CS498 Homework 6

March 9, 2026

1 Problem 1: Interior Point / Log Barrier for Portfolio Optimization (Gradescope — 100 points)

In Lecture 12 we saw the penalty method: replace each constraint $g_i(x) \leq 0$ with a penalty $\frac{\rho}{2}\max(0, g_i(x))^2$ added to the objective, and crank $\rho \to \infty$. It works, but there's a serious practical problem: as ρ grows the objective becomes increasingly ill-conditioned (think of a narrow, steep canyon), and gradient-based solvers struggle. Can we find a smoother way to keep iterates strictly inside the feasible region?

1.1 From Penalties to Barriers

The penalty method has a fundamental tension: we need ρ large enough that the penalty term dominates any constraint violations, but large ρ distorts the objective's curvature and slows convergence.

Key idea: Instead of punishing constraint *violations* from the outside, what if we placed a "wall" on the *inside* of each constraint boundary that repels iterates away from the boundary?

Consider the function $\phi(u) = -\log(-u)$ for $u < 0$:

- As $u \to 0^-$ (approaching the boundary from inside), $\phi(u) \to +\infty$
- Deep inside the feasible region ($u \ll 0$), $\phi(u)$ is small and well-behaved
- ϕ is smooth and convex on its domain $u < 0$

This is our **log barrier**. For a set of inequality constraints $g_i(x) \leq 0$, $i = 1, \dots, m$, we define:

$$B(x) = -\sum_{i=1}^{m} \log(-g_i(x))$$

This function is defined only when all constraints are strictly satisfied ($g_i(x) < 0$ for all i) — i.e., x is in the **strict interior** of the feasible region. As x approaches any constraint boundary, $B(x) \to +\infty$, creating a smooth "force field" that keeps iterates away from the boundary.

1.2 The Barrier Problem and the Central Path

Given the original constrained problem:

$$\min_x \ f(x) \quad \text{s.t.} \quad g_i(x) \leq 0, \ \ i = 1, \dots, m$$

we form the **barrier problem** for parameter $t > 0$:

$$\min_x \; t \cdot f(x) + B(x) = t \cdot f(x) - \sum_{i=1}^{m} \log(-g_i(x))$$

For any fixed $t > 0$, the barrier term keeps the minimizer $x^*(t)$ strictly inside the feasible region. As t increases:

- The objective term $t \cdot f(x)$ dominates: we care more about optimality
- The barrier term becomes relatively less important: we allow iterates closer to the boundary
- The minimizer $x^*(t)$ traces a smooth curve called the **central path**
- In the limit $t \to \infty$, $x^*(t)$ converges to the constrained optimum x^*

Duality gap bound: At any point on the central path, we have a certificate of suboptimality:

$$f(x^*(t)) - f(x^*) \leq \frac{m}{t}$$

where m is the number of inequality constraints. This tells us exactly how far we are from optimal!

1.3 Path-Following Interior Point Method

Rather than solving one hard problem (huge t), we solve a **sequence of easier problems** with increasing t:

Algorithm: 1. Start with a strictly feasible point $x^{(0)}$, initial $t = t_0 > 0$, and a growth factor $\mu > 1$ 2. **Inner loop (centering step):** In iteration $k+1$, starting from $x^{(k)}$, approximately minimize $t \cdot f(x) + B(x)$ using gradient descent. Call the result $x^{(k+1)}$. 3. **Outer loop:** Increase $t \leftarrow \mu \cdot t$ 4. **Stop** when $m/t < \epsilon$ (desired accuracy)

Why this works: Each centering step is a smooth unconstrained minimization (no ill-conditioning if t is moderate). By warm-starting each centering step from the previous solution, each step only needs a few iterations. The outer loop drives t up geometrically, so we converge in $O(\sqrt{m}\log(1/\epsilon))$ outer iterations.

1.4 What is Portfolio Optimization?

Before we dive into the implementation, let us step back and understand what we are actually optimizing.

The setup: You have money to invest across n assets (stocks). The fundamental question is: how should you split your money? If you put everything into one high-growth tech stock, you might make a lot of money, or lose a lot (F in the chat for everyone who shorted NVDA including me). If you spread your money across many safe bonds, you will not lose much, but you will not gain much either. Portfolio optimization is about finding the sweet spot.

Expected returns: Each asset i has an expected return μ_i: this is how much you expect to make per dollar invested. For example, $\mu_i = 0.08$ means you expect an 8% annual return on asset i. We estimate these from historical data.

Risk and covariance: Returns are uncertain. The risk of an asset is measured by how much its return varies over time (its variance). But here is the crucial insight: assets are *correlated*. When one tech stock goes down, other tech stocks tend to go down too. When interest rates rise, bond prices all fall together. This correlation structure is captured by the **covariance matrix** Σ, where Σ_{ij} measures how much assets i and j move together. The diagonal entries Σ_{ii} are the variances of individual assets, and the off-diagonal entries Σ_{ij} are the covariances between pairs.

The portfolio: Let $w \in \mathbb{R}^n$ be the vector of portfolio weights, where w_i is the fraction of your money invested in asset i. Then:

- The expected return of your portfolio is $\mu^\top w = \sum_i \mu_i w_i$
- The variance (risk) of your portfolio is $w^\top \Sigma w = \sum_{i,j} w_i \Sigma_{ij} w_j$

The optimization problem: Harry Markowitz's key insight (which won him the Nobel Prize in Economics) was to formalize the tradeoff between risk and return as a mathematical optimization problem:

$$\min_w \; w^\top \Sigma w \quad \text{s.t.} \quad \mu^\top w \geq r_{\min}, \;\; \sum_i w_i = 1, \;\; w_i \geq 0$$

Let us understand each piece:

- **The objective** $w^\top \Sigma w$ measures the **variance** (risk) of your portfolio. We want to minimize risk.
- **The constraint** $\mu^\top w \geq r_{\min}$ says "I want at least $r_{\min}$ expected return." You specify $r_{\min}$ based on your goals.
- **The constraint** $w_i \geq 0$ says "no short selling" — you cannot bet *against* a stock, only invest in it.
- **The constraint** $\sum_i w_i = 1$ says "invest all your money" — the weights must add up to 100%.

The tradeoff: If you set $r_{\min}$ very high, you are forcing the optimizer to put money into high-return (but risky) assets, which increases the portfolio variance. If you set $r_{\min}$ low, the optimizer can spread money across safe, low-correlation assets to minimize risk. This tradeoff between risk and return is the heart of modern portfolio theory.

1.5 Application: Log Barrier for Markowitz

You will apply the log barrier method to the Markowitz portfolio optimization problem stated above.

Where: - $w \in \mathbb{R}^n$ are the portfolio weights (fraction of wealth in each asset) - $\Sigma \in \mathbb{R}^{n \times n}$ is the covariance matrix of asset returns - $\mu \in \mathbb{R}^n$ is the vector of expected returns - $r_{\min}$ is the minimum required expected return

We will reformulate the equality constraint by substitution and handle the remaining inequality constraints with log barriers.

Data: You are given `sp500_returns.csv`, a CSV file containing daily returns for a selection of S&P 500 stocks. You will compute μ (sample mean returns) and Σ (sample covariance matrix) from this data.

1.6 What You Must Implement

Submit a file named **`hw6_p1_barrier.py`** with the following function.

Note on gradients: The gradients in this problem are simple enough to compute analytically by hand (we provide them below). You are welcome to compute gradients analytically, or if you prefer, you may use PyTorch autograd to compute them automatically. Either approach is acceptable.

Recall from the theory above: The barrier objective for parameter t is:

$$\phi_t(w) = t \cdot w^\top \Sigma w - \sum_i \log(w_i) - \log(\mu^\top w - r_{\min})$$

Its gradient with respect to w is:

$$\nabla \phi_t(w) = 2t\, \Sigma w - \frac{1}{w} \text{ (element-wise)} - \frac{\mu}{\mu^\top w - r_{\min}}$$

We handle the equality constraint $\sum_i w_i = 1$ by working in a reduced space: set $w_n = 1 - \sum_{i=1}^{n-1} w_i$. The barrier handles $w_i \geq 0$ for $i = 1, \ldots, n$ and $\mu^T w \geq r_{\min}$.

```
import numpy as np


def barrier_method(Sigma, mu, r_min, t_init=1.0, mu_factor=10.0,
                   tol=1e-6, max_outer=50, max_inner=200, lr=1e-3):
    """
    Solve the Markowitz portfolio optimization using the log barrier method.

    Problem:
        min_w  w^T Sigma w
        s.t.   mu^T w >= r_min
               w >= 0
               sum(w) = 1

    We handle sum(w) = 1 by working in a reduced space: set w_n = 1 - sum(w_{1..n-1}).
    The barrier handles w >= 0 and mu^T w >= r_min.

    The barrier objective for parameter t is:
        phi_t(w) = t * w^T Sigma w - sum_i log(w_i) - log(mu^T w - r_min)

    Algorithm:
        For each outer iteration:
            1. Form the barrier objective: t * w^T Sigma w - sum log(w_i) - log(mu^T w - r_min
            2. Run gradient descent for the inner loop (centering step)
            3. Check stopping criterion: (n+1)/t < tol  (n+1 = number of inequality constraint
            4. Increase t: t *= mu_factor

    Parameters
    ----------
    Sigma : np.ndarray, shape (n, n)
        Covariance matrix of asset returns.
    mu : np.ndarray, shape (n,)
        Expected return for each asset.
    r_min : float
        Minimum required expected return.
    t_init : float
        Initial barrier parameter.
    mu_factor : float
        Multiplicative increase for t each outer iteration.
    tol : float
        Stop when duality gap bound (n+1)/t < tol.
    max_outer : int
        Maximum number of outer (barrier) iterations.
    max_inner : int
        Maximum number of inner (gradient descent) iterations per centering step.
    lr : float
        Learning rate for inner gradient descent.

    Returns
    -------
    result : dict with keys:
        'w'         : np.ndarray, shape (n,) - optimal portfolio weights
        'objective' : float - optimal portfolio variance w^T Sigma w
    """
    raise NotImplementedError
```

1.7 Hints

- **Initialization:** Start with a uniform portfolio $w = \mathbf{1}/n$. If this violates the return constraint, shift weight toward the highest-return asset (don't worry too much, the tests won't give you crazy $r_{\min}$). Example:

```
w = np.ones(n) / n
if mu @ w <= r_min:
    idx = np.argmax(mu)
    w = np.ones(n) * 0.01 / (n - 1)
    w[idx] = 1.0 - 0.01
    w = w / np.sum(w)
```

- **Reduced space:** To enforce $\sum w_i = 1$, optimize over $v = w_{1:n-1}$ and set $w_n = 1 - \sum v_i$. The reduced gradient is $g_{v_i} = g_i^{\text{full}} - g_n^{\text{full}}$ (chain rule).
- **Inner loop:** Use gradient descent on $\phi_t(w)$. After each step, check that you haven't stepped outside the feasible region (all $w_i > 0$ and $\mu^\top w > r_{\min}$). If you have, reduce the step size or reject the step.
- **Stopping:** The duality gap bound is $(n + 1)/t$ where $n + 1$ is the number of inequality constraints (n non-negativity + 1 return constraint). Stop when this drops below `tol`.

1.8 Sanity Checks

```
import numpy as np

# Load real S&P 500 data
returns = np.genfromtxt('sp500_returns.csv', delimiter=',', skip_header=1)
mu = np.mean(returns, axis=0)
Sigma = np.cov(returns, rowvar=False)
r_min = np.median(mu) * 0.8  # achievable target return

result = barrier_method(Sigma, mu, r_min)
w = result['w']
print(f"Optimal weights: {np.round(w, 4)}")
print(f"Portfolio variance: {result['objective']:.6f}")
print(f"Sum of weights: {np.sum(w):.6f}")       # Should be ~1.0
print(f"Expected return: {mu @ w:.6f}")           # Should be >= r_min
print(f"All non-negative: {np.all(w >= -1e-6)}")  # Should be True
```

---

2 Problem 2: Predicting California Housing Prices with Ridge & Lasso (Gradescope — 100 points)

Throughout this course, we've always given you the data: the capacity of an edge, the cost of shipping, the demand at each node, the weight of each item. But in the real world, this data often isn't handed to you — you need to ***predict*** *it. How much will this house sell for? What will demand be next quarter? How many customers will churn this month? This is where* ***Machine Learning*** *comes in.*

In this problem, you'll step into the role of a data scientist: given raw features about California housing districts (location, income levels, house age, etc.), your job is to build a model that predicts median house prices. Along the way, you'll learn the full ML pipeline — from feature engineering to train/test evaluation — and see firsthand how the optimization techniques you've learned in this course power real-world prediction.

You'll implement both ***Ridge regression*** *(L2 regularization) and* ***Lasso regression*** *(L1 regularization, via the proximal gradient descent you learned in HW5), and discover that Lasso performs* ***automatic feature selection*** *— zeroing out irrelevant features while keeping the important ones.*

2.1 Background

2.1.1 What Are "Features"?

In machine learning, **features** are the measurable properties or attributes of each data point that we use to make predictions. Think of features as the information you would look at when making a decision.

For example, if you are trying to predict the price of a house, you would naturally consider: - How big is it? (square footage, number of rooms) - Where is it located? (latitude, longitude, neighborhood) - How old is it? (year built) - What is the neighborhood like? (median income, population density)

Each of these is a **feature**. We organize all of our data into a matrix X where: - Each **row** is one data point (one house, one district, one observation) - Each **column** is one feature (income, latitude, number of rooms, etc.)

So if we have n houses and p features, X is an $n \times p$ matrix. The entry X_{ij} is the value of feature j for house i.

2.1.2 What Is a "Target"?

The **target** (also called the **label** or **response variable**) is the thing we want to predict. In our problem, the target is the **median house price** in a California census block. We collect all the targets into a vector y of length n, where y_i is the median house price for district i.

2.1.3 What Is "Supervised Learning"?

Supervised learning means we have a dataset of *labeled examples* — pairs (X, y) where we know both the features and the correct answer. Our goal is to learn a function f such that $f(X) \approx y$. Once we have learned f, we can use it to predict targets for new data points where we only know the features.

In our case, we want to learn a function that takes in information about a California district (income, house age, location, etc.) and outputs a prediction of the median house price. We will use **linear models**, where:

$$f(x) = w_1 x_1 + w_2 x_2 + \cdots + w_p x_p = w^\top x$$

The weights w tell us how important each feature is. Learning the model means finding the best weights w by solving an optimization problem — which is exactly what you have been doing all semester!

2.1.4 Why Do We Split Into Training and Test Sets?

Imagine a student who memorizes every answer in a practice exam without understanding the material. They would score perfectly on that practice exam, but poorly on the real exam because they did not actually *learn* the underlying concepts.

The same thing happens with machine learning models. If we evaluate our model on the same data we used to train it, we might just be measuring how well it memorized the training data, not how well it actually learned the underlying patterns. This is called **overfitting**.

To guard against this, we **hold out** a portion of our data (typically 20%) as a **test set** that the model never sees during training. We train the model on the remaining 80% (the **training set**), and then evaluate its performance on the test set. If the model performs well on test data it has never seen, we have evidence that it has truly learned something useful.

2.1.5 What Is "Standardization"?

Our features have wildly different scales: - Median income might range from \$0 to \$15 (in units of \$10,000s) - Latitude ranges from about 32 to 42 (degrees) - Average number of rooms might be 1 to 15

When we add regularization (like Ridge or Lasso), the penalty term $\lambda\|w\|$ treats all weights equally. But if feature j has values 1000x larger than feature k, then the weight w_j will naturally be 1000x smaller, and the regularization will penalize them unequally.

Standardization fixes this: for each feature, we subtract its mean and divide by its standard deviation:

$$x_j^{\text{standardized}} = \frac{x_j - \text{mean}(x_j)}{\text{std}(x_j)}$$

After standardization, every feature has mean 0 and standard deviation 1, so the regularization treats them all fairly. One important detail: we compute the mean and standard deviation from the **training set only**, and then apply the same transformation to the test set. This prevents information from the test set from "leaking" into the training process.

2.1.6 What Is "Feature Engineering"?

Raw features are not always the most informative representation of the data. **Feature engineering** is the art of creating new features from existing ones that might be more useful for prediction.

For example, the raw dataset gives us “average number of rooms” and “average number of bedrooms” as separate features. But perhaps what really matters for predicting house price is the *ratio* of bedrooms to total rooms (a house where most rooms are bedrooms is very different from one with lots of living space). So we can create a new feature:

$$\text{BedroomRatio} = \frac{\text{AveBedrms}}{\text{AveRooms}}$$

This new feature captures information that neither original feature captures alone. Good feature engineering can dramatically improve model performance.

2.1.7 Ridge Regression (L2 Regularization)

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

- Has a closed-form solution: $w^* = (X^\top X/n + \lambda I)^{-1} X^\top y/n$
- Shrinks all coefficients toward zero, but never exactly to zero
- Controlled by $\lambda \geq 0$: larger λ = more shrinkage

2.1.8 Lasso Regression (L1 Regularization)

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

- No closed-form solution (the ℓ_1 norm is non-differentiable)
- Solved via **proximal gradient descent** (ISTA) — you implemented this in HW5!
- Drives some coefficients exactly to zero: **automatic feature selection**
- Controlled by $\lambda \geq 0$: larger λ = more sparsity

2.1.9 The Data: California Housing

We use the California Housing dataset (available via `sklearn.datasets.fetch_california_housing`). Each sample represents a census block group in California. The target is the **median house value** (in \$100,000s).

Raw features: | Feature | Description | |———|————-| | MedInc | Median income in block group | | HouseAge | Median house age in block group | | AveRooms | Average number of rooms per household | | AveBedrms | Average number of bedrooms per household | | Population | Block group population | | AveOccup | Average number of household members | | Latitude | Block group latitude | | Longitude | Block group longitude |

2.2 What You Must Implement

Submit a file named **`hw6_p2_housing.py`** with the following three functions.

Note: You already implemented `ridge_closed_form` and `lasso_proximal_gd` in HW5. You should **reuse** those implementations (copy them into your submission file). The train/test split and standardization are provided for you in the sanity checks below.

```
import numpy as np
```

```python
def load_and_engineer_features():
    """
    Load the California Housing dataset and engineer additional features.

    Steps:
        1. Load data using sklearn.datasets.fetch_california_housing()
        2. Create a feature matrix X (as np.ndarray) and target vector y
        3. Engineer at least 2-3 new features by creating ratios or interactions
           from the existing columns. Some suggestions:
           - BedroomRatio = AveBedrms / AveRooms
           - RoomsPerPerson = AveRooms / AveOccup
           - IncomePerRoom = MedInc / AveRooms
           Feel free to get creative! Try other combinations that you think
           might help predict house prices.
        4. Return X (with original + engineered features) and y

    Returns
    -------
    X : np.ndarray, shape (n_samples, p)
        Feature matrix (8 original + your engineered features).
    y : np.ndarray, shape (n_samples,)
        Target values (median house value in $100,000s).
    feature_names : list of str, length p
        Names of all features.
    """
    raise NotImplementedError


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

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

    Solution: w* = (X^T X / n + lam * I)^{-1} X^T y / n

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

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

```python
def lasso_proximal_gd(X, y, lam, lr=None, num_iters=5000, tol=1e-6):
    """
    Solve Lasso regression via proximal gradient descent (ISTA).

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

    Algorithm (at each iteration):
        1. Compute gradient of smooth part: grad = (1/n) X^T (Xw - y)
        2. Gradient step: v = w - lr * grad
        3. Proximal step: w_new = sign(v) * max(|v| - lr * lam, 0)
        4. Check convergence: max |w_new - w_old| < tol

    Parameters
    ----------
    X : np.ndarray, shape (n, p)
    y : np.ndarray, shape (n,)
    lam : float
    lr : float or None
        If None, set lr = 1/L where L = largest eigenvalue of X^T X / n.
    num_iters : int
    tol : float

    Returns
    -------
    result : dict with keys:
        'w'         : np.ndarray, shape (p,) - final coefficients
        'objective' : list of float - Lasso objective at each iteration
    """
    raise NotImplementedError
```

2.3 Hints

- **`ridge_closed_form`**: Use `np.linalg.solve` instead of computing the matrix inverse explicitly.
- **`lasso_proximal_gd`**: Precompute $X^\top X/n$ and $X^\top y/n$ for efficiency. The soft-thresholding (proximal step) is just `np.sign(v) * np.maximum(np.abs(v) - lr * lam, 0)`.

2.4 Sanity Checks

```python
import numpy as np

# --- Provided helper code (you don't need to implement these) ---
def train_test_split(X, y, test_fraction=0.2, seed=42):
    rng = np.random.RandomState(seed)
    perm = rng.permutation(X.shape[0])
    n_test = int(X.shape[0] * test_fraction)
    return X[perm[n_test:]], X[perm[:n_test]], y[perm[n_test:]], y[perm[:n_test]]

def standardize(X_train, X_test):
    mu, sigma = X_train.mean(0), X_train.std(0) + 1e-10
    return (X_train - mu) / sigma, (X_test - mu) / sigma
# ----------------------------------------------------------------

X, y, names = load_and_engineer_features()
print(f"Features: {X.shape[1]} ({names})")  # Should be >= 10
X_train, X_test, y_train, y_test = train_test_split(X, y)
X_train_s, X_test_s = standardize(X_train, X_test)

# Ridge
w_ridge = ridge_closed_form(X_train_s, y_train, lam=0.1)
mse_train = np.mean((X_train_s @ w_ridge - y_train)**2)
mse_test = np.mean((X_test_s @ w_ridge - y_test)**2)
print(f"Ridge MSE (train): {mse_train:.4f}")
print(f"Ridge MSE (test):  {mse_test:.4f}")

# Lasso
result = lasso_proximal_gd(X_train_s, y_train, lam=0.01)
w_lasso = result['w']
mse_lasso = np.mean((X_test_s @ w_lasso - y_test)**2)
print(f"Lasso MSE (test):  {mse_lasso:.4f}")
print(f"Lasso nonzero coefficients: {np.sum(np.abs(w_lasso) > 1e-6)}")
```

3 Problem 3: Projection onto the Simplex & Proximal Methods (Gradescope — 100 points)

Many optimization problems require solutions that live on the ***probability simplex****: all components non-negative and summing to 1. Think of portfolio weights, probability distributions, or resource allocations. In this problem, you'll* ***derive*** *the projection onto the simplex from first principles using KKT conditions, implement it, and then use it inside projected gradient descent.*

3.1 3.1 Deriving the Simplex Projection from KKT Conditions

The **probability simplex** in $\mathbb{R}^n$ is:

$$\Delta^n = \left\{ x \in \mathbb{R}^n : \sum_{i=1}^{n} x_i = 1, \;\; x_i \geq 0 \;\; \forall i \right\}$$

Given any point $y \in \mathbb{R}^n$, the **projection onto the simplex** is:

$$\operatorname{proj}_{\Delta^n}(y) = \arg\min_{x} \; \frac{1}{2}\|x - y\|_2^2 \quad \text{s.t.} \quad \mathbf{1}^\top x = 1, \;\; x \geq 0$$

This is a convex QP, so the KKT conditions are necessary and sufficient.

3.1.1 Part A: Write the Lagrangian (10 points, handgraded)

Write the Lagrangian for this problem, introducing: - A multiplier $\tau \in \mathbb{R}$ for the equality constraint $\mathbf{1}^\top x = 1$ - Multipliers $\nu_i \geq 0$ for the inequality constraints $-x_i \leq 0$ (i.e., $x_i \geq 0$)

$$L(x, \tau, \nu) = \; ???$$

Hint: The constraints $x_i \geq 0$ can be written as $-x_i \leq 0$, so the KKT multipliers $\nu_i \geq 0$ enter with a minus sign on x_i.

3.1.2 Part B: Derive the KKT Conditions (15 points, handgraded)

Write down all four KKT conditions:

1. **Stationarity:** $\frac{\partial L}{\partial x_i} = 0$ for all i. Show that this gives:

$$x_i - y_i + \tau - \nu_i = 0 \quad \Longrightarrow \quad x_i = y_i - \tau + \nu_i$$

2. **Primal feasibility:** $\sum_i x_i = 1$ and $x_i \geq 0$
3. **Dual feasibility:** $\nu_i \geq 0$ for all i
4. **Complementary slackness:** $\nu_i \cdot x_i = 0$ for all i

3.1.3 Part C: Derive the Projection Formula (25 points, handgraded)

Use the KKT conditions to derive a closed-form for x in terms of y and a single threshold τ.

Hint 1: From complementary slackness, for each i exactly one of two cases holds: - **Case 1:** $x_i > 0$, which forces $\nu_i = 0$ (by complementary slackness), so $x_i = y_i - \tau$ - **Case 2:** $x_i = 0$, which means $\nu_i = \tau - y_i \geq 0$ (from stationarity + dual feasibility), so $y_i \leq \tau$

Hint 2: Combining both cases: $x_i = \max(y_i - \tau, 0)$.

Hint 3: Use the constraint $\sum_i x_i = 1$ to determine τ:

$$\sum_{i=1}^{n} \max(y_i - \tau, 0) = 1$$

Hint 4: Sort y in decreasing order: $y_{(1)} \geq y_{(2)} \geq \ldots \geq y_{(n)}$. Let ρ be the number of strictly positive components in the projection. Then:

$$\tau = \frac{\left(\sum_{i=1}^{\rho} y_{(i)}\right) - 1}{\rho}$$

where ρ is the largest index such that $y_{(\rho)} - \tau > 0$, i.e.:

$$\rho = \max\left\{ k \in \{1, \ldots, n\} : y_{(k)} > \frac{\left(\sum_{i=1}^{k} y_{(i)}\right) - 1}{k} \right\}$$

Your task: Show all the steps that lead from the KKT conditions to this algorithm. Specifically: 1. Show how complementary slackness gives $x_i = \max(y_i - \tau, 0)$ 2. Show how $\sum x_i = 1$ determines τ 3. Explain why sorting y and finding ρ correctly identifies τ

3.1.4 Part D: Implement the Projection (10 points, autograded)

Now implement the algorithm you derived.

3.1.5 Part E: Projected Gradient Descent (20 points, autograded)

Use your simplex projection inside projected gradient descent to solve:

$$\min_{x \in \Delta^n} \; f(x) = \frac{1}{2} x^\top Q x + c^\top x$$

3.1.6 Part F: Application — Optimal Mixing of Strategies (20 points, autograded)

Apply projected gradient descent to find the optimal mixture of n base strategies that minimizes expected cost.

3.2 What You Must Implement

Submit a file named **hw6_p3_simplex_proj.py** with the following functions.

```python
import numpy as np


def project_simplex(y):
    """
    Project a vector y onto the probability simplex:
        argmin_x  (1/2)||x - y||^2
        s.t.      sum(x) = 1, x >= 0

    Uses the algorithm derived from KKT conditions:
        1. Sort y in decreasing order
        2. Find rho = max{k : y_(k) > (sum_{i=1}^k y_(i) - 1) / k}
        3. Compute tau = (sum_{i=1}^rho y_(i) - 1) / rho
        4. Return x = max(y - tau, 0)

    Parameters
    ----------
    y : np.ndarray, shape (n,)
        Point to project.

    Returns
    -------
    x : np.ndarray, shape (n,)
        Projection of y onto the simplex.
    """
    raise NotImplementedError


def projected_gradient_descent(Q, c, x0, lr, num_iters, project_fn):
    """
    Projected gradient descent for:
        min  (1/2) x^T Q x + c^T x
        s.t. x in C  (handled by project_fn)

    Algorithm:
        x_{t+1} = project_fn(x_t - lr * (Q x_t + c))

    Parameters
    ----------
    Q : np.ndarray, shape (n, n)
        Symmetric PSD matrix.
    c : np.ndarray, shape (n,)
        Linear term.
    x0 : np.ndarray, shape (n,)
```

```
        Initial point (should be feasible).
    lr : float
        Step size.
    num_iters : int
        Number of iterations.
    project_fn : callable
        Projection function: project_fn(y) -> x on the constraint set.

    Returns
    -------
    result : dict with keys:
        'x_final'    : np.ndarray, shape (n,) - final iterate
        'f_final'    : float - objective at final iterate
        'objectives' : list of float - objective at each iterate (length num_iters+1)
    """
    raise NotImplementedError


def optimal_strategy_mix(cost_matrix, num_iters=5000, lr=0.01):
    """
    Find the optimal probability distribution over n strategies that minimizes
    expected quadratic cost.

    Given a cost matrix C of shape (n, n) where C[i,j] represents the cost of
    strategy i in scenario j, find the mixture weights x in the simplex that
    minimize:
        f(x) = (1/2) x^T (C C^T) x

    This corresponds to minimizing the variance of the mixed strategy cost.

    Parameters
    ----------
    cost_matrix : np.ndarray, shape (n, n)
        Cost matrix.
    num_iters : int
    lr : float

    Returns
    -------
    result : dict with keys:
        'x'          : np.ndarray, shape (n,) - optimal strategy mixture
        'objective'  : float - optimal objective value
        'objectives' : list of float - convergence history
    """
    raise NotImplementedError
```

3.3 Hints

- **`project_simplex`**: The algorithm is exactly what you derived in Part C. Use `np.sort` (descending) and `np.cumsum` to find ρ efficiently.
- **`projected_gradient_descent`**: Standard GD, but after each gradient step, project back onto the simplex.
- **`optimal_strategy_mix`**: Compute $Q = CC^\top$, then use projected GD with your simplex projection.

---

4 Problem 4: Deriving LP Duality from Lagrangian Duality (100 points, handgraded)

In this course, we have been solving constrained optimization problems using various techniques. But there is a beautiful and deep framework underlying all of them: ***Lagrangian Duality****. In this problem, you will learn Lagrangian duality from scratch and then apply it to derive the dual of a linear program, showing that LP duality (which you have used throughout this course) is just a special case of a much more general beautiful theory.*

4.1 The Lagrangian Framework

Consider a general constrained optimization problem:

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & g_i(x) \leq 0, \quad i = 1, \dots, m \\ & h_j(x) = 0, \quad j = 1, \dots, p \end{aligned}$$

The **Lagrangian** is a single function that combines the objective and all the constraints:

$$L(x, \lambda, \nu) = f(x) + \sum_{i=1}^{m} \lambda_i \, g_i(x) + \sum_{j=1}^{p} \nu_j \, h_j(x)$$

where: - $\lambda_i \geq 0$ are called **dual variables** (or **Lagrange multipliers**) for the inequality constraints - $\nu_j \in \mathbb{R}$ are dual variables for the equality constraints (these are unconstrained in sign)

The requirement $\lambda_i \geq 0$ is crucial: it means we *reward* constraint satisfaction ($g_i(x) < 0$ makes the $\lambda_i g_i(x)$ term negative, lowering the Lagrangian) and *penalize* constraint violation ($g_i(x) > 0$ makes the term positive, raising the Lagrangian).

4.2 The Dual Function and Weak Duality

The **Lagrangian dual function** is defined by minimizing the Lagrangian over x:

$$g(\lambda, \nu) = \min_x \; L(x, \lambda, \nu)$$

This function has a remarkable property: **for any $\lambda \geq 0$ and any ν, the dual function gives a lower bound on the optimal value p^* of the original (primal) problem.** That is:

$$g(\lambda, \nu) \leq p^*$$

This is called **weak duality**. The proof is straightforward: if x^* is the optimal primal solution, then $g_i(x^*) \leq 0$ and $h_j(x^*) = 0$, so:

$$g(\lambda, \nu) = \min_x L(x, \lambda, \nu) \le L(x^*, \lambda, \nu) = f(x^*) + \underbrace{\sum_i \lambda_i g_i(x^*)}_{\le 0 \text{ since } \lambda_i \ge 0, g_i(x^*) \le 0} + \underbrace{\sum_j \nu_j h_j(x^*)}_{=0} \le f(x^*) = p^*$$

Since the dual function always provides a lower bound, the best lower bound comes from the **dual problem**:

$$\max_{\lambda,\nu} \ g(\lambda, \nu) \quad \text{s.t.} \quad \lambda \ge 0$$

The optimal value of this problem, d^*, satisfies $d^* \le p^*$.

4.3 Strong Duality

A natural question is: how tight is this bound? In general, there can be a gap $p^* - d^* > 0$ (the **duality gap**). But under certain conditions, the bound is tight and we get $p^* = d^*$. This is called **strong duality**.

For **convex problems** (where f and g_i are convex and h_j are affine), strong duality holds under a mild regularity condition called **Slater's condition**: there exists a strictly feasible point $\hat{x}$ such that $g_i(\hat{x}) < 0$ for all i (strict inequality) and $h_j(\hat{x}) = 0$ for all j.

What does strong duality buy us? It means we can solve the dual problem instead of the primal, and get the same optimal value. Sometimes the dual is much easier to solve! It also provides certificates of optimality: if we find primal and dual feasible solutions with the same objective value, both must be optimal.

4.4 Applying Lagrangian Duality to a Linear Program

Now let us apply this framework to derive the dual of a standard-form linear program.

Consider the LP:

$$\begin{aligned} \text{(P)} \quad \min_x \quad & c^\top x \\ \text{s.t.} \quad & Ax \ge b \\ & x \ge 0 \end{aligned}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$.

4.5 Part A: Write the Lagrangian and Simplify (40 points)

Introduce dual variables: - $\lambda \in \mathbb{R}^m$, $\lambda \ge 0$ for the constraints $Ax \ge b$ (rewritten as $b - Ax \le 0$) - $\mu \in \mathbb{R}^n$, $\mu \ge 0$ for the constraints $x \ge 0$ (rewritten as $-x \le 0$)

Task:

1. Write the Lagrangian $L(x, \lambda, \mu)$ by adding the dual variable terms:

$$L(x, \lambda, \mu) = c^\top x + \lambda^\top (b - Ax) + \mu^\top(-x)$$

2. Simplify by collecting all terms that involve x together:

$$L(x, \lambda, \mu) = (c - A^\top \lambda - \mu)^\top x + b^\top \lambda$$

Your task: Verify this simplification step by step. Show clearly how the terms rearrange. In particular, show that $\lambda^\top(b - Ax) = b^\top\lambda - \lambda^\top Ax = b^\top\lambda - (A^\top\lambda)^\top x$ and that $\mu^\top(-x) = -\mu^\top x$. Then combine the x-dependent terms.

4.6 Part B: Derive the Dual Problem (60 points)

Task:

1. Compute the dual function $g(\lambda, \mu) = \min_x L(x, \lambda, \mu)$.

   Notice that $L(x, \lambda, \mu) = (c - A^\top\lambda - \mu)^\top x + b^\top\lambda$ is **linear in** x with no constraints on x (we relaxed both $Ax \geq b$ and $x \geq 0$ into the Lagrangian). The minimum of a linear function over all of $\mathbb{R}^n$ is:

   - $-\infty$ if the coefficient of x is nonzero in any component (we can send x in the direction that makes the linear function go to $-\infty$)
   - A finite value (just the constant term) if the coefficient of x is exactly zero

   Therefore:

$$g(\lambda, \mu) = \begin{cases} b^\top\lambda & \text{if } c - A^\top\lambda - \mu = 0 \\ -\infty & \text{otherwise} \end{cases}$$

2. Write the dual problem. Since we want to maximize $g(\lambda, \mu)$ subject to $\lambda \geq 0$, $\mu \geq 0$, and we need g to be finite, the dual problem is:

$$\begin{aligned} \max_{\lambda,\mu} \quad & b^\top\lambda \\ \text{s.t.} \quad & c - A^\top\lambda - \mu = 0 \\ & \lambda \geq 0,\ \mu \geq 0 \end{aligned}$$

3. Eliminate μ: since $\mu = c - A^\top\lambda$ and we need $\mu \geq 0$, this gives $A^\top\lambda \leq c$. Substituting, the dual becomes the **standard dual LP**:

$$\text{(D)} \quad \begin{aligned} \max_{\lambda} \quad & b^\top\lambda \\ \text{s.t.} \quad & A^\top\lambda \leq c \\ & \lambda \geq 0 \end{aligned}$$

Your task: Write out each of these steps in full detail. Show all work for computing $g(\lambda, \mu)$ (explain carefully why the minimum over x is $-\infty$ when the coefficient is nonzero). Then write the complete dual LP after eliminating μ. Finally, briefly explain why this result means that every feasible dual solution λ (satisfying $A^\top \lambda \leq c$ and $\lambda \geq 0$) provides a **lower bound** $b^\top \lambda \leq p^*$ on the optimal primal value. This is weak duality applied to LPs.

---

5 Problem 5: Augmented Lagrangian for Resource Allocation (Gradescope — 100 points)

You've now seen two approaches to constrained optimization: the Lagrangian method and the penalty method. Each has drawbacks. The pure Lagrangian method (dual ascent) can oscillate and converge slowly. The penalty method needs $\rho \to \infty$, causing ill-conditioning. In this problem, you'll learn the ***Augmented Lagrangian method*** *(also called the* ***Method of Multipliers****), which combines the best of both worlds: it converges with a moderate, fixed penalty parameter ρ.*

5.1 Review: What We Know So Far

5.1.1 The Lagrangian Method (Dual Ascent)

For the problem $\min_x f(x)$ s.t. $g_i(x) \leq 0$, the Lagrangian is:

$$L(x, \lambda) = f(x) + \sum_i \lambda_i g_i(x), \quad \lambda_i \geq 0$$

The idea: alternate between minimizing L over x (primal step) and increasing λ for violated constraints (dual step):

$$x^{k+1} = \arg\min_x L(x, \lambda^k), \qquad \lambda_i^{k+1} = \max(0, \lambda_i^k + \alpha\, g_i(x^{k+1}))$$

Problem: This can oscillate. The primal minimization may give wildly different x values from step to step, and convergence requires careful tuning of the dual step size α.

5.1.2 The Penalty Method

Replace constraints with penalties:

$$\min_x \; f(x) + \frac{\rho}{2} \sum_i \max(0, g_i(x))^2$$

Problem: Needs $\rho \to \infty$ for exactness, which makes the objective ill-conditioned (very steep near constraint boundaries).

5.2 The Key Idea: Combine Both!

What if we use Lagrange multipliers λ **AND** a quadratic penalty? The multipliers handle the "correctness" (converging to the right constraint satisfaction), while the penalty improves conditioning and prevents oscillation.

5.2.1 The Augmented Lagrangian

For inequality constraints $g_i(x) \leq 0$, the **augmented Lagrangian** is:

$$\mathcal{L}_\rho(x,\lambda) = f(x) + \frac{\rho}{2}\sum_{i=1}^{m}\left[\max\left(0,\ g_i(x) + \frac{\lambda_i}{\rho}\right)\right]^2 - \frac{1}{2\rho}\sum_{i=1}^{m}\lambda_i^2$$

This can equivalently be written as:

$$\mathcal{L}_\rho(x,\lambda) = f(x) + \sum_{i=1}^{m}\psi_\rho(g_i(x),\lambda_i)$$

where:

$$\psi_\rho(g,\lambda) = \begin{cases} \lambda g + \frac{\rho}{2}g^2 & \text{if } g + \frac{\lambda}{\rho} \geq 0 \\ -\frac{\lambda^2}{2\rho} & \text{if } g + \frac{\lambda}{\rho} < 0 \end{cases}$$

Intuition: When $g_i(x) + \lambda_i/\rho \geq 0$ (constraint is active or violated), the augmented Lagrangian applies both a Lagrange multiplier term $\lambda_i g_i$ and a quadratic penalty $\frac{\rho}{2}g_i^2$. When $g_i(x) + \lambda_i/\rho < 0$ (constraint is well satisfied), the penalty is turned off.

5.3 The Algorithm: Method of Multipliers

Input: Initial x^0, $\lambda^0 \geq 0$, penalty parameter $\rho > 0$

Repeat for $k = 0, 1, 2, \ldots$:

1. **Primal update (inner loop):** Approximately minimize $\mathcal{L}_\rho(x, \lambda^k)$ over x using gradient descent (with projection for box constraints, see below).
2. **Dual update:** Update the multipliers:

$$\lambda_i^{k+1} = \max\left(0,\ \lambda_i^k + \rho\, g_i(x^{k+1})\right)$$

3. **Check convergence:** Stop when the maximum constraint violation is below tolerance.

Why it works: The λ updates "learn" the correct multiplier values over time. Unlike the pure penalty method, we don't need $\rho \to \infty$ — a moderate, fixed ρ suffices because the multipliers λ carry the information about how much each constraint matters.

Key advantage over penalty method: Converges with **moderate, fixed** ρ — no ill-conditioning!

Key advantage over dual ascent: The quadratic penalty term makes the primal subproblem better conditioned and prevents oscillation.

5.4 Application: Multi-Department Resource Allocation

A company has m shared resources (budget, server capacity, warehouse space) with limits b_i. Each of n departments chooses an activity level $x_j \geq 0$ to maximize its own benefit $f_j(x_j)$, but the total resource usage must not exceed supply:

$$
\begin{aligned}
\min_{x} \quad & \sum_{j=1}^{n} f_j(x_j) \quad \text{(sum of department costs)} \\
\text{s.t.} \quad & \underbrace{\sum_{j} A_{ij} x_j \le b_i \quad \text{for } i = 1, \ldots, m}_{\text{coupling constraints}\to\text{augmented Lagrangian}} \\
& \underbrace{0 \le x_j \le u_j \quad \text{for } j = 1, \ldots, n}_{\text{box constraints}\to\text{projection (np.clip)}}
\end{aligned}
$$

We use quadratic costs $f_j(x_j) = c_j x_j + \frac{d_j}{2} x_j^2$ (representing diminishing returns).

5.5 Handling Two Types of Constraints

Our resource allocation problem has two fundamentally different types of constraints:

1. Linear coupling constraints: $Ax \le b$

These link multiple variables together. For example, "the total server usage across all departments cannot exceed capacity." Changing one department's activity level x_j affects multiple resource constraints simultaneously. These are the **hard constraints** — they couple the variables and cannot be handled by simple per-variable operations.

2. Box constraints: $0 \le x \le u$

These only involve individual variables. For example, "Department 3 can run between 0 and 5 units of activity." Each bound involves a single variable, so they are extremely easy to enforce — just clip each variable to its allowed range.

Our strategy: - Handle the **coupling constraints** $Ax \le b$ with the augmented Lagrangian (the sophisticated machinery with Lagrange multipliers and penalty terms) - Handle the **box constraints** $0 \le x \le u$ with simple **projection** (just `np.clip`) after each gradient step in the inner loop

This is called **projected gradient descent**: take a gradient step on the augmented Lagrangian (which only includes the $Ax \le b$ constraints), then project back onto the box $[0, u]$. The projection step is trivial (the clipping operation):

$$x_j \leftarrow \text{clip}(x_j,\ 0,\ u_j) = \max(0, \min(x_j, u_j))$$

Why separate them? The box constraints are so simple that it would be wasteful to bring out the full augmented Lagrangian machinery for them. Projection is exact, instantaneous, and adds no complexity. The augmented Lagrangian is reserved for the coupling constraints that actually need it.

The augmented Lagrangian for this problem (only the coupling constraints $Ax \le b$ appear):

$$\mathcal{L}_\rho(x, \lambda) = \underbrace{\sum_j \left(c_j x_j + \frac{d_j}{2} x_j^2 \right)}_{\text{objective } f(x)} + \frac{\rho}{2} \sum_{i=1}^{m} \left[\max\left(0,\ (Ax)_i - b_i + \frac{\lambda_i}{\rho}\right) \right]^2 - \frac{1}{2\rho} \sum_{i=1}^{m} \lambda_i^2$$

Its gradient with respect to x:

$$\nabla_x \mathcal{L}_\rho = \underbrace{c + d \odot x}_{\nabla f(x)} + \rho \cdot A^\top s$$

where $s_i = \max(0,\ (Ax)_i - b_i + \lambda_i/\rho)$ and $\odot$ denotes element-wise multiplication.

5.6 What You Must Implement

Submit a file named **`hw6_p5_auglag.py`** with the following function.

```python
import numpy as np


def augmented_lagrangian_method(c, d, A, b, ub, rho=10.0,
                                max_outer=50, max_inner=500,
                                lr=1e-3, tol=1e-5):
    """
    Solve the resource allocation problem using the augmented Lagrangian method.

    Problem:
        min  sum_j (c_j x_j + (d_j/2) x_j^2)
        s.t. A x <= b           (coupling constraints - handled by augmented Lagrangian)
             0 <= x <= ub       (box constraints - handled by projection)

    Algorithm:
        Initialize x = 0, lam = 0
        For each outer iteration:
            1. Inner loop: minimize L_rho(x, lam) over x using projected gradient descent
               - Gradient step: x = x - lr * grad_x L_rho
               - Projection step: x = clip(x, 0, ub)
            2. Dual update: lam_i = max(0, lam_i + rho * ((Ax)_i - b_i))
            3. Check convergence: stop if max constraint violation < tol

    Parameters
    ----------
    c : np.ndarray, shape (n,)
        Linear cost coefficients.
    d : np.ndarray, shape (n,)
        Quadratic cost coefficients (>= 0).
    A : np.ndarray, shape (m, n)
        Resource usage matrix.
    b : np.ndarray, shape (m,)
        Resource limits.
    ub : np.ndarray, shape (n,)
        Upper bounds on x.
    rho : float
        Penalty parameter.
    max_outer : int
        Maximum number of outer (multiplier update) iterations.
    max_inner : int
        Maximum number of inner (projected gradient descent) iterations.
    lr : float
        Learning rate for inner gradient descent.
    tol : float
        Stop when max constraint violation < tol.

    Returns
    -------
    result : dict with keys:
        'x'         : np.ndarray, shape (n,) - optimal activity levels
        'lam'       : np.ndarray, shape (m,) - final Lagrange multipliers
        'objective' : float - optimal objective value sum_j (c_j x_j + (d_j/2) x_j^2)
    """
    raise NotImplementedError
```

5.7 Hints

- **Gradient of the augmented Lagrangian:** Compute $g_i = (Ax)_i - b_i$ (constraint residuals), then $s_i = \max(0, g_i + \lambda_i/\rho)$. The gradient is $\nabla_x \mathcal{L}_\rho = c + d \odot x + \rho \cdot A^\top s$.
- **Inner loop (projected GD):** At each step: (1) compute gradient, (2) take a step $x \leftarrow x - \text{lr} \cdot \nabla_x \mathcal{L}_\rho$, (3) project $x \leftarrow \text{clip}(x, 0, u)$.
- **Dual update:** After the inner loop converges, update $\lambda_i \leftarrow \max(0, \lambda_i + \rho \cdot g_i)$.
- **Convergence check:** Compute $\max_i \max(0, (Ax)_i - b_i)$. If this is below `tol`, stop.

6 Problem 6: Checking Sum-of-Squares Polynomials via SDP (Gradescope — 100 points)

When is a polynomial guaranteed to be nonnegative everywhere? This fundamental question arises throughout optimization, control theory, and combinatorics. Checking nonnegativity directly is NP-hard, but there's a beautiful sufficient condition: if a polynomial can be written as a ***sum of squares*** *(SOS), then it's obviously nonneg. The remarkable fact is that checking whether a polynomial is SOS reduces to solving a* ***semidefinite program*** *(SDP). In this problem, you'll implement this connection.*

6.1 Background

6.1.1 Nonnegative Polynomials

A polynomial $p(\mathbf{x})$ in n variables is **nonnegative** if $p(\mathbf{x}) \geq 0$ for all $\mathbf{x} \in \mathbb{R}^n$. For example:

- $p(x) = x^2 + 1$ is nonneg (obviously)
- $p(x, y) = x^2 + y^2$ is nonneg
- $p(x) = x^2 - 1$ is NOT nonneg ($p(0) = -1$)

Checking whether a polynomial is nonnegative is **NP-hard** in general (for degree ≥ 4 and $n \geq 2$). We need a tractable sufficient condition.

6.1.2 Sum of Squares (SOS)

A polynomial p is a **sum of squares** (SOS) if it can be written as:

$$p(\mathbf{x}) = \sum_k q_k(\mathbf{x})^2$$

for some polynomials $q_1, q_2, \ldots$

SOS $\Rightarrow$ nonneg — obvious, since a sum of squares is always ≥ 0.

Is the converse true? It depends:

- **Yes** for univariate polynomials (any degree), quadratics (any number of variables), and bivariate quartics
- **No** in general (Hilbert, 1888). The first explicit counterexample was the **Motzkin polynomial** (1967):

$$M(x, y) = x^4y^2 + x^2y^4 - 3x^2y^2 + 1$$

This polynomial is nonneg everywhere (provable by AM-GM) but **cannot** be written as a sum of squares of polynomials.

6.1.3 The Gram Matrix Condition

Here's the key insight that connects SOS to SDP. Let $p(\mathbf{x})$ have degree $2d$ in n variables. Define the **monomial vector** $z(\mathbf{x})$ containing all monomials of degree $\leq d$:

$$z(\mathbf{x}) = [1,\ x_1,\ \dots,\ x_n,\ x_1^2,\ x_1x_2,\ \dots]^T$$

The dimension of z is $s = \binom{n+d}{d}$.

Theorem: p is SOS if and only if there exists a **positive semidefinite** matrix $Q \in \mathbb{R}^{s\times s}$ (the **Gram matrix**) such that:

$$p(\mathbf{x}) = z(\mathbf{x})^T Q\, z(\mathbf{x})$$

Why? If $Q \succeq 0$, write $Q = \sum_k u_k u_k^T$ (eigendecomposition). Then $p = \sum_k (u_k^T z)^2$, which is explicitly a sum of squares. Conversely, if $p = \sum_k q_k^2$ where each $q_k = u_k^T z$, then $p = z^T(\sum_k u_k u_k^T)z$ with $Q = \sum_k u_k u_k^T \succeq 0$.

6.1.4 Checking SOS via SDP

Expanding $z(\mathbf{x})^T Q\, z(\mathbf{x})$ and matching each monomial coefficient with p gives **linear equations** in the entries of Q:

For each monomial $\mathbf{x}^\gamma$ (of degree $\le 2d$):

$$\sum_{\alpha+\beta=\gamma} Q_{\alpha,\beta} = p_\gamma$$

where the sum is over all pairs of basis monomials (α, β) in z whose exponents add up to γ, and p_γ is the coefficient of $\mathbf{x}^\gamma$ in p (0 if absent).

So checking SOS is an **SDP feasibility problem**:

$$\text{Find } Q \in \mathbb{R}^{s\times s} \quad \text{s.t.} \quad \sum_{\alpha+\beta=\gamma} Q_{\alpha,\beta} = p_\gamma \ \ \forall\,\gamma, \quad Q \succeq 0$$

Feasible $\Rightarrow$ p is SOS (Cholesky of Q gives the decomposition). **Infeasible** $\Rightarrow$ p is not SOS.

6.1.5 Worked Example: Univariate Degree 4

Consider $p(x) = x^4 - 2x^3 + 3x^2 - 2x + 1$. Since $\deg(p) = 4$, we have $d = 2$.

Monomial vector: $z = [1,\ x,\ x^2]^T$ (degree ≤ 2), so Q is 3×3.

Expand $z^T Q z$:

$$z^T Q z = Q_{00} \cdot 1 + 2Q_{01} \cdot x + (Q_{11} + 2Q_{02}) \cdot x^2 + 2Q_{12} \cdot x^3 + Q_{22} \cdot x^4$$

Match coefficients with p:

Monomial	Equation	Result
1	$Q_{00} = 1$	$Q_{00} = 1$
x	$2Q_{01} = -2$	$Q_{01} = -1$

Monomial	Equation	Result
x^2	$Q_{11} + 2Q_{02} = 3$	one free parameter
x^3	$2Q_{12} = -2$	$Q_{12} = -1$
x^4	$Q_{22} = 1$	$Q_{22} = 1$

5 equations in 6 unknowns $\Rightarrow$ **one free parameter** Q_{02}. Setting $Q_{02} = 1$ gives $Q_{11} = 1$:

$$Q = \begin{pmatrix} 1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & -1 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 & -1 & 1 \end{pmatrix} \succeq 0$$

The SDP found a valid Q, so p is SOS: $p(x) = (1 - x + x^2)^2$.

6.1.6 The Motzkin Polynomial

The Motzkin polynomial $M(x, y) = x^4y^2 + x^2y^4 - 3x^2y^2 + 1$ has degree 6 in 2 variables, so $d = 3$. The monomial vector has $\binom{2+3}{3} = 10$ entries, making Q a 10×10 matrix.

Setting up the coefficient matching constraints and requiring $Q \succeq 0$ gives an **infeasible** SDP — confirming that M is not SOS, despite being nonneg.

6.2 What You Must Implement

Submit a file named **`hw6_p6_sos.py`** with the following function. You will need `cvxpy` (`pip install cvxpy`).

```python
import numpy as np
import cvxpy as cp
from itertools import combinations_with_replacement


# --- Provided helper (you don't need to implement this) ---
def monomial_list(n_vars, max_degree):
    """
    Return list of all monomial exponent tuples of degree <= max_degree.

    Example: monomial_list(2, 2) returns:
        [(0,0), (1,0), (0,1), (2,0), (1,1), (0,2)]
    representing: 1, x, y, x^2, xy, y^2
    """
    monoms = []
    for deg in range(max_degree + 1):
        for combo in combinations_with_replacement(range(n_vars), deg):
            exp = [0] * n_vars
            for v in combo:
                exp[v] += 1
            monoms.append(tuple(exp))
    return monoms
```

```
# ------------------------------------------------------------


def check_sos(poly_coeffs, n_vars):
    """
    Check if a polynomial is a sum of squares (SOS) using semidefinite programming.

    Given a polynomial p(x) specified by its coefficients, determine whether there
    exists a positive semidefinite Gram matrix Q such that p(x) = z(x)^T Q z(x),
    where z(x) is the vector of all monomials of degree <= deg(p)/2.

    Parameters
    ----------
    poly_coeffs : dict
        Polynomial coefficients as {exponent_tuple: coefficient}.
        Each key is a tuple of length n_vars giving the exponent of each variable.
        Example for p(x,y) = x^4*y^2 + x^2*y^4 - 3*x^2*y^2 + 1:
            {(4,2): 1, (2,4): 1, (2,2): -3, (0,0): 1}
        Example for p(x) = x^4 - 2x^3 + 3x^2 - 2x + 1:
            {(4,): 1, (3,): -2, (2,): 3, (1,): -2, (0,): 1}
    n_vars : int
        Number of variables in the polynomial.

    Returns
    -------
    result : dict with keys:
        'is_sos' : bool - True if the polynomial is SOS, False otherwise
        'Q'      : np.ndarray or None - the Gram matrix (s x s) if SOS, None otherwise
    """
    raise NotImplementedError
```

6.3 Hints

- **Step 1: Determine the degree.** The polynomial's degree is max $\sum_j \alpha_j$ over all exponent tuples α in `poly_coeffs`. If the degree is odd, the polynomial cannot be SOS (return `False` immediately). Set $d = \text{degree}/2$.
- **Step 2: Build the monomial list** of degree $\leq d$ using `monomial_list(n_vars, d)`. Create a dictionary mapping each monomial tuple to its index: `mono_to_idx = {m: i for i, m in enumerate(monoms)}`.
- **Step 3: Create the Gram matrix** as a `cvxpy` symmetric PSD variable:

  ```
  Q = cp.Variable((s, s), symmetric=True)
  constraints = [Q >> 0]
  ```

- **Step 4: Coefficient matching.** For each target monomial γ that can appear in $z^T Q z$, add the constraint:

  ```
  expr = 0
  ```

```
for i, alpha in enumerate(monoms):
    beta = tuple(g - a for g, a in zip(gamma, alpha))
    if all(b >= 0 for b in beta) and beta in mono_to_idx:
        j = mono_to_idx[beta]
        expr += Q[i, j]
constraints.append(expr == poly_coeffs.get(gamma, 0.0))
```

- **Step 5: Solve** the feasibility SDP:

  ```
  prob = cp.Problem(cp.Minimize(0), constraints)
  prob.solve(solver=cp.SCS)
  ```

  Check `prob.status` — if `'optimal'` or `'optimal_inaccurate'`, the polynomial is SOS. Or alternatively, Consider:

  ```
  try:
    prob.solve(solver=cp.SCS, verbose=False, max_iters=10000)
  except Exception:
    return {'is_sos': False, 'Q': None}
  ```

- **Expected Q sizes:** For $p(x)$ of degree 4 in 1 variable: Q is 3×3. For Motzkin (degree 6, 2 variables): Q is 10×10.

6.4 Sanity Checks

```
import numpy as np

# SOS: p(x) = (1 - x + x^2)^2 = x^4 - 2x^3 + 3x^2 - 2x + 1
result = check_sos({(4,): 1, (3,): -2, (2,): 3, (1,): -2, (0,): 1}, n_vars=1)
print(f"x^4 - 2x^3 + 3x^2 - 2x + 1 is SOS: {result['is_sos']}")  # True
if result['Q'] is not None:
    eigvals = np.linalg.eigvalsh(result['Q'])
    print(f"Q shape: {result['Q'].shape}")              # (3, 3)
    print(f"Min eigenvalue of Q: {min(eigvals):.6f}")  # >= 0 (or very small negative)

# NOT SOS: Motzkin polynomial
motzkin = {(4,2): 1, (2,4): 1, (2,2): -3, (0,0): 1}
result2 = check_sos(motzkin, n_vars=2)
print(f"Motzkin polynomial is SOS: {result2['is_sos']}")  # False
```