CS498 Homework 5

February 24, 2026

1 Problem 1: Convex Optimization Toolkit (Gradescope — 100 points)

Before we optimize anything, we need to know what "nice" problems look like. In this problem you will implement core convex optimization building blocks: computing gradients, checking convexity, running gradient descent, and solving Ridge regression. These will be your tools for the rest of the homework.

1.1 Background

Recall: A function $f : \mathbb{R}^n \to \mathbb{R}$ is **convex** if for all $x, y \in \mathbb{R}^n$ and all $\lambda \in [0, 1]$:

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

If f is twice differentiable, then f is convex if and only if the Hessian $\nabla^2 f(x) \succeq 0$ (positive semidefinite) for all x.

The **gradient** $\nabla f(x)$ gives the direction of steepest ascent. At a minimum of a differentiable convex function, $\nabla f(x^*) = 0$.

1.2 What You Must Implement

Submit a file named **`hw5_p1_convex.py`** with the following functions.

Important: You must derive the gradients yourself. Use calculus (matrix calculus, chain rule, etc.).

```
import numpy as np

def gradient_quadratic(A, b, x):
    """
    Compute the gradient of f(x) = (1/2) x^T A x + b^T x with respect to x.

    Parameters
    ----------
    A : np.ndarray, shape (n, n)
        Symmetric matrix.
    b : np.ndarray, shape (n,)
        Linear coefficient vector.
```

```
    x : np.ndarray, shape (n,)
        Point at which to evaluate the gradient.

    Returns
    -------
    grad : np.ndarray, shape (n,)
    """
    raise NotImplementedError


def gradient_least_squares(X, y, w):
    """
    Compute the gradient of f(w) = (1/(2n)) ||Xw - y||^2 with respect to w.

    Parameters
    ----------
    X : np.ndarray, shape (n, p)
        Data matrix (n samples, p features).
    y : np.ndarray, shape (n,)
        Target vector.
    w : np.ndarray, shape (p,)
        Weight vector.

    Returns
    -------
    grad : np.ndarray, shape (p,)
    """
    raise NotImplementedError


def gradient_ridge(X, y, w, lam):
    """
    Compute the gradient of the Ridge regression objective:
        f(w) = (1/(2n)) ||Xw - y||^2 + (lam/2) ||w||^2
    with respect to w.

    Parameters
    ----------
    X : np.ndarray, shape (n, p)
    y : np.ndarray, shape (n,)
    w : np.ndarray, shape (p,)
    lam : float
        Regularization parameter (>= 0).

    Returns
    -------
    grad : np.ndarray, shape (p,)
    """
```

```
    raise NotImplementedError


def gradient_smooth_2d(x, y, lam):
    """
    Compute the gradient of:

        f(x) = (1/2)||x - y||_F^2
               + lam * sum of (x[i,j] - x[i',j'])^2
                         over all horizontally and vertically adjacent pairs

    where x and y are 2D arrays of shape (H, W), and adjacent means
    pairs (i,j)-(i+1,j) and (i,j)-(i,j+1).

    Parameters
    ----------
    x : np.ndarray, shape (H, W)
    y : np.ndarray, shape (H, W)
    lam : float
        Weight on the neighbor-difference penalty (>= 0).

    Returns
    -------
    grad : np.ndarray, shape (H, W)

    Hints
    -----
    1. The gradient of (1/2)||x - y||_F^2 with respect to x is simply ???.

    2. For the smoothness term, each pixel x[i,j] appears in multiple
       neighbor-difference terms. Be careful to account for *all*
       contributions involving x[i,j].

    3. For an interior pixel, x[i,j] interacts with up to four neighbors:
           (i-1,j), (i+1,j), (i,j-1), (i,j+1).
       Think about how many times x[i,j] appears in squared differences
       and what derivative each contributes.

    4. Boundary pixels have fewer neighbors, handle edges carefully.

    You may find it helpful to derive the gradient for a single interior
    pixel first before implementing it vectorized.

    There is a very beautiful vectorized solution for this.
    """
    raise NotImplementedError
```

```python
def check_convexity_quadratic(A):
    """
    Determine whether f(x) = (1/2) x^T A x + b^T x + c is convex
    for ANY b and c. (This depends only on A.)

    Parameters
    ----------
    A : np.ndarray, shape (n, n)
        Symmetric matrix.

    Returns
    -------
    is_convex : bool
        True if the quadratic is convex (for all b, c), False otherwise.
    min_eigenvalue : float
        The smallest eigenvalue of A. (Hint: Hint)
    """
    raise NotImplementedError


def lipschitz_constant_least_squares(X):
    """
    Compute the Lipschitz constant of the gradient of
        f(w) = (1/(2n)) ||Xw - y||^2.

    The Lipschitz constant L satisfies:
        ||grad f(w1) - grad f(w2)|| <= L ||w1 - w2||   for all w1, w2.

    This determines the maximum safe learning rate: lr = 1/L guarantees
    that gradient descent will not diverge.

    Hint: The gradient of f is a linear function of w. What is the
    operator norm of that linear map? (look it up!)

    Parameters
    ----------
    X : np.ndarray, shape (n, p)

    Returns
    -------
    L : float
        The Lipschitz constant.
    """
    raise NotImplementedError


def gradient_descent_quadratic(A, b, x0, lr, num_iters):
    """
```

```
    Run gradient descent on f(x) = (1/2) x^T A x + b^T x.

    Update rule: x_{t+1} = x_t - lr * grad f(x_t)

    Parameters
    ----------
    A : np.ndarray, shape (n, n)
        Symmetric PSD matrix.
    b : np.ndarray, shape (n,)
    x0 : np.ndarray, shape (n,)
        Starting point.
    lr : float
        Learning rate.
    num_iters : int
        Number of iterations to run.

    Returns
    -------
    result : dict with keys:
        'x_final'    : np.ndarray, shape (n,)  - final iterate x_{num_iters}
        'f_final'    : float                    - f evaluated at the final iterate
        'trajectory' : list of np.ndarray       - [x_0, x_1, ..., x_{num_iters}]
                                                  (length num_iters + 1)
        'objectives' : list of float            - [f(x_0), f(x_1), ..., f(x_{num_iters})]
                                                  (length num_iters + 1)
    """
    raise NotImplementedError


def solve_ridge_closed_form(X, y, lam):
    """
    Solve Ridge regression in closed form.

    The Ridge objective is:
        min_w  (1/(2n)) ||Xw - y||^2 + (lam/2) ||w||^2

    Derive the closed-form solution by setting the gradient to zero.
    Consider looking up numpy.linalg.solve.

    Parameters
    ----------
    X : np.ndarray, shape (n, p)
    y : np.ndarray, shape (n,)
    lam : float (>= 0)

    Returns
    -------
    w : np.ndarray, shape (p,)
```

```
        Optimal weight vector.
    """
    raise NotImplementedError


def solve_ridge_gd(X, y, lam, lr=None, num_iters=1000, tol=1e-8):
    """
    Solve Ridge regression via gradient descent.

    The Ridge objective is:
        min_w  (1/(2n)) ||Xw - y||^2 + (lam/2) ||w||^2

    If lr is None, choose a safe learning rate automatically.
    (Hint: what is the Lipschitz constant of the Ridge gradient?)

    Initialize w = 0.

    Stop early if ||grad f(w)||_2 < tol.

    Parameters
    ----------
    X : np.ndarray, shape (n, p)
    y : np.ndarray, shape (n,)
    lam : float (>= 0)
    lr : float or None
        Learning rate. If None, choose automatically use 1/(Lipschitz Constant) of gradient.
    num_iters : int
    tol : float

    Returns
    -------
    result : dict with keys:
        'w'          : np.ndarray, shape (p,)  - final weight vector
        'num_iters' : int                       - iterations actually performed
        'converged' : bool                      - True if stopped early (||grad|| < tol)
        'objectives': list of float             - objective value recorded at each iteration
    """
    raise NotImplementedError
```

1.3 Hints

- For `gradient_quadratic`: recall from multivariable calculus how to differentiate $x^T Ax$.
- For `gradient_least_squares`: expand the squared norm and differentiate with respect to w.
- For `gradient_ridge`: combine your answers from the previous two.
- For `gradient_smooth_2d`: the data fidelity term $(1/2)\|x - y\|_F^2$ is easy. For the neighbor penalty, think about what each pixel $x_{i,j}$ contributes: it appears in differences with each of its (up to 4) neighbors. You can compute this efficiently using array slicing (e.g., `x[1:,:] -`

`x[:-1,:]`).

- For `check_convexity_quadratic`: think about when a quadratic form is convex. What property of A determines this?
- For `lipschitz_constant_least_squares`: the gradient of f is a linear function of w. The Lipschitz constant is the operator norm of the corresponding matrix.
- For `solve_ridge_closed_form`: set $\nabla f(w) = 0$ and solve for w.
- For `solve_ridge_gd`: use your gradient function and iterate. The Lipschitz constant of the Ridge gradient is $L + \lambda$ where L is the Lipschitz constant of the least-squares gradient.

2 Problem 2: Inside PyTorch: Custom Autograd Functions (Gradescope — 100 points)

*PyTorch's autograd computes gradients automatically, but how? Under the hood, every operation defines a **forward pass** (compute the output) and a **backward pass** (compute the gradient). In this problem you'll build three custom autograd operations from scratch, then compose them into larger computational graphs. By the end, you'll understand exactly what happens when you call* `.backward()`.

2.1 Background: `torch.autograd.Function`

PyTorch lets you define custom differentiable operations by subclassing `torch.autograd.Function`. You implement two static methods:

- **`forward(ctx, ...)`**: Compute the output. Save anything needed for the backward pass using `ctx.save_for_backward(...)`.
- **`backward(ctx, grad_output)`**: Given the upstream gradient `grad_output` (the chain rule factor from later in the graph), compute and return the gradient with respect to each input.

Here's a simple example, a custom "scaled square" function:

```
import torch

class ScaledSquare(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, a):
        # Save what we need for backward
        ctx.save_for_backward(x)
        ctx.a = a  # treat 'a' as a constant (no grad)
        return a * x ** 2 # f(x)=ax^2

    @staticmethod
    def backward(ctx, grad_output):
        x, = ctx.saved_tensors
        a = ctx.a

        # y = a * x^2
        # dy/dx = a * 2x
        grad_x = grad_output * (a * 2 * x)

        # We are NOT computing gradients w.r.t. 'a' (hyperparameter),
        # so return None for that slot.
        return grad_x, None

# Usage:
x = torch.tensor([3.0], requires_grad=True)
a = 5.0  # constant scale (not requires_grad)

y = ScaledSquare.apply(x, a)  # forward: y = 5 * 9 = 45
```

```
y.backward()

print("y:", y)          # tensor([45.])
print("x.grad:", x.grad)  # dy/dx = 5 * 2 * 3 = 30 -> tensor([30.])
```

The key idea in backward: `grad_output` carries the chain rule from everything *downstream*. Your backward just multiplies by the local derivative and passes it along.

2.2 What You Must Implement

Submit a file named **hw5_p2_autograd.py** with the following three classes and one function.

```
import torch
import numpy as np


class SmoothPenalty2D(torch.autograd.Function):
    """
    Computes the neighbor-difference penalty on a 2D grid:
        f(x) = lam * sum of (x[i,j] - x[i',j'])^2
               over all horizontally and vertically adjacent pairs

    where adjacent means pairs (i,j)-(i+1,j) and (i,j)-(i,j+1).

    Forward
    -------
    Inputs:
        x   : torch.Tensor, shape (H, W)
        lam : float (passed as a Python float, not a tensor)

    Output:
        scalar torch.Tensor - the penalty value

    Backward
    --------
    Returns gradient with respect to x only (lam is not a tensor).
    The gradient has shape (H, W).

    Hint: You derived this gradient by hand in Problem 1 (gradient_smooth_2d).
    """

    @staticmethod
    def forward(ctx, x, lam):
        raise NotImplementedError

    @staticmethod
    def backward(ctx, grad_output):
        raise NotImplementedError
```

```
class HuberLoss(torch.autograd.Function):
    """
    Computes the sum of element-wise Huber losses:

        f(x) = sum_i h_delta(x_i)

    where the Huber function is:

        h_delta(r) = r^2 / 2              if |r| <= delta
                     delta*(|r| - delta/2) if |r| > delta

    This is a smooth approximation to the absolute value: quadratic near
    zero, linear in the tails. It's widely used in robust regression because
    it's less sensitive to outliers than squared loss.

    Forward
    -------
    Inputs:
        x     : torch.Tensor, any shape
        delta : float (passed as a Python float, not a tensor)

    Output:
        scalar torch.Tensor - the sum of Huber losses

    Backward
    --------
    Returns gradient with respect to x only.

    The derivative of h_delta(r) is:
        r                 if |r| <= delta
        delta * sign(r)   if |r| > delta

    (This is just clipping the derivative to [-delta, delta].)
    """

    @staticmethod
    def forward(ctx, x, delta):
        raise NotImplementedError

    @staticmethod
    def backward(ctx, grad_output):
        raise NotImplementedError


class LogSumExp(torch.autograd.Function):
    """
    Computes the log-sum-exp of a vector:
```

```
        f(x) = log( sum_i exp(x_i) )

    This function appears everywhere in machine learning (softmax,
    cross-entropy, convex relaxations).

    NUMERICAL STABILITY: A naive implementation will overflow for large x_i.
    Use the standard trick:

        log(sum_i exp(x_i)) = m + log(sum exp(x_i - m))

    where m = max(x).

    Forward
    -------
    Inputs:
        x : torch.Tensor, shape (n,)  - a 1D vector

    Output:
        scalar torch.Tensor - the log-sum-exp value

    Backward
    --------
    Returns gradient with respect to x.

    Hint: The gradient of log(sum exp(x_i)) with respect to x_j is
    exp(x_j) / sum_i exp(x_i). Use the same numerical
    stability trick.
    """

    @staticmethod
    def forward(ctx, x):
        raise NotImplementedError

    @staticmethod
    def backward(ctx, grad_output):
        raise NotImplementedError


def compose_and_gradient(A, x, b, delta, lam, H, W):
    """
    Build a computation graph that composes your custom functions with
    standard PyTorch operations, then compute the gradient of the total
    loss with respect to x.

    The computation graph is:

        residual = A @ x + b          (standard PyTorch matmul + add)
```

```
    loss1 = HuberLoss(residual, delta)     (your custom node)
    loss2 = LogSumExp(x)                   (your custom node)
    z = x[:H*W].reshape(H, W)             (standard reshape)
    loss3 = SmoothPenalty2D(z, lam)        (your custom node)
    total = loss1 + loss2 + loss3

Then compute d(total)/dx via backpropagation.

Parameters
----------
A : np.ndarray, shape (m, n)
x : np.ndarray, shape (n,) Initial values of x tensor.
b : np.ndarray, shape (m,)
delta : float
lam : float
H : int
W : int
    Must satisfy H * W <= n. We take the first H*W entries of x
    for the 2D penalty.

Returns
-------
grad : np.ndarray, shape (n,)
    The gradient d(total)/dx, as a numpy array.
total : float
    The total loss value.
"""
raise NotImplementedError
```

2.3 Implementation Hints

2.3.1 `SmoothPenalty2D`

- **Forward:** Compute horizontal differences `x[:, 1:] - x[:, :-1]` and vertical differences `x[1:, :] - x[:-1, :]`, square them, sum, and multiply by `lam`. Save `x` and `lam` for backward.
- **Backward:** You derived this in Problem 1. Multiply your local gradient by `grad_output` (the upstream scalar gradient).

2.3.2 `HuberLoss`

- **Forward:** Use `torch.where` to compute the piecewise function. Save `x` and `delta` for backward.
- **Backward:** Hint: `torch.clamp(....)`. Multiply by `grad_output`.

2.3.3 `LogSumExp`

- **Forward:** Compute `m = x.max()`, then `m + torch.log(torch.sum(torch.exp(x - m)))`. Save the softmax probabilities for backward.

- **Backward:** The gradient is the softmax vector `exp(x - m) / sum(exp(x - m))`. Multiply by `grad_output`.

2.3.4 `compose_and_gradient`

- Convert all numpy inputs to `torch.float64` tensors. Make `x` require gradients.
- Build the graph using standard torch ops + your `.apply()` calls.
- Call `total.backward()`, then return `x.grad.numpy()` and `total.item()`.

2.4 Sanity Checks

```
# Test SmoothPenalty2D with gradcheck
x = torch.randn(4, 5, dtype=torch.float64, requires_grad=True)
torch.autograd.gradcheck(lambda x: SmoothPenalty2D.apply(x, 2.0), (x,))

# Test HuberLoss with gradcheck
x = torch.randn(10, dtype=torch.float64, requires_grad=True)
torch.autograd.gradcheck(lambda x: HuberLoss.apply(x, 1.0), (x,))

# Test LogSumExp with gradcheck
x = torch.randn(8, dtype=torch.float64, requires_grad=True)
torch.autograd.gradcheck(lambda x: LogSumExp.apply(x), (x,))

# Test numerical stability of LogSumExp
x_big = torch.tensor([1000.0, 1001.0, 1002.0], dtype=torch.float64)
result = LogSumExp.apply(x_big)
print(result)  # Should be ~1002.41, NOT inf

# Test compose_and_gradient
A = np.random.randn(6, 10)
x = np.random.randn(10)
b = np.random.randn(6)
grad, total = compose_and_gradient(A, x, b, delta=1.0, lam=0.5, H=2, W=3)
print(f"Total loss: {total:.4f}")
print(f"Gradient shape: {grad.shape}")  # (10,)
```

3 Problem 3: Lasso Regression via Proximal Gradient Descent (Gradescope — 100 points)

*The Lasso (ℓ_1-regularized regression) is one of the most important tools in statistics and machine learning. Unlike Ridge regression, Lasso promotes **sparsity** — it drives some coefficients exactly to zero, performing automatic feature selection. In this problem you'll implement a Lasso solver from scratch.*

3.1 Background

The **Lasso** objective is:

$$\min_w \quad \frac{1}{2n}\|Xw - y\|_2^2 + \lambda\|w\|_1$$

where: - $X \in \mathbb{R}^{n\times p}$ is the data matrix (n samples, p features), - $y \in \mathbb{R}^n$ is the response vector, - $w \in \mathbb{R}^p$ are the regression coefficients, - $\lambda > 0$ is the regularization parameter, - $\|w\|_1 = \sum_{j=1}^p |w_j|$.

This objective is **convex** (why?). However, it is **not differentiable** at points where some $w_j = 0$, because $|w_j|$ has a kink there. So plain gradient descent doesn't directly apply.

3.2 The Proximal Gradient Method (ISTA)

We want to minimize an objective of the form

$$\min_w \; F(w) = f(w) + g(w)$$

where:

- $f(w) = \frac{1}{2n}\|Xw - y\|_2^2$ — smooth and differentiable
- $g(w) = \lambda\|w\|_1$ — convex but non-differentiable

Because of the non-smooth ℓ_1 term, ordinary gradient descent does not apply directly. The solution is proximal gradient descent, also known as ISTA (Iterative Shrinkage-Thresholding Algorithm).

1. Definition of the Proximal Operator

For a convex function g, the proximal operator with parameter $\alpha > 0$ is defined as

$$\text{prox}_{\alpha g}(v) = \arg\min_w \left\{ g(w) + \frac{1}{2\alpha}\|w - v\|_2^2 \right\}$$

It finds a point w that keeps $g(w)$ small and stays close to v.
The quadratic term penalizes moving too far from v.
You can think of it as a regularized projection step.

2. Proximal Gradient Update

Each iteration performs:

Step 1: Gradient step on smooth part

$$v = w^{(t)} - \alpha \nabla f(w^{(t)})$$

Step 2 — Proximal step on non-smooth part

$$w^{(t+1)} = \text{prox}_{\alpha g}(v)$$

So we solve

$$w^{(t+1)} = \arg\min_{w} \left\{ \lambda \|w\|_1 + \frac{1}{2\alpha} \|w - v\|_2^2 \right\}$$

3. Derivation of the Proximal Operator for $g(w) = \lambda \|w\|_1$

The objective separates across coordinates because

$$\|w\|_1 = \sum_j |w_j|, \qquad \|w - v\|_2^2 = \sum_j (w_j - v_j)^2$$

Thus the minimization decouples coordinate-wise:

$$w_j^{(t+1)} = \arg\min_{w_j} \left\{ \lambda |w_j| + \frac{1}{2\alpha} (w_j - v_j)^2 \right\}$$

Your first task is to do a case analysis on whether $w_j > 0$, $w_j < 0$, or $w_j = 0$, and compute the proximal projection. For example: Now solve this one-dimensional problem.

Case 1: $w_j > 0$, then $|w_j| = w_j$ and the objective becomes

$$\lambda w_j + \frac{1}{2\alpha} (w_j - v_j)^2$$

Taking the derivative:

$$\lambda + \frac{1}{\alpha} (w_j - v_j) = 0$$

Solving:

$$w_j = v_j - \alpha\lambda$$

Valid when $v_j > \alpha\lambda$.

Continue other cases….

4. Final Result: Soft-Thresholding

$$\text{prox}_{\alpha\lambda\|\cdot\|_1}(v)_j = ???$$

This operator is called soft-thresholding. The threshold in the case of lasso regression is equal to $\alpha \cdot \lambda$.

5. Big Picture Insight

Proximal gradient descent is essentially a gradient step, plus a structured proximal projection step. The gradient step reduces data misfit. The proximal step enforces the regularizer.

For ℓ_1, the proximal operator produces automatic feature selection through soft-thresholding. ## What You Must Implement

Submit a file named **hw5_p3_lasso.py** with the following two functions.

```
import numpy as np


def soft_threshold(v, threshold):
    """
    Apply the soft-thresholding (shrinkage) operator element-wise:

        S(v, t)_j = ???

    Parameters
    ----------
    v : np.ndarray
        Input array, any shape.
    threshold : float
        Non-negative threshold value. For lasso, it is \alpha (or learning rate) times lambda.

    Returns
    -------
    np.ndarray, same shape as v.
    """
    raise NotImplementedError


def lasso_proximal_gd(X, y, lam, lr=None, num_iters=1000, tol=1e-6,
                      w_init=None):
    """
    Solve the Lasso problem using proximal gradient descent (ISTA).

        min_w  (1/(2n)) ||Xw - y||^2  +  lam * ||w||_1

    Algorithm (each iteration):
        1. Compute the gradient of the smooth part f(w) = (1/(2n))||Xw - y||^2.
        2. Take a gradient step:  v = w - lr * grad_f(w).
        3. Apply soft-thresholding:  w_new = soft_threshold(v, lr * lam).
```

```
    4. Check convergence.

Parameters
----------
X : np.ndarray, shape (n, p)
    Feature matrix.
y : np.ndarray, shape (n,)
    Response vector.
lam : float
    Regularization parameter (>= 0).
lr : float or None
    Step size. If None, set lr = 1/L where L is the largest eigenvalue
    of (X^T X / n). This is the Lipschitz constant of grad f and
    guarantees convergence.
num_iters : int
    Maximum number of iterations.
tol : float
    Stop early if  max_j |w_new_j - w_old_j| < tol  (infinity-norm
    of the change in w).
w_init : np.ndarray, shape (p,), or None
    Initial coefficients. If None, initialize to zeros.

Returns
-------
result : dict with keys:
    'w'         : np.ndarray, shape (p,) - final coefficient vector
    'objective' : list of float - the Lasso objective value at each
                  iteration (BEFORE the update), length <= num_iters
    'num_iters' : int - number of iterations actually performed
    'converged' : bool - True if stopped early because the change
                  in w fell below tol
"""
raise NotImplementedError
```

3.3 Implementation Hints

3.3.1 `soft_threshold`

This is a one-liner. Think `np.sign` and `np.maximum`.

3.3.2 `lasso_proximal_gd`

Precompute for efficiency:

```
XtX = X^T X / n
Xty = X^T y / n
```

Then the gradient of the smooth part at any w is just `XtX @ w - Xty`: no need to recompute $X^T X$ every iteration.

Step size: If `lr` is `None`, compute $L = \lambda_{\max}(X^T X/n)$ using `np.linalg.eigvalsh` and set `lr = 1/L`.

Objective value: At each iteration (before updating), record:

$$\text{obj} = \frac{1}{2n}\|Xw - y\|_2^2 + \lambda\|w\|_1$$

Convergence: Stop if $\|w_{\text{new}} - w_{\text{old}}\|_\infty < \text{tol}$.

3.4 Sanity Checks

```
np.random.seed(0)
n, p = 100, 5
X = np.random.randn(n, p)
w_true = np.array([3.0, -2.0, 0.0, 0.0, 1.5])
y = X @ w_true + 0.1 * np.random.randn(n)

# Test 1: lam=0 should give OLS
result = lasso_proximal_gd(X, y, lam=0.0, num_iters=5000)
ols = np.linalg.lstsq(X, y, rcond=None)[0]
print("Lasso (lam=0):", np.round(result['w'], 4))
print("OLS:          ", np.round(ols, 4))
# Should be very close

# Test 2: Large lam should give all zeros
result = lasso_proximal_gd(X, y, lam=10.0)
print("Lasso (lam=10):", result['w'])
# Should be all zeros

# Test 3: Moderate lam should produce a sparse solution
result = lasso_proximal_gd(X, y, lam=0.05)
print("Lasso (lam=0.05):", np.round(result['w'], 4))
# Should zero out features 2 and 3 (the true zeros)
```

4 Problem 4: Frank–Wolfe via Linear Programming (Gradescope — 100 points)

In this problem, you will implement the Frank–Wolfe (FW) algorithm to minimize a smooth convex function over a polytope defined by linear constraints.

4.1 The Optimization Problem

We consider

$$\min_{x \in \mathcal{D}} f(x)$$

where

$$\mathcal{D} = \{x \in \mathbb{R}^n : Ax \leq b,\ Ex = d,\ \ell \leq x \leq u\}$$

is a nonempty compact polytope, and

$$f : \mathbb{R}^n \to \mathbb{R}$$

is convex, differentiable, and has a gradient oracle.

In code, we will treat

$$f$$

as:

- a Python function `f(x)` that returns a scalar
- a Python function `grad_f(x)` that returns a NumPy array of shape `(n,)` which is the gradient of f with respect to x.

The algorithm is not allowed to assume any special structure of

$$f$$

beyond smoothness, convexity, and differentiability.

4.2 What Unconstrained Gradient Descent Would Like To Do

If there were no constraints, steepest descent takes:

$$x_{t+1} = x_t - \eta \nabla f(x_t).$$

This is locally optimal because

$$-\nabla f(x_t)$$

is the direction of fastest decrease in Euclidean geometry.

But with constraints, this update generally leaves

$$\mathcal{D}$$

.

A standard fix is projected gradient descent:

$$x_{t+1} = \Pi_{\mathcal{D}}(x_t - \eta \nabla f(x_t)).$$

The problem is that projection onto a general polytope is an optimization problem:

$$\Pi_{\mathcal{D}}(v) \in \arg\min_{z \in \mathcal{D}} \|z - v\|_2^2.$$

That is a quadratic program. Doing that at every iteration is expensive, and it defeats the point of "cheap" first-order methods.

Frank–Wolfe avoids projection completely.

4.3 The Geometric Insight Behind Frank–Wolfe

At the current iterate

$$x_t$$

, convexity and differentiability give the supporting hyperplane inequality:

$$f(s) \geq f(x_t) + \langle \nabla f(x_t), s - x_t \rangle.$$

Interpretation:

- The function lies above its tangent plane at

$$x_t$$

  .
- So if you want to find a feasible point that looks promising, minimize the tangent plane instead of the true function.

Since

$$f(x_t)$$

is constant w.r.t.

$$s$$

, the "best point according to the tangent plane" solves:

$$\arg\min_{s \in \mathcal{D}} \langle \nabla f(x_t), s - x_t \rangle$$

which is equivalent to

$$s_t \in \arg\min_{s \in \mathcal{D}} \langle s, \nabla f(x_t) \rangle.$$

Geometric picture:

- $$g_t = \nabla f(x_t)$$
  defines a direction.
- The linear functional
  $$\langle s, g_t \rangle$$
  defines parallel hyperplanes.
- Minimizing
  $$\langle s, g_t \rangle$$
  over a polytope means sliding that hyperplane "downward" until it first touches
  $$\mathcal{D}$$
  .
- A linear objective over a polytope always attains its optimum at an extreme point (a vertex).
- That touching point is
  $$s_t$$
  .

That point

$$s_t$$

is the "most downhill" feasible point for the *linearized* problem.

4.4 Why We Need the LMO (And Why It Must Be an LP)

Frank–Wolfe's whole trick is:

- Do NOT step outside
  $$\mathcal{D}$$
  and then project back (projection is hard).
- Instead, pick a feasible "target" point inside
  $$\mathcal{D}$$
  that is best according to the local linear approximation.
- Then move toward it using a convex combination.

The target point comes from the Linear Minimization Oracle (LMO):

$$s_t = \arg\min_{s \in \mathcal{D}} \langle s, \nabla f(x_t) \rangle.$$

This is an LP because:

- the objective is linear in
  $$s$$
- the constraints describing
  $$\mathcal{D}$$
  are linear

So each FW iteration is "one LP + one convex combination update".

The LMO is necessary because without it, you do not get a guaranteed feasible descent direction without projection. The feasible descent direction in Frank–Wolfe is

$$d_t = s_t - x_t.$$

If

$$x_t$$

is not optimal, this direction is guaranteed to decrease the function for small enough step sizes (because it decreases the tangent plane).

4.5 The Frank–Wolfe Update

Once you have

$$s_t$$

, update:

$$x_{t+1} = (1 - \gamma_t)x_t + \gamma_t s_t$$

with

$$\gamma_t \in [0, 1]$$

.

Because

$$\mathcal{D}$$

is convex,

$$x_{t+1}$$

remains feasible automatically.

No projection is ever performed.

4.6 Step Size Rule Used in This Homework

You will use the classic diminishing step size:

$$\gamma_t = \frac{2}{t + 2}.$$

This guarantees convergence for smooth convex functions over compact convex sets.

4.7 What You Must Implement

Submit: `hw5_p4_frank_wolfe.py`

You must implement the following functions exactly.

4.8 1. solve_lmo_gurobi

```
import numpy as np

def solve_lmo_gurobi(grad, A, b, E, d, lb, ub):
    """
    Solves the Frank-Wolfe linear subproblem:

    $$
    \min_{s \in \mathcal{D}} \langle s, \text{grad} \rangle
    $$

    i.e.

    $$
    \min_s \sum_{i=0}^{n-1} s_i \, \text{grad}_i
    $$

    subject to

    $$
    A s \le b
    $$
    $$
    E s = d
    $$
    $$
    lb \le s \le ub
    $$

    WARNING: Remember Gurobi's default zero lower bound. When initializing a variable,
    use lb=-GRB.INFINITY otherwise lb<=s might not be enforced.

    Parameters:
        grad (np.ndarray): shape (n,)
        A (np.ndarray or None): shape (m,n) or None
        b (np.ndarray or None): shape (m,) or None
        E (np.ndarray or None): shape (p,n) or None
        d (np.ndarray or None): shape (p,) or None
        lb (np.ndarray or None): shape (n,) or None
        ub (np.ndarray or None): shape (n,) or None

    Returns:
        s (np.ndarray): optimal solution of shape (n,)
    """
    pass
```

Requirements:

- Build a new Gurobi model inside this function.
- Add n decision variables for s.
- Objective: minimize sum_i grad[i] * s[i].
- Add inequality constraints if A is not None.
- Add equality constraints if E is not None.
- Add bounds if provided.
- Set: model.Params.OutputFlag = 0
- Optimize.
- Return solution as NumPy array.
- You may assume the LP is feasible and bounded on all test cases.

Important: The autograder will check that you truly solve the LP using Gurobi here. If you bypass this, you will receive zero credit.

4.9 2. frank_wolfe

```
def frank_wolfe(f, grad_f,
                A, b, E, d, lb, ub,
                x0,
                num_iters=100):
    """
    Runs Frank-Wolfe on:

    $$
    \min_{x \in \mathcal{D}} f(x)
    $$

    using the step size:

    $$
    \gamma_t = \frac{2}{t+2}.
    $$

    Parameters:
        f (callable): f(x) -> float
        grad_f (callable): grad_f(x) -> np.ndarray shape (n,), gradient of f wrt x
        A, b, E, d, lb, ub: define D
        x0 (np.ndarray): initial feasible point, shape (n,)
        num_iters (int): number of FW iterations

    Returns:
        dict with keys:
            'x' : final iterate (np.ndarray shape (n,))
            'f' : list of objective values (length num_iters+1)
```

```
    """
    pass
```

Implementation requirements:

- Start with x = x0.copy().
- Initialize history list with f(x).
- For t in range(num_iters):
  1. Compute gradient g = grad_f(x).
  2. Solve LMO: s = solve_lmo_gurobi(g, A, b, E, d, lb, ub).
  3. Step size gamma = 2/(t+2).
  4. Update x = (1 - gamma)x + $gamma$s.
  5. Append f(x) to history.
- Return {'x': x, 'f': history}.

You may assume x0 is feasible. Do NOT project, and do NOT call Gurobi anywhere except inside solve_lmo_gurobi.

4.10 3. check_feasible

```
def check_feasible(x, A, b, E, d, lb, ub, tol=1e-8):
    """
    Returns True if x satisfies the constraints within tolerance.
    """
    pass
```

This helper makes debugging easier and will be used in some visible tests.

4.11 Autograding

You will be graded on:

- Correct LP formulation and correct Gurobi usage in solve_lmo_gurobi.
- Correct Frank–Wolfe iteration and step size.
- Correct use of f and grad_f as black boxes (no assumptions about structure).
- Numerical correctness within tolerance (1e-6 to 1e-8 depending on test).
- Deterministic behavior.

You may NOT:

- Use projections.
- Replace the LMO with heuristics.
- Solve the full problem directly using Gurobi (QP/NLP).
- Call Gurobi outside solve_lmo_gurobi.