COMP0083

NMWX9, SN: 20114649

December 31, 2024

Contents

1 Questions with multiple answers

1.1 Question 1

Answer: (a)
Justification:

Function a: $f(x) = \max\{ax + b, x^4 - 5, e^{-x}\}$

- the maximum of convex functions is convex lets check each individual functions
- $ax + b$ is a linear function which is convex
- $x^4 - 5$ is convex because its second derivative $12x^2$ is non-negative
- e^{-x} is convex because its second derivative e^{-x} is postive for all x
- since all three functions are convex, their maximum is also convex

Function b: $g(x) = \min\{ax + b, x^4 - 5, e^{x}\}$

- The minimum of convex functions is not necessarily convex. While each of the individual functions is convex, the minimum operation can result in a function that is not convex, particularly when the functions intersect in a way that creates concave regions. Therefore, this function is not necessarily convex.

Function c: $h(x) = x^4 - 5 - e^{x}$

- This is the sum of $x^4 - 5$ and e^{-x}
- $x^4 - 5$ is convex because its second derivative $12x^2$ is non-negative
- $-e^{x}$ is concave because its second derivative $e{-}^{x}$ is negative for all x
- the sum of a convex and a concave function is not necessarily convex and in this case the concave term $-e^{x}$ will dominate at large value of x making the function concave at these region

1.2 Question 2

Answer: (a)
Justification:
let's compute the derivative of $f(x)$ first:

$$f'(x) = \begin{cases} -1 & x \in]-1, 0] \\ 2x & x \geq 0 \end{cases} \tag{1}$$

we know that $f(x)$ is a proper convex function on $]-1, +\infty]$ from the lecture notes **Proposition 3.1.8.** at $x = 0$ we have: $\partial f(0) = [f'_-(0), f'_+(0)] \cap]-1, +\infty] = [-1, 0]$

1.3 Question 3

Answer: (a)
Justification:
Take the derivative of $f(x) = \langle Ax, x\rangle + \langle x, b\rangle + c$ we got:

$$f'(x) = A^{*}x + Ax + b \tag{2}$$

1.4 Question 4

Answer: (a)
Justification:

The Fenchel conjugate of a function $h(x)$ is defined as:

$$h^*(u) = \sup_{x \in \mathbb{R}} (ux - h(x))$$

We are given the function $f(x) = g(3x)$, and we need to find its Fenchel conjugate $f^*(u)$.

$$f^*(u) = \sup_{x \in \mathbb{R}} (ux - g(3x))$$

Perform a change of variable:

$$y = 3x \quad \Rightarrow \quad x = \frac{y}{3}$$

Substituting into the expression:

$$f^*(u) = \sup_{y \in \mathbb{R}} \left(u \left(\frac{y}{3} \right) - g(y) \right) = \sup_{y \in \mathbb{R}} \left(\frac{u}{3} y - g(y) \right)$$

This is now in the standard form for the Fenchel conjugate of $g(y)$:

$$f^*(u) = g^* \left(\frac{u}{3} \right)$$

Thus, the Fenchel conjugate of $f(x) = g(3x)$ is:

$$f^*(u) = g^* \left(\frac{u}{3} \right)$$

2 Theory on convex analysis and optimization

2.1 Problem 1

Compute (showing a complete derivation) the Fenchel conjugates of the following functions:

1. $f : \mathbb{R} \to (-\infty, +\infty]$, with
$$f(x) = \begin{cases} +\infty & \text{if } x \leq 0, \\ -\log x & \text{if } x > 0. \end{cases}$$

2. $f(x) = 2x^2$.

3. $\iota_{[-1,1]}(x)$, the indicator function of the interval $[-1, 1]$, defined as
$$\iota_{[-1,1]}(x) = \begin{cases} 0 & \text{if } x \in [-1, 1], \\ +\infty & \text{otherwise.} \end{cases}$$

2.1.1 1. Fenchel Conjugate of $f(x) = \begin{cases} +\infty & x \leq 0, \\ -\log x & x > 0 \end{cases}$

The Fenchel conjugate $f^*(u)$ of a function f is defined as:
$$f^*(u) = \sup_{x \in \mathbb{R}} \left(ux - f(x) \right).$$

Given:
$$f(x) = \begin{cases} +\infty & \text{if } x \leq 0, \\ -\log x & \text{if } x > 0. \end{cases}$$

Thus,
$$f^*(u) = \sup_{x > 0} \left(ux + \log x \right).$$

To find the supremum, we first find the critical point by taking the derivative with respect to x and setting it to zero:
$$\frac{d}{dx} \left(ux + \log x \right) = u + \frac{1}{x} = 0.$$

Solving for x:
$$u + \frac{1}{x} = 0 \quad \Rightarrow \quad x = -\frac{1}{u}.$$

Since $x > 0$, we require:
$$-\frac{1}{u} > 0 \quad \Rightarrow \quad u < 0.$$

Thus, the supremum is attained at $x = -\frac{1}{u}$ for $u < 0$. Substituting back:
$$f^*(u) = u \left(-\frac{1}{u} \right) + \log \left(-\frac{1}{u} \right) = -1 + \log \left(-\frac{1}{u} \right).$$

Simplifying:
$$f^*(u) = -1 + \log \left(\frac{1}{-u} \right) = -1 - \log(-u).$$

For $u \geq 0$, the supremum is $+\infty$ because $f(x) = +\infty$ for $x \leq 0$, and as $x \to +\infty$, $ux + \log x \to +\infty$. Therefore, the Fenchel conjugate is:
$$f^*(u) = \begin{cases} -1 - \log(-u) & \text{if } u < 0, \\ +\infty & \text{if } u \geq 0. \end{cases}$$

2.1.2 2. Fenchel Conjugate of $f(x) = 2x^2$

The Fenchel conjugate $f^*(u)$ is given by:

$$f^*(u) = \sup_{x \in \mathbb{R}} \left(ux - 2x^2 \right).$$

To find the supremum, we take the derivative with respect to x and set it to zero:

$$\frac{d}{dx}\left(ux - 2x^2 \right) = u - 4x = 0 \quad \Rightarrow \quad x = \frac{u}{4}.$$

Substituting $x = \frac{u}{4}$ back into the expression:

$$f^*(u) = u\left(\frac{u}{4}\right) - 2\left(\frac{u}{4}\right)^2 = \frac{u^2}{4} - 2 \cdot \frac{u^2}{16} = \frac{u^2}{4} - \frac{u^2}{8} = \frac{u^2}{8}.$$

Therefore, the Fenchel conjugate is:

$$f^*(u) = \frac{u^2}{8}.$$

2.1.3 3. Fenchel Conjugate of $\iota_{[-1,1]}(x)$

The indicator function $\iota_{[-1,1]}(x)$ is defined as:

$$\iota_{[-1,1]}(x) = \begin{cases} 0 & \text{if } x \in [-1,1], \\ +\infty & \text{otherwise.} \end{cases}$$

The Fenchel conjugate $f^*(u)$ is:

$$f^*(u) = \sup_{x \in \mathbb{R}} \left(ux - \iota_{[-1,1]}(x) \right).$$

Since $\iota_{[-1,1]}(x) = 0$ for $x \in [-1,1]$ and $+\infty$ otherwise, the supremum reduces to:

$$f^*(u) = \sup_{x \in [-1,1]} ux.$$

The supremum of ux over $x \in [-1,1]$ depends on the sign of u:
- If $u \geq 0$, the supremum is attained at $x = 1$:

$$f^*(u) = u \cdot 1 = u.$$

- If $u \leq 0$, the supremum is attained at $x = -1$:

$$f^*(u) = u \cdot (-1) = -u.$$

Therefore, the Fenchel conjugate is:

$$f^*(u) = \begin{cases} u & \text{if } u \geq 0, \\ -u & \text{if } u \leq 0. \end{cases}$$

This can be succinctly written as:

$$f^*(u) = |u|.$$

2.2 Problem 2

Let $f : X \to (-\infty, +\infty]$ be a proper convex function.

2.2.1 1. Proof of Jensen's Inequality by Induction

Jensen's inequality states that for any convex function f and real numbers $x_1, x_2, \ldots, x_n \in X$ and weights $\lambda_1, \lambda_2, \ldots, \lambda_n \in \mathbb{R}^+$ such that $\sum_{i=1}^n \lambda_i = 1$, the following inequality holds:

$$f\left(\sum_{i=1}^n \lambda_i x_i\right) \leq \sum_{i=1}^n \lambda_i f(x_i).$$

We will prove this inequality by induction on n, the number of terms.

Base Case: $n = 2$
For the case $n = 2$, the inequality becomes:

$$f(\lambda_1 x_1 + \lambda_2 x_2) \leq \lambda_1 f(x_1) + \lambda_2 f(x_2),$$

where $\lambda_1 + \lambda_2 = 1$ and $\lambda_1, \lambda_2 \in \mathbb{R}^+$.
Since f is convex, by the definition of convexity, we know that:

$$f(\lambda_1 x_1 + \lambda_2 x_2) \leq \lambda_1 f(x_1) + \lambda_2 f(x_2).$$

Thus, the base case holds.

Inductive Hypothesis
Now, assume that the inequality holds for $n = k$. That is, for any $x_1, x_2, \ldots, x_k \in X$ and weights $\lambda_1, \lambda_2, \ldots, \lambda_k \in \mathbb{R}^+$ such that $\sum_{i=1}^k \lambda_i = 1$, we assume that:

$$f\left(\sum_{i=1}^k \lambda_i x_i\right) \leq \sum_{i=1}^k \lambda_i f(x_i).$$

Inductive Step: $n = k + 1$
Now, we prove the inequality for $n = k + 1$. Let $x_1, x_2, \ldots, x_{k+1} \in X$ and $\lambda_1, \lambda_2, \ldots, \lambda_{k+1} \in \mathbb{R}^+$ such that $\sum_{i=1}^{k+1} \lambda_i = 1$. We want to show that:

$$f\left(\sum_{i=1}^{k+1} \lambda_i x_i\right) \leq \sum_{i=1}^{k+1} \lambda_i f(x_i).$$

We can split the sum on the left-hand side as follows:

$$\sum_{i=1}^{k+1} \lambda_i x_i = \left(\sum_{i=1}^k \lambda_i x_i\right) + \lambda_{k+1} x_{k+1}.$$

Now, applying the convexity of f to the two terms $\left(\sum_{i=1}^k \lambda_i x_i\right)$ and x_{k+1}, we get:

$$f\left(\sum_{i=1}^{k+1} \lambda_i x_i\right) = f\left(\left(\sum_{i=1}^k \lambda_i x_i\right) + \lambda_{k+1} x_{k+1}\right).$$

By the convexity of f, we have:

$$f\left(\left(\sum_{i=1}^k \lambda_i x_i\right) + \lambda_{k+1} x_{k+1}\right) \leq \sum_{i=1}^k \lambda_i f(x_i) + \lambda_{k+1} f(x_{k+1}).$$

This completes the inductive step.

Conclusion
By induction, we have proven that Jensen's inequality holds for all n. That is, for any convex function f and any $x_1, x_2, \ldots, x_n \in X$ with weights $\lambda_1, \lambda_2, \ldots, \lambda_n \in \mathbb{R}^+$ such that $\sum_{i=1}^n \lambda_i = 1$, the inequality:

$$f\left(\sum_{i=1}^{n}\lambda_i x_i\right) \leq \sum_{i=1}^{n}\lambda_i f(x_i)$$

holds.

2.2.2 2. Proof that $-\log(x)$ is Convex

We will prove that the function $f(x) = -\log(x)$, defined on $x > 0$, is convex using the characterization of convexity for differentiable functions.

A function $f : \mathbb{R} \to \mathbb{R}$ is convex if for all $x_1, x_2 \in \mathbb{R}$ and for all $\lambda \in [0, 1]$, the following inequality holds:

$$f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda)f(x_2).$$

Alternatively, for differentiable functions, convexity can be established by showing that the second derivative of $f(x)$ is non-negative for all $x \in \text{domain}(f)$.

Compute the First and Second Derivatives
The function $f(x) = -\log(x)$ is differentiable for $x > 0$.
- First, compute the first derivative:

$$f'(x) = \frac{d}{dx}(-\log(x)) = -\frac{1}{x}.$$

- Next, compute the second derivative:

$$f''(x) = \frac{d}{dx}\left(-\frac{1}{x}\right) = \frac{1}{x^2}.$$

the Second Derivative
The second derivative $f''(x) = \frac{1}{x^2}$ is always positive for all $x > 0$, since $x^2 > 0$ for $x > 0$. Therefore, we have:

$$f''(x) > 0 \quad \text{for all} \quad x > 0.$$

Conclusion
Since the second derivative of $f(x) = -\log(x)$ is positive for all $x > 0$, this implies that $f(x)$ is convex on $(0, \infty)$ by the characterization of convexity for twice-differentiable functions.

Thus, $f(x) = -\log(x)$ is convex on $(0, \infty)$.

2.2.3 2. Arithmetic-Geometric Inequality via Jensen's Inequality

We aim to prove the following arithmetic-geometric inequality:

$$\sqrt[n]{x_1 x_2 \dots x_n} \leq \frac{1}{n}\sum_{i=1}^{n} x_i,$$

for all $x_1, x_2, \dots, x_n \in \mathbb{R}^+$ (positive real numbers).

Define a Convex Function
Consider the convex function $f(x) = -\log(x)$, which is convex on $\mathbb{R}^+$ as we have shown previously. Since $f(x)$ is convex, we can apply Jensen's inequality.

Apply Jensen's Inequality
Jensen's inequality states that for a convex function f, and for weights $\lambda_1, \lambda_2, \dots, \lambda_n \in \mathbb{R}^+$ such that $\sum_{i=1}^{n} \lambda_i = 1$, the following inequality holds:

$$f\left(\sum_{i=1}^{n} \lambda_i x_i\right) \leq \sum_{i=1}^{n} \lambda_i f(x_i).$$

In our case, let $f(x) = -\log(x)$ and choose the weights $\lambda_i = \frac{1}{n}$ for all i. Notice that the weights satisfy:

$$\sum_{i=1}^{n} \lambda_i = \sum_{i=1}^{n} \frac{1}{n} = 1.$$

Apply Jensen's Inequality to $f(x) = -\log(x)$
Applying Jensen's inequality to $f(x) = -\log(x)$ with $\lambda_i = \frac{1}{n}$, we get:

$$-\log\left(\sum_{i=1}^{n} \frac{1}{n} x_i\right) \leq \sum_{i=1}^{n} \frac{1}{n}(-\log(x_i)),$$

which simplifies to:

$$-\log\left(\frac{1}{n}\sum_{i=1}^{n} x_i\right) \leq \frac{1}{n}\sum_{i=1}^{n} -\log(x_i).$$

Multiplying both sides by -1 (since the logarithm function is negative, this reverses the inequality):

$$\log\left(\frac{1}{n}\sum_{i=1}^{n} x_i\right) \geq \frac{1}{n}\sum_{i=1}^{n} \log(x_i).$$

Exponentiate Both Sides
Next, exponentiate both sides of the inequality to remove the logarithms:

$$\exp\left(\log\left(\frac{1}{n}\sum_{i=1}^{n} x_i\right)\right) \geq \exp\left(\frac{1}{n}\sum_{i=1}^{n} \log(x_i)\right),$$

which simplifies to:

$$\frac{1}{n}\sum_{i=1}^{n} x_i \geq \exp\left(\frac{1}{n}\sum_{i=1}^{n} \log(x_i)\right).$$

By the properties of the geometric mean and the exponential of the average logarithm, we have:

$$\frac{1}{n}\sum_{i=1}^{n} x_i \geq \sqrt[n]{x_1 x_2 \dots x_n}.$$

Thus, we have proved the **Arithmetic-Geometric Inequality**:

$$\sqrt[n]{x_1 x_2 \dots x_n} \leq \frac{1}{n} \sum_{i=1}^{n} x_i.$$

This completes the proof.

2.3 Problem 3

Given a polytope $C = \mathrm{co}(a_1, \dots, a_m)$ in X, where $\mathrm{co}(a_1, \dots, a_m)$ denotes the convex hull of the points $a_1, \dots, a_m$, we are to prove that the maximum of a convex function f on C is attained at one of the vertices $a_1, \dots, a_m$.

Definition of Polytope and Convex Function
Let $C = \mathrm{co}(a_1, \dots, a_m)$ be the convex hull of the points $a_1, \dots, a_m$. The polytope C is the set of all convex combinations of the points $a_1, \dots, a_m$, i.e.,

$$C = \left\{ x \in X \mid x = \sum_{i=1}^{m} \lambda_i a_i,\ \lambda_i \geq 0,\ \sum_{i=1}^{m} \lambda_i = 1 \right\}.$$

The function f is convex on C, which means for any two points $x, y \in C$ and for any $\lambda \in [0,1]$, we have:

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y).$$

Consider the Maximum of f
Let $x^* \in C$ be the point where the maximum of f is attained on C, i.e.,

$$f(x^*) = \max_{x \in C} f(x).$$

We need to prove that x^* must be one of the vertices $a_1, \dots, a_m$ of the polytope C.

Assume x^* is Not a Vertex
Assume that x^* is not a vertex of C. This means that x^* can be written as a strict convex combination of two distinct points, say a_i and a_j, where $i \neq j$. That is, there exist $\lambda_1, \lambda_2 \in (0,1)$ such that

$$x^* = \lambda_1 a_i + \lambda_2 a_j \quad \text{with} \quad \lambda_1 + \lambda_2 = 1.$$

Use Convexity of f
Since f is convex, we can apply the convexity property of f to the points a_i and a_j:

$$f(x^*) = f(\lambda_1 a_i + \lambda_2 a_j) \leq \lambda_1 f(a_i) + \lambda_2 f(a_j).$$

But by the assumption that $f(x^*) = \max_{x \in C} f(x)$, we know that $f(x^*) \geq f(a_i)$ and $f(x^*) \geq f(a_j)$. Thus, we must have:

$$f(x^*) = \lambda_1 f(a_i) + \lambda_2 f(a_j).$$

Contradiction
However, this implies that $f(x^*)$ is a weighted average of $f(a_i)$ and $f(a_j)$, and hence strictly less than $f(x^*)$ unless $f(a_i) = f(x^*)$ and $f(a_j) = f(x^*)$. But this contradicts the assumption that x^* is not a vertex because it shows that f is constant on the entire segment joining a_i and a_j, implying f is constant in this region. This cannot happen unless x^* is one of the vertices of the polytope.

Conclusion
Therefore, the maximum of f must occur at one of the vertices $a_1, \dots, a_m$ of the polytope C.
This completes the proof.

2.4 Problem 4

We are given the function $f : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ defined by:

$$f(x, y) = \|2x - y\|_2^2.$$

Our goal is to prove that $f(x, y)$ is jointly convex in both x and y.

Write $f(x, y)$ Explicitly
The function $f(x, y)$ is defined as the squared Euclidean norm of $2x - y$:

$$f(x, y) = \|2x - y\|_2^2 = (2x - y)^\top (2x - y).$$

Expanding this expression, we get:

$$f(x, y) = 4x^\top x - 4x^\top y + y^\top y.$$

Thus, we can write:

$$f(x, y) = 4\|x\|_2^2 - 4x^\top y + \|y\|_2^2.$$

Convexity of Each Term
We now examine the convexity of each term in the expanded form:
- The term $4\|x\|_2^2 = 4\sum_{i=1}^n x_i^2$ is a quadratic function in x, and quadratic functions of the form $\|x\|_2^2$ are convex (since the Hessian of $\|x\|_2^2$ is $2I$, which is positive semi-definite). - The term $-4x^\top y$ is linear in both x and y, and linear functions are convex (and concave). - The term $\|y\|_2^2 = \sum_{i=1}^n y_i^2$ is also quadratic in y, and as with x, quadratic functions of the form $\|y\|_2^2$ are convex.

Convexity of $f(x, y)$
The function $f(x, y) = 4\|x\|_2^2 - 4x^\top y + \|y\|_2^2$ is the sum of convex functions:
- $4\|x\|_2^2$ is convex in x, - $\|y\|_2^2$ is convex in y, - $-4x^\top y$ is linear in both x and y (and thus convex in both).
Since the sum of convex functions is convex, we conclude that $f(x, y)$ is jointly convex in x and y.

Conclusion
Therefore, the function $f(x, y) = \|2x - y\|_2^2$ is jointly convex in x and y.
This completes the proof.

2.5 Problem 5

2.5.1 1. Deriving the Dual Problem Using Fenchel-Rockafellar Duality

We consider the following optimization problem:

$$\begin{aligned}\min_{x\in\mathbb{R}^d} \quad & \frac{1}{2}\|x\|_2^2 \\ \text{subject to} \quad & \|Ax-b\|_\infty \le \varepsilon,\end{aligned}$$

where $\varepsilon > 0$, $A \in \mathbb{R}^{n\times d}$, and $b \in \mathbb{R}^n$.

Our goal is to compute the dual problem using the Fenchel-Rockafellar duality theory.

Reformulate the Primal Problem

First, we express the primal problem in the form suitable for applying Fenchel-Rockafellar duality, i.e.,

$$\min_x \{f(x) + g(Ax)\},$$

where:

$$f(x) = \frac{1}{2}\|x\|_2^2,$$

and

$$g(z) = \iota_C(z),$$

with $C = \{z \in \mathbb{R}^n \mid \|z-b\|_\infty \le \varepsilon\}$ being the feasible set defined by the constraint.

Identify the Fenchel Conjugates

To apply Fenchel-Rockafellar duality, we need to determine the Fenchel conjugates f^* and g^*.

Fenchel Conjugate of $f(x)$

The Fenchel conjugate of f, denoted $f^*(u)$, is defined as:

$$f^*(u) = \sup_{x\in\mathbb{R}^d}\left(u^\top x - \frac{1}{2}\|x\|_2^2\right).$$

To compute this, we complete the square:

$$u^\top x - \frac{1}{2}\|x\|_2^2 = -\frac{1}{2}\|x-u\|_2^2 + \frac{1}{2}\|u\|_2^2.$$

Taking the supremum over $x \in \mathbb{R}^d$:

$$f^*(u) = \sup_x\left(-\frac{1}{2}\|x-u\|_2^2 + \frac{1}{2}\|u\|_2^2\right) = \frac{1}{2}\|u\|_2^2.$$

Fenchel Conjugate of $g(z)$

The Fenchel conjugate of g, denoted $g^*(y)$, is defined as:

$$g^*(y) = \sup_{z\in\mathbb{R}^n}\left(y^\top z - \iota_C(z)\right).$$

Since $\iota_C(z)$ is the indicator function of the set C, the supremum is achieved when $z \in C$. Therefore,

$$g^*(y) = \sup_{z\in C} y^\top z.$$

The set $C = \{z \mid \|z-b\|_\infty \le \varepsilon\}$ can be rewritten as $z = b + w$, where $\|w\|_\infty \le \varepsilon$. Thus,

$$g^*(y) = \sup_{\|w\|_\infty\le\varepsilon} y^\top(b+w) = y^\top b + \sup_{\|w\|_\infty\le\varepsilon} y^\top w.$$

The supremum $\sup_{\|w\|_\infty\le\varepsilon} y^\top w$ is equal to $\varepsilon\|y\|_1$. Therefore,

$$g^*(y) = y^\top b + \varepsilon\|y\|_1.$$

Formulate the Dual Problem

According to Fenchel-Rockafellar duality, the dual problem is given by:

$$\max_y \left(-f^*(-A^\top y) - g^*(y)\right).$$

Substituting the expressions for f^* and g^*:

$$\begin{aligned} &\max_y \quad -\frac{1}{2}\|-A^\top y\|_2^2 - \left(y^\top b + \varepsilon\|y\|_1\right) \\ = &\max_y \quad -\frac{1}{2}\|A^\top y\|_2^2 - y^\top b - \varepsilon\|y\|_1. \end{aligned}$$

This can be equivalently written as a minimization problem:

$$\min_y \left(\frac{1}{2}\|A^\top y\|_2^2 + y^\top b + \varepsilon\|y\|_1\right).$$

Final Dual Problem

Thus, the dual problem is:

$$\begin{aligned} &\max_y \quad -\frac{1}{2}\|A^\top y\|_2^2 - y^\top b - \varepsilon\|y\|_1 \\ \text{or equivalently} \quad &\min_y \left(\frac{1}{2}\|A^\top y\|_2^2 + y^\top b + \varepsilon\|y\|_1\right). \end{aligned}$$

2.5.2 2. Strong Duality and Qualification Conditions

To determine if strong duality holds for the optimization problem:

$$\min_{x\in\mathbb{R}^d} \quad \frac{1}{2}\|x\|_2^2 \quad \text{subject to} \quad \|Ax-b\|_\infty \leq \varepsilon,$$

we need to check whether the Slater's condition holds, which is a sufficient condition for strong duality to hold in convex optimization problems.

Primal Problem and Feasibility
The primal problem is given by:

$$\min_{x\in\mathbb{R}^d} \quad \frac{1}{2}\|x\|_2^2 \quad \text{subject to} \quad \|Ax-b\|_\infty \leq \varepsilon.$$

We see that the objective function $\frac{1}{2}\|x\|_2^2$ is convex and the constraint $\|Ax-b\|_\infty \leq \varepsilon$ defines a polytope (or a box) which is convex as well. Hence, the primal problem is a convex optimization problem.

Slater's Condition
Slater's condition is a regularity condition that guarantees strong duality holds. It states that there must exist an interior point that strictly satisfies the constraints. In our case, the constraint is $\|Ax-b\|_\infty \leq \varepsilon$, which implies:

$$-\varepsilon \leq Ax-b \leq \varepsilon.$$

This means that the vector $Ax-b$ should strictly lie within the open box $(-\varepsilon, \varepsilon)$, not touching the boundaries of this box. If such an x exists, then the constraints are strictly feasible, and Slater's condition holds.

Feasibility
First, we need to ensure that the feasible set C is non-empty:

$$C = \{x \in \mathbb{R}^d : \|Ax-b\|_\infty \leq \epsilon\}.$$

Assuming C is non-empty (which is typically the case if ϵ is sufficiently large), we proceed to check Slater's condition.

Strict Feasibility
A point $\hat{x} \in C$ is **strictly feasible** if:

$$\|A\hat{x}-b\|_\infty < \epsilon.$$

To demonstrate the existence of such a point:

- If $\epsilon > \|b\|_\infty$, then choosing $\hat{x} = 0$ yields:

  $$\|A\hat{x}-b\|_\infty = \|-b\|_\infty = \|b\|_\infty < \epsilon.$$

  Hence, $\hat{x} = 0$ is strictly feasible.

- If $\epsilon \leq \|b\|_\infty$, one can consider perturbing x slightly from a feasible point to ensure strict inequality. For instance, if there exists x' such that $\|Ax'-b\|_\infty = \epsilon$, then for a small $\delta > 0$, the point:

  $$\hat{x} = x' - \delta \frac{A^\top(Ax'-b)}{\|A^\top(Ax'-b)\|_2}$$

  will satisfy $\|A\hat{x}-b\|_\infty < \epsilon$, provided A has full rank and δ is chosen appropriately.

Thus, **Slater's condition is satisfied** provided that the feasible set C is non-empty.

Strong Duality
When Slater's condition holds, strong duality holds by **Fenchel-Rockafellar duality theory**. This means that the optimal value of the primal problem equals the optimal value of the dual problem.

Thus, if there exists an interior point satisfying the inequality $-\varepsilon < Ax-b < \varepsilon$, then strong duality holds.

Conclusion
Strong duality holds for the given optimization problem. provided that the feasible set is non-empty. This is justified by the problem's convexity and the satisfaction of Slater's condition, ensuring that the optimal primal value equals the optimal dual value.

2.5.3 3. KKT Conditions

For the optimization problem

$$\min_{x\in\mathbb{R}^d} \quad \frac{1}{2}\|x\|_2^2 \quad \text{subject to} \quad \|Ax-b\|_\infty \leq \varepsilon,$$

we can apply the KKT conditions for optimality. These conditions provide conditions for a solution to be optimal for a convex optimization problem with inequality constraints.

The Lagrangian
The Lagrangian of the problem is given by:

$$\mathcal{L}(x,\lambda) = \frac{1}{2}\|x\|_2^2 + \sum_{i=1}^{n} \lambda_i \left(\|Ax-b\|_\infty - \varepsilon\right),$$

where $\lambda_i \geq 0$ are the Lagrange multipliers associated with the inequality constraints $\|Ax-b\|_\infty \leq \varepsilon$.

However, since $\|Ax-b\|_\infty \leq \varepsilon$ involves a vector, the Lagrangian should incorporate the component-wise constraints:

$$\mathcal{L}(x,\lambda) = \frac{1}{2}\|x\|_2^2 + \sum_{i=1}^{n} \lambda_i \left(|(Ax-b)_i| - \varepsilon\right).$$

KKT Conditions
The KKT conditions for this problem are:
1. **Stationarity**: The gradient of the Lagrangian with respect to x must be zero at the optimum.

$$\nabla_x \mathcal{L}(x,\lambda) = x - A^\top \lambda = 0.$$

This leads to the condition:

$$x = A^\top \lambda.$$

2. **Primal feasibility**: The inequality constraints must be satisfied at the optimum.

$$\|Ax-b\|_\infty \leq \varepsilon.$$

3. **Dual feasibility**: The Lagrange multipliers must be non-negative.

$$\lambda_i \geq 0, \quad \forall i = 1,\ldots,n.$$

4. **Complementary slackness**: The Lagrange multipliers must satisfy the complementary slackness condition. This means that for each i, either the constraint is active, i.e., $\|(Ax-b)_i\| = \varepsilon$, or the corresponding Lagrange multiplier is zero.

$$\lambda_i \left(|(Ax-b)_i| - \varepsilon\right) = 0, \quad \forall i = 1,\ldots,n.$$

Summary of KKT Conditions
Thus, the KKT conditions for this problem are:

$$\begin{aligned}
\text{Stationarity:} &\quad x = A^\top \lambda, \\
\text{Primal feasibility:} &\quad \|Ax-b\|_\infty \leq \varepsilon, \\
\text{Dual feasibility:} &\quad \lambda_i \geq 0, \quad \forall i = 1,\ldots,n, \\
\text{Complementary slackness:} &\quad \lambda_i \left(|(Ax-b)_i| - \varepsilon\right) = 0, \quad \forall i = 1,\ldots,n.
\end{aligned}$$

These are the necessary conditions for optimality under the assumption that the problem is convex and the constraints satisfy Slater's condition (for strong duality).

2.5.4 4. Deriving the Rate of Convergence on Primal Iterates Using FISTA on the Dual Problem

In this section, we derive the rate of convergence for the primal iterates when applying the Fast Iterative Shrinkage-Thresholding Algorithm (FISTA) to the dual problem derived earlier. We utilize the relationship between the dual objective values and the distance of the primal iterates to the primal solution.

recall the Dual Problem
From Problem 5.1, the dual problem is formulated as:

$$\min_{y\in\mathbb{R}^n} \left(\frac{1}{2}\|A^\top y\|_2^2 + y^\top b + \varepsilon\|y\|_1 \right).$$

Let y^* denote the optimal solution to the dual problem, and let x^* denote the corresponding primal optimal solution, satisfying the KKT conditions.

FISTA Convergence Rate on Dual Objective Values
FISTA is known to achieve an objective value convergence rate of $\mathcal{O}\left(\frac{1}{k^2}\right)$, where k is the iteration number. Specifically, after k iterations, the dual objective satisfies:

$$f_D(y_k) - f_D(y^*) \leq \frac{C}{k^2},$$

where $f_D(y) = \frac{1}{2}\|A^\top y\|_2^2 + y^\top b + \varepsilon\|y\|_1$ is the dual objective function, and C is a constant that depends on the initial conditions.

Bounding the Distance to the Primal Solution
To relate the dual objective values to the primal iterates, we utilize the following relationship derived from Fenchel duality and strong duality (as strong duality holds in q2):

$$\frac{1}{2}\|x_k - x^*\|_2^2 \leq f_D(y_k) - f_D(y^*).$$

This inequality indicates that the squared distance between the primal iterate x_k and the optimal primal solution x^* is bounded by the dual objective suboptimality.

Deriving the Convergence Rate for Primal Iterates
Using the bound from Step 3 and the convergence rate of FISTA from Step 2, we substitute the dual suboptimality bound into the distance bound:

$$\frac{1}{2}\|x_k - x^*\|_2^2 \leq \frac{C}{k^2}.$$

Multiplying both sides by 2 gives:

$$\|x_k - x^*\|_2^2 \leq \frac{2C}{k^2}.$$

Taking the square root of both sides yields:

$$\|x_k - x^*\|_2 \leq \sqrt{\frac{2C}{k^2}} = \frac{\sqrt{2C}}{k}.$$

Conclusion
Thus, we have established that the distance between the primal iterates x_k and the optimal primal solution x^* converges at a rate of $\mathcal{O}\left(\frac{1}{k}\right)$:

$$\|x_k - x^*\|_2 \leq \frac{\sqrt{2C}}{k}.$$

This demonstrates that by applying FISTA to the dual problem, the primal iterates converge to the optimal solution x^* with a convergence rate of $\mathcal{O}\left(\frac{1}{k}\right)$.
Summary of the Convergence Rate:

$$\|x_k - x^*\|_2 = \mathcal{O}\left(\frac{1}{k}\right).$$

3 Solving the Lasso problem

Full Python implementation is included in the appendix I have also put the full python script in the accompanying zip file in the submission.

The result of the PSGA, RCPGA, and ergodic PSGA is presented in the figures below:

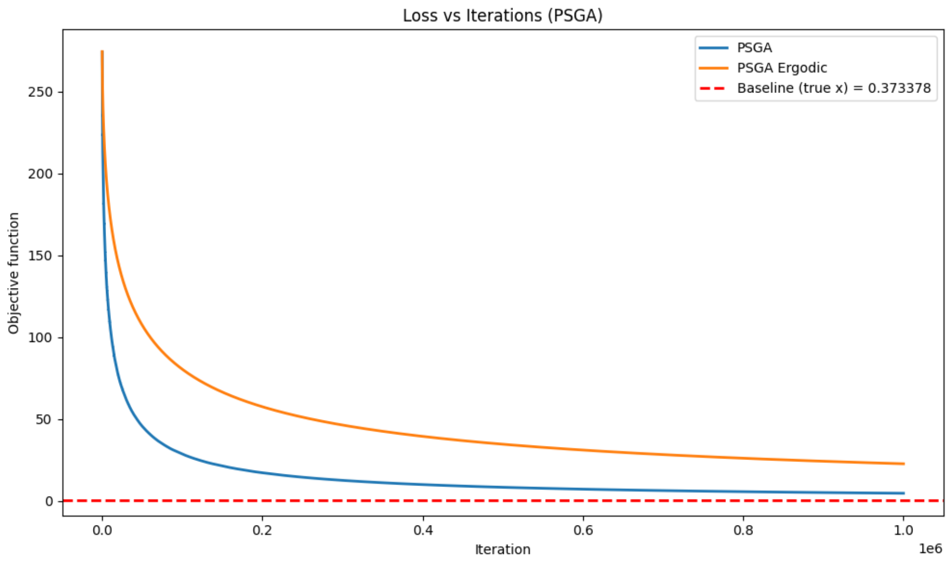

Figure 1: Value Change of the Objective Function (Loss) for PGSA and Ergodic PSGA, a baseline using the true solution is showed in red dash-lines

We can see that the PSGA need a large number of iterations to approach the baseline objective function and using the ergodic mean have make the performance worse with even slower convergence. In this run we used $\lambda = 0.01$. Although with 1 million iterations, both PSGA and Ergodic PSGA still cannot recover the true sparse vectors as shown below

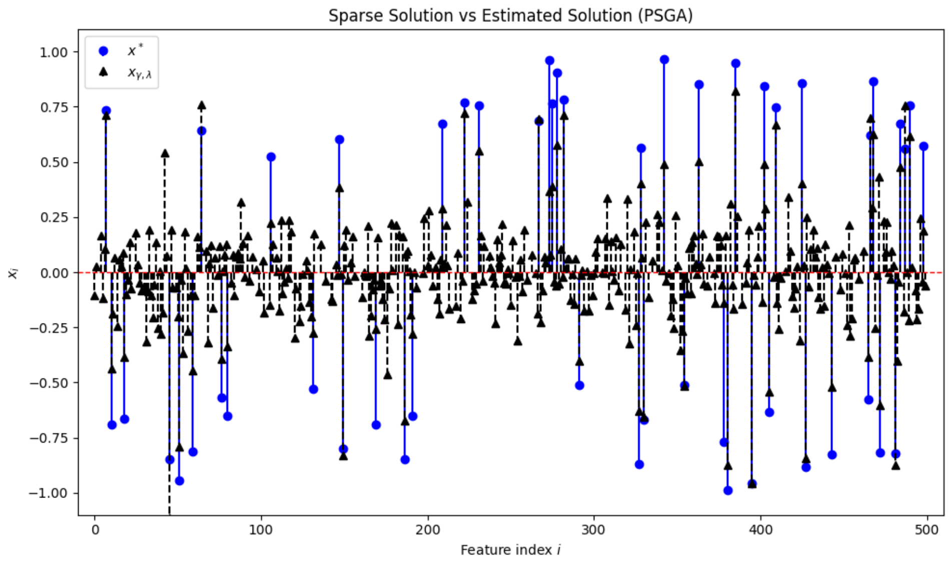

Figure 2: Solution of PSGA compared with true sparse vectors

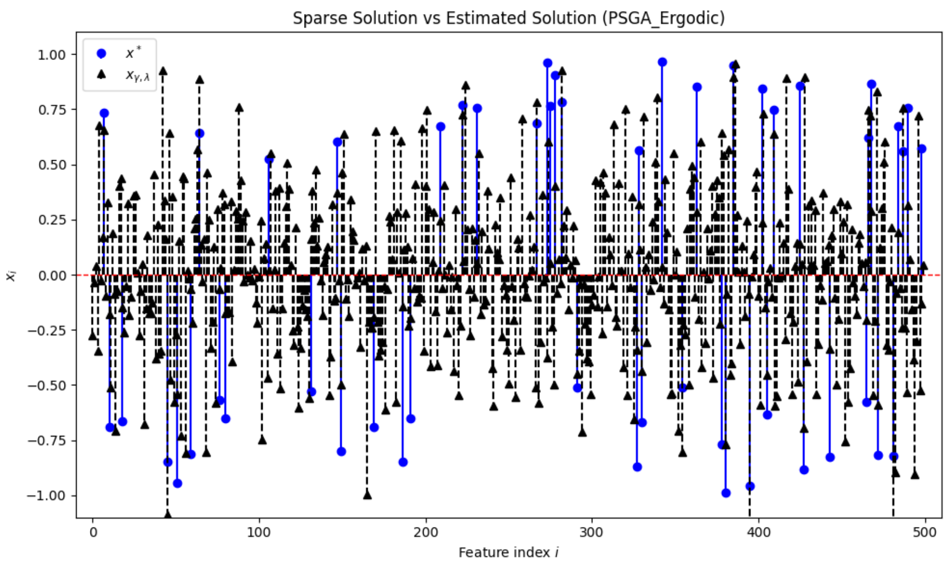

Figure 3: Solution of Ergodic PSGA compared with true sparse vectors

For RCPGA with just 10 thousand iterations, we were able to recover the true sparse vector solutions, in this run we set $\lambda = 0.05$

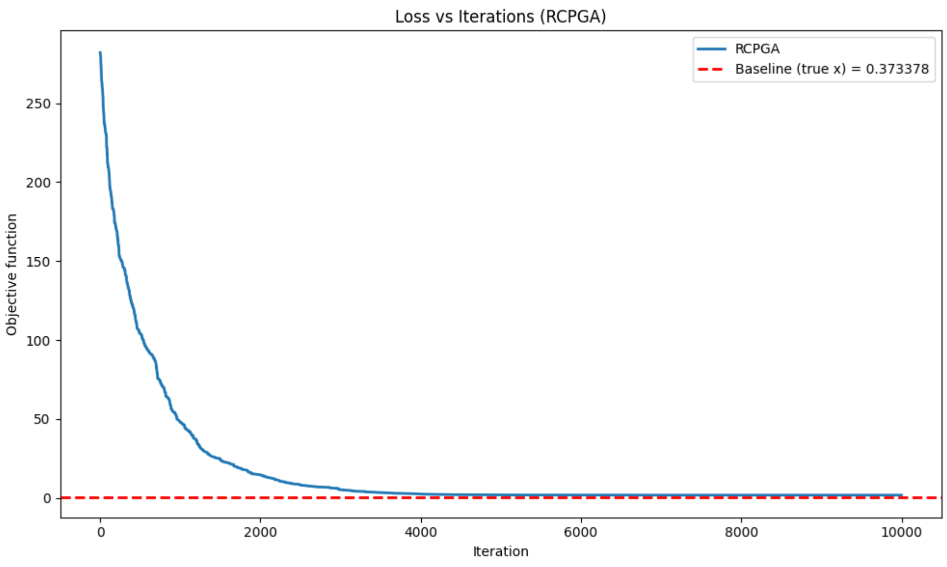

Figure 4: Value Change of the Objective Function (Loss) for RCPGA, baseline is showed in red dash-lines

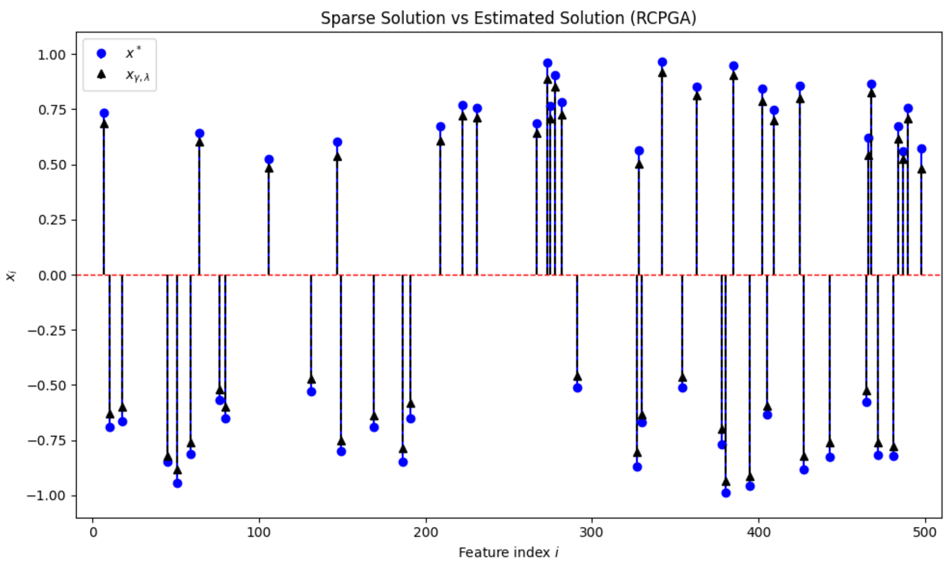

Figure 5: Solution of RCPGA compared with true sparse vectors

To see the difference in performance between PSGAs and RCPGA we add a combined loss plot at the end.

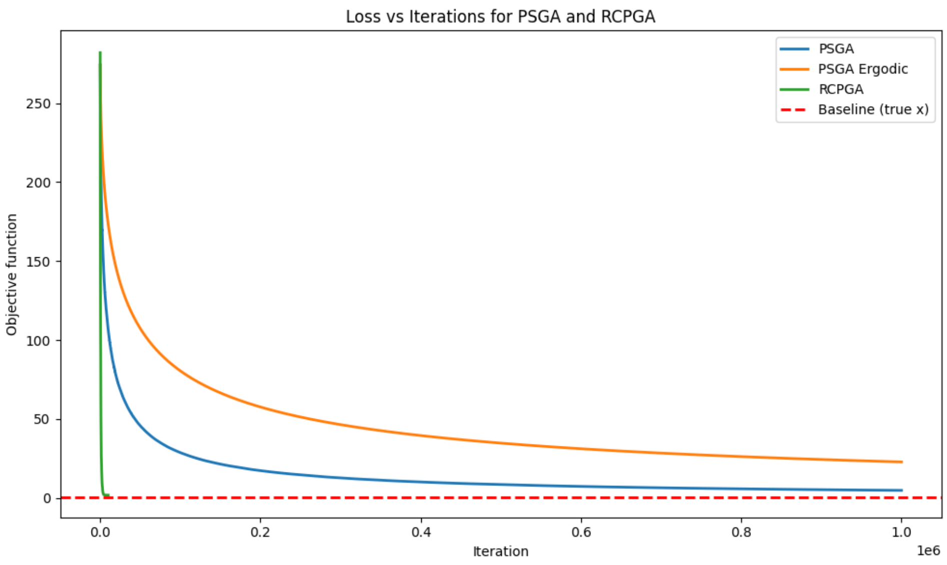

Figure 6: A combined Loss plot to show the difference in performance between RCPGA and the two PSGAs

```python
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm


def generate_lasso_problem(n, d, s, std=0.06):
    """
    Generates the synthetic Lasso problem with specified parameters.

    Args:
        n (int): Number of data points.
        d (int): Number of features.
        s (int): Sparsity of the true solution (number of non-zero elements).
        std (float): Standard deviation of the noise.

    Returns:
        tuple: A tuple containing:
            - true_solution (np.ndarray): The true sparse solution vector.
            - design_matrix (np.ndarray): The design matrix A.
            - noisy_target (np.ndarray): The noisy target vector y.
    """
    assert s % 2 == 0, "s␣needs␣to␣be␣divisible␣by␣2"

    # Create sparse true solution with positive and negative values
    positive_vals = 0.5 * (np.random.rand(s // 2) + 1)
    negative_vals = -0.5 * (np.random.rand(s // 2) + 1)
    true_solution = np.hstack([positive_vals, negative_vals, np.zeros(d - s)])
    np.random.shuffle(true_solution)  # Use numpy's shuffle for consistency

    # Generate design matrix A
    design_matrix = np.random.randn(n, d)

    # Generate noisy target vector y
    noisy_target = design_matrix @ true_solution + std * np.random.randn(n)

    return true_solution, design_matrix, noisy_target


def soft_thresholding(x, gamma):
    """
    Applies the soft-thresholding operator element-wise to a vector.

    Args:
        x (np.ndarray): The input vector.
        gamma (float): The threshold parameter.

    Returns:
        np.ndarray: The thresholded vector.
    """
    return np.sign(x) * np.maximum(np.abs(x) - gamma, 0.0)


def compute_lasso_loss(A, x_sol, y, lam, n):
    """
    Computes the Lasso objective function value.

    Args:
        A (np.ndarray): The design matrix.
        x_sol (np.ndarray): The coefficient vector.
        y (np.ndarray): The target vector.
        lam (float): The regularization parameter.
        n (int): Number of data points.

    Returns:
        float: The value of the Lasso loss function.
    """
    residual = A @ x_sol - y
    loss = (1.0 / (2.0 * n)) * np.linalg.norm(residual) ** 2 + lam * np.linalg.norm(
        x_sol, ord=1)
    return loss
```

```python
def proximal_stochastic_gradient_algorithm(x_init, y, n, A, lam, num_iterations,
    is_ergodic=False):
    """
    Proximal Stochastic Gradient Algorithm (PSGA) for solving the Lasso problem.

    Args:
        x_init (np.ndarray): The initial solution vector.
        y (np.ndarray): The target vector.
        n (int): Number of data points.
        A (np.ndarray): The design matrix.
        lam (float): The regularization parameter.
        num_iterations (int): The number of iterations to run the algorithm.
        is_ergodic (bool): Whether to compute the ergodic mean or not.

    Returns:
        tuple: A tuple containing:
            - solution (np.ndarray): The estimated solution after the specified number
                of iterations.
            - loss_history (list): The history of the Lasso loss function over
                iterations.
    """
    FroNormAsqrd = np.linalg.norm(A, ord='fro') ** 2
    x = x_init.copy()
    loss_history = []

    # Initialize for ergodic mean
    gamma_sum = 0.0
    gamma_x_sum = np.zeros_like(x)

    # Initialize progress bar
    with tqdm(total=num_iterations, desc="PSGA␣Progress", unit="iter") as pbar:
        for k in range(1, num_iterations + 1):
            # Randomly select an index i_k uniformly from {0, ..., n-1}
            ik = np.random.randint(0, n)

            # Step size gamma_k
            gamma_k = n / (FroNormAsqrd * np.sqrt(k))

            # Gradient step for PSGA
            gradient = (A[ik, :] @ x - y[ik]) * A[ik, :]
            t = x - gamma_k * gradient

            # Proximal operator (soft-thresholding)
            x = soft_thresholding(t, lam * gamma_k)

            # Compute and store loss
            current_loss = compute_lasso_loss(A, x, y, lam, n)
            loss_history.append(current_loss)

            # Update ergodic mean if needed
            if is_ergodic:
                gamma_sum += gamma_k
                gamma_x_sum += gamma_k * x
                ergodic_mean = gamma_x_sum / gamma_sum
                ergodic_loss = compute_lasso_loss(A, ergodic_mean, y, lam, n)
                loss_history[-1] = ergodic_loss  # overwrite with ergodic loss

            # Update progress bar
            pbar.update(1)

    if is_ergodic:
        x = gamma_x_sum / gamma_sum

    return x, loss_history


def randomized_coordinate_proximal_gradient_algorithm(x_init, y, n, d, A, lam,
    num_iterations):
    """
    Randomized Coordinate Proximal Gradient Algorithm (RCPGA) for solving the Lasso
        problem.

    Args:
        x_init (np.ndarray): The initial solution vector.
        y (np.ndarray): The target vector.
        n (int): Number of data points.
        d (int): Number of features.
        A (np.ndarray): The design matrix.
        lam (float): The regularization parameter.
        num_iterations (int): The number of iterations to run the algorithm.

    Returns:
        tuple: A tuple containing:
            - solution (np.ndarray): The estimated solution after the specified number
              of iterations.
            - loss_history (list): The history of the Lasso loss function over
              iterations.
    """
    x = x_init.copy()
    loss_history = []

    with tqdm(total=num_iterations, desc="RCPGA␣Progress", unit="iter") as pbar:
        for k in range(1, num_iterations + 1):
            # Randomly select a coordinate j_k uniformly from {0, ..., d-1}
            jk = np.random.randint(0, d)

            # Step size gamma_j
            gamma_j = n / np.sum(A[:, jk] ** 2)

            # Gradient w.r.t. x_j
            grad_j = (A[:, jk] @ (A @ x - y)) / n

            # Gradient step for coordinate j_k
            t = x[jk] - gamma_j * grad_j

            # Proximal operator (soft-thresholding)
            x[jk] = soft_thresholding(t, lam * gamma_j)

            # Compute and store loss
            current_loss = compute_lasso_loss(A, x, y, lam, n)
            loss_history.append(current_loss)

            # Update progress bar
            pbar.update(1)

    return x, loss_history


def plot_solution(x_estimated, x_true, algorithm_name, d):
    """
    Plots the estimated solution against the true sparse solution.

    Args:
        x_estimated (np.ndarray): The estimated solution.
        x_true (np.ndarray): The true sparse solution.
        algorithm_name (str): The name of the algorithm for the plot title.
        d (int): Number of features.
    """
    plt.figure(figsize=(10, 6), dpi=100)

    # Plot true non-zero elements
    true_nonzero = np.where(np.abs(x_true) > 0)[0]
    plt.stem(true_nonzero, x_true[true_nonzero], linefmt='b-', markerfmt='bo', basefmt='
        ␣', label='$x^*$')

    # Plot estimated non-zero elements
    estimated_nonzero = np.where(np.abs(x_estimated) > 1e-3)[0]
    plt.stem(estimated_nonzero, x_estimated[estimated_nonzero], linefmt='k--', markerfmt
        ='k^', basefmt='␣', label=r'$x_{\gamma,␣\lambda}$')

    plt.axhline(0.0, color='red', linestyle='--', linewidth=1)
    plt.xlim([-10, d + 10])
    plt.ylim([-1.1, 1.1])
    plt.xlabel("Feature␣index␣$i$")
    plt.ylabel("$x_i$")
    plt.title(f"Sparse␣Solution␣vs␣Estimated␣Solution␣({algorithm_name})")
    plt.legend()
    plt.tight_layout()
    plt.savefig(f"{algorithm_name}_solution.png")
    plt.close()


def plot_loss(loss_histories, legends, baseline_obj, algorithm_name):
    """
    Plots the loss history during the optimization process.

    Args:
        baseline_obj:  The baseline objective function value.
        loss_histories (list of lists): The history of the Lasso loss for different runs
            .
        legends (list of str): Labels for the loss curves.
        algorithm_name (str): The name of the algorithm for the plot title.
    """
    plt.figure(figsize=(10, 6), dpi=100)
    for loss, label in zip(loss_histories, legends):
        plt.plot(loss, label=label, linewidth=2)
    plt.axhline(
    y=baseline_obj,
    color=’red’,
    linewidth=2,
    linestyle=’--’,
    label=f"Baseline␣(true␣x)␣=␣{baseline_obj:.6f}"
    )
    plt.xlabel("Iteration")
    plt.ylabel("Objective␣function")
    plt.title(f"Loss␣vs␣Iterations␣({algorithm_name})")
    plt.legend()
    plt.tight_layout()
    plt.savefig(f"{algorithm_name}_loss.png")
    plt.close()


def main():
    # Generate synthetic data
    n = 1000
    d = 500
    s = 50
    std = 0.06
    true_x, A, y = generate_lasso_problem(n, d, s, std)

    # Compute baseline objective function value using true x
    baseline_obj = compute_lasso_loss(A, true_x, y, lam=0.01, n=n)
    print(f"Baseline␣objective␣with␣true␣x:␣{baseline_obj:.6f}")

    # ------------------------
    # Proximal Stochastic Gradient Algorithm (PSGA)
    # ------------------------
    max_iters_psga = 1000000  # Max number of iterations for PSGA
    lam_psga = 0.05  # Regularization parameter for PSGA
    x0_psga = np.random.randn(d)  # Initialization of x0

    # Run PSGA without ergodic mean
    print("Running␣PSGA␣without␣ergodic␣mean...")
    estimated_x_psga, loss_history_psga = proximal_stochastic_gradient_algorithm(
        x_init=x0_psga,
        y=y,
        n=n,
        A=A,
        lam=lam_psga,
        num_iterations=max_iters_psga,
        is_ergodic=False
    )

    # Run PSGA with ergodic mean
    print("Running␣PSGA␣with␣ergodic␣mean...")
    estimated_x_psga_ergo, loss_history_psga_ergo =
        proximal_stochastic_gradient_algorithm(
        x_init=x0_psga,
        y=y,
        n=n,
        A=A,
        lam=lam_psga,
        num_iterations=max_iters_psga,
        is_ergodic=True
    )

    # Plot PSGAs loss histories
    plot_loss(
        loss_histories=[loss_history_psga, loss_history_psga_ergo],
        baseline_obj = baseline_obj,
        legends=["PSGA", "PSGA␣Ergodic"],
        algorithm_name="PSGA"
    )

    # Plot PSGA estimated solution vs true solution
    plot_solution(
        x_estimated=estimated_x_psga,
        x_true=true_x,
        algorithm_name="PSGA",
        d=d
    )

    # Plot PSGA with ergodic x estimated solution vs true solution
    plot_solution(
        x_estimated=estimated_x_psga_ergo,
        x_true=true_x,
        algorithm_name="PSGA_Ergodic",
        d=d
    )

    # ------------------------
    # Randomized Coordinate Proximal Gradient Algorithm (RCPGA)
    # ------------------------
    max_iters_rcpga = 10000  # Max number of iterations for RCPGA
    lam_rcpga = 0.05  # Regularization parameter for RCPGA
    x0_rcpga = np.random.randn(d)  # Initialization of x0

    # Run RCPGA
    print("Running␣RCPGA...")
    estimated_x_rcpga, loss_history_rcpga =
        randomized_coordinate_proximal_gradient_algorithm(
        x_init=x0_rcpga,
        y=y,
        n=n,
        d=d,
        A=A,
        lam=lam_rcpga,
        num_iterations=max_iters_rcpga
    )

    # Plot RCPGA loss history
    plot_loss(
        loss_histories=[loss_history_rcpga],
        baseline_obj = baseline_obj,
        legends=["RCPGA"],
        algorithm_name="RCPGA"
    )

    # Plot RCPGA estimated solution vs true solution
    plot_solution(
        x_estimated=estimated_x_rcpga,
        x_true=true_x,
        algorithm_name="RCPGA",
        d=d
    )

    # ------------------------
    # Combined Loss Plot (Optional)
    # ------------------------
    plt.figure(figsize=(10, 6), dpi=100)
    plt.plot(loss_history_psga, label='PSGA', linewidth=2)
    plt.plot(loss_history_psga_ergo, label='PSGA␣Ergodic', linewidth=2)
    plt.plot(loss_history_rcpga, label='RCPGA', linewidth=2)
    plt.axhline(y=baseline_obj, color='red', linestyle='--', linewidth=2, label='
        Baseline␣(true␣x)')
    plt.xlabel("Iteration")
    plt.ylabel("Objective␣function")
    plt.title("Loss␣vs␣Iterations␣for␣PSGA␣and␣RCPGA")
    plt.legend()
    plt.tight_layout()
    plt.savefig("Combined_Loss.png")
    plt.close()


if __name__ == "__main__":
    main()
```