experiments

To compare our experiments to previous work we used problems and instances similar to those in (Lacoste-Julien & Jaggi, 2015; Garber & Meshi, 2016; Rao et al., 2015; Braun et al., 2017; Lan et al., 2017). These problems include structured regression, sparse regression, video colocalization, sparse signal recovery, matrix completion, and Lasso. In particular, we compared our algorithm to the Pairwise Frank–Wolfe algorithm from (Lacoste-Julien & Jaggi, 2015; Garber & Meshi, 2016) and the lazified Pairwise Frank–Wolfe algorithm from (Braun et al., 2017). We also benchmarked against the lazified versions of the vanilla Frank–Wolfe and the Away-step Frank–Wolfe as presented in (Braun et al., 2017) for completeness. We implemented our code in `Python 3.6` using `Gurobi` (see (Gurobi Optimization, 2016)) as the LP solver for complex feasible regions; as well as obvious direct implementations for the probability simplex, the cube and the ℓ_1-ball. As feasible regions, we used instances from MIPLIB2010 (see (Koch et al., 2011)), as done before in (Braun et al., 2017), along with some of the examples in (Bashiri & Zhang, 2017). We used quadratic objective functions for the tests with random coefficients, making sure that the global minimum lies outside the feasible region, to make the optimization problem

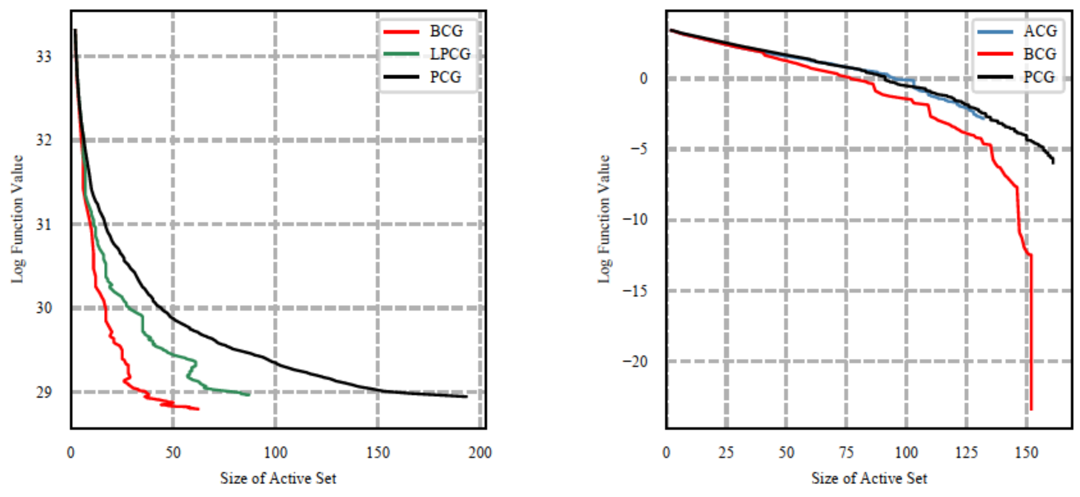

Figure 2. Comparison of ACG, PCG and LPCG against BCG in function value and size of the active set. Left: Video Co-Localization instance. Right: Sparse signal recovery.

non-trivial; see below in the respective sections for more details.

Every plot contains four diagrams depicting results of a single instance. The upper row measures progress in the logarithm of the function value, while the lower row does so in the logarithm of the gap estimate. The left column measures performance in the number of iterations, while the right column does so in wall-clock time. In the graphs we will compare various algorithms denoted by the following abbreviations: Pairwise Frank–Wolfe (PCG), Away-step Frank–Wolfe (ACG), (vanilla) Frank–Wolfe (CG), blended conditional gradients (BCG); we indicate the lazified versions of (Braun et al., 2017) by prefixing with an ‘L’. All tests were conducted with an instance-dependent, fixed time limit, which can be easily read off the plots.

The value Φ_t provided by the algorithm is an estimate of the primal gap $f(x_t) - f(x^*)$. The lazified versions (including BCG) use it to estimate the required stepwise progress, halving it occasionally, which provides a stair-like appearance in the graphs for the dual progress. Note that if the certification in the weak-separation oracle that $c(z - x) \geq \Phi$ for all $z \in P$ is obtained from the original LP oracle (which computes the actual optimum of cy over $y \in P$), then we update the gap estimate Φ_{t+1} with that value; otherwise the oracle would continue to return **false** anyway until Φ drops below that value. For the non-lazified algorithms, we plot the dual gap $\max_{v \in P} \nabla f(x_t)(x_t - v)$.

Performance comparison

We implemented Algorithm 1 as outlined above and used SiGD for the descent steps as described in Section 4. For line search in Line 13 of Algorithm 2 we perform standard backtracking line search, and for Line 16 of Algorithm 1, we do ternary search. We provide four representative example plots in Figure 1 to summarize our results.

Lasso. We tested BCG on lasso instances and compared them to vanilla Frank–Wolfe, Away-step Frank–Wolfe, and Pairwise Frank–Wolfe. We generated Lasso instances similar to (Lacoste-Julien & Jaggi, 2015), which has also also been used by several follow-up papers as benchmark. Here we solve $\min_{x \in P} \|Ax - b\|^2$ with P being the (scaled) ℓ_1-ball. We considered instances of varying sizes and the results (as well as details about the instance) can be found in Figure 3. Note that we did not benchmark any of the lazified versions of (Braun et al., 2017) here, because the linear programming oracle is so simple that lazification is not beneficial and we used the LP oracle directly.

Video co-localization instances. We also tested BCG on video co-localization instances as done in (Lacoste-Julien & Jaggi, 2015). It was shown in (Joulin et al., 2014) that video co-localization can be naturally reformulated as optimizing a quadratic function over a flow (or path) polytope. To this end, we run tests on the same flow polytope instances as used in (Lan et al., 2017) (obtained from `http://lime.cs.elte.hu/~kpeter/data/mcf/road/`). We depict the results in Figure 4.

Structured regression. We also compared BCG against PCG and LPCG on structured regression problems, where we minimize a quadratic objective function over polytopes corresponding to hard optimization problems used as benchmarks in e.g., (Braun et al., 2017; Lan et al., 2017; Bashiri

& Zhang, 2017). As in Lasso, we minimize a least-squares objective but instead of the ℓ_1-ball, the feasible regions are the polytopes from MIPLIB2010 (see (Koch et al., 2011)). Additionally, we compare ACG, PCG, and vanilla CG over the Birkhoff polytope for which linear optimization is fast (we are using the Hungarian algorithm), so that there is little gain to be expected from lazification. See Figures 5 and 6 for results.

Matrix completion. Clearly, our algorithm also works directly over compact convex sets, even though with a weaker theoretical bound of $O(1/\varepsilon)$ as convex sets need not have a pyramidal width bounded away from 0, and linear optimization might dominate the cost, and hence the advantage of lazification and BCG might be even greater empirically.

To this end, we also considered Matrix Completion instances over the spectrahedron $S = \{X \succeq 0 : \mathrm{Tr}[X] = 1\} \subseteq \mathbb{R}^{n\times n}$, where we solve the problem:

$$\min_{X\in S} \sum_{(i,j)\in L} (X_{i,j} - T_{i,j})^2,$$

where $D = \{T_{i,j} \mid (i,j) \in L\} \subseteq \mathbb{R}$ is a data set. In our tests we used the data sets Movie Lens 100k and Movie Lens 1m from `https://grouplens.org/datasets/movielens/` We subsampled in the 1m case to generate 3 different instances.

As in the case of the Lasso benchmarks, we benchmark against ACG, PCG, and CG, as the linear programming oracle is simple and there is no gain to be expected from lazification. In the case of matrix completion, the performance of BCG is quite comparable to ACG, PCG, and CG in iterations, which makes sense over the spectrahedron, because the gradient approximations computed by the linear optimization oracle are essentially identical to the actual gradient, so that there is no gain from the blending with descent steps. In wall-clock time, vanilla CG performs best as the algorithm has the lowest implementation overhead beyond the oracle calls compared to BCG, ACG, and PCG (see Figure 7) and in particular does not have to maintain the (large) active set.

Sparse signal recovery. We also performed computational experiments on the sparse signal recovery instances from (Rao et al., 2015), which have the following form:

$$\hat{x} = \operatorname*{argmin}_{x\in\mathbb{R}^n : \|x\|_1 \le \tau} \|y - \Phi x\|_2^2.$$

We chose a variety of parameters in our tests, including one test that matches the setup in (Rao et al., 2015). As in the case of the Lasso benchmarks, we benchmark against ACG, PCG, and CG, as the linear programming oracle is simple and there is no gain to be expected from lazification. The results are shown in Figure 8.

PGD vs. SiGD as subroutine

To demonstrate the superiority of SiGD over PGD we also tested two implementations of BCG, once with standard PGD as subroutine and once with SiGD as subroutine. The results can be found in Figure 9 (right): while PGD and SiGD compare essentially identical in per-iteration progress, in terms of wall clock time the SiGD variant is much faster. For comparison, we also plotted LPCG on the same instance.

Pairwise steps vs. Frank–Wolfe steps

As pointed out in Section B.2, a natural extension is to replace the Frank–Wolfe steps in Line 16 of Algorithm 1 with pairwise steps, since the information required is readily available. In Figure 9 (left) we depict representative behavior: Little to no advantage when taking the more complex pairwise step. This is expected as the Frank–Wolfe steps are only needed to add new vertices as the drop steps are subsumed the steps from the SiDO oracle. Note that BCG with Frank–Wolfe steps is slightly faster per iteration, allowing for more steps within the time limit.

Comparison between lazified variants and BCG

For completeness, we also ran tests for BCG against various other lazified variants of conditional gradient descent. The results are consistent with our observations from before which we depict in Figure 10.

Standard vs. accelerated version

Another natural variant of our algorithm is to replace the SiDO subroutine with its accelerated variant (both possible for PGD and SiGD). As expected, due to the small size of the subproblem, we did not observe any significant speedup from acceleration; see Figure 11.

Comparison to Fully-Corrective Frank–Wolfe

As mentioned in the introduction, BCG is quite different from FCFW. BCG is much faster and, in fact, FCFW is usually already outpeformed by the much more efficient Pairwise-step CG (PCG), except in some special cases. In Figure 12, the left column compares FCFW and BCG *only across those iterations where FW steps were taken*; for completeness, we also implemented a variant *FCFW (fixed steps)* where only a fixed number of descent steps in the correction subroutine are performed. As expected FCFW has a better "per-FW-iteration performance," because it performs *full* correction. The excessive cost of FCFW's correction routine shows up in the wall-clock time (right column), where FCFW is outperformed even by vanilla pairwise-step CG. This becomes even more apparent when the iterations in the correction subroutine are broken out and reported as well (see middle column). For purposes of comparison, BCG and FCFW used both SiGD steps in the subroutine. (This actually gives an advantage to FCFW, as SiGD was not known until the current paper.) The per-iteration progress of FCFW is poor, due to spending many iterations to optimize over active sets that are irrelevant for the optimal solution. Our tests highlight the fact that correction steps do not have constant cost in practice.

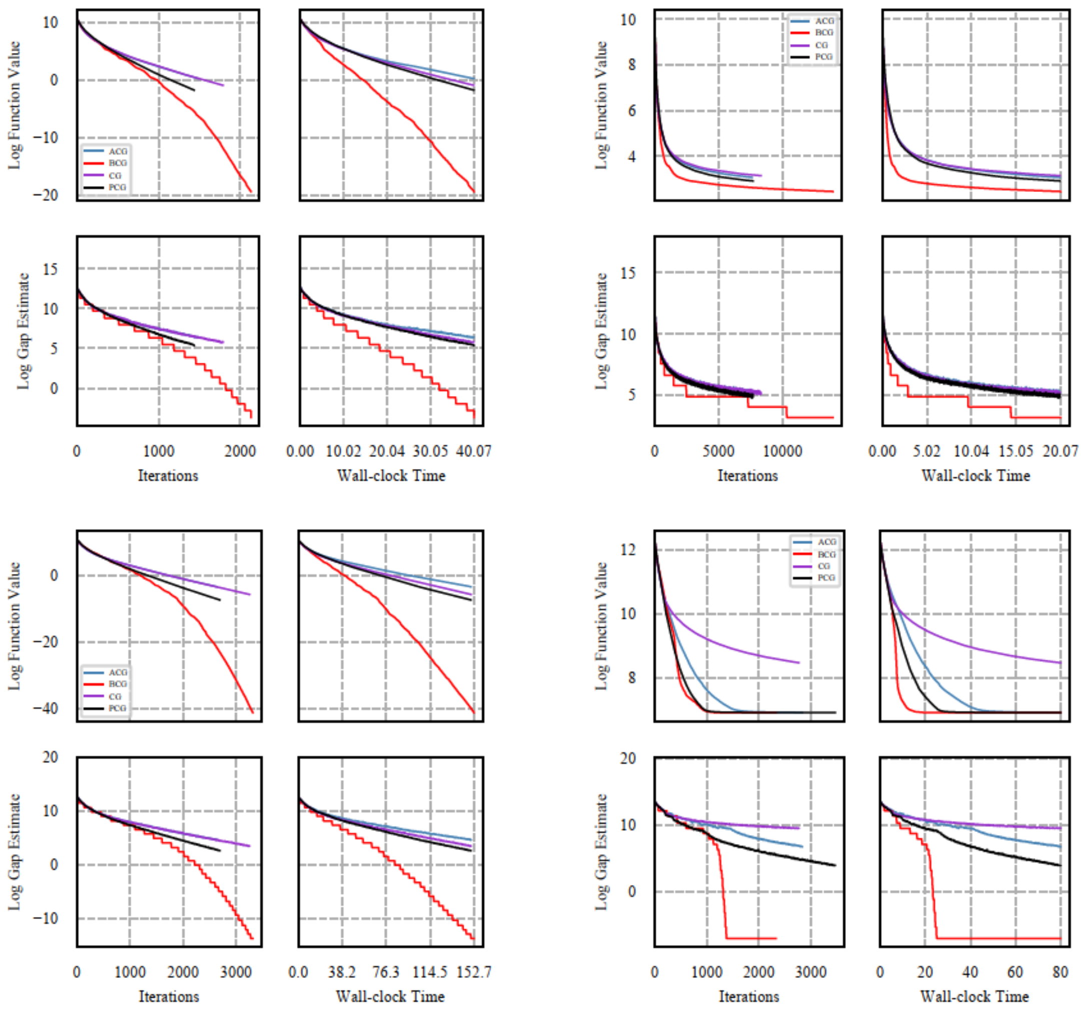

Figure 3. Comparison of BCG, ACG, PCG and CG on Lasso instances. Upper-left: A is a 400×2000 matrix with 100 non-zeros. BCG made 2130 iterations, calling the LP oracle 477 times, with the final solution being a convex combination of 462 vertices giving the sparsity. Upper-right: A is a 200×200 matrix with 100 non-zeros. BCG made 13952 iterations, calling the LP oracle 258 times, with the final solution being a convex combination of 197 vertices giving the sparsity. Lower-left: A is a 500×3000 matrix with 100 non-zeros. BCG made 3314 iterations, calling the LP oracle 609 times, with the final solution being a convex combination of 605 vertices giving the sparsity. Lower-right: A is a 1000×1000 matrix with 200 non-zeros. BCG made 2328 iterations, calling the LP oracle 1007 times, with the final solution being a convex combination of 526 vertices giving the sparsity.

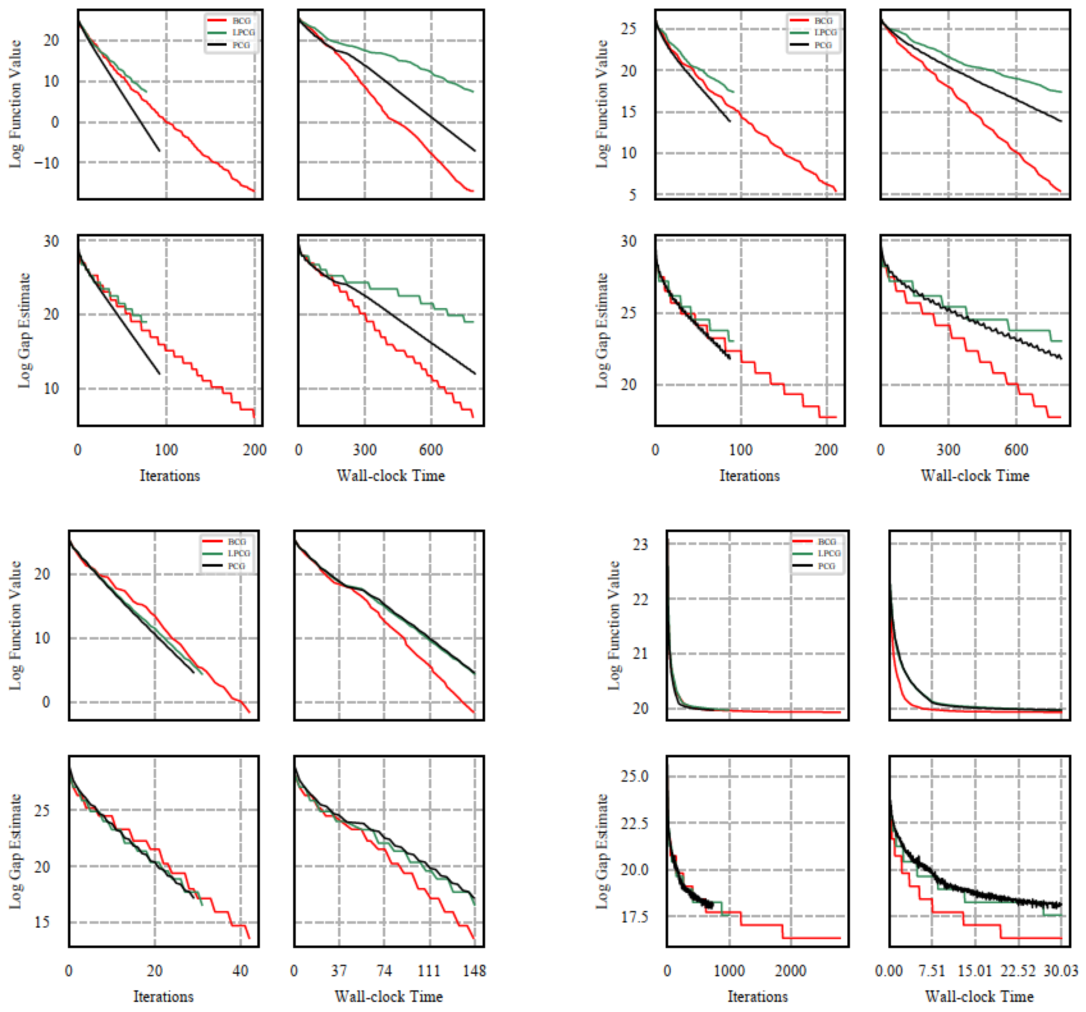

Figure 4. Comparison of PCG, Lazy PCG, and BCG on video co-localization instances. Upper-Left: `netgen_12b` for a 3000-vertex graph. BCG made 202 iterations, called LPsep_P 56 times and the final solution is a convex combination of 56 vertices. Upper-Right: `netgen_12b` over a 5000-vertex graph. BCG did 212 iterations, LPsep_P was talked 58 times, and the final solution is a convex combination of 57 vertices. Lower-Left: `road_paths_01_DC_a` over a 2000-vertex graph. Even on instances where lazy PCG gains little advantage over PCG, BCG performs significantly better with empirically higher rate of convergence. BCG made 43 iterations, LPsep_P was called 25 times, and the final convex combination has 25 vertices Lower-Right: `netgen_08a` over a 800-vertex graph. BCG made 2794 iterations, LPsep_P was called 222 times, and the final convex combination has 106 vertices.

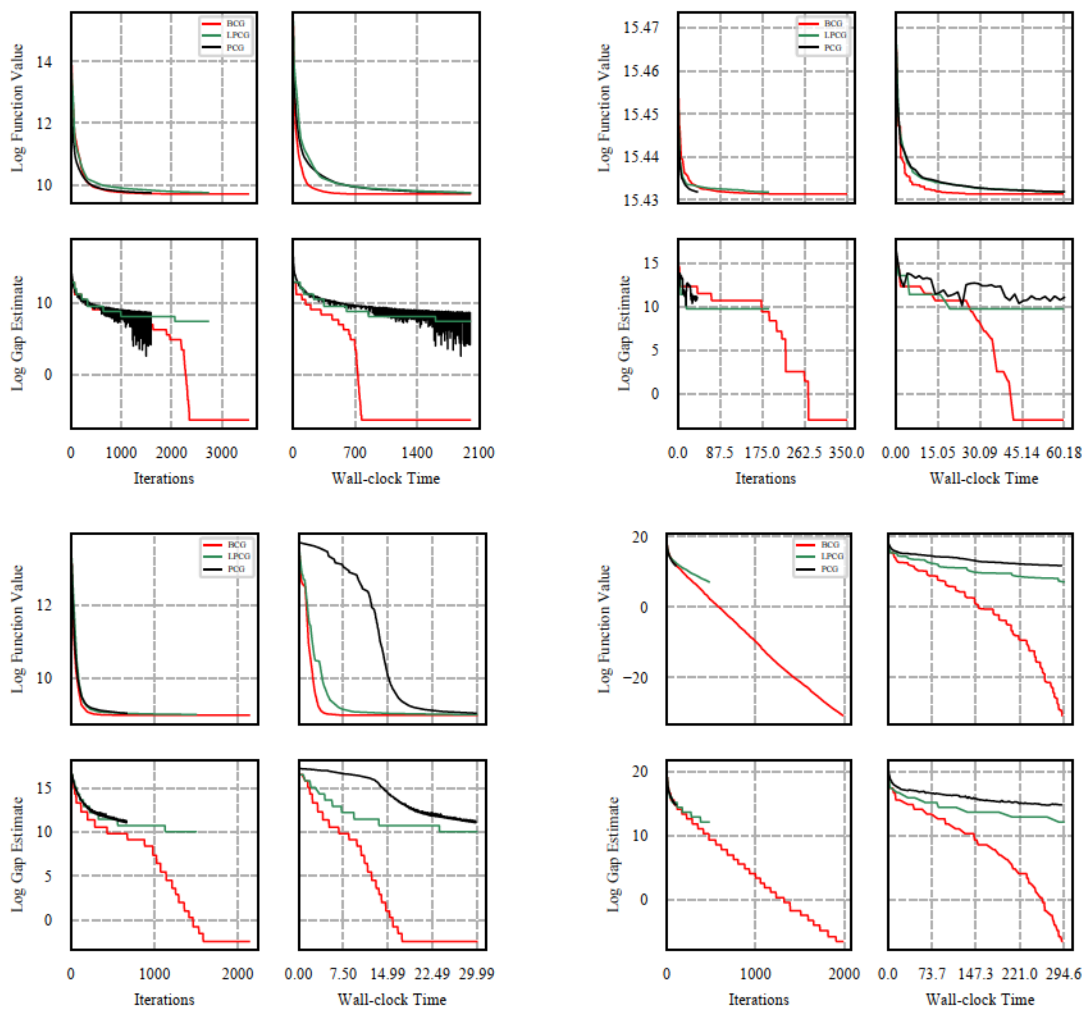

Figure 5. Comparison of BCG, LPCG and PCG on structured regression instances. Upper-Left: Over the `disctom` polytope. BCG made 3526 iterations with 1410 LPsep_P calls and the final solution is a convex combination of 85 vertices. Upper-Right: Over a `maxcut` polytope over a graph with 28 vertices. BCG made 76 LPsep_P calls and the final solution is a convex combination of 13 vertices. Lower-Left: Over the `m100n500k4r1` polytope. BCG made 2137 iterations with 944 LPsep_P calls and the final solution is a convex combination of 442 vertices. Lower-right: Over the spanning tree polytope over the complete graph with 10 nodes. BCG made 1983 iterations with 262 LPsep_P calls and the final solution is a convex combination of 247 vertices. BCG outperforms LPCG and PCG, even in the cases where LPCG is much faster than PCG.

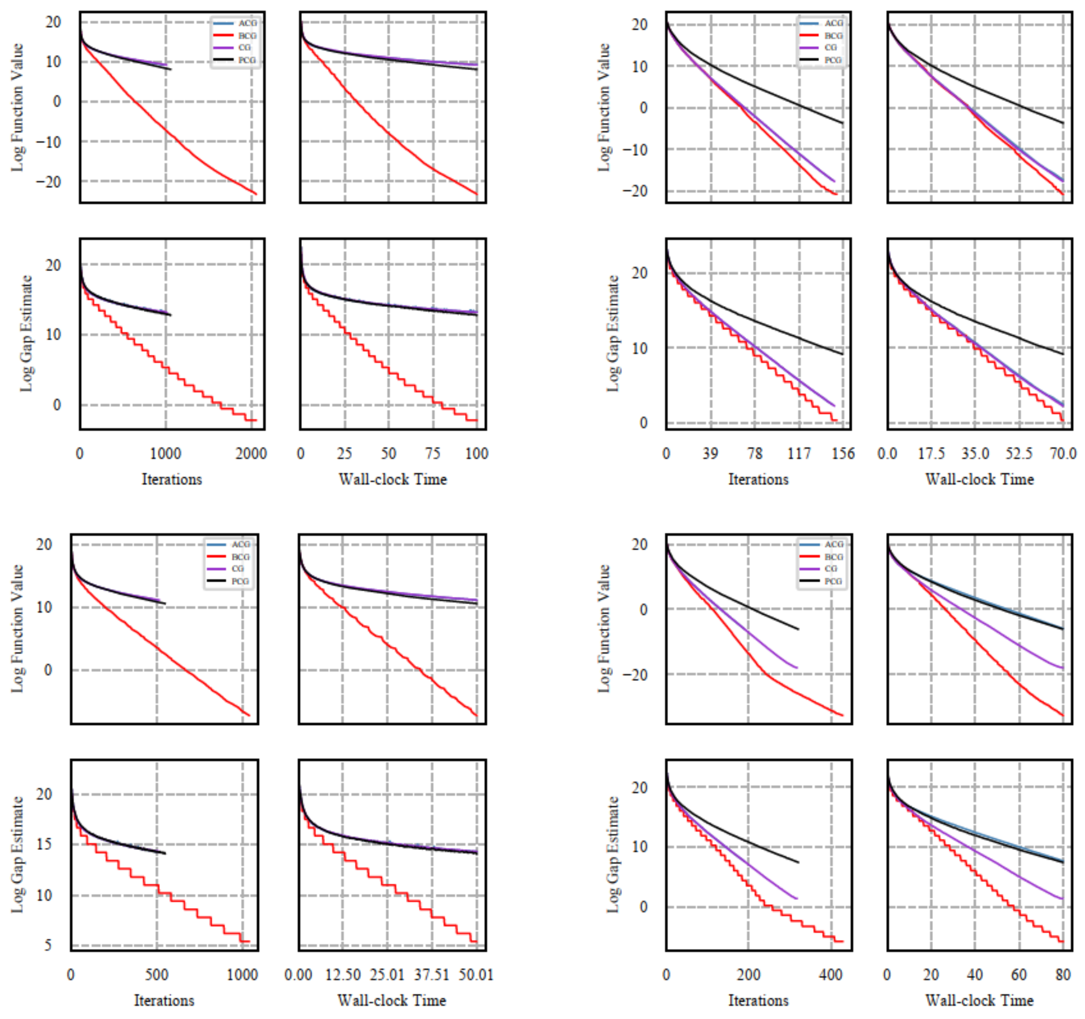

Figure 6. Comparison of BCG, ACG, PCG and CG over the Birkhoff polytope. Upper-Left: Dimension 50. BCG made 2057 iterations with 524 LPsep_P calls and the final solution is a convex combination of 524 vertices. Upper-Right: Dimension 100. BCG made 151 iterations with 134 LPsep_P calls and the final solution is a convex combination of 134 vertices. Lower-Left: Dimension 50. BCG made 1040 iterations with 377 LPsep_P calls and the final solution is a convex combination of 377 vertices. Lower-right: Dimension 80. BCG made 429 iterations with 239 LPsep_P calls and the final solution is a convex combination of 239 vertices. BCG outperforms ACG, PCG and CG in all cases.

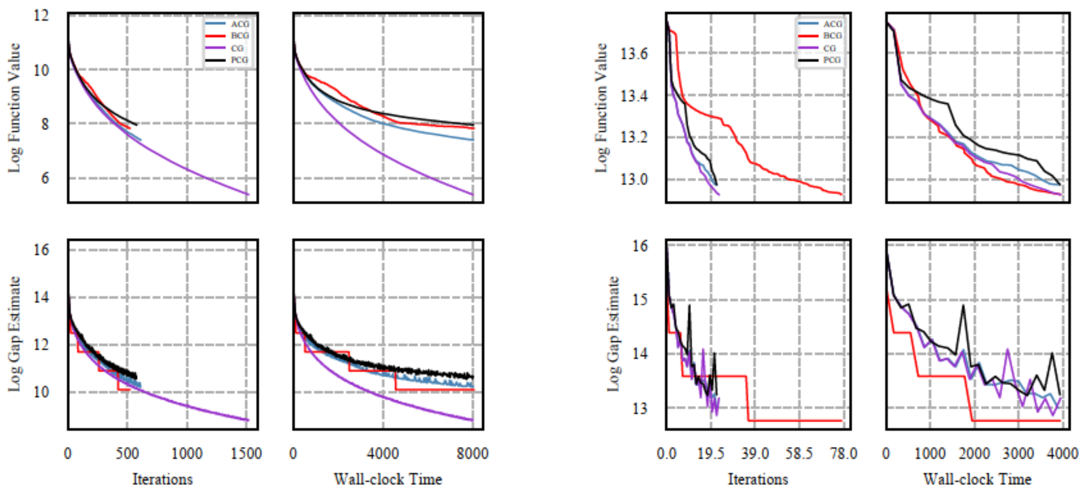

Figure 7. Comparison of BCG, ACG, PCG and CG on matrix completion instances over the spectrahedron. Upper-Left: Over the movie lens 100k data set. BCG made 519 iterations with 346 LPsep_P calls and the final solution is a convex combination of 333 vertices. Upper-Right: Over a subset of movie lens 1m data set. BCG made 78 iterations with 17 LPsep_P calls and the final solution is a convex combination of 14 vertices. BCG performs very similar to ACG, PCG, and vanilla CG as discussed.

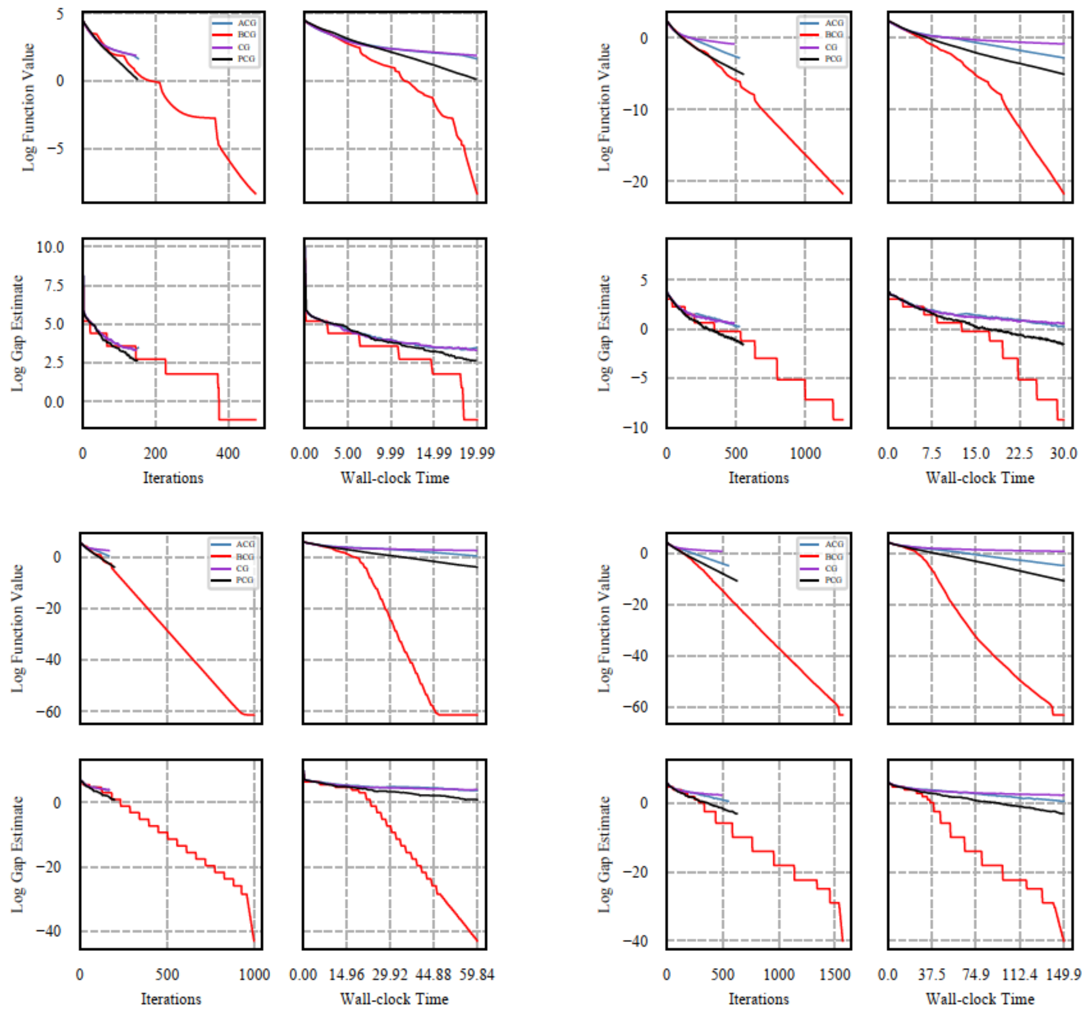

Figure 8. Comparison of BCG, ACG, PCG and CG on a sparse signal recovery problem. Upper-Left: Dimension is 5000×1000 density is 0.1. BCG made 547 iterations with 102 LPsep_P calls and the final solution is a convex combination of 102 vertices. Upper-Right: Dimension is 1000×3000 density is 0.05. BCG made 1402 iterations with 155 LPsep_P calls and the final solution is a convex combination of 152 vertices. Lower-Left: Dimension is 10000×1000 density is 0.05. BCG made 997 iterations with 87 LPsep_P calls and the final solution is a convex combination of 52 vertices. Lower-right: dimension is 5000×2000 density is 0.05. BCG made 1569 iterations with 124 LPsep_P calls and the final solution is a convex combination of 103 vertices. BCG outperforms all other algorithms in all examples significantly.

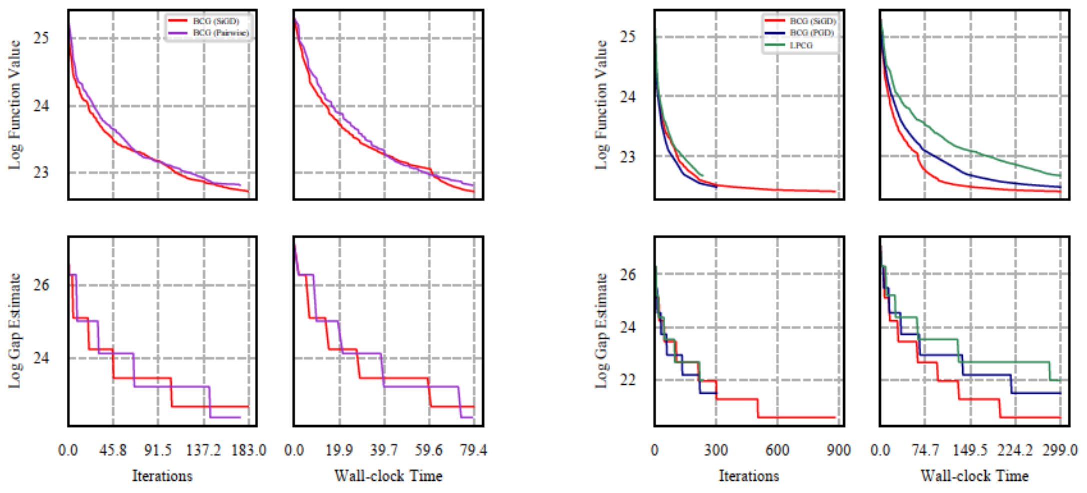

Figure 9. Comparison of BCG variants on a small video co-localization instance (instance `netgen_10a`). Left: BCG with vanilla Frank–Wolfe steps (red) and with pairwise steps (purple). Performance is essentially equivalent here which matches our observations on other instances. Right: Comparison of oracle implementations PGD and SiGD. SiGD is significantly faster in wall-clock time.

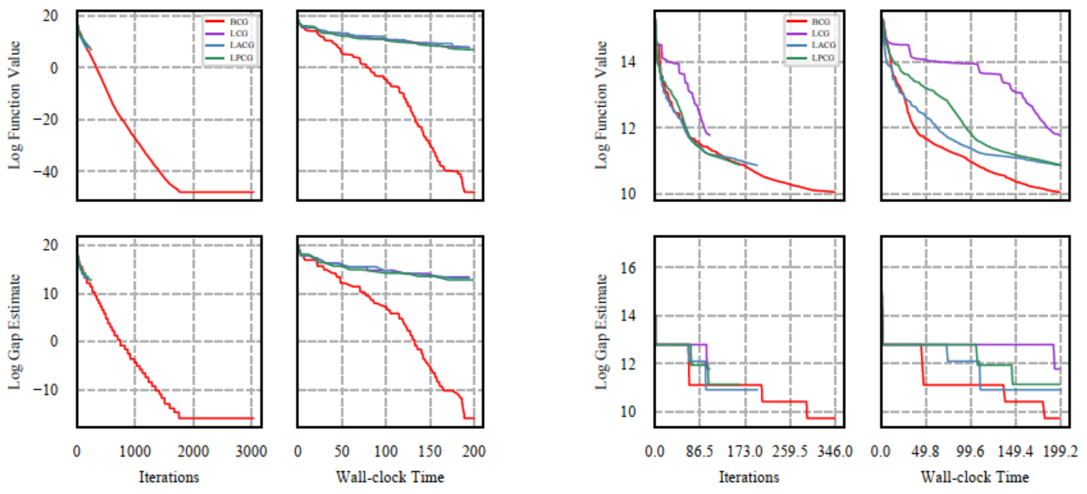

Figure 10. Comparison of BCG, LCG, ACG, and PCG. Left: Structured regression instance over the spanning tree polytope over the complete graph with 11 nodes demonstrating significant performance difference in improving the function value and closing the dual gap; BCG made 3031 iterations, LPsep_P was called 1501 times (almost always terminated early) and final solution is a convex combination of 232 vertices only. Right: Structured regression over the `disctom` polytope; BCG made 346 iterations, LPsep_P was called 71 times, and final solution is a convex combination of 39 vertices only. Observe that not only the function value decreases faster, but the gap estimate, too.

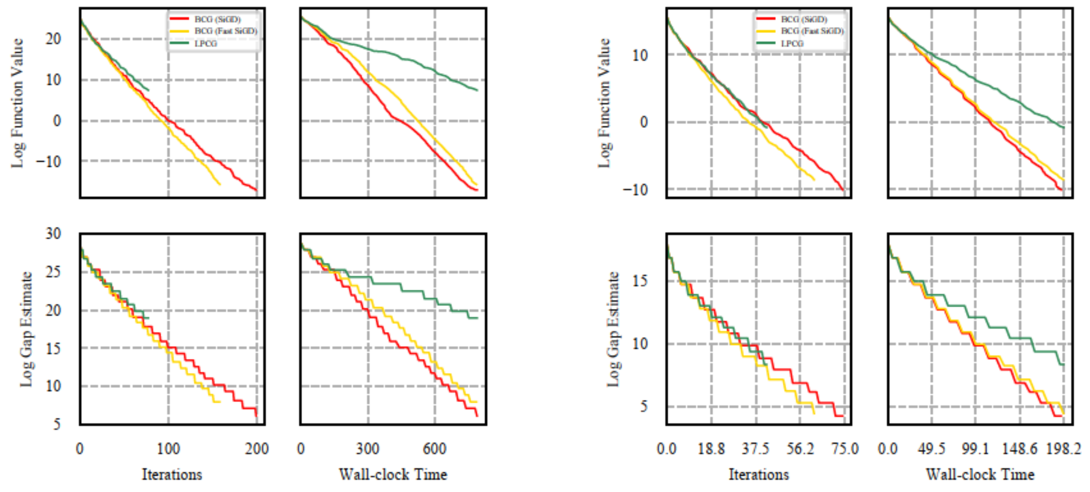

Figure 11. Comparison of BCG, accelerated BCG and LPCG. Left: On a medium size video co-localization instance (`netgen_12b`). Right: On a larger video co-localization instance (`road_paths_01_DC_a`). Here the accelerated version is (slightly) better in iterations but not in wall-clock time though. These findings are representative of all our other tests.

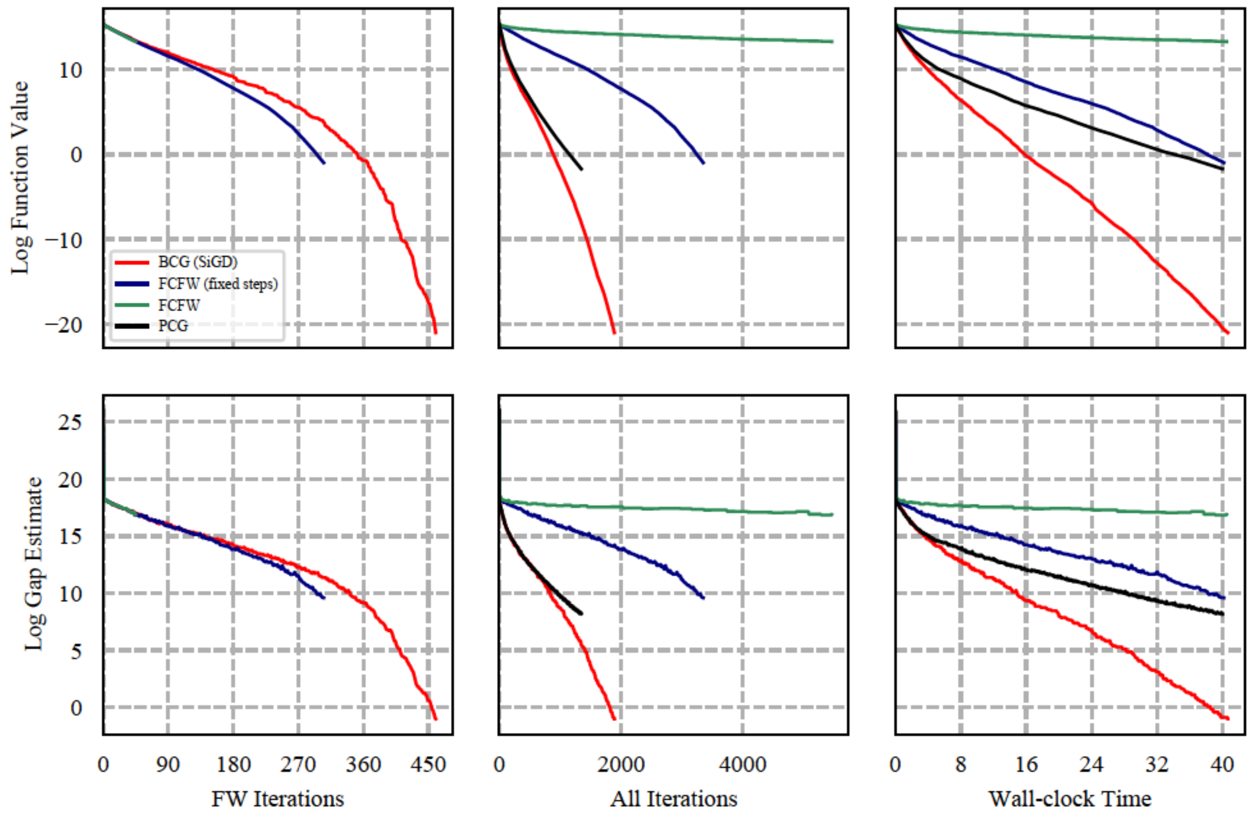

Figure 12. Comparison to FCFW across FW iterations, (all) iterations, and wall-clock time on a Lasso instance. Test run with 40s time limit. In this test we explicitly computed the dual gap of BCG, rather than using the estimate Φ_t.