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*

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

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 $f_S(\alpha) \coloneqq f(A\alpha)$, where A is the linear operator defined via $A\alpha \coloneqq \sum_{i=1}^k \alpha_i v_i$. We need to show

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

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

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

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

$$\|\alpha-\beta\|_2^2 = \|\gamma_+ - \gamma_-\|_2^2 = \|\gamma_+\|_2^2 + \|\gamma_-\|_2^2. \tag{17}$$

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

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

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

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

By (18) and (19) we conclude that $\|A\alpha - A\beta\|^2 \le kD^2\|\alpha-\beta\|_2^2/4$, 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*

$$C \le 2C^\Delta.$$

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 $S \coloneqq S_w \cup S_z$ and $|S| \le 2\dim P$.

Reusing the notation from the proof of Lemma A.1, let $k \coloneqq |S|$ 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 $L_{f_S} \le C^\Delta$ and $\|\beta - \alpha\| \le \sqrt{2}$:

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

showing that $C \le 2C^\Delta$ 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 $f(y) \ge f(x)$ to $f(y) \ge f(x) - \varepsilon$, where $\varepsilon \coloneqq \min\{\frac{\max\{p,0\}}{2}, \varepsilon_0\}$ with $\varepsilon_0 \in \mathbb{R}$ some upper bound on the accepted potential increase in objec-

tive 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 Δ^k for $t = 1, \dots, T$

1: $x_0 = e_1$
2: **for** $t = 0$ **to** $T - 1$ **do**
3: $S_t \leftarrow \{i \colon x_{t,i} > 0\}$
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: $w_t \leftarrow \operatorname{argmin}_{1 \le i \le k} \nabla f(x_t)_i$
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: $\eta = \max\{\gamma : x_t - \gamma d \ge 0\}$ {ratio test}
10: $y = x_t - \eta d$
11: **if** $f(x_t) \ge f(y)$ **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

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

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

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

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 $W \ge 2/\sqrt{k}$, so that the geometric strong convexity of f is $\mu \ge 4\alpha/k$. The diameter of Δ^k is $D = \sqrt{2}$, and it is easily seen that $C^\Delta = L_f$ and $C \le L_f D^2/2 = L_f$.

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 $h_t \coloneqq f(x_t) - f(x^*)$.

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

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

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

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

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

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

By Lemma 4.1, we have

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

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

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

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

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

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 co-localization, 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

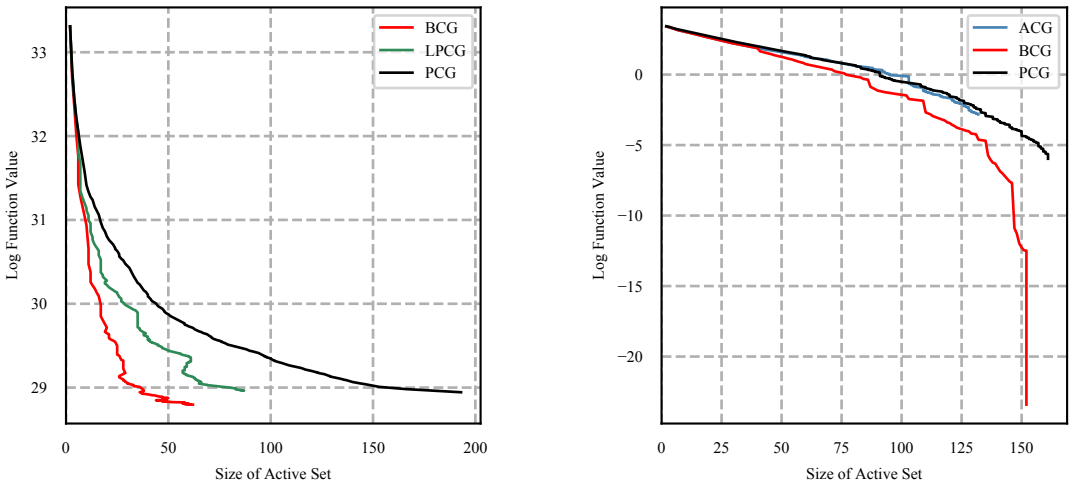

Figure 2. Comparison of ACG, PCG and LPCG against BCG in function value and size of the active set. Left: Video Co-Localization instance. Right: Sparse signal recovery.

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 $S = \{X \succeq 0 : \operatorname{Tr}[X] = 1\} \subseteq \mathbb{R}^{n\times n}$, 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:

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

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.

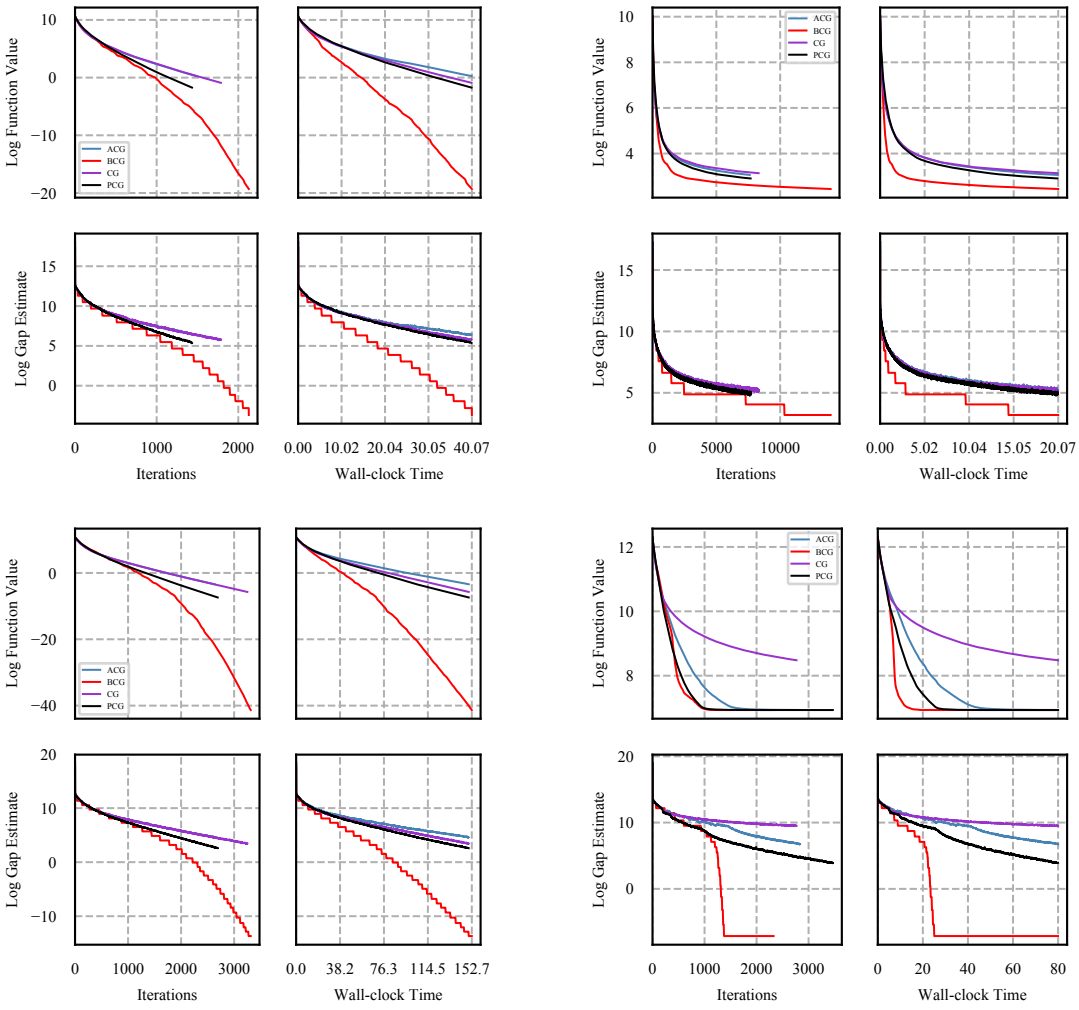

Figure 3. Comparison of BCG, ACG, PCG and CG on Lasso instances. Upper-left: A is a 400×2000 matrix with 100 non-zeros. BCG made 2130 iterations, calling the LP oracle 477 times, with the final solution being a convex combination of 462 vertices giving the sparsity. Upper-right: A is a 200×200 matrix 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.

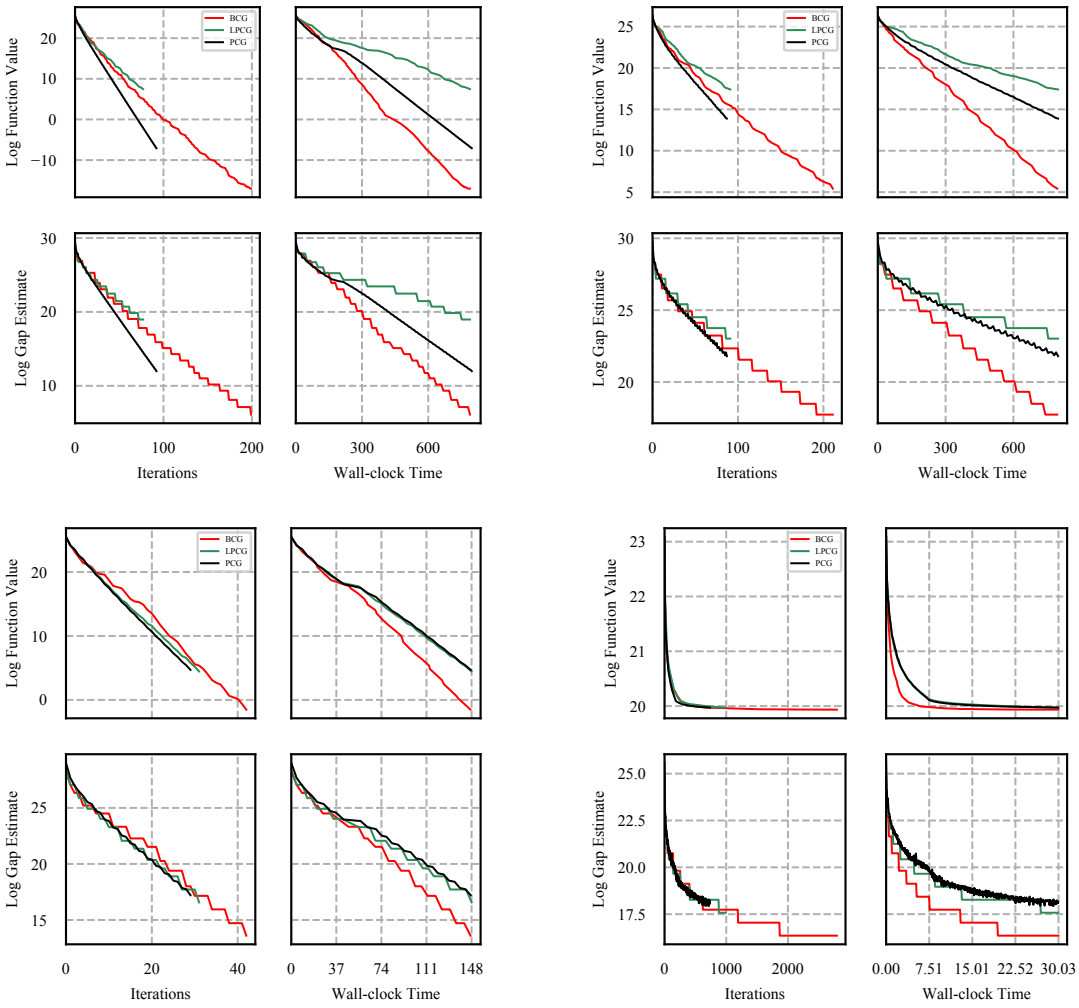

Figure 4. Comparison of PCG, Lazy PCG, and BCG on video co-localization instances. Upper-Left: `netgen_12b` for a 3000-vertex graph. BCG made 202 iterations, called LPsep$_P$ 56 times and the final solution is a convex combination of 56 vertices. Upper-Right: `netgen_12b` over a 5000-vertex graph. BCG 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.

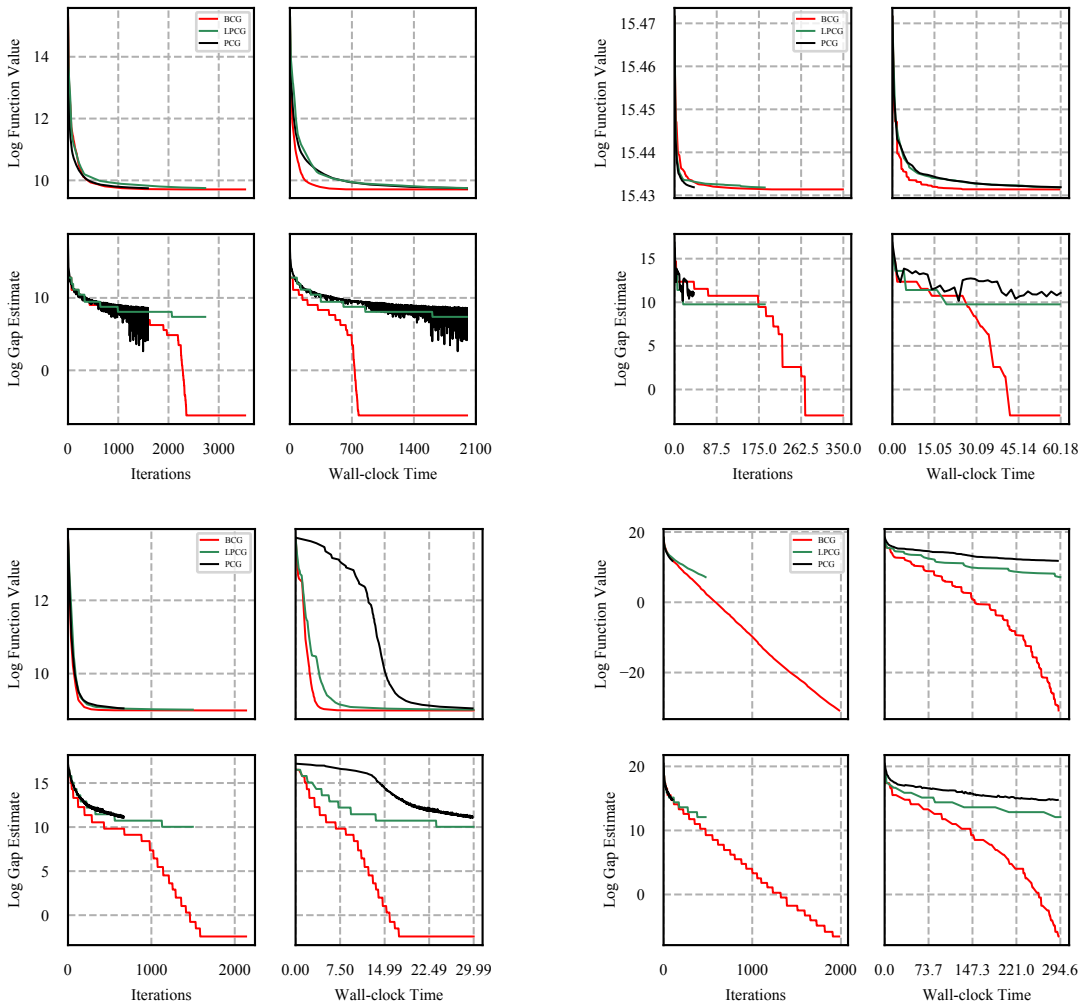

Figure 5. Comparison of BCG, LPCG and PCG on structured regression instances. Upper-Left: Over the `disctom` polytope. BCG made 3526 iterations with 1410 LPsep_P calls and the final solution is a convex combination of 85 vertices. Upper-Right: Over a `maxcut` polytope over a graph with 28 vertices. BCG 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.

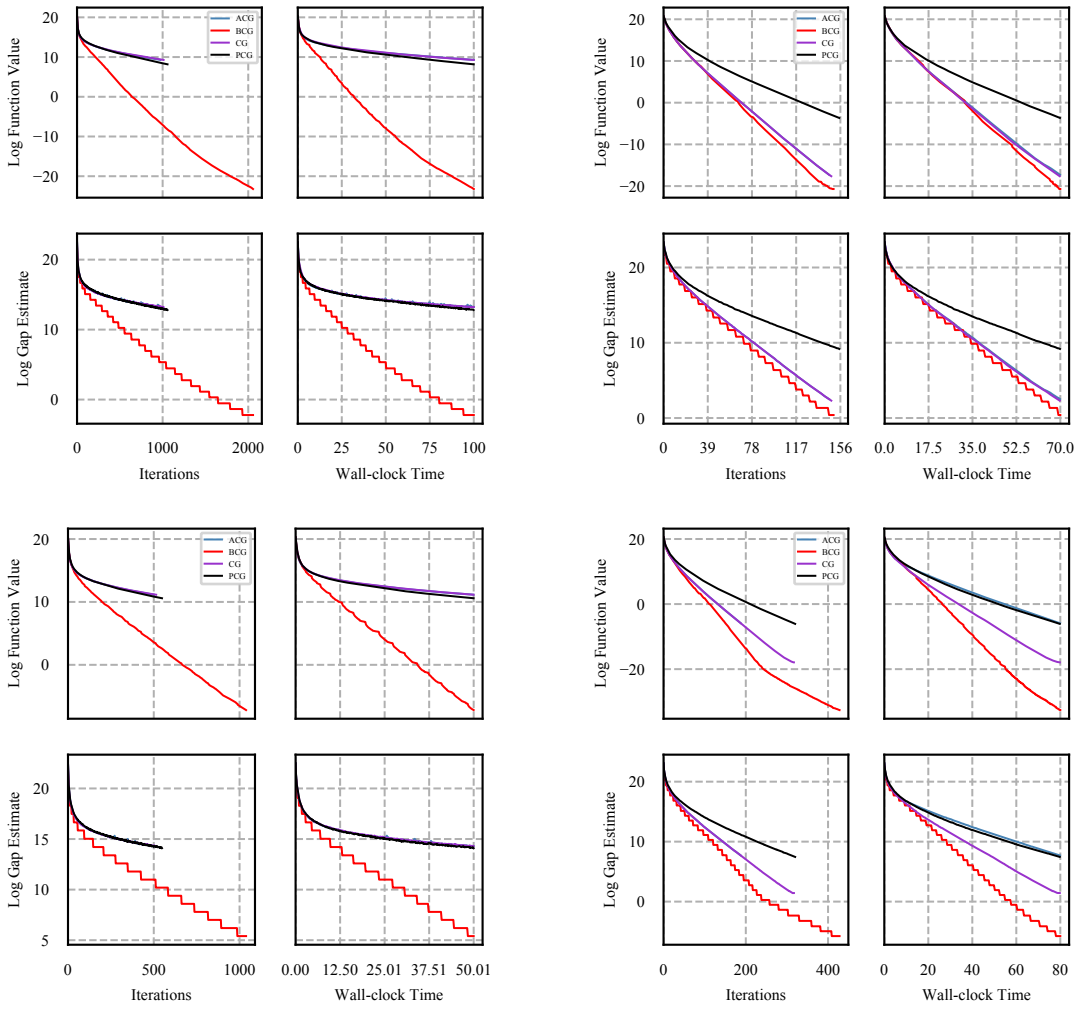

Figure 6. Comparison of BCG, ACG, PCG and CG over the Birkhoff polytope. Upper-Left: Dimension 50. BCG made 2057 iterations with 524 LPsep_P calls and the final solution is a convex combination of 524 vertices. Upper-Right: Dimension 100. BCG made 151 iterations with 134 LPsep_P calls and the final solution 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.

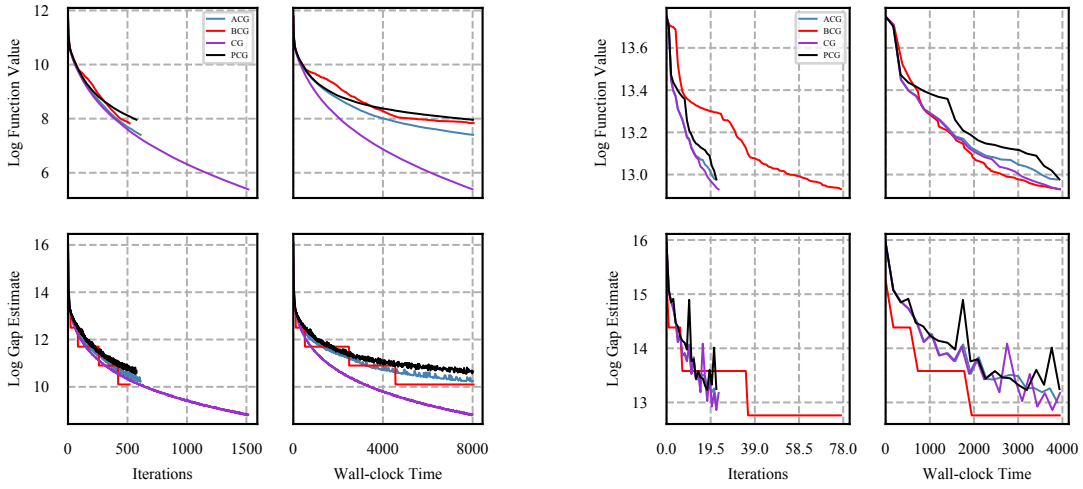

Figure 7. Comparison of BCG, ACG, PCG and CG on matrix completion instances over the spectrahedron. Upper-Left: Over the movie lens 100k data set. BCG made 519 iterations with 346 LPsep_P calls and the final solution is a convex combination of 333 vertices. Upper-Right: Over a subset of movie lens 1m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

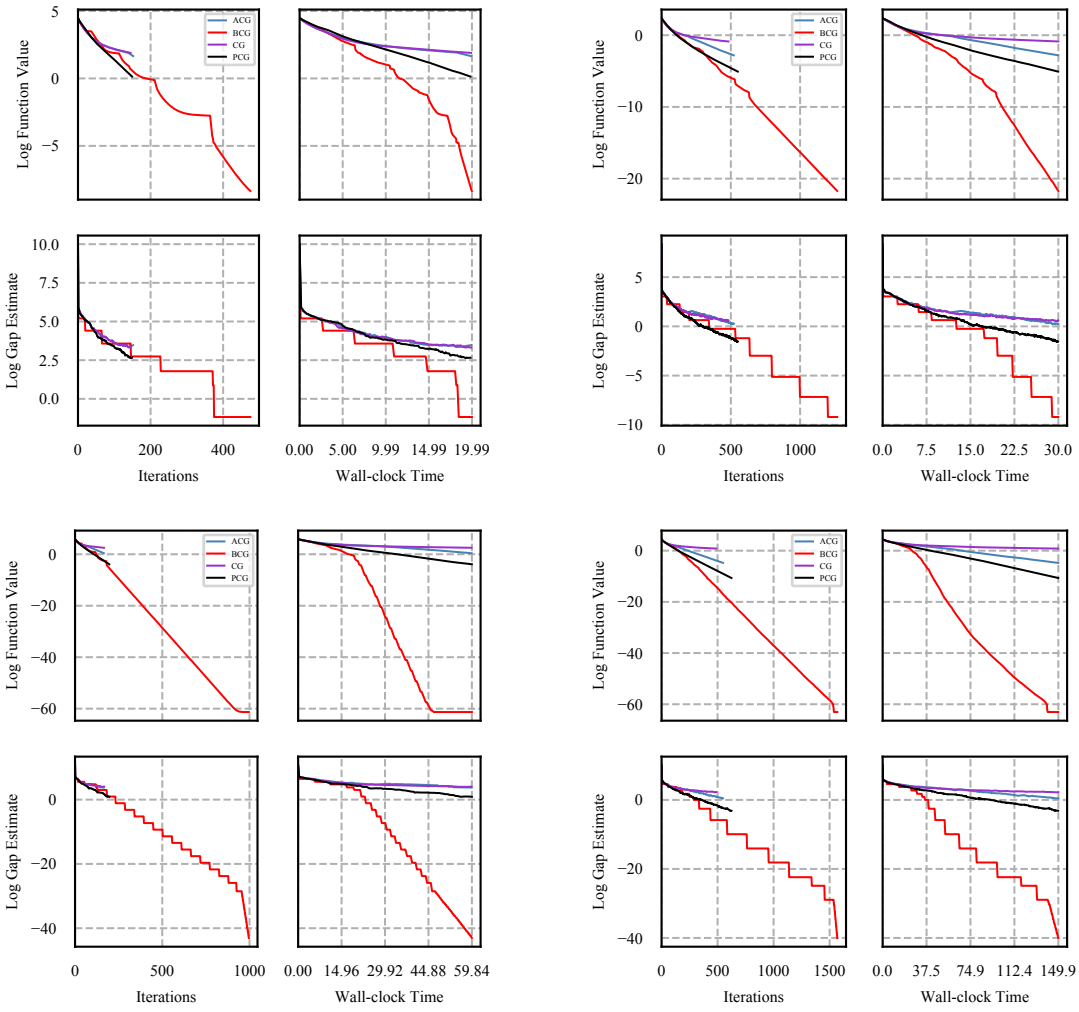

Figure 8. Comparison of BCG, ACG, PCG and CG on a sparse signal recovery problem. Upper-Left: Dimension is 5000×1000 density is 0.1. BCG made 547 iterations with 102 LPsep_P calls and the final solution is a convex combination of 102 vertices. Upper-Right: Dimension is 1000×3000 density is 0.05. BCG 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.

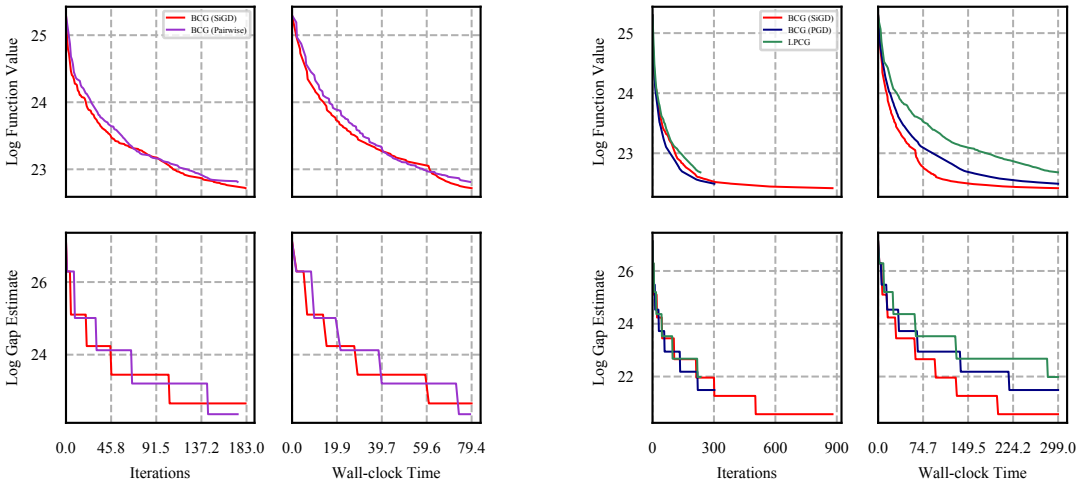

Figure 9. Comparison of BCG variants on a small video co-localization instance (instance `netgen_10a`). Left: BCG with vanilla Frank–Wolfe steps (red) and with pairwise steps (purple). Performance is essentially equivalent here which matches our observations on other instances. Right: Comparison of oracle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

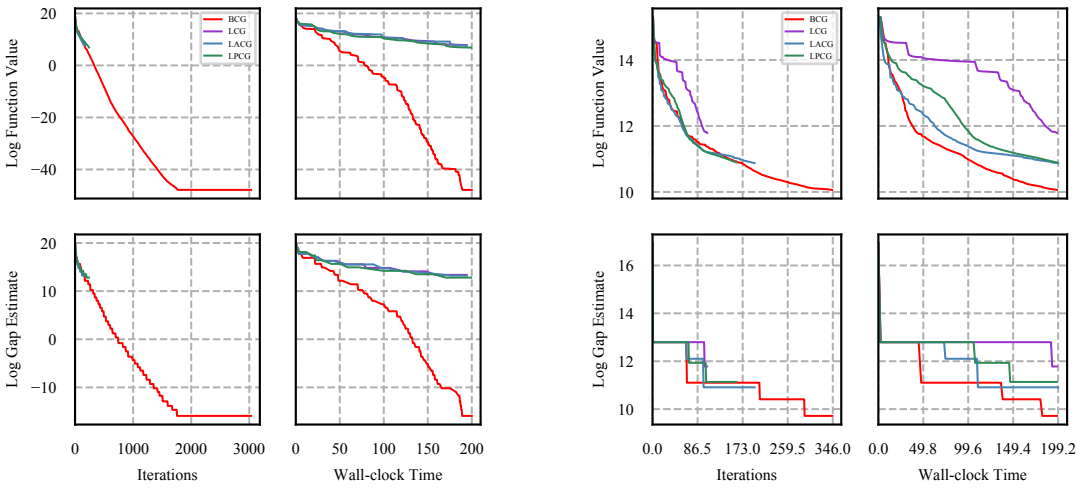

Figure 10. Comparison of BCG, LCG, ACG, and PCG. Left: Structured regression instance over the spanning tree polytope over the complete graph with 11 nodes demonstrating significant performance difference in improving the function value and closing the dual gap; BCG made 3031 iterations, LPsep_P was 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.

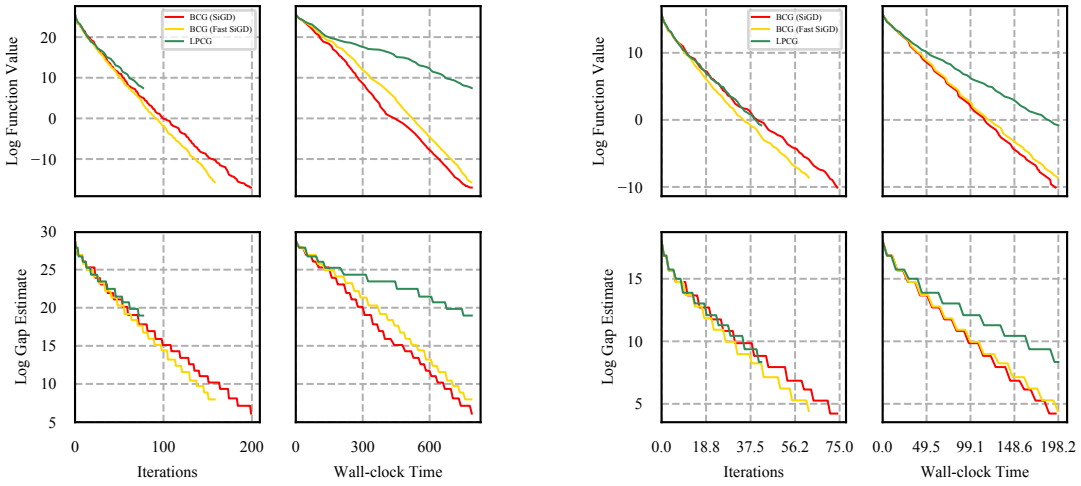

Figure 11. Comparison of BCG, accelerated BCG and LPCG. Left: On a medium size video co-localization instance (`netgen_12b`). Right: On a larger video co-localization instance (`road_paths_01_DC_a`). Here the accelerated version is (slightly) better in iterations but not in wall-clock time though. These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

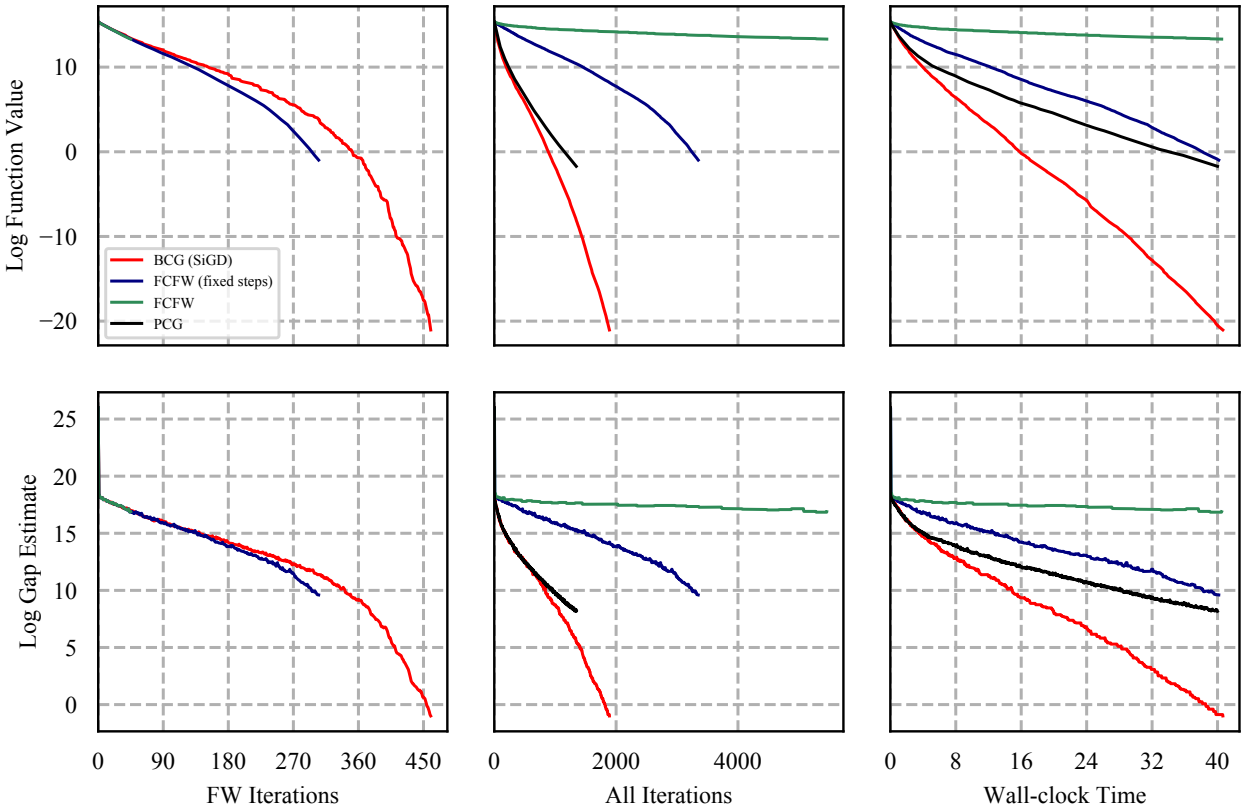

Figure 12. Comparison to FCFW across FW iterations, (all) iterations, and wall-clock time on a Lasso instance. Test run with 40s time limit. In this test we explicitly computed the dual gap of BCG, rather than using the estimate Φ_t.