UNIVERSITÀ DI PISA

Solutions for the Convex Quadratic Knapsack Problem Leveraging a Frank-Wolfe-based Algorithm

Calcolo Magico 2023/2024

Lorenzo Leuzzi

Nicola Pitzalis

Contents

0 Notation

In this report, we use bold letters for vectors $\boldsymbol{v}$ and matrices $\boldsymbol{A}$, and italic for constants c. When referring to sets we will use curly letters $\mathcal{X}$. Whenever a new symbol is introduced, we will explicitly state whether it denotes a vector, a constant, or a matrix, ensuring that the reader can easily understand the specific type of mathematical entity being referred to, despite the shared notation for vectors and constants.

Also, in order to conform to the usual mathematical literature, we will describe some thought processes as "straightforward", very "simple", "clear", or "immediate" just to hide the incredible strain of eyes and neural activity used to get to that very conclusion, whilst not exploiting the most powerful mathematical tool of "the trivial proof is left to the reader".

1 Problem

(P) is the convex quadratic nonseparable knapsack problem

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{x}^T \boldsymbol{Q} \boldsymbol{x} + \boldsymbol{q}^T \boldsymbol{x} \\ \text{subject to} \quad & \boldsymbol{a}^T \boldsymbol{x} \geq b,\ \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u} \end{aligned} \tag{P}$$

where $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{q}$, $\boldsymbol{0} \leq \boldsymbol{l} < \boldsymbol{u}$ and $\boldsymbol{a} > \boldsymbol{0}$ are fixed vectors of $\mathbb{R}^n$, b is a fixed positive real number, and $\boldsymbol{Q}$ is an $n \times n$ Positive Semidefinite matrix.

The aim of this report is to present a tailored solution algorithm falling under the category of conditional gradient methods, often referred to as *Frank-Wolfe* type methods. This approach necessitates the development of a customized solution strategy. Specifically, the linear programming tasks inherent within the algorithm must be tackled through an ad-hoc method that leverages the unique structural aspects of the problem at hand.

2 Frank-Wolfe

Frank-Wolfe methods are a class of iterative algorithms used for solving differentiable optimization problems with convex constraints. The algorithm starts from a feasible point and at each iteration, it linearizes the objective function around the current point. Then, it solves a linear program over the feasible set to find a new direction along which it moves to form the next point. Procedural steps of this method are shown in Algorithm 1, where the primary focus lies in solving the linear problem depicted in line **2**.

2.1 Linear Sub-problem

We define the linear problem presented at line **2** of Algorithm 1 as follows:

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{c}^T \boldsymbol{x} \\ \text{subject to} \quad & \boldsymbol{a}^T \boldsymbol{x} \geq b, \\ & l_i \leq x_i \leq u_i, \forall i \in \{1, \ldots, n\}. \end{aligned}$$

Algorithm 1 Frank-Wolfe Method

Require: $\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{l}, \boldsymbol{u}, b, f, \epsilon$
1: **while** $\langle \nabla f(\boldsymbol{x}), \boldsymbol{d} \rangle \geq -\epsilon$ **do**
2: $\quad \boldsymbol{s} \leftarrow \arg\min_{\boldsymbol{z}} \left\{ \langle \nabla f(\boldsymbol{x}), \boldsymbol{z} \rangle : \boldsymbol{a}^T \boldsymbol{z} \geq b,\ \boldsymbol{l} \leq \boldsymbol{z} \leq \boldsymbol{u} \right\}$
3: $\quad \boldsymbol{d} \leftarrow \boldsymbol{s} - \boldsymbol{x}$
4: $\quad \alpha \leftarrow \texttt{step}(f, \boldsymbol{x}, \boldsymbol{d}, 1, \epsilon)$
5: $\quad \boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha \boldsymbol{d}$
6: **end while**
7: **return** $\boldsymbol{x}$

To effectively solve the Linear Problem inherent in the Frank-Wolfe method, it is essential to take advantage of the structure of the feasible region $\mathcal{X}$, as demonstrated in Algorithm 2.

This particular problem is constrained by a box defined by the bounds $\boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}$. A fundamental property of such box constraints is that the minimum of a convex function over this region is always located at one of the vertices of the box. Given this insight, we adopt a strategy that selects, for each coordinate of the solution vector x, the corresponding element from either the lower bound vector $\boldsymbol{l}$ or the upper bound vector $\boldsymbol{u}$. The decision criterion for this selection is based on the sign of the gradient of the objective function for that dimension: if the gradient is positive, indicating an increase in the function value, the corresponding element from $\boldsymbol{l}$ is chosen as it will reduce the objective function. Conversely, if the gradient is negative, the element from $\boldsymbol{u}$ is selected. By systematically applying this reasoning to each dimension, we construct a candidate vertex that aligns with the descent direction of the objective function, thereby progressing towards the optimal solution within the confines of the box constraints. If this candidate also satisfies the additional constraint $\boldsymbol{a}^T \boldsymbol{x} \geq b$, then it represents the optimal solution $\boldsymbol{x}^*$.

Theorem 2.1 (Fundamental Theorem of Linear Programming)**.** Every feasible linear program that is bounded below has an optimal solution in a zero-dimensional face (a vertex) of the feasible polyhedron.

Because of the possibility of the minimum not being in any of the vertices of the hyperbox, from the presence of the additional constraint $\boldsymbol{a}^T \boldsymbol{x} \geq b$, we must provide a more flexible solution. Theorem 2.1 assures us that even with this added constraint, the minimum solution will still be located at a critical point of the feasible region. Specifically, it will either be at a vertex of the hyperbox or at a point where the hyperbox intersects with the inequality constraint. This intersection forms a polyhedron, ensuring that the search for the optimal solution, although potentially more complex, remains tractable by focusing on the vertices and edges of this polyhedron.

Also, by the structure of the linear inequality constraint

$$\boldsymbol{a}^T \boldsymbol{x} \geq b \equiv a_1 x_1 + \ldots + a_n x_n \geq b,$$

it is pretty straightforward to see that the modification of some of the components of $\boldsymbol{x}$ leads to the satisfaction of the constraint.

Observation 2.2. since the minimum must lie in one of the vertices of the polyhedron, we might as well reduce the constraint to an equality:

$$\boldsymbol{a}^T\boldsymbol{x} = b \equiv a_1x_1 + \ldots + a_nx_n = b.$$

and at this point it is clear that only one of the components of $\boldsymbol{x}$ will be different from its respective lower/upper bound, i.e. if the inequality constraint intersects the hyperbox in such a way that it reduces the feasible region (hence not considering the cases it intersects a vertex of the hyperbox), then $\forall i \in \{1, \ldots, n\} \setminus \{j\},\ x_i \in \{l_i, u_i\}$ and $l_j \leq x_j \leq u_j$.

So, now the non trivial task to address is to determine which components of $\boldsymbol{x}$ must be changed in order to achieve the optimal feasible solution.

To draw a comparison to our current problem, we can reference the well-known Knapsack Problem, which shares notable similarities with our situation. More specifically, our challenge aligns closely with one of its most recognized variants: the Continuous (or Fractional) Knapsack Problem, defined as:

$$\begin{aligned} \max_{\boldsymbol{x}} \quad & \boldsymbol{c}^T\boldsymbol{x} \\ \text{subject to} \quad & \boldsymbol{a}^T\boldsymbol{x} \leq b, \\ & 0 \leq x_i \leq 1, \forall i \in \{1, \ldots, n\}, \end{aligned}$$

where $\boldsymbol{a}, \boldsymbol{c} \in \mathbb{R}^n_+$.

We could rewrite our problem in order to match as close as possible the formulation of the Continuous Knapsack Problem seen above, in order to get a parallel understanding of the solution proposed to our problem:

$$\begin{aligned} \max_{x} \quad & (-\boldsymbol{c} + \boldsymbol{p})^T\boldsymbol{x} \\ \text{subject to} \quad & \boldsymbol{a}^T\boldsymbol{x} = b, \\ & l_i \leq x_i \leq u_i, \forall i \in \{1, \ldots, n\}, \end{aligned}$$

where $0 \leq l_i < u_i$ for $i = 1, \ldots, n$ and $\boldsymbol{p} \in \mathbb{R}^n_+$ is a vector such that $-\boldsymbol{c} + \boldsymbol{p} > \boldsymbol{0}$.

We transformed our minimisation problem into a maximisation one, by inverting the sign on the $\boldsymbol{c}$ term, and then added a positive constant to each of its component, in order to match the formulation of the Knapsack Problem for which $\boldsymbol{c} > \boldsymbol{0}$. This does not change the maximisation process, which is invariant of constant scaling. Regarding what happened to the inequality constraint, we refer to Observation 2.2.

Observation 2.3. We could also reformulate the constraint $\boldsymbol{a}^T\boldsymbol{x} \geq b$ as $\boldsymbol{a}^T\boldsymbol{x} \leq b$ to align seamlessly with the structure of the Continuous Knapsack Problem, focusing our interest on the scenarios where the constraint is met with equality. This reformulation is justified because, in solving the Continuous Knapsack Problem, the constraint typically achieves equality at the optimal solution. The only exception — when the total weight of all items is less than the threshold b — leads to the trivial solution of selecting all items. Therefore, this transformation does not alter the nature of the optimal solution sought within the problem's framework

By now we should be convinced that the problem we are facing is not very different from a Continuous Knapsack Problem, thus by following the strategy of the very known Greedy Algorithm, we build a nuanced solution which take cares of the main differences between the two tasks at hand. We will address later on, in section 2.2, the proof of correctness, but as far as intuition goes, the above formulation should suffice to understand the main reasoning behind the chosen implementation.

The proposed solution strategically optimizes the component selection for the solution vector $\boldsymbol{x}$, focusing on both the objective function's gradient and the imposed constraints. Components are prioritized based on the ratio:

$$\frac{c_i}{a_i},$$

where c_i represents the gradient of the objective function, $\nabla(\boldsymbol{c}^T\boldsymbol{x})$, with respect to the i-th component, and a_i signifies the coefficient's impact in the constraint $\boldsymbol{a}^T\boldsymbol{x} \geq b$. This ratio, reminiscent of the value-to-weight ratio in the Continuous Knapsack Problem, informs the efficient inclusion of each component, balancing objective function optimization with constraint satisfaction.

After sorting, indices are utilized to traverse the variables. More precisely, we bring every component to its upper bound, since it will be the minimum for negative gradients, and the (least) maximum for positive gradients. The objective is to achieve the least possible increment in the objective function while progressing into the feasible region. Following the adjustments, if the condition $\boldsymbol{a}^T\boldsymbol{x} > b$ persists, indicating that the current solution exceeds the threshold set by the inequality constraint, the algorithm proceeds to refine the solution further. It recalibrates the most recently modified variable, x_i, aligning it precisely with the intersection point where the boundary of the feasible box region and the inequality constraint's hyperplane converge.

This intersection is significant because it represents the exact point along the edge of the feasible region where $\boldsymbol{x}$ remains within the bounds of the constraint while contributing to the minimization of the objective function. Essentially, it is the optimal point on the feasible region's boundary for that specific variable. If the adjustment to $\boldsymbol{x}$ fails to place the point within the feasible region, the iterative process continues. It systematically progresses through the remaining dimensions of $\boldsymbol{x}$ following the previously specified order.

Algorithm 2 Linear Subproblem $\mathcal{LP}$

Require: $\boldsymbol{a}, \boldsymbol{c}, b$
for $i < n$ **do**
  if $c_i \geq 0$ **then**
    $x_i \leftarrow l_i$
  else
    $x_i \leftarrow u_i$
  end if
end for
if $\boldsymbol{a}^T\boldsymbol{x} \geq b$ **then**
  return $\boldsymbol{x}$
end if
$\mathcal{P} \leftarrow permutation\{1, 2, ..., n\}$ that orders the indices s.t. $i < j \Leftrightarrow \frac{c_i}{a_i} < \frac{c_j}{a_j}$
for $p \in \mathcal{P}$ **do**
  $x_p \leftarrow u_p$
  if $\boldsymbol{a}^T\boldsymbol{x} > b$ **then**
    $x_p = (b - \sum_{i \neq p} a_i x_i)/a_p$
    return $\boldsymbol{x}$
  end if
end for

2.2 Of course it's going to work! Is it?

We start by defining the Lagrangian $\mathcal{L}(\boldsymbol{x}; \boldsymbol{\lambda})$ for the given optimization problem. In order to facilitate the notation, we will introduce the variables $\lambda_0 \in \mathbb{R}_+$, $\boldsymbol{\lambda_l}, \boldsymbol{\lambda_u} \in \mathbb{R}^n_+$, which combined form the Lagrange multipliers altogether. This way we can reformulate the Lagrangian as $\mathcal{L}(\boldsymbol{x}; \lambda_0, \boldsymbol{\lambda_l}, \boldsymbol{\lambda_u})$. The Lagrangian combines the objective function $f(\boldsymbol{x})$ with the constraints of the problem, represented by the functions $g_i(\boldsymbol{x})$, and associates them with the previously mentioned Lagrange multipliers.

$$
\begin{aligned}
&\mathcal{L}(\boldsymbol{x}; \lambda_0, \boldsymbol{\lambda_l}, \boldsymbol{\lambda_u}) \\
&\qquad = f(\boldsymbol{x}) + \sum_{i \in \mathcal{I}} \lambda_i g_i(\boldsymbol{x}) \\
&\qquad = \boldsymbol{c}^T\boldsymbol{x} + \lambda_0(b - \boldsymbol{a}^T\boldsymbol{x}) \\
&\qquad\quad + \lambda_{l_1}(l_1 - x_1) + ... + \lambda_{l_n}(l_n - x_n) \\
&\qquad\quad + \lambda_{u_1}(-u_1 + x_1) + ... + \lambda_{u_n}(-u_n + x_n) \\
&\qquad = \boldsymbol{c}^T\boldsymbol{x} + \lambda_0(b - \boldsymbol{a}^T\boldsymbol{x}) + \boldsymbol{\lambda_l}^T(\boldsymbol{l} - \boldsymbol{x}) + \boldsymbol{\lambda_u}^T(\boldsymbol{-u} + \boldsymbol{x})
\end{aligned}
$$

This sets the stage for applying the Karush-Kuhn-Tucker (KKT) conditions, which are pivotal in determining the optimality of potential solutions. The KKT conditions serve as a bridge between feasible regions defined by linear constraints and the optimal solutions sought through linear objectives. For a linear problem, these conditions are not just necessary but also sufficient for optimality given the convexity of the problem.

$\exists \lambda_0 \in \mathbb{R}_+$, $\boldsymbol{\lambda_l}, \boldsymbol{\lambda_u} \in \mathbb{R}^n_+$ s.t:

$$\boldsymbol{a}^T\boldsymbol{x} \geq b,\ \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u} \tag{KKT-F}$$

$$\nabla_{\boldsymbol{x}}\mathcal{L}(\boldsymbol{x}; \lambda_0, \boldsymbol{\lambda_l}, \boldsymbol{\lambda_u}) = \boldsymbol{c} - \lambda_0\boldsymbol{a} - \boldsymbol{\lambda_l} + \boldsymbol{\lambda_u} = \boldsymbol{0} \tag{KKT-G}$$

$$\lambda_0(b - \boldsymbol{a}^T\boldsymbol{x}) + \boldsymbol{\lambda_l}^T(\boldsymbol{l} - \boldsymbol{x}) + \boldsymbol{\lambda_u}^T(-\boldsymbol{u} + \boldsymbol{x}) = 0 \tag{KKT-CS}$$

2.2.1 "Trivial" Case: on the corner of the box

Showing the correctness of the case in which the minimum lies on the vertex of the hyperbox is pretty straightforward. This situation arises when the point representing the minimum not only coincides with a vertex of the hyperbox but also satisfies the additional constraint, specifically $\boldsymbol{a}^T\boldsymbol{x} \geq b$, which implies either that the hyperplane intersects the box on the vertex or that the latter is completely incorporated in the area delimited by the constraint. In particular this scenario occurs when at line `9` of Algorithm 2 the $\boldsymbol{x}$ is returned.

The primal feasibility condition (KKT-F) states that all the constraints of the problem must be satisfied at the optimal point. In the context of a corner of the box, this implies that the optimal solution $\boldsymbol{x}$ lies within the bounds $\boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}$ and in particularly some components of $\boldsymbol{x}$ will be equal either to their respective lower bound or to their respective upper bound, i.e. $x_i \in \{l_i, x_i\}$. The check at line `8` also ensures the satisfaction of the other inequality constraint.

To find non-negative values of λ_0, $\boldsymbol{\lambda_l}$, and $\boldsymbol{\lambda_u}$ given that the minimum point is at a corner of the box constraint and the stationarity condition $\boldsymbol{c} + \lambda_0\boldsymbol{a} + \boldsymbol{\lambda_l} - \boldsymbol{\lambda_u} = \boldsymbol{0}$ holds, we need to consider the nature of the constraints at a corner of the box. Some components of $\boldsymbol{x}$ will be equal to their corresponding components in $\boldsymbol{l}$ and others will be equal to their corresponding components in $\boldsymbol{u}$.

The stationarity condition from the Karush-Kuhn-Tucker (KKT) optimality conditions for each component i is given by:

$$c_i - \lambda_0 a_i - \lambda_{l_i} + \lambda_{u_i} = 0.$$

This equation can be dissected component-wise, depending on the sign of c_i:

- For $c_i > 0$, we have $x_i = l_i$, and thus $\lambda_{u_i} = 0$. The resulting equation is: $\lambda_{l_i} = c_i - \lambda_0 a_i$. To ensure $\lambda_{l_i} > 0$, we require $\lambda_0 < \frac{c_i}{a_i}$.
- For $c_i < 0$, $x_i = u_i$, making $\lambda_{l_i} = 0$. The equation simplifies to: $\lambda_{u_i} = -c_i + \lambda_0 a_i$. Given $c_i < 0 \Leftrightarrow -c_i > 0$, we have that λ_{u_i} will be positive if λ_0 is positive.

Thus, λ_0 must be chosen to satisfy both conditions. If the constraint $\boldsymbol{a}^T\boldsymbol{x} \geq b$ is inactive ($\lambda_0 = 0$), we get $\lambda_{l_i} = c_i$ and $\lambda_{u_i} = -c_i$; remember that while it seems that we are assigning a negative value to λ_{u_i}, this is not the case because when $\lambda_{u_i} \neq 0$, c_i is a negative quantity. To conform to the KKT conditions, λ_0 must be positive. This can be ensured by taking the absolute value of c_i in the numerator, given that a_i is always positive. Therefore, we select

λ_0 as:

$$\lambda_0 < \min_i \left(\frac{|c_i|}{a_i} \right)$$

This guarantees that all components of λ_l and λ_u are non-negative, satisfying the gradient stationarity condition of the KKT requirements.

We said that for the complementary slackness (KKT-CS) to hold, each non-zero $(l_i - x_i)$ must be paired with a zero λ_{l_i}, and each non-zero $(-u_i + x_i)$ must be paired with a zero λ_{u_i}. This clearly makes both $\lambda_{l_i}(l_i - x_i)$ and $\lambda_{u_i}(-u_i + x_i)$ equal to 0 for all i. Regarding the other constraint $\boldsymbol{a}^T\boldsymbol{x} \geq b$, it could either be inactive but still satisfied, resulting in λ_0 being 0, or it could be active, leading to $b - \boldsymbol{a}^T\boldsymbol{x} = 0$. In either case, the product between the two factors is 0.

We have established the KKT conditions for the simple scenario where the minimum lies at a corner of the box constraint.

2.2.2 "Wish we could leave the proof to the reader" Case: not on the corner of the box

To demonstrate that the (KKT-F) condition is still applicable in this scenario, consider when we enter the **for loop** at line **12** of Algorithm 2. It's important to examine the reassignment of the $\boldsymbol{x}$ vector components. Given that $\boldsymbol{x}$ is sorted in ascending order based on $\frac{c_i}{a_i}$, the initial positions might contain components with a negative gradient. For these components, maintaining their value at the upper bound (i.e., $x_i = u_i$ for all components with negative gradients) is crucial, as initially set, because this ensures they are minimized as much as possible. Generally, the strategy involves setting as many components as possible to their upper bounds until reaching a point where one component causes the equation $\boldsymbol{a}^T\boldsymbol{x} = b$ to be surpassed. At this critical juncture, illustrated at line **14** of Algorithm 2, the component's value is adjusted to solve the linear system that represents the intersection between the hyperbox and the plane of the equality constraint. This way we prove the feasibility condition, since in the worst case every component will be assigned to its upper bound, still being inside the feasible region:

$$b \leq \boldsymbol{a}^T\boldsymbol{x} \leq \boldsymbol{a}^T\boldsymbol{u}.$$

Observation 2.4. In line with the outlined approach, we arrange a vector so that its initial $j-1$ components are assigned their respective upper bounds. The j-th component is determined by the point of intersection between the constraint hyperplane and the hyperbox. For the remaining components, from position $j+1$ to n, each is set to its corresponding lower bound.

$$\underbrace{\frac{c_1}{a_1} \quad \cdots}_{x_{1:j-1}=u_{1:j-1}}, \qquad \underbrace{\frac{c_j}{a_j}}_{x_j=(b-\sum_{i\neq j} a_i \bar{x}_i)/a_j}, \qquad \underbrace{\cdots \quad \frac{c_n}{a_n}}_{x_{j+1:n}=u_{j+1:n}}$$

To verify the Karush-Kuhn-Tucker Gradient Stationarity (KKT-G) condition, we examine

the following equation:

$$\boldsymbol{c} - \lambda_0 \boldsymbol{a} - \boldsymbol{\lambda_l} + \boldsymbol{\lambda_u} = \boldsymbol{0}.$$

Let's analyse the equation component-wise, but following the same ordering established at line **11** of Algorithm 2, meaning we will be ordering the constraints on the Lagrangian in such a way that they mimic the indices of the ordered $\boldsymbol{x}$.

For the j-th element, where both lower and upper bound constraints are inactive, the corresponding Lagrange multipliers are zero. This leads to:

$$c_j - \lambda_0 a_j = 0,$$

which yields:

$$\lambda_0 = \frac{c_j}{a_j}.$$

Let's observe that the value of λ_0 is surely positive by ordering of $\boldsymbol{x}$ (at line **11** of Algorithm 2); it is clear that if c_j were to be negative then the corresponding component would have been set to its upper bound $x_i = u_i$. Thus, we proceed to analyze the preceding components, denoted by the index p ($p \in \{1, \ldots, j-1\}$), and succeeding components, denoted by the index i ($i \in \{j+1, \ldots, n\}$), based on the index ordering previously established.

For all components before the j-th element (p), where $x_p = u_p$ (as set in line **13** of Algorithm 2), we have $\lambda_{l_p} = 0$, resulting in:

$$c_p - \frac{c_j}{a_j} a_p + \lambda_{u_p} = 0,$$

which simplifies to:

$$\lambda_{u_p} = -c_p + \frac{c_j}{a_j} a_p.$$

To ensure $\lambda_{u_p} > 0$, the ratio $\frac{c_j}{a_j}$ must exceed $\frac{c_p}{a_p}$, which clearly follows from Observation 2.4.

For all components after the j-th element (i), where $x_i = l_i$ (those are the ones that did not get a chance to be reassigned at line **13** of Algorithm 2, thus being set to their respective lower bound) we have $\lambda_{u_i} = 0$, hence:

$$\lambda_{l_i} = c_i - \frac{c_j}{a_j} a_i.$$

For λ_{l_i} to remain positive, $\frac{c_j}{a_j} < \frac{c_i}{a_i}$, which also follows from Observation 2.4.

This way we also proved the KKT-G condition, by showing the existence of $\lambda_0 \in \mathbb{R}_+, \boldsymbol{\lambda_l}, \boldsymbol{\lambda_u} \in \mathbb{R}^n_+$ that makes the point a stationary one.

Finally, to verify the (KKT-CS) condition, the reasoning parallels that of the simple case. Given that we are in a scenario where the point lies on the hyperplane $b - \boldsymbol{a}^T \boldsymbol{x} = 0$, the corresponding constraint is active and λ_0 is nonzero. We observe then that $\lambda_0(b - \boldsymbol{a}^T \boldsymbol{x}) = 0$. Similarly, for dimensions where either $x_i = l_i$ or the corresponding constraint is inactive, $\lambda_{li}(l_i - x_i)$ equals 0 for all $i \neq j$. The same holds true for the upper bound $x_i = u_i$ and λ_{ui}. In the j-th dimension, neither the lower nor the upper constraint are active, we have

$\lambda_{lj} = 0$ and $\lambda_{uj} = 0$. Everything sums up to 0 thus we have proven the complementary slack condition.

Apparently, the KKT conditions are satisfied by our solution, indicating that our solution is optimal. It works!

2.3 The OG algorithm

In the following, we are going to briefly touch on the workings of the Frank-Wolfe (FW) algorithm, and some of its most known variants.

We'll use the general formulation of a convex constrained optimization problem:

$$\min_{x \in \mathcal{X}} f(\boldsymbol{x}), \tag{R}$$

where $\mathcal{X}$ is a compact convex set, that in our case happens to be a polytope, due the linearity of our constraints. More precisely, we say that $\mathcal{A} \subseteq \mathbb{R}^d$ is a finite set of vectors — vertices of the polytope actually (we will refer to them as atoms) — and $\mathcal{X} = \text{conv}(\mathcal{A})$, where $\text{conv}(\mathcal{A})$ is the convex hull of the set $\mathcal{A}$. Given the construction of $\mathcal{X}$, we are sure that the set is convex and bounded.

The FW algorithm, as presented in Algorithm 1, works as follows: at the current iterate $\boldsymbol{x}_k$ the algorithm solves the linear problem $\boldsymbol{s}_k = \arg\min_{\boldsymbol{s} \in \mathcal{X}} \langle \nabla f(\boldsymbol{x}_k, \boldsymbol{s} \rangle$, where $\boldsymbol{s}_k$ is a vertex of the polytope $\mathcal{X}$ (an atom); the next iterate $\boldsymbol{x}_{k+1}$ is computed as a convex combination of the current iterate $\boldsymbol{x}_k$ and the selected vertex $\boldsymbol{s}_k$: $\boldsymbol{x}_{k+1} = (1 - \alpha_k \boldsymbol{x}_k + \alpha_k \boldsymbol{s}_k$, where α_k is the step size at iteration k.

This means, that $\boldsymbol{x}_k$ can be written as a convex combination of at most $k+1$ atoms in the set of previously discovered atoms $\mathcal{S}_k \subseteq \mathcal{A}$, i.e. $\boldsymbol{x}_k = \sum_{\boldsymbol{s} \in \mathcal{S}_k} \beta_{\boldsymbol{s}_k} \boldsymbol{s}$, where $\beta_{\boldsymbol{s}_k} > 0$ is the non-zero weight associated to the atom $\boldsymbol{s}$ at iteration k.

From this standpoint, it is quite easy to see where the algorithm starts to struggle, i.e. when the optimal solution lies at the relative boundary of $\mathcal{X}$, more precisely when it lies on a low-dimensional face of the polytope. The reason for this case being the worst situation is that the algorithm will start zig-zagging between the vertices defining the face containing the solution $\boldsymbol{x}^*$. In fact, as we will see next in more detail, the FW algorithm, under some assumptions, yields linear convergence when the optimal solution lies in the relative interior of $\mathcal{X}$, and worsens whenever it gets close to the relative boundary of $\mathcal{X}$, breaking to sublinear convergence whenever the optimal point lies on a low-dimensional face of the polytope.

2.3.1 Convergence analysis

Under L-smoothness and vanilla Frank-Wolfe algorithm converges sublinearly, i.e. it reaches the ϵ-optimal solution in $\mathcal{O}(1/\epsilon)$. In fact, the $1/k$ rate is tight for a large class of functions [1].

We'll briefly recall the duality gap's definition and state the theorem regarding the primal-dual convergence next.

Definition 2.5. For any constrained convex optimization problem of the form (R), and a feasible point $\boldsymbol{x} \in \mathcal{X}$, we define the duality gap as:

$$g(\boldsymbol{x}) := \max_{\boldsymbol{y} \in \mathcal{X}} \langle \nabla f(\boldsymbol{x}), \boldsymbol{x} - \boldsymbol{y} \rangle . \tag{1}$$

Now, given the convexity of f, it follows that $f(\boldsymbol{y}) \geq f(\boldsymbol{z}) + \nabla f(\boldsymbol{z})^T(\boldsymbol{y} - \boldsymbol{z}) \quad \forall \boldsymbol{z}, \boldsymbol{y} \in \mathcal{X}$, i.e. the linear approximation $f(\boldsymbol{z}) + \nabla f(\boldsymbol{z})^T(\boldsymbol{y} - \boldsymbol{z})$ always lies below the graph of the function f. From this point, we obtain:

$$\begin{aligned} f(\boldsymbol{y}) - f(\boldsymbol{z}) &\geq \nabla f(\boldsymbol{z})^T(\boldsymbol{y} - \boldsymbol{z}) \\ f(\boldsymbol{x}) - f(\boldsymbol{x}^*) &\leq \langle \nabla f(\boldsymbol{x}), \boldsymbol{x} - \boldsymbol{x}^* \rangle \leq \max_{\boldsymbol{y} \in \mathcal{X}} \langle \nabla f(\boldsymbol{x}), \boldsymbol{x} - \boldsymbol{y} \rangle \end{aligned}$$

where in the second line, first we took $\boldsymbol{y} \leftarrow \boldsymbol{x}$ and $\boldsymbol{z} \leftarrow \boldsymbol{x}^*$ and rearranged signs. So, the first inequality holds because of convexity, and the second holds because of maximality.

This clearly shows that the primal gap $f(\boldsymbol{x}) - f(\boldsymbol{x}^*)$ is indeed bounded by the dual gap in equation 1. Since we usually don't know the optimal value of the function beforehand, the dual gap is extremely helpful as both stopping criterion and bound on the convergence analysis.

Theorem 2.6 (Primal-Dual Convergence)**.** If Algorithm 1 is run for $K \geq 2$ iterations, then the algorithm has an iterate $\boldsymbol{x}_{\hat{k}}$, $1 \leq \hat{k} \leq K$, with duality gap bounded by

$$g(\boldsymbol{x}_{\hat{k}}) \leq \frac{2\gamma C_f}{K+2}(1+\delta),$$

where C_f is the curvature constant of the function f, $\gamma = \frac{27}{8} = 3.375$, and $\delta \geq 0$ is the accuracy to which the linear subproblems are solved.

For a proof of Theorem 2.6 we refer to [2].

Notice that the only assumption needed here is that the curvature constant C_f of the function is bounded. This means the function cannot curve infinitely fast. A weaker assumption is that the gradient of the function is Lipschitz continuous over the feasible region C, which implies that C_f is bounded because the gradient cannot vary arbitrarily fast. In our case, since the function is quadratic, the gradient is Lipschitz continuous, satisfying this condition.

Stronger results are obtained when both the function f and the feasible region $\mathcal{X}$ is μ-strongly-convex, reaching a convergence rate of $1/k^2$ [3]. Although very interesting, our feasible region is not μ-strongly-convex, and for that reason we won't expand our discussion further.

2.4 Frank Wolfe variants

2.4.1 Away-Step Frank Wolfe

In order to mitigate the zig-zagging problem of the vanilla Frank-Wolfe algorithm, we use a variation of the latter, namely Away-Step Frank-Wolfe, which adds the possibility to move away from an active atom in $\mathcal{S}_k$. This, as we'll be motivated later, is sufficient to make the algorithm linear convergent for strongly convex functions and, eventually, also for non-strongly convex ones.

Algorithm 3 Frank-Wolfe Away-Step Method

Require: $\boldsymbol{x}_0, \boldsymbol{a}, \boldsymbol{l}, \boldsymbol{u}, b, f, \epsilon$
1: $\mathcal{S} \leftarrow \{\boldsymbol{x}_0\}$, $\beta_{\boldsymbol{x}_0} \leftarrow 1$ and $\boldsymbol{x} \leftarrow \boldsymbol{x}_0$
2: **while** $\langle \nabla f(\boldsymbol{x}), \boldsymbol{d} \rangle \geq -\epsilon$ **do**
3: $\quad \boldsymbol{s} \leftarrow \arg\min_{\boldsymbol{z}} \left\{ \langle \nabla f(\boldsymbol{x}), \boldsymbol{z} \rangle : \boldsymbol{a}^T \boldsymbol{z} \geq b,\ \boldsymbol{l} \leq \boldsymbol{z} \leq \boldsymbol{u} \right\}$
4: $\quad \boldsymbol{d}^{FW} \leftarrow \boldsymbol{s} - \boldsymbol{x}$
5: $\quad \boldsymbol{v} \leftarrow \arg\max_{\boldsymbol{z}} \{ \langle \nabla f(\boldsymbol{x}), \boldsymbol{z} \rangle : \boldsymbol{z} \in \mathcal{S} \}$
6: $\quad \boldsymbol{d}^A = \boldsymbol{x} - \boldsymbol{v}$
7: $\quad$ **if** $\langle -\nabla f(\boldsymbol{x}), \boldsymbol{d}^{FW} \rangle \geq \langle -\nabla f(\boldsymbol{x}), \boldsymbol{d}^A \rangle$ **then**
8: $\quad\quad \boldsymbol{d} \leftarrow \boldsymbol{d}^{FW}$ and $\alpha_{\max} \leftarrow 1$
9: $\quad$ **else**
10: $\quad\quad \boldsymbol{d} \leftarrow \boldsymbol{d}^A$ and $\alpha_{\max} \leftarrow \frac{\beta_{\boldsymbol{v}}}{1-\beta_{\boldsymbol{v}}}$
11: $\quad$ **end if**
12: $\quad \alpha \leftarrow \texttt{step}(f, \boldsymbol{x}, \boldsymbol{d}, \alpha_{\max}, \epsilon)$
13: $\quad \boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha \boldsymbol{d}$
14: $\quad$ **if** $\boldsymbol{d} = \boldsymbol{d}^{FW}$ **then**
15: $\quad\quad \beta_{\boldsymbol{s}} \leftarrow (1-\alpha)\beta_{\boldsymbol{s}} + \alpha$ if $\boldsymbol{s} \in \mathcal{S}$ else $\beta_{\boldsymbol{s}} \leftarrow \alpha$
16: $\quad\quad \beta_{\boldsymbol{y}} \leftarrow (1-\alpha)\beta_{\boldsymbol{y}}$ for all $\boldsymbol{y} \in \mathcal{S} \setminus \{\boldsymbol{s}\}$
17: $\quad\quad \mathcal{S} \leftarrow \mathcal{S} \cup \{\boldsymbol{s}\}$ if $\alpha < \alpha_{\max}$ else $\mathcal{S} \leftarrow \{\boldsymbol{s}\}$
18: $\quad$ **else**
19: $\quad\quad \beta_{\boldsymbol{v}} \leftarrow (1+\alpha)\beta_{\boldsymbol{v}} - \alpha$
20: $\quad\quad \beta_{\boldsymbol{y}} \leftarrow (1+\alpha)\beta_{\boldsymbol{y}}$ for all $\boldsymbol{y} \in \mathcal{S} \setminus \{\boldsymbol{v}\}$
21: $\quad\quad \mathcal{S} \leftarrow \mathcal{S} \setminus \{\boldsymbol{v}\}$ if $\alpha > \alpha_{\max}$
22: $\quad$ **end if**
23: **end while**
24: **return** $\boldsymbol{x}$

The away direction $\boldsymbol{d}_k^A$ is defined in line **4** of Algorithm 3, by finding the atom $\boldsymbol{v}_k$ in $\mathcal{S}_k$ that maximizes the potential of descent given by $\boldsymbol{g}_k^A := \langle -\nabla f(\boldsymbol{x}_k, \boldsymbol{x}_k - \boldsymbol{v}_k \rangle$. Note that the search happens over the active set $\mathcal{S}_k$, which is typically small sized, thus easy to search. The maximum step-size $\alpha_{\max}$ as defined in line **10** of Algorithm 3 ensures that the new iterate $\boldsymbol{x}_k + \alpha_{\max} \boldsymbol{d}_k^A$ stays inside $\mathcal{X}$, more precisely it guarantees that the convex representation is maintained in order to stay inside $\text{conv}(\mathcal{S}_k) \subseteq \mathcal{X}$.

When $\alpha_k = \alpha_{\max}$, we have a drop step, i.e. the atom $\boldsymbol{v}_k$ is fully removed from the current active set of atoms $\mathcal{S}_k$, by setting its weight to zero. Weights are updated in lines **16** and **20**: for a FW step, we have $\mathcal{S}_{k+1} = \{\boldsymbol{s}_k\}$ if $\alpha_k = 1$ and $\mathcal{S}_{k+1} = \mathcal{S}_k \cup \{\boldsymbol{s}_k\}$ otherwise. Also, $\beta_{\boldsymbol{s}_k, k+1} := (1-\alpha_k)\beta_{\boldsymbol{s}_k, k} + \alpha_k$ and $\beta_{\boldsymbol{v}, k+1} := (1-\alpha_k)\beta_{\boldsymbol{v}, k}$ for $v \in \mathcal{S}_k \backslash \{\boldsymbol{s}_k\}$. For an away step, we have $\mathcal{S}_{k+1} = \mathcal{S}_k \backslash \{\boldsymbol{v}_k\}$ if $\alpha_k = \alpha_{\max}$ (drop step) and $\mathcal{S}_{k+1} = \mathcal{S}_k$ otherwise. Also, we have $\beta_{\boldsymbol{v}_k, k+1} := (1+\alpha_k)\beta_{\boldsymbol{v}_k, k} - \alpha_k$ and $\beta_{\boldsymbol{v}, k+1} := (1-\alpha_k)\beta_{\boldsymbol{v}, k}$ for $v \in \mathcal{S}_k \backslash \{\boldsymbol{v}_k\}$.

2.4.2 Pairwise Frank Wolfe

Another variant is the pairwise method, which focuses on redistributing the weight mass between two specific atoms in each iteration.

More specifically, the method presented in Algorithm 4 transfers weight from the "away" atom $\boldsymbol{v}$ to the Frank-Wolfe atom $\boldsymbol{s}$, while keeping all other weight coefficients β unchanged. In contrast to the classical Frank-Wolfe method, which reduces all active weights during each iteration, the pairwise variant only modifies the weights of the two selected atoms.

Algorithm 4 Frank-Wolfe Pairwise Method

Require: $\boldsymbol{x}_0, \boldsymbol{a}, \boldsymbol{l}, \boldsymbol{u}, b, f, \epsilon$
$\mathcal{S} \leftarrow \{\boldsymbol{x}_0\}$, $\beta_{\boldsymbol{x}_0} \leftarrow 1$, and $\boldsymbol{x} \leftarrow \boldsymbol{x}_0$
while $\langle \nabla f(\boldsymbol{x}), \boldsymbol{d} \rangle \geq -\epsilon$ **do**
  $\boldsymbol{s} \leftarrow \arg\min_{\boldsymbol{z}} \left\{ \langle \nabla f(\boldsymbol{x}), \boldsymbol{s} \rangle : \boldsymbol{a}^T \boldsymbol{z} \geq b,\ \boldsymbol{l} \leq \boldsymbol{z} \leq \boldsymbol{u} \right\}$
  $\boldsymbol{v} \leftarrow \arg\max_{\boldsymbol{z}} \left\{ \langle \nabla f(\boldsymbol{x}), \boldsymbol{z} \rangle : \boldsymbol{z} \in \mathcal{S} \right\}$
  $\boldsymbol{d} \leftarrow \boldsymbol{s} - \boldsymbol{v}$
  $\alpha \leftarrow \texttt{step}(f, \boldsymbol{x}, \boldsymbol{d}, \beta_{\boldsymbol{v}}, \epsilon)$
  $\boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha \boldsymbol{d}$
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{\boldsymbol{s}\}$
  $\beta_{\boldsymbol{s}} \leftarrow \beta_{\boldsymbol{s}} + \alpha$ and $\beta_{\boldsymbol{v}} \leftarrow \beta_{\boldsymbol{v}} - \alpha$
  $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\boldsymbol{v}\}$ if $\beta_{\boldsymbol{v}} = 0$
end while
return $\boldsymbol{x}$

2.4.3 Convergence analysis

We here state the result for the global linear convergence rate of both the Away-Step and Pairwise Frank-Wolfe (AFW) algorithms. We'll use $h_k := f(\boldsymbol{x}_k) - f(\boldsymbol{x}^*)$ to refer to the primal error at iteration k, and $\boldsymbol{g}_k$ as defined in 1 to refer the dual one.

Theorem 2.7. Suppose that f has L-Lipschitz gradient and is μ-strongly convex over $\mathcal{X} = \text{conv}(\mathcal{A})$. Let $D = \text{diam}(\mathcal{X})$ and $\delta = \text{PWidth}(\mathcal{A})$ as defined by (9). Then the suboptimality h_k of the iterates of the algorithms decrease geometrically at each step that is not a drop step nor a swap step (i.e. when $\alpha_k < \alpha_{\max}$, called a *good step*), that is

$$h_{k+1} \leq (1 - \rho)\, h_k, \quad \text{where} \quad \rho := \frac{\mu}{4L} \left(\frac{\delta}{D} \right)^2.$$

Let $\xi(k)$ be the number of "good steps" up to iteration k. We have $\xi(k) \geq k/2$ for AFW, and $\xi(k) \geq k/(3|\mathcal{A}|! + 1)$ for PFW. This yields a global linear convergence rate of $h_k \leq h_0 \exp(-\rho\, \xi(k))$. If $\mu = 0$ (general convex), then $h_k = O(1/\xi(k))$ instead.

However, this result bounds the primal error only, which we cannot easily track. The following result will prove sublinear convergence with rate $\mathcal{O}(1/\sqrt{k})$ on the dual error.

Theorem 2.8. Suppose that f has L-Lipschitz gradient over $\mathcal{X}$ with $D := \text{diam}(\mathcal{X})$. Then the dual gap g_k for the algorithms is upper bounded by the primal error h_k as follows:

$$g_k \leq h_k + LD^2/2 \text{ when } h_k > LD^2/2, \quad g_k \leq D\sqrt{2h_k L} \text{ otherwise.}$$

For a formal definition of the pyramidal width we refer to section 3.5.2 and to [4]. Up to now, we just need to know that, paired with the diameter of the feasible region, it represents some sort of conditioning number of the polytope.

Still, we need to generalize Theorem 2.7 to non-strongly convex functions, which is our most general case, given $\boldsymbol{Q}$ being positive semi-definite. For that we will refer to **Lemma 9** of [4], which states that we can generalize our previous result with objective $f(\boldsymbol{x}) := g(\boldsymbol{A}\boldsymbol{x}) + \langle \boldsymbol{b}, \boldsymbol{x} \rangle$, where g is μ_g-strongly convex w.r.t. the Euclidean norm over the domain $\boldsymbol{A}\mathcal{X}$ with strong convexity constant $\mu_g > 0$.

To satisfy the criterion, we recall the simple fact that the Euclidean norm squared $||\cdot||_2^2$ is indeed μ-strong-convex with $\mu = 2$. So we just need to find a matrix $\boldsymbol{A}$ such that g results in the Euclidean norm squared. This is actually easy to obtain, through the Eigenvalue Decomposition of the matrix $\boldsymbol{Q}$. In fact, since $\boldsymbol{Q}$ is symmetric positive-definite we can write $\boldsymbol{Q} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T$, obtaining $f(\boldsymbol{x}) = \boldsymbol{x}^T\boldsymbol{Q}\boldsymbol{x} + \boldsymbol{q}^T\boldsymbol{x} = \boldsymbol{x}^T\boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{x}$. Now, let us further decompose the diagonal matrix $\boldsymbol{\Lambda} = \sqrt{\boldsymbol{\Lambda}}\sqrt{\boldsymbol{\Lambda}}$ to get $\boldsymbol{x}^T\boldsymbol{U}\sqrt{\boldsymbol{\Lambda}}\sqrt{\boldsymbol{\Lambda}}\boldsymbol{U}^T\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{x}$, and finally set $\boldsymbol{z} = \sqrt{\boldsymbol{\Lambda}}\boldsymbol{U}\boldsymbol{x}$. We can finally write $f(\boldsymbol{x}) = g(\boldsymbol{z}) + \boldsymbol{b}^T\boldsymbol{x}$, where $g(\boldsymbol{z})$ is the squared Euclidean norm and $\boldsymbol{z} = \boldsymbol{A}\boldsymbol{x}$ with $\boldsymbol{A} = \sqrt{\boldsymbol{\Lambda}}\boldsymbol{U}$.

Given that our function can be generalized as such, we can generalize Theorem 2.7 substituting μ with:

$$\tilde{\mu} := \frac{1}{\left(2\theta^2\left(\|\boldsymbol{b}\|D + 3GD_{\mathcal{A}} + \frac{2}{\mu_g}(G^2+1)\right)\right)},$$

where θ is the Hoffman constant associated with the matrix $[\boldsymbol{A}; \boldsymbol{b}^T; \boldsymbol{B}]$ (where the rows of $\boldsymbol{B}$ are the linear inequality constraints defining the set $\mathcal{X}$), $G := \max_{\boldsymbol{x}\in\mathcal{X}} ||\nabla g(\boldsymbol{A}\boldsymbol{x})||$ is the maximal norm of the gradient of g over $\boldsymbol{A}\mathcal{X}$, D is the diamater of $\mathcal{X}$ and $D_{\mathcal{A}}$ is the diameter of $\boldsymbol{A}\mathcal{X}$. Again, we will not enter in any detail, but refer to [4].

To conclude our analysis, we emphasize that all the assumptions are satisfied and, for that reason, we expect linear convergence of the primal error and sublinear convergence of the dual error (with rate $\mathcal{O}(1/\sqrt{k})$.

3 Implementation

In the following we are going to motivate some of the design choices made for the algorithm's implementation.

3.1 Stopping criterion

As discussed before, we will use the duality gap defined as (1) as a stopping criterion, i.e. we will stop whenever $f(\boldsymbol{x}_k) - f(\boldsymbol{x}^*) \leq \epsilon$. To compute the gap we will keep the procedure shown in Algorithm 5.

Algorithm 5 Gap Update

1: lb $\leftarrow f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T(\boldsymbol{s} - \boldsymbol{x})$
2: **if** lb $>$ bestlb **then**
3: bestlb $\leftarrow$ lb
4: **end if**
5: gap $\leftarrow \frac{f(\boldsymbol{x}) - \text{bestlb}}{\max(|f(\boldsymbol{x})|, 1)}$

The idea is that we can keep the best lower bound in memory and use that to compute the gap. Also, notice that we compute the relative bound whenever the function value does not become too small in absolute value, i.e. less than one.

To see why the algorithm is correct, we use a symmetric argument to that of the duality gap upper bounding the primal gap. In fact,

$$f(\boldsymbol{x}^*) \geq f(\boldsymbol{x}_k) + \nabla f(\boldsymbol{x}_k)^T(\boldsymbol{x}^* - \boldsymbol{x}_k) \geq f(\boldsymbol{x}_k) + \min_{\boldsymbol{y} \in \mathcal{X}} \nabla f(\boldsymbol{x}_k)^T(\boldsymbol{y} - \boldsymbol{x}_k),$$

where the first inequality follows from convexity and the second holds for minimality.

To make it a little more clear, let $\boldsymbol{s}_k := \arg\min_{\boldsymbol{y} \in \mathcal{X}} \nabla f(\boldsymbol{x}_k)^T(\boldsymbol{y} - \boldsymbol{x}_k)$, then

$$f(\boldsymbol{x}_k) + \nabla f(\boldsymbol{x}_k)^T(\boldsymbol{s}_k - \boldsymbol{x}_k) \leq f(\boldsymbol{x}^*).$$

Since the inequality holds for all k, then the best lower bound:

$$\text{bestlb} := \max_{\boldsymbol{x} \in \mathcal{X}} \left\{ f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T(\boldsymbol{s}_k - \boldsymbol{x}) \right\} \leq f(\boldsymbol{x}^*)$$

also holds, and indeed justifies the choice of keeping the best lower bound in memory, as presented in lines **2-4** of Algorithm 5. Thus the gap, as computed in line **5** of Algorithm 5 is always non-negative, since $f(\boldsymbol{x}_k) \geq f(\boldsymbol{x}^*) \geq \max_{\boldsymbol{x} \in \mathcal{X}} \left\{ f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T(\boldsymbol{s}_k - \boldsymbol{x}) \right\}$.

To conclude our discussion we confront the dual gap with the one we compute in the algorithm and notice that

$$\frac{f(\boldsymbol{x}_k) - f(\boldsymbol{x}^*)}{\max\left(|f(\boldsymbol{x})|, 1\right)} \leq \frac{f(\boldsymbol{x}_k) - \text{bestlb}}{\max\left(|f(\boldsymbol{x})|, 1\right)} =: \text{gap}_k,$$

which implies that

$$f(\boldsymbol{x}_k) - f(\boldsymbol{x}^*) \leq \omega \cdot \text{gap}_k, \tag{2}$$

for some ω. And finally, when the rhs of (2) goes to 0, also the lhs does. This is sufficient to notice that setting the rhs to be smaller than a certain ϵ will cut it for us. We have a very simple stopping criterion!

3.2 Step size

Regarding the step size choice, we decided to implement the exact line search, as shown in Algorithm 6. To briefly recall the idea of the exact line search, the goal would be that of solving the unconstrained problem:

$$\alpha_k := \underset{\alpha \in [0, \alpha_{\max}]}{\arg\min}\ f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k),$$

where $\boldsymbol{d}_k := \boldsymbol{s}_k - \boldsymbol{x}_k$.

To find the minimum we just differentiate with respect to α_k, set it to 0, and find the value of α_k. We omit all the steps and state the result, i.e.

$$\alpha_k = -\frac{\boldsymbol{d}_k^T \nabla f(\boldsymbol{x})}{2 \cdot \boldsymbol{d}_k^T \boldsymbol{Q} \boldsymbol{d}_k}.$$

Notice that the numerator is precisely the dual gap defined in (1) — inverting the sign. Then we must take care of the special in case in which the denominator is zero, which can happen given $\boldsymbol{Q}$ being positive semi-definite. In fact, when the denominator is small enough (up to machine precision) we know the function is linear along $\boldsymbol{d}_k$ and thus we can take the maximum step-size.

Algorithm 6 Exact Line Search

1: $\boldsymbol{d} \leftarrow \boldsymbol{y} - \boldsymbol{x}$
2: **if** $2 \cdot \boldsymbol{d}^T \boldsymbol{Q} \boldsymbol{d} \leq$ 1e-16 **then**
3: $\quad \alpha \leftarrow \alpha_{\max}$ *(If $\boldsymbol{d}^T \boldsymbol{Q} \boldsymbol{d} = 0$, then f is linear along $\boldsymbol{d}$)*
4: **else**
5: $\quad \alpha \leftarrow \min\left(\frac{-\boldsymbol{g}^T \boldsymbol{d}}{\text{den}}, \alpha_{\max}\right)$
6: **end if**

3.3 Selecting the initial point

To select the initial point we actually run the linear optimizer $\mathcal{LP}(\boldsymbol{a}, \boldsymbol{0}, b)$. This is sufficient to take one of the vertices of the polytope as our first initial guess. That's it! Isn't that just great?

3.4 Problem generation

We need to generate the problem variables in such a way that they represent the problem described in (P).

In order to generate the matrix $\boldsymbol{Q} \in \mathbb{S}_+^n$, which has to be positive semi-definite, while also controlling its conditioning, we used the Matlab function `sprandsym(n, density, rc, kind)`, where `n` is the size of the matrix, `dens` is the desired matrix density —how many entries are non-zeros—, `rc` is the inverse of the condition number and `kind` is set to `1` to guarantee the condition number is precisely obtained.

To construct $\boldsymbol{q} \in \mathbb{R}^n$ we notice that by having $\boldsymbol{q} = -2\boldsymbol{Q}\boldsymbol{z}$, and setting the gradient of the function to zero, i.e. $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = 2\boldsymbol{Q}\boldsymbol{x} + \boldsymbol{q} \overset{\text{set}}{=} 0$, we get $= 2\boldsymbol{Q}\boldsymbol{x} - 2\boldsymbol{Q}\boldsymbol{z} \equiv \boldsymbol{x} = \boldsymbol{z}$,

implying that $\boldsymbol{z}$ is the solution to the unconstrained original problem. This way, $\boldsymbol{z}$ can be constructed in such a way that it regulates the amount of violated constraints, so that we can experiment different behaviours of the algorithms. To do so, we use `actv` as shown in Algorithm 7.

Last but not least, we need to generate $\boldsymbol{a} \in \mathbb{R}^n, \boldsymbol{b} \in \mathbb{R}$ so that the linear inequality constraint $\boldsymbol{a}^T\boldsymbol{x}^* \geq b$ is satisfied. For that to happen, we just take $\boldsymbol{a} \in (0, 1]$ and $\boldsymbol{x}_0 : \boldsymbol{l} < \boldsymbol{x}_0 < \boldsymbol{u}$. Then, we compute $b = \boldsymbol{a}^T\boldsymbol{x}_0$.

Algorithm 7 Problem Generation

Require: $n, actv, density, \lambda_{\max}, l_{min}, l_{max}, u_{min}, u_{max}$

$\boldsymbol{l} \leftarrow (l_1, l_2, \ldots, l_n)$, where $l_i \sim \mathcal{U}(l_{\min}, l_{\max})$

$\boldsymbol{u} \leftarrow (u_1, u_2, \ldots, u_n)$, where $u_i \sim \mathcal{U}(u_{\min}, u_{\max})$

$\boldsymbol{rc} \leftarrow (rc_1, rc_2, \ldots, rc_n)$, where $rc_i \sim \mathcal{U}(0, \lambda_{\max})$

$\boldsymbol{Q} \leftarrow \texttt{sprandsym}(n, density, \boldsymbol{rc})$

$$\boldsymbol{z} \leftarrow \begin{cases} z_i \sim \mathcal{U}(u_i, 2u_i) \text{ or } z_i \sim \mathcal{U}(0, l_i) & \text{for some elements according to } actv \\ z_i \sim \mathcal{U}(l_i, u_i) & \text{for the rest} \end{cases}$$

$\boldsymbol{q} \leftarrow -2\boldsymbol{Q}\boldsymbol{z}$

$\boldsymbol{a} \leftarrow (a_1, a_2, \ldots, a_n)$, where $a_i \sim \mathcal{U}(0, 1)$

$\boldsymbol{x} \leftarrow (x_1, x_2, \ldots, x_n)$, where $x_i = l_i + (u_i - l_i)r_i$, and $r_i \sim \mathcal{U}(0, 1)$

$\boldsymbol{b} \leftarrow \boldsymbol{a}^T\boldsymbol{x}$

return $\boldsymbol{l}, \boldsymbol{u}, \boldsymbol{Q}, \boldsymbol{q}, \boldsymbol{a}, b$

3.5 Side quest: Estimating the theoretical bounds

In this section, we are going to discuss how we can estimate the theoretical bounds on the proposed algorithms' errors. The estimates are going to be helpful when analyzing the difference between theory and practice altogether.

3.5.1 The OG Frank Wolfe dual gap's estimate

We know how the dual gap of the original FW algorithm is theoretically bounded (see Theorem 2.6). We will briefly discuss how the curvature constant is defined and how it is computed in practice.

Definition 3.1. The curvature constant C_f of the convex and differentiable function f: $\mathbb{R}^n \to \mathbb{R}$, w.r.t. a compact domain $\mathcal{D}$ is defined as:

$$C_f := \sup_{\substack{x,s\in\mathcal{D},\\ \gamma\in[0,1],\\ y=x+\gamma(s-x)}} \frac{2}{\gamma^2}\left(f(y) - f(x) - \langle y - x, \nabla f(x)\rangle\right).$$

To compute the curvature constant, we utilized the Matlab function `fmincon`, which solves generic non-linear optimization problems. The initial guesses for x, y are generated by selecting a random point within the lower and upper bounds, followed by projecting onto the linear inequality constraint if this is not satisfied. The initial value for γ is set to 0.5. The δ variable that appears in the theorem represents the precision to which the linear

subproblem is solved, which in our case is 0. Thus, we finally obtain:

$$g(\boldsymbol{x}_{\hat{k}}) \leq \frac{2 \cdot \frac{27}{8} C_f}{K+2},$$

where K is the maximum number iterations. Solving for $K \leq \epsilon$ leads to:

$$K \geq \frac{27}{4} \cdot \frac{C_f}{\epsilon} - 2.$$

It is clear from this point on, that the greater the curvature, the higher the number of iterations required for the algorithm to converge.

3.5.2 AFW and PFW dual gaps' estimates

To estimate the theoretical upper bounds to the dual error on the proposed FW variants, we need a little bit more work. We are going to briefly define some necessary concepts and discuss how their implementation differs. To define the Pyramidal Width, we first need some other definitions.

Definition 3.2 (Directional Width)**.** The directional width of a set $\mathcal{A}$ w.r.t. a direction $\boldsymbol{r}$ is defined as $\text{dirW}(\mathcal{A}, \boldsymbol{r}) := \max_{\boldsymbol{s},\boldsymbol{v} \in \mathcal{A}} \left\langle \frac{\boldsymbol{r}}{||\boldsymbol{r}||}, \boldsymbol{s} - \boldsymbol{v} \right\rangle$. The width of $\mathcal{A}$ is the minimum directional width over all possible directions in its affine hull.

Definition 3.3 (Pyramidal Directional Width)**.** We define the pyramidal directional width of a set $\mathcal{A}$ w.r.t. a direction $\boldsymbol{r}$ and a base point $\boldsymbol{x} \in \mathcal{X}$ to be

$$\text{PdirW}(\mathcal{A}, \boldsymbol{r}, \boldsymbol{x}) := \min_{\mathcal{S} \in \mathcal{S}_{\boldsymbol{x}}} \text{dirW}(\mathcal{S} \cup \{\boldsymbol{s}(\mathcal{A}, \boldsymbol{r})\}, \boldsymbol{r}) = \min_{\mathcal{S} \in \mathcal{S}_{\boldsymbol{x}}} \max_{\boldsymbol{s} \in \mathcal{A}, \boldsymbol{v} \in \mathcal{S}} \left\langle \frac{\boldsymbol{r}}{||\boldsymbol{r}||}, \boldsymbol{s} - \boldsymbol{v} \right\rangle,$$

where $\mathcal{S} \in \mathcal{S}_{\boldsymbol{x}} := \{\mathcal{S} | \mathcal{S} \subseteq \mathcal{A} \text{ and } \boldsymbol{x} = \sum_i^{|\mathcal{S}|} \alpha_i x_i \text{ with } \alpha_i > 0 \text{ for } i \in \{1, \ldots, |\mathcal{S}|\}\}$, and $\boldsymbol{s}(\mathcal{A}, \boldsymbol{r}) := \arg\max_{\boldsymbol{v} \in \mathcal{A}} \langle \boldsymbol{r}, \boldsymbol{v} \rangle$.

Definition 3.4 (Pyramidal Width)**.** To define the pyramidal width of a set, we take the minimum over the cone of possible feasible directions $\boldsymbol{r}$ (to avoid the problem of zero width). A direction $\boldsymbol{r}$ is feasible for $\mathcal{A}$ from $\boldsymbol{x}$ if it points inwards $\text{conv}(\mathcal{A})$ (i.e. $\boldsymbol{r} \in \text{cone}(\mathcal{A} - \boldsymbol{x})$). We define the pyramidal width of a set $\mathcal{A}$ to be the smallest pyramidal width of all its faces, i.e.

$$\text{PWidth}(A) := \min_{\substack{\mathcal{K} \in \text{faces}(\text{conv}(\mathcal{A})) \\ \boldsymbol{x} \in \mathcal{K} \\ \boldsymbol{r} \in \text{cone}(\mathcal{K} - \boldsymbol{x}) \setminus \{\boldsymbol{0}\}}} \text{PdirW}(\mathcal{K} \cap \mathcal{A}, \boldsymbol{r}, \boldsymbol{x}).$$

We'll make a short observation that will come in handy when implementing the function to approximate the pyramidal width.

Observation 3.5. Notice that, using Definition 3.3, we can rewrite the pyramidal directional width of $\mathcal{A}$ w.r.t. a direction $\boldsymbol{r}$ and a base point $\boldsymbol{x} \in \mathcal{X}$ as:

$$\text{PdirW}(\mathcal{A}, \boldsymbol{r}, \boldsymbol{x}) = \min_{\mathcal{S} \in \mathcal{S}_{\boldsymbol{x}}} \max_{\boldsymbol{s} \in \mathcal{A}} \left\langle \frac{\boldsymbol{r}}{||\boldsymbol{r}||}, \boldsymbol{s} \right\rangle - \min_{\boldsymbol{v} \in \mathcal{S}} \left\langle \frac{\boldsymbol{r}}{||\boldsymbol{r}||}, \boldsymbol{v} \right\rangle. \tag{3}$$

Also notice, that we need to talk about approximation of the pyramidal width, since the exact computation can become easily intractable as the dimension of the problem increases. In fact, we do not know how the set of vertices of the polytope $\mathcal{A}$ is composed, and to find out we would use some algorithm of vertex enumeration. Though, the number of vertices of a cut hyper box is in $\mathcal{O}(2^n)$, where n is the size of the problem, which makes the computation hard to be tackled in practical implementation.

Given the numerous difficulties that arise with this task, we will simplify the pyramidal width computation with a very crude approximation, thus we will simplify the following:

$$\text{PWidth}(A) := \min_{\substack{\mathcal{K} \in \text{faces}(\text{conv}(\mathcal{A})) \\ \boldsymbol{x} \in \mathcal{K} \\ \boldsymbol{r} \in \text{cone}(\mathcal{K} - \boldsymbol{x}) \setminus \{\boldsymbol{0}\}}} \min_{\mathcal{K} \cap \mathcal{A} \in \mathcal{S}_{\boldsymbol{x}}} \left\{ \max_{\boldsymbol{s} \in \mathcal{A}} \left\langle \frac{\boldsymbol{r}}{||\boldsymbol{r}||}, \boldsymbol{s} \right\rangle - \min_{\boldsymbol{v} \in \mathcal{K} \cap \mathcal{A}} \left\langle \frac{\boldsymbol{r}}{||\boldsymbol{r}||}, \boldsymbol{v} \right\rangle \right\},$$

approximating as:

$$\text{PWidth}(A) \approx \min_{\boldsymbol{r} \in \tilde{\boldsymbol{F}}_{\mathcal{X}}(\boldsymbol{x})} \left\{ \max_{\boldsymbol{x} \in \mathcal{V}} \left\langle \frac{\boldsymbol{r}}{||\boldsymbol{r}||}, \boldsymbol{x} \right\rangle - \min_{\boldsymbol{x} \in \mathcal{V}} \left\langle \frac{\boldsymbol{r}}{||\boldsymbol{r}||}, \boldsymbol{x} \right\rangle \right\},$$

where $\mathcal{V}$ is the set of sampled vertices of the polytope and $\tilde{\boldsymbol{F}}_{\mathcal{X}}(\boldsymbol{x})$ is the sampled set of feasible directions. Indeed, we know that the vertices of the polytope are the solutions to the linear subproblem $\mathcal{LP}$ through the feasible directions. Thus, we just need to sample a direction $\boldsymbol{r}$ and find the vertex through that direction.

3.5.3 The fun stuff: estimating the Hoffman constant

We will start by defining the Hoffman constant that we are going to estimate.

Definition 3.6. Given a point $\boldsymbol{x} \in \mathbb{R}^n$, the distance from $\boldsymbol{x}$ to a non-empty polyhedron $\mathcal{P}_{\boldsymbol{A}}(\boldsymbol{b}) := \{\boldsymbol{z} \in \mathbb{R}^n : \boldsymbol{A}\boldsymbol{z} \leq \boldsymbol{b}\}$ can be bounded above in terms of the residual vector $(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})_+ := \max(0, \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})$. More precisely, for $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ there exists a Hoffman constant $H(\boldsymbol{A})$ that depends only on $\boldsymbol{A}$ such that for all $\boldsymbol{b} \in \boldsymbol{A}(\mathbb{R}^n) + \mathbb{R}^m_+$ and all $\boldsymbol{x} \in \mathbb{R}^n$:

$$\text{dist}(\boldsymbol{x}, \mathcal{P}_{\boldsymbol{A}}(\boldsymbol{b})) \leq H(\boldsymbol{A}) \cdot ||(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})_+||, \tag{4}$$

where $\boldsymbol{A}(\mathbb{R}^n) := \{\boldsymbol{A}\boldsymbol{z} : \boldsymbol{z} \in \mathbb{R}^n\}$ and $\text{dist}(\boldsymbol{x}, \mathcal{P}_{\boldsymbol{A}}(\boldsymbol{b})) := \min\{||\boldsymbol{x} - \boldsymbol{z}|| : \boldsymbol{z} \in \mathcal{P}_{\boldsymbol{A}}(\boldsymbol{b})\}$.

We recall that our problem is a quite simple one, since we just have a single inequality constraint and n box constraints. For this reason, we can bound the distance from $\boldsymbol{x}$ to $\mathcal{P}_{\boldsymbol{A}}(\boldsymbol{b})$ by the sum of the distance to satisfy both the box constraints and the inequality

constraint. So, we will rewrite (4) in a slightly different manner:

$$\text{dist}(\boldsymbol{x}, \mathcal{P}_{\boldsymbol{A}}(\boldsymbol{b})) \leq H(\boldsymbol{A}) \cdot ||r(\boldsymbol{x})||,$$

where $r(\boldsymbol{x})$ is the positive residual defined as $r(\boldsymbol{x}) := ||r_{\text{box}}|| + r_{\text{ineq}}$, and $r_{\text{box}} := \max(\boldsymbol{l} - \boldsymbol{x}, 0) + \max(\boldsymbol{x} - \boldsymbol{u}, 0)$, $r_{\text{ineq}} := \max(\boldsymbol{b} - \boldsymbol{a}^T\boldsymbol{x}, 0)$. We also define the distance $\text{dist}(\boldsymbol{x}, \mathcal{P}_{\boldsymbol{A}}(\boldsymbol{b})) := \sqrt{d_{\text{box}}^2 + d_{\text{ineq}}^2}$, where $d_{\text{box}} = ||r_{\text{box}}||$ since projecting $\boldsymbol{x}$ onto the box involves adjusting each coordinate independently, and the Euclidean distance is the norm of these adjustments; and $d_{\text{ineq}} = \frac{\max(\boldsymbol{b} - \boldsymbol{a}^T\boldsymbol{x}, 0)}{||\boldsymbol{a}||}$ is the perpendicular distance from $\boldsymbol{x}$ to the hyperplane $\boldsymbol{a}^T\boldsymbol{x} = \boldsymbol{b}$. To find H then:

$$H = \frac{\text{dist}(\boldsymbol{x}, \mathcal{P}_{\boldsymbol{A}}(\boldsymbol{b}))}{||r(\boldsymbol{x})||} = \frac{\sqrt{d_{\text{box}}^2 + d_{\text{ineq}}^2}}{\sqrt{r_{\text{box}}^2 + r_{\text{ineq}}^2}} = \frac{\sqrt{d_{\text{box}}^2 + \left(\frac{r_{\text{ineq}}}{||\boldsymbol{a}||}\right)^2}}{\sqrt{d_{\text{box}}^2 + r_{\text{ineq}}^2}}.$$

Now, let's consider the squared Hoffman constant:

$$H^2 = \frac{d_{\text{box}}^2 + \left(\frac{r_{\text{ineq}}}{||\boldsymbol{a}||}\right)^2}{d_{\text{box}}^2 + r_{\text{ineq}}^2} = \frac{d_{\text{box}}^2 + \frac{r_{\text{ineq}}^2}{||\boldsymbol{a}||^2}}{d_{\text{box}}^2 + r_{\text{ineq}}^2}.$$

Here, we will make the assumption that $r_{\text{ineq}} > 0$, i.e. we are assuming there is always a small residual on the inequality constraint. Although this is not alway true, we need to make such an assumption in order to have some sort of upper bound. In fact, we notice that when $r_{\text{ineq}} \to 0$, then $H \to 1$.

Given the assumption, we know divide both numerator and denominator by r_{ineq}^2:

$$H^2 = \frac{\left(\frac{d_{\text{box}}}{r_{\text{ineq}}}\right)^2 + \frac{1}{||\boldsymbol{a}||^2}}{\left(\frac{d_{\text{box}}}{r_{\text{ineq}}}\right)^2 + 1}.$$

Now, let $\gamma = \frac{d_{\text{box}}}{r_{\text{ineq}}}$, and let $H(\gamma)$ be a function on γ:

$$H^2(\gamma) = \frac{\gamma^2 + \frac{1}{||\boldsymbol{a}||^2}}{\gamma^2 + 1}.$$

Since we care about the maximum —the upper bound to the Hoffman constant– we will take the derivative of the function w.r.t. γ:

$$\frac{dH^2(\gamma)}{d\gamma} = \frac{2\gamma(1 - \frac{1}{||\boldsymbol{a}||^2})}{(\gamma^2 + 1)^2},$$

and set its derivative to zero. The derivative equates to zero, either when $\gamma = 0$ and $||\boldsymbol{a}|| = 1$. Since $||\boldsymbol{a}||$ is constant, we have the maximum of $H^2(\gamma)$ when $\gamma = 0$, but also when $\gamma \to \infty$. In fact, when $\gamma = 0$:

$$H^2(0) = \frac{1}{||\boldsymbol{a}||^2} \implies H(0) = \frac{1}{||\boldsymbol{a}||},$$

while, when $\gamma \to \infty$:

$$\lim_{\gamma\to\infty} H^2(\gamma) = \lim_{\gamma\to\infty} \frac{2\gamma(1 - \frac{1}{||\boldsymbol{a}||^2})}{(\gamma^2+1)^2} = 1 \implies \lim_{\gamma\to\infty} H(\gamma) = 1.$$

Thus we know that, when $\gamma = 0$, i.e. when the box constrained is satisfied but the inequality constraint is not, $H = \frac{1}{||\boldsymbol{a}||}$; whilst when $\gamma \to \infty$, i.e. the inequality constraint is very close to be satisfied (by assumption it cannot be strictly satisfied) but the box constraints are not, $H = 1$. We can finally bound the Hoffman constant as:

$$H_{\max} \leq \sqrt{1 + \frac{1}{||\boldsymbol{a}||^2}},$$

since we might expect this value to be between the extremes of γ. There we go: an easy estimate of the Hoffman constant!

4 Experiments

In this section, we present the results of our experiments aimed at evaluating the performance of the Frank-Wolfe algorithm and its variants (Away-Step Frank-Wolfe and Pairwise Frank-Wolfe) on the convex quadratic nonseparable knapsack problem. The experiments were designed to explore key characteristics of the algorithms and their behavior under different problem configurations. Specifically, we analyze the influence of the percentage of active constraints, sparsity, condition number of the matrix $\boldsymbol{Q}$, and problem size on both convergence and CPU time.

All experiments were conducted using various synthetically generated problem instances. For each experiment, we performed 100 trials and analyzed the geometric mean of the duality gap progression over these runs. Notice that we chose to use the geometric mean because, in logarithmic space, it becomes equivalent to the arithmetic mean on a linear scale. The different trials for each experiment were executed in parallel to improve computational efficiency. The various experiments focus on isolating and varying a single feature at a time, while keeping all other parameters fixed according to a set of base values shown in Table 1. Unless otherwise specified, these base values were applied across all experiments.

In the experiments, ϵ represents the tolerance for the stopping criterion, where the algorithm terminates once the duality gap is smaller than this threshold, ensuring that the solution is sufficiently accurate. A very small value of $\epsilon = 1 \times 10^{-16}$ was chosen to prioritize precision. The `max_iter` parameter, set to 1500, defines the maximum number of iterations allowed for the algorithm. This prevents the algorithm from running indefinitely in cases where convergence is slow or the stopping criterion is not met.

The bounds for the variable's components were set such that the lower bound $\boldsymbol{l}$ and upper bound $\boldsymbol{u}$ of the box constraints are fixed across all dimensions. Specifically, `l_min` = `l_max` = 0 and `u_min` = `u_max` = 1, meaning the variables are constrained within the interval $[\mathbf{0}, \mathbf{1}]$.

Parameter	Base Value
Problem size (`n`)	100
Active constraints (`actv`)	0.7
Matrix density (`dens`)	0.9
Reciprocal of Condition number (`rc`)	0.1

Table 1: Base case problem parameters used in the experiments.

4.1 Base Case Problem

We begin by comparing the convergence of the Frank-Wolfe variants on the base problem.

Initially, the vanilla Frank-Wolfe algorithm demonstrates rapid progress in reducing the duality gap, particularly in the first few hundred iterations, as shown in Figure 1 top-left. However, as expected, the convergence rate slows down significantly after this early phase. This behavior is typical for this algorithm, which exhibits sublinear convergence, especially when the solution approaches the boundary of the feasible region. The variance in performance across trials, represented by the spread of the dashed gray lines, remains relatively low, indicating consistent behavior across different problem instances with the base parameters.

The Away-Step variant shows a faster convergence (Figure 1 top-right), handling constraints more effectively by allowing steps away from active atoms, thus reducing the zig-zagging behavior seen in the classic algorithm. The variance is wider compared to the original version

The Pairwise variant also achieves faster and more consistent convergence (Figure 1 bottom-left) by redistributing the weight between two atoms in each iteration, avoiding the inefficiencies of classic Frank-Wolfe. By iteration 1500, the duality gap has decreased to around 10^{-14} in many trials, with some trials stopping early due to meeting the optimal condition. The broader variance across trials reflects the varying performance depending on the problem instance.

Comparing the algorithms (Figure 1 bottom-right), the Away-Step and Pairwise Frank-Wolfe variants improve over the classic Frank-Wolfe by mitigating zig-zagging and handling constraints more efficiently, resulting in linear convergence.

4.2 Problem Size

Next, we analyze how the algorithms behave as we vary the size of the problem. We begin with smaller matrices and increase the size gradually, starting from 50 and progressing through 100, 200, and 500. Even larger sizes will be explored in later experiments.

For all problem sizes, the Pairwise and Away-Step variants converge faster than the vanilla Frank-Wolfe, with Pairwise showing the steepest reduction in the duality gap. As the problem size increases, the performance difference between the algorithms appears to narrows within the first 1500 iterations.

In smaller problems, the Pairwise and Away-Step variants the linear convergence is clear,

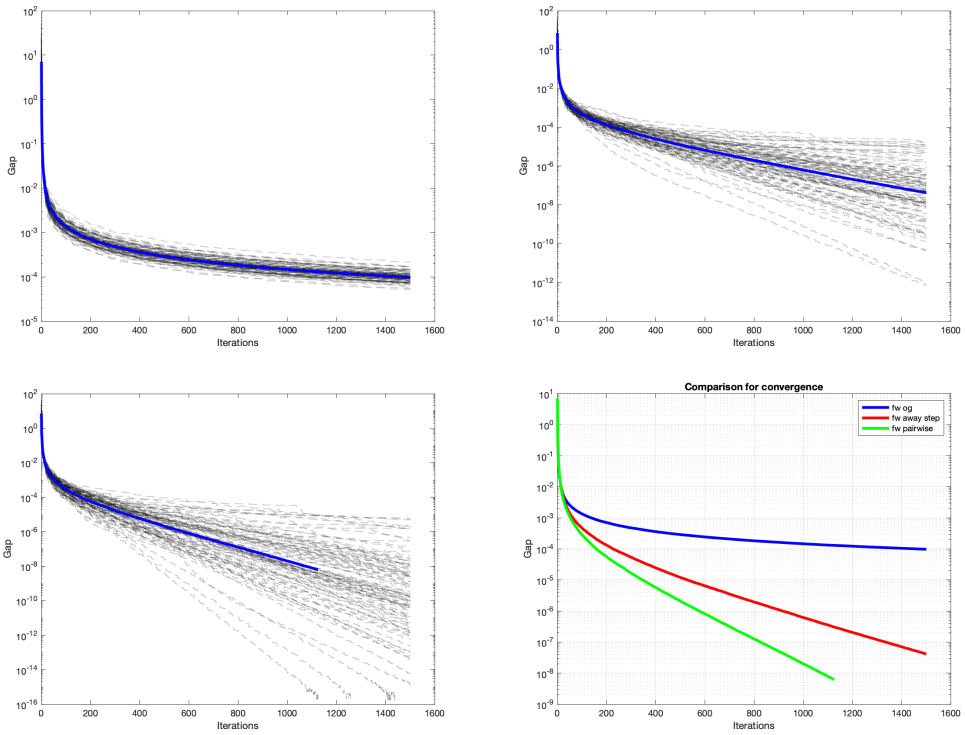

Figure 1: Convergence of the Frank-Wolfe algorithms for the base case problem. Top-left: vanilla Frank-Wolfe. Top-right: Away-Step Frank-Wolfe. Bottom-left: Pairwise Frank-Wolfe. Bottom-right: variants comparison.

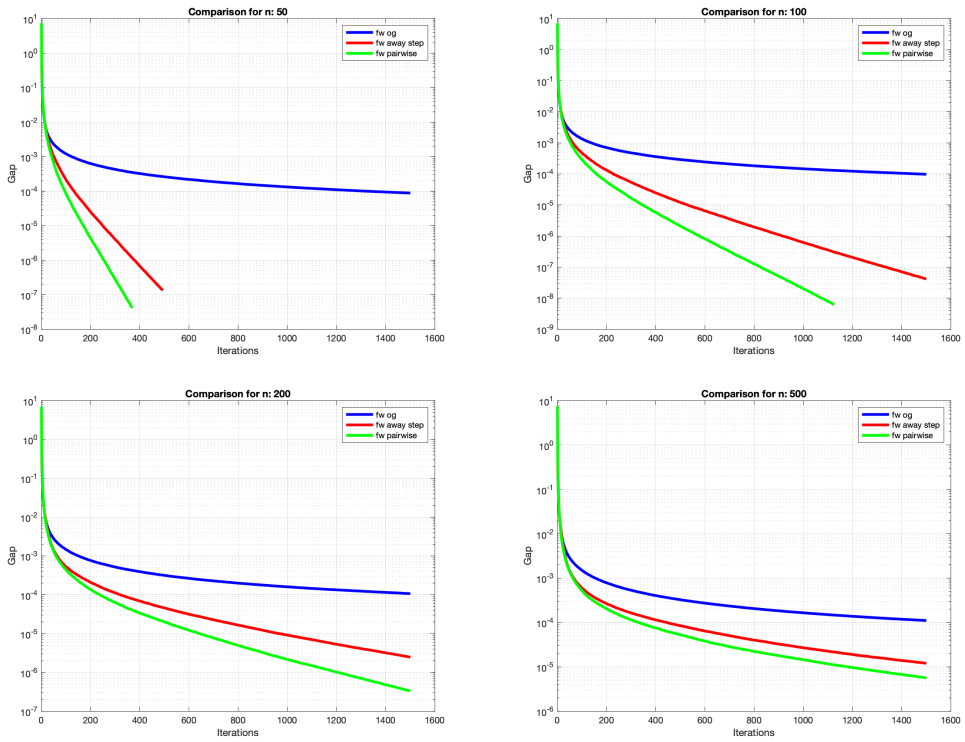

Figure 2: Convergence of the Frank-Wolfe algorithms as problem size increases.

likely due to the more well-behaved polytope structure. Although convergence slows down as the problem size increases, both of the advanced variants continue to significantly outperform the vanilla Frank-Wolfe. Notice that we do not actually expect strict linear convergence of the dual error from any of the variants, as seen in Theorem 2.8, rather we have a sublinear upper bound. In practice, however, in the vast majority of cases, linearity of convergence is obtained. A noticeable difference between the standard algorithm and its variants, is definitely the polytope conditioning that we have seen in Theorem 2.7. For this reason, we expect converge to be linear in case of well-conditioned polytopes and to be closer to sublinear whenever we have a ill-conditioned one.

As expected, the computation time increases with problem size for all algorithms as illustrated in Figure 3. The Pairwise variant takes the longest time, followed by the Away-Step variant, with the vanilla Frank-Wolfe being the fastest. While the more advanced versions take longer to complete the first 1500 iterations, they ultimately achieve better solutions compared to the OG version. This is consistent with the fact that the more advanced variants perform additional computational steps per iteration managing the active set. This becomes more pronounced as the problem size grows. However, we observe that all algorithms scale similarly with the size.

4.3 Active Constraints

Now, let's analyze the effect of varying the percentage of active constraints while keeping all other parameters fixed at their base values. We will consider the following cases: 10%,

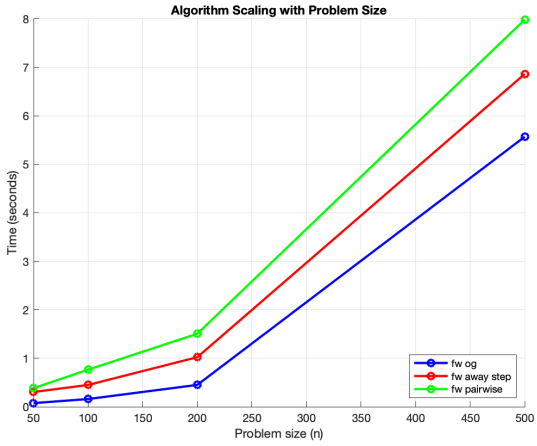

Figure 3: CPU time of the Frank-Wolfe algorithms for the first 1500 iterations as problem size increases.

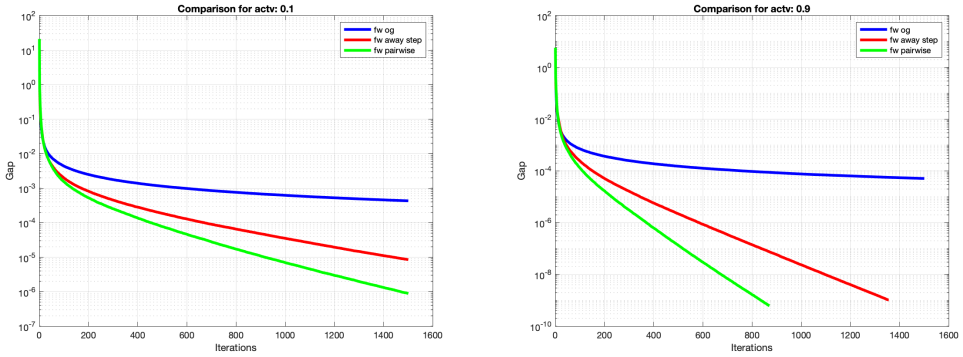

Figure 4: Convergence of the Frank-Wolfe algorithms varying the number of active constraints.

30%, 50%, 70%, and 90% of the constraints being active.

The convergence and computation times do not vary significantly between similar cases, but the difference becomes clear, when comparing the convergence on the extremes of the range, specifically 10% and 90% active constraints in Figure 4. With fewer active constraints, the solution has more freedom to move within the feasible region, making it harder for the algorithm to find the optimal solution quickly and resulting in slower convergence. In contrast, a large number of active constraints restricts the solution to a smaller, lower-dimensional space, reducing the search space and enabling faster convergence by guiding the algorithm towards the optimal solution. Because of that, as the number of active constraints increases, the global linear convergence of the Pairwise and Away-Step algorithms becomes more pronounced. However, even with fewer active constraints, the algorithms maintain linear convergence, though likely with a slower convergence rate.

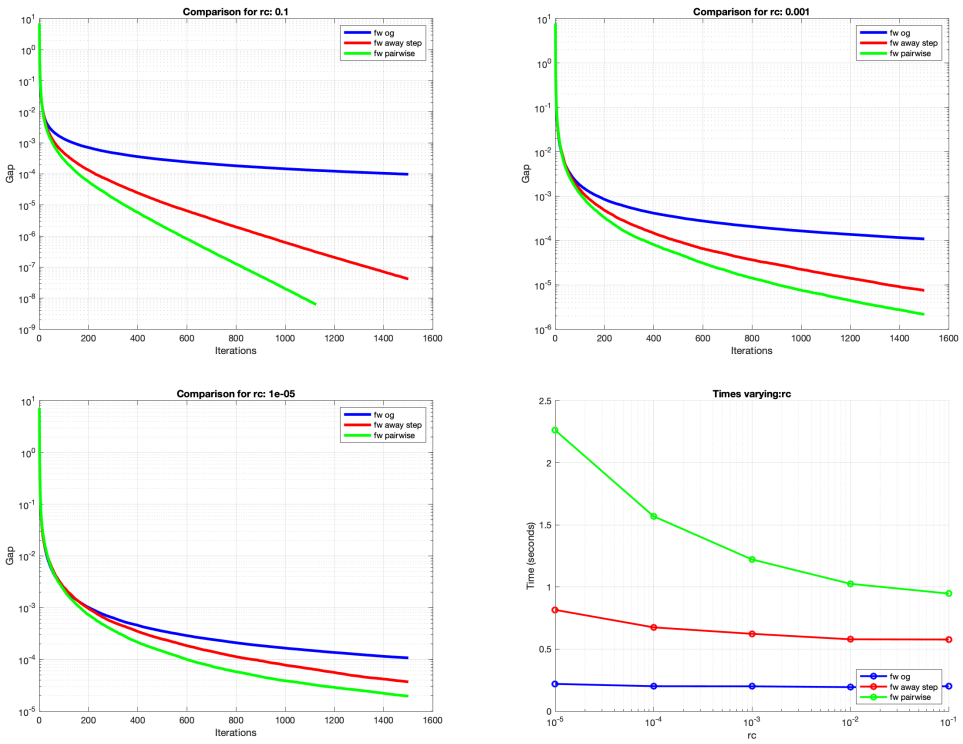

Figure 5: Convergence of the Frank-Wolfe algorithms varying the reciprocal of the condition number. Bottom-right: CPU times.

4.4 Condition Number

We can also analyze the behaviour varying the condition number or, the reciprocal, `rc`. The values that we explored are $10^{-i}, i \in \{1, 2, 3, 4, 5\}$.

The plots in Figure 5 demonstrate the effect of varying the reciprocal of the condition number on the convergence behavior of the Frank-Wolfe variants. The condition number influences how well-conditioned or ill-conditioned the problem is, with higher values indicating a more challenging problem to optimize (respectively, low values for `rc`).

At a higher values of `rc` (well-conditioned problem), the Pairwise and Away-Step variants converge the fastest, reducing the duality gap at a global linear rate. The classic Frank-Wolfe lags behind, with significantly slower convergence. The performance gap between the advanced algorithms is substantial, with Pairwise converging nearly 5 orders of magnitude faster than the vanilla. As `rc` decreases, making the problem more ill-conditioned, the overall convergence rate slows down, but the advanced variants still outperform the vanilla version, converging at a better rate. At very low condition numbers (highly ill-conditioned problems), even the advanced algorithms approach sublinear convergence, and the performance differences between them and the classic version become less distinct: while Pairwise and Away-Step are still faster than Frank-Wolfe, the performance gap narrows. This highlights the importance of the condition number: as the problem becomes more ill-conditioned, all algorithms face greater difficulty in converging efficiently.

Considering the CPU time of the first 1500 iterations (Figure 5 bottom-right), as `rc` ap-

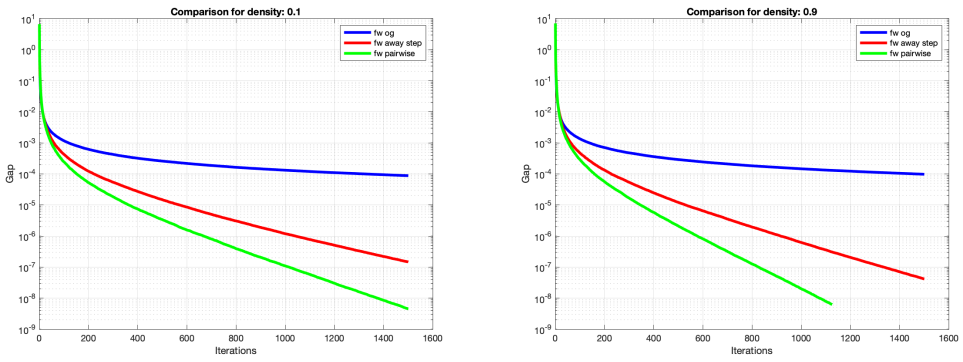

Figure 6: Convergence of the Frank-Wolfe algorithms varying the density.

proaches 0, the time taken by all algorithms increases, particularly for the Pairwise variant. The Away-Step algorithm also experiences an increase in time, though less drastic. The vanilla Frank-Wolfe, is faster in general and shows the least increase in time as the problem becomes more ill-conditioned, suggesting that its computational time is not sensitive to changes in the condition number compared to the other variants.

This is something we actually would expect since, by construction, the decrease of the `rc` value leads to the Hessian being close to be singular.

4.5 Density

Another parameter we can vary is the density of the matrix $\boldsymbol{Q}$. Keeping all other factors constant, we explore density values of 0.1, 0.3, 0.5, 0.7, and 0.9. From the experiments, we observe that density has a smaller impact on both convergence and computation time compared to other factors we've tested. As shown in Figure 6, there is no substantial difference in convergence even when comparing the extremes of the range. While higher density slightly improves the performance of the Away-Step and Pairwise variants, resulting in faster convergence, all densities exhibit linear convergence for these versions. Additionally, the maximum difference in computation time between the experiments with very sparse and very dense matrices for the Pairwise variant is only 0.14 seconds. This moderate difference is reasonable because the problem size is fixed to a moderate value like 100. Sparsity has a greater influence as the matrix size increases, as we will see.

4.6 Bigger Matrices

We now proceed to experiments involving larger matrices. Due to the dimensionality of these matrices, we will focus on sparse cases with fewer entries to both accelerate computation and assess the impact of sparsity. We define a sparse matrix as having a density of 0.1 and will conduct tests with problem sizes of 500 and 1000. For a very sparse matrix, with a density of 0.01, we will explore larger problem sizes of 5000 and 10,000. Additionally, a matrix with a density of 0.001, considered extremely sparse, will be tested with sizes of 5000 and 10,000. For each combination of size and sparsity, we will perform paired experiments—one with 70% active constraints and another with all constraints active. All other parameters are consistent with the base problem described in Table 1.

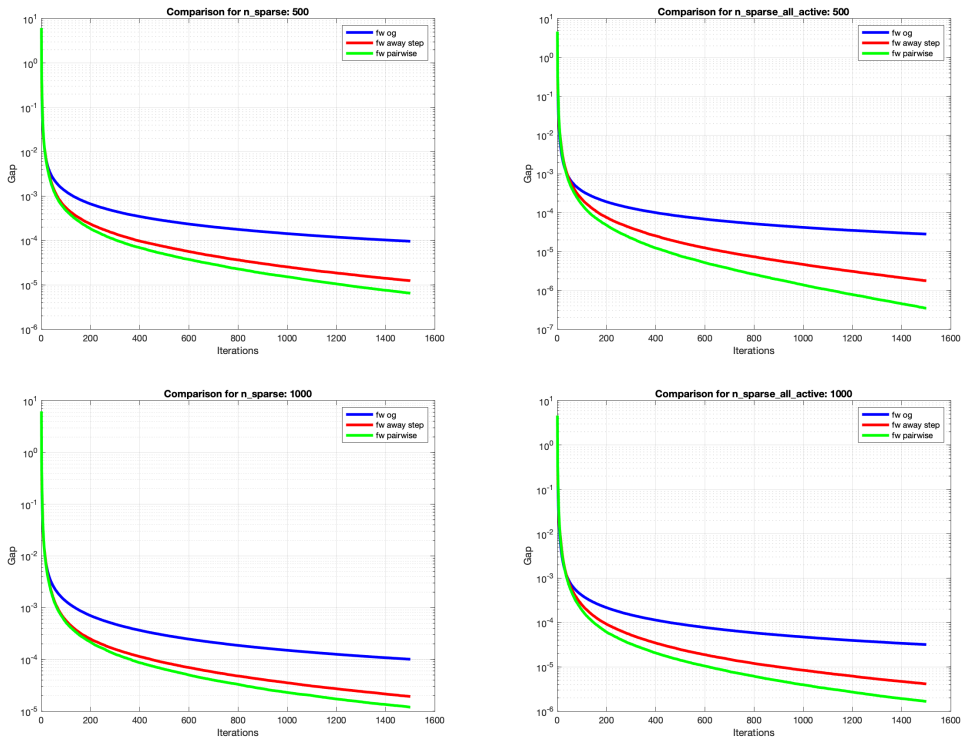

Figure 7: Convergence of the Frank-Wolfe algorithms for sparse matrices with problem sizes 500 (top) and 1000 (bottom). Left plots: 70% active constraints. Right plots: 100% (all active) constraints.

The results from the experiments with sparse matrices of sizes 500 and 1000 (Figure 7) confirm previous observations: increasing the problem size slows down convergence, while utilizing more active constraints accelerates convergence, reaching lower objective values.

When comparing the plot for size 500 from the previous section, where only the size varied, we observe that sparsity has little effect on convergence. However, having all active constraints significantly improves convergence, making the advanced variant of the Frank-Wolfe algorithm appear more linear. In terms of runtime, as shown in Table 7, sparse matrices lead to faster computations compared to those with base density. Additionally, experiments with all active constraints further reduce runtime. Note that these times are recorded for the algorithms completing the first 1500 iterations where each algorithm achieves different levels of precision in the solution. This explains why OG is the fastest—the iterations are quicker, but it converges to a less accurate solution.

Let's now analyze even larger matrices, specifically of sizes 5000 and 10,000. The results can be found in Figure 8. It is evident that with such large matrices, the likelihood of encountering ill-conditioned problems increases, causing the Away-step and Pairwise variants to perform more similarly to the original Frank-Wolfe algorithm, though they still outperform it. These experiments also confirm that reducing matrix density and increasing the number of active constraints improves overall convergence for problems of the same size. Additionally, as shown in Table 3, reducing density and increasing active constraints significantly

Algorithm, n=500	base	dens=0.1 actv=0.7	dens=0.1 actv=1
Frank-Wolfe OG	5.5692	0.6508	0.4376
Frank-Wolfe Away Step	6.8633	1.8045	1.5101
Frank-Wolfe Pairwise	7.9847	3.1385	2.5865

Table 2: CPU time comparison for the first 1500 Frank-Wolfe algorithms iterations on a problem size of 500, with varying matrix densities (base and 0.1) and different percentages of active constraints (70% and 100%).

improves the computational time. Here too, OG is faster in the initial iterations but reaches a lower precision in the solution.

Algorithm, n=5,000	dens=0.01 actv=0.7	dens=0.01 actv=1	dens=0.001 actv=0.7	dens=0.001 actv=1
Frank-Wolfe OG	9.3665	7.5935	3.8999	3.6322
Frank-Wolfe Away Step	17.8026	15.4016	10.5244	12.0314
Frank-Wolfe Pairwise	26.1624	23.2525	17.6753	35.0571

Algorithm, n=10,000	dens=0.01 actv=0.7	dens=0.01 actv=1	dens=0.001 actv=0.7	dens=0.001 actv=1
Frank-Wolfe OG	32.9256	32.05387	9.6177	3.3241
Frank-Wolfe Away Step	52.6993	50.6043	24.7906	20.4551
Frank-Wolfe Pairwise	70.0331	67.4073	40.1603	45.2057

Table 3: CPU time comparison for the first Frank-Wolfe algorithms iterations on problem sizes of 5,000 and 10,000, with varying matrix densities (0.01 and 0.001) and different percentages of active constraints (70% and 100%).

Interestingly, with these large matrices, the high variance between different trials observed in the base problem (Figure 1) is no longer present. This is evident in Figure 9, for the case of a 10,000-size matrix that is extremely sparse with all active constraints. No single run deviates significantly from the others, indicating that all algorithms and problem instances face similar challenges and perform consistently across trials.

4.7 Upper Bound

In the previous section, we defined a method to estimate an upper bound for the convergence of the algorithms. Now, based on Theorems 2.6, 2.7, and 2.8, we examine how these estimations perform in practice. Figure 10 shows the convergence of the dual gap for the algorithms using the base-case problem, clearly demonstrating that the upper bound is respected. The upper bound consistently remains higher than the dual gap throughout the iterations, with some gradual decrease. This gap indicates that, while the theoretical worst-case performance is bounded, the algorithm performs much better in practice.

For the classic Frank-Wolfe algorithm, the upper bound follows a similar curve to the dual gap, while the other variants show more linear behavior. Although still optimistic, the upper bound for the Frank-Wolfe algorithm is tighter compared to the other variants. This is likely due to the more approximate nature of the estimations used in their calculation. For example, in estimating the pyramidal width, we used sampling for random directions,

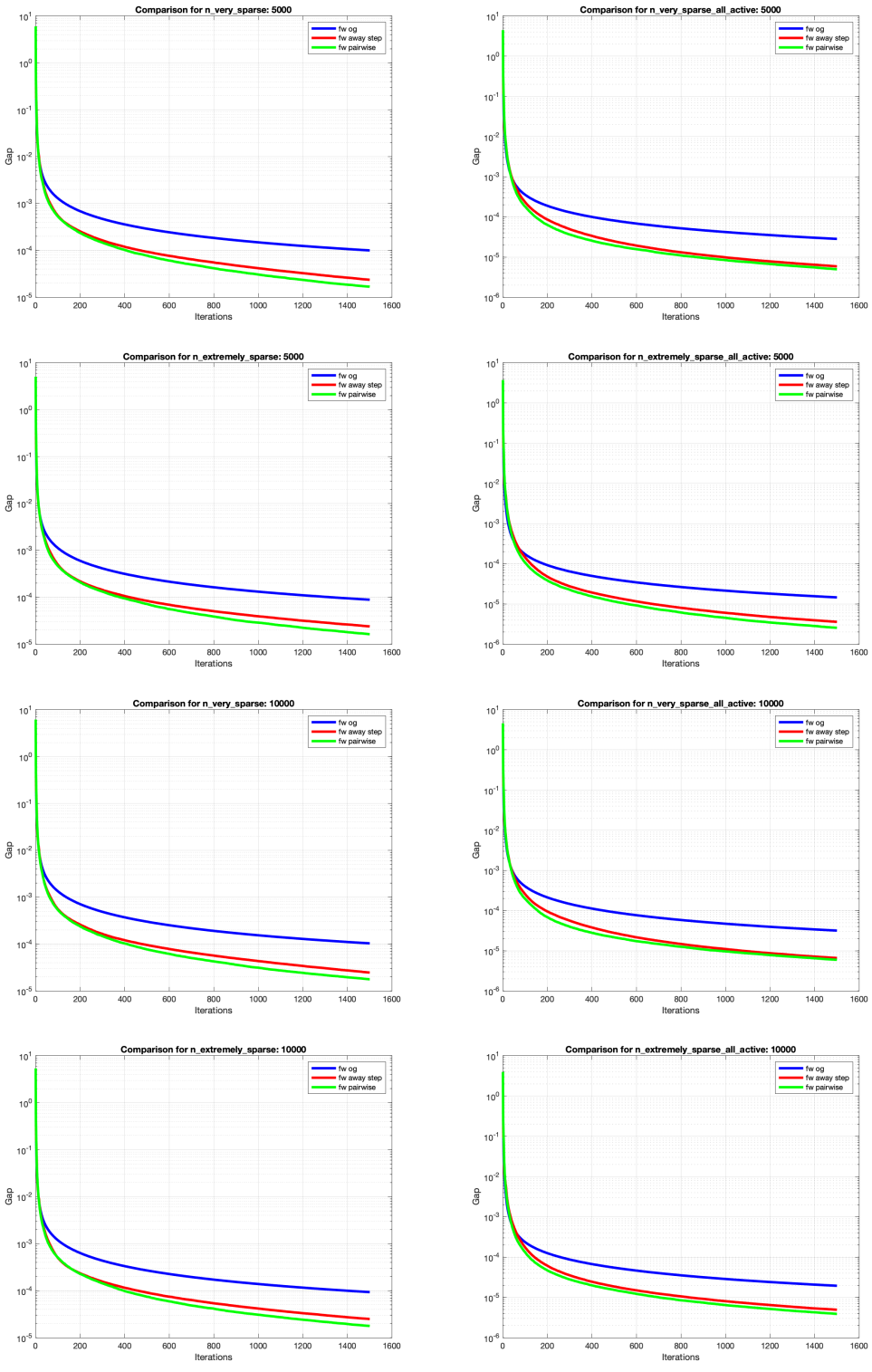

Figure 8: Convergence comparison of Frank-Wolfe algorithms for sparse matrices with problem sizes of 5,000 and 10,000. The top two rows display results for n=5,000, while the bottom two rows display results for n=10,000. Left plots: 0.01 and 0.001 density with 70% active constraints. Right plots: 0.01 and 0.001 density with 100% active constraints.

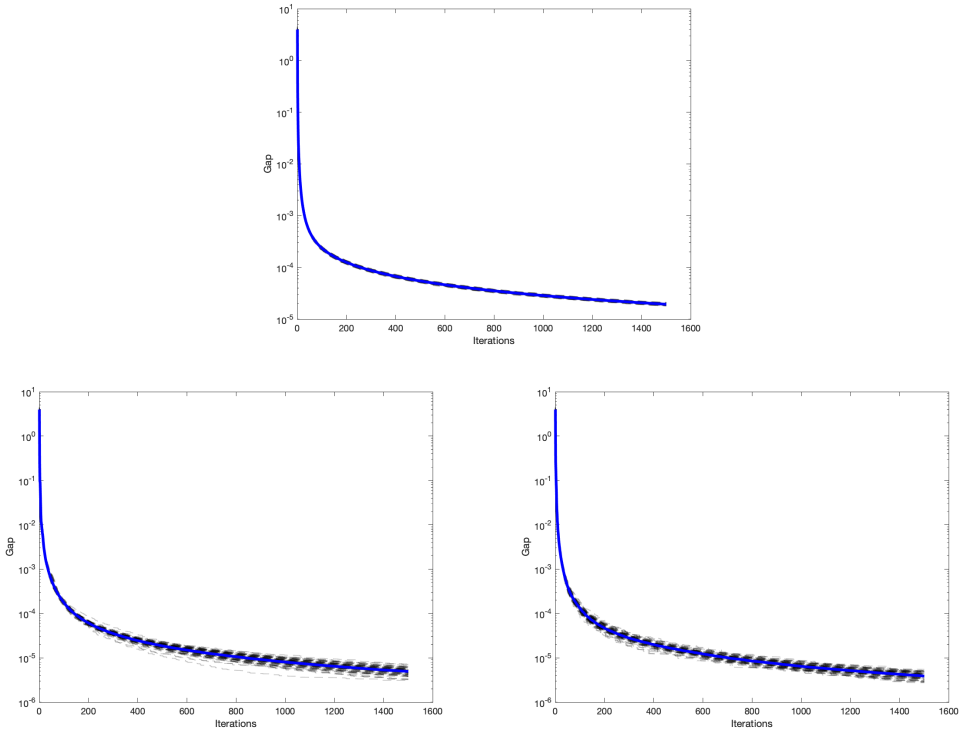

Figure 9: Convergence of the Frank-Wolfe algorithms for a 10,000-size matrix that is extremely sparse with all active constraints. Top: vanilla Frank-Wolfe. Bottom-left: Away-Step Frank Wolfe. Bottom-right: Pairwise Frank-Wolfe.

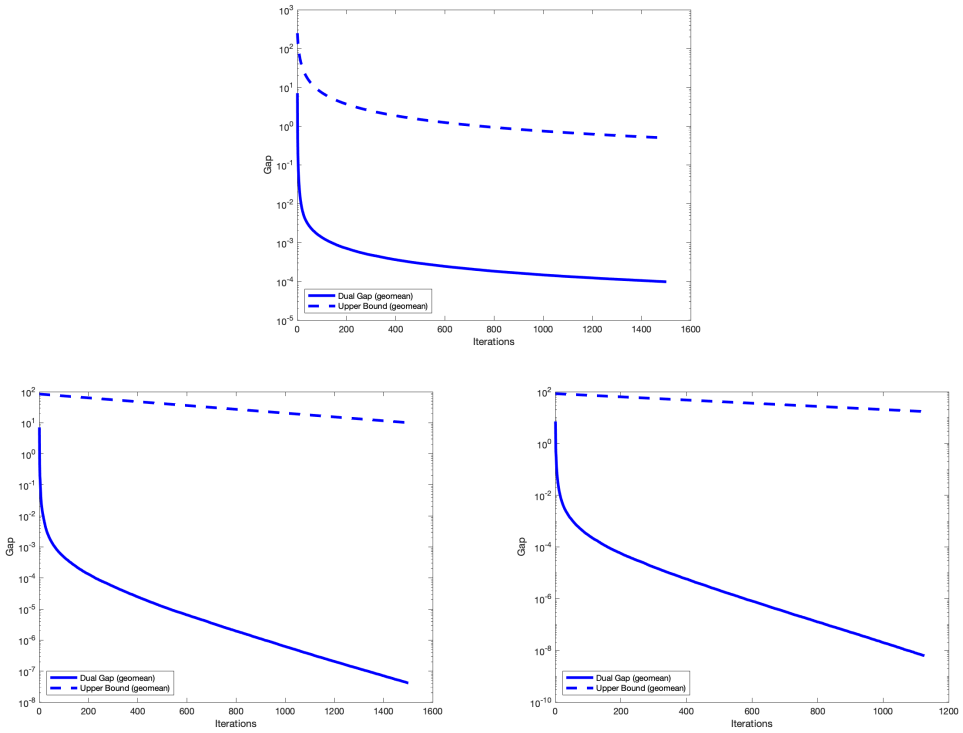

Figure 10: Convergence and upper bound of the Frank-Wolfe algorithms for the base-case problem. Top-left: vanilla Frank-Wolfe. Top: vanilla Frank-Wolfe. Bottom-left: Away-Step Frank Wolfe. Bottom-right: Pairwise Frank-Wolfe.

and the polytope diameter considers only the box constraints, not the linear inequality constraints.

4.8 And the Winner is...

We observed that during the first 1500 iterations, both the Away-Step and Pairwise algorithms consistently perform better in terms of convergence, as the dual gap they minimize reaches a lower point, leading to more accurate solutions. However, based on the time plots, the original Frank-Wolfe algorithm completes the first 1500 iterations more quickly, though it reaches a lower precision. So, how do we determine the winner? Which algorithm is the fastest at reaching a target precision?

To answer this, we allowed the algorithms to run for an indefinite number of iterations and recorded how long they took to reach a precision goal of 10^{-5}. This experiment was conducted for both the base case problem and a larger sparse matrix (size 1000, density 0.1).

For the base case problem, Table 4 presents the average time and number of iterations over 100 trials for each algorithm. While each iteration of the Frank-Wolfe OG is faster, it requires significantly more iterations, resulting in longer overall computation time to achieve a dual gap of 10^{-5}. The Frank-Wolfe Pairwise algorithm requires the fewest iterations, but each iteration is slower (likely due to the larger active set). In comparison, the Away Step

algorithm strikes a balance between iteration speed and count, making it the fastest overall and the most efficient in this experiment (the **winner**).

Algorithm	time (s)	n. of iterations
Frank-Wolfe OG	0.9571	15051.9
Frank-Wolfe Away Step	**0.2171**	665.76
Frank-Wolfe Pairwise	0.2648	**415.75**

Table 4: CPU time comparison for the Frank-Wolfe algorithms to reach a precision on 10^{-5} on the base case problem.

We conducted the same analysis for a sparse problem of size 1000, with the results presented in Table 5. The conclusions remain largely the same: OG is the slowest, and although the Away-Step method requires slightly more iterations than Pairwise, it emerges as the overall **winner** in terms of efficiency.

Algorithm	time (s)	n. of iterations
Frank-Wolfe OG	26.3885	15289.26
Frank-Wolfe Away Step	**8.1218**	2262.31
Frank-Wolfe Pairwise	8.4528	**1709.79**

Table 5: CPU time comparison for the Frank-Wolfe algorithms to reach a precision on 10^{-5} on a big and sparse problem.

5 Conclusion

In this report, we have investigated the performance of the Frank-Wolfe algorithm and its two prominent variants—Away-Step and Pairwise Frank-Wolfe—in tackling the convex quadratic nonseparable knapsack problem. This comprehensive study is divided into several key sections, covering the problem formulation, algorithmic details, implementation strategies, and experimental results.

We begun by introducing the convex quadratic nonseparable knapsack problem. The problem is defined as minimizing a quadratic objective function subject to box constraints and a linear inequality constraint. This forms the basis for applying Frank-Wolfe-based algorithms, known for handling differentiable optimization problems under convex constraints. The main challenge addressed in this work is how to efficiently solve such a problem using a linear programming approach at each iteration, taking advantage of the problem's structure.

The standard Frank-Wolfe method is known to be prone to a zig-zagging behavior, particularly when the solution lies on the boundary of the feasible region, which slows down convergence. To address this, two key variants of Frank-Wolfe were introduced. The Away-Step Frank-Wolfe variant introduces an "away" direction, which allows the algorithm to move away from active constraints, mitigating the zig-zagging behavior and improving convergence. The Pairwise Frank-Wolfe variant redistributes weight between two atoms (vertices), rather than just moving towards a single vertex as in the classical algorithm. This ensures better weight management and faster convergence.

Both variants provide improvements in the convergence rate compared to the original Frank-Wolfe algorithm, particularly in structured optimization problems.

Before proceeding to the experiments, we addressed practical aspects of the project implementation, including stopping criteria, step size determination, initial point selection, and problem generation.

With a thorough experimental section we evaluated the performance of the Frank-Wolfe algorithms on various problem configurations. The experiments include:

- Convergence of Base Case: The base case problem is used to compare the three algorithms. The Away-Step and Pairwise variants show faster and linear convergence and reduced zig-zagging behavior, while the classic Frank-Wolfe struggles when the solution is near the boundary of the feasible region.
- Problem Size: This experiment examines how increasing the size of the matrix impacts convergence and computation time. While smaller problem sizes exhibited clear linear convergence for the advanced variants, larger problem sizes, as expected, lead to slower convergence across all algorithms. The performance gap in convergence rates between the advanced variants and the original Frank-Wolfe algorithm narrows as the problem size increases.
- Active Constraints: Varying the percentage of active constraints affects convergence. Higher percentages of active constraints lead to faster convergence for all algorithms, especially for the Pairwise and Away-Step variants.
- Condition Number: The condition number of the matrix Q is varied to explore how well-conditioned or ill-conditioned problems impact performance. With well-conditioned problems, the advanced Frank-Wolfe variants exhibit linear convergence, significantly outperforming the original Frank-Wolfe algorithm. As the condition number worsens, all algorithms struggle more. While also in these cases the convergence difference between algorithms becomes less pronounced the advanced variants continue to hold an edge.
- Density: The effect of matrix sparsity is tested by varying the density of the matrix Q. Lower density (more sparsity) leads to faster computation times but has a relatively small impact on convergence rates. The Away-Step and Pairwise variants consistently show better performance, and their linear convergence is maintained across varying levels of sparsity.
- Big Matrix Experiments: Larger matrices, with sizes 500, 1000, 5000 and 10,000, are analyzed under different sparsity and active constraint levels. The results confirm that reducing matrix density, and increasing the active constraints enhance convergence and lower computational times. The advanced Frank-Wolfe variants continue to outperform the original, although their behavior becomes more similar to the original algorithm in the case of ill-conditioned problems.
- Winner: In both the base case and the larger sparse problem, the Frank-Wolfe OG algorithm takes significantly more iterations and time to reach the desired precision of 10^{-5}. The Away-Step algorithm is the fastest overall, with fewer iterations and less

CPU time than Pairwise, making it the most efficient choice in both scenarios.

In essence, this report presents a comprehensive analysis of the Frank-Wolfe algorithm and its variants, demonstrating their applicability and superiority in efficiently solving large-scale convex optimization problems with constraints.

References

[1] M. D. Canon and C. D. Cullum. A tight upper bound on the rate of convergence of frank-wolfe algorithm. *SIAM Journal on Control*, 6(4):509–516, 1968.

[2] Martin Jaggi. Revisiting Frank-Wolfe: Projection-free sparse convex optimization. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 427–435, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[3] Dan Garber and Elad Hazan. Faster rates for the frank-wolfe method over strongly-convex sets, 2015.

[4] Simon Lacoste-Julien and Martin Jaggi. On the global linear convergence of frank-wolfe optimization variants, 2015.