Optimal Arbitrage of Energy Storage Systems

A Challenge Proposal for TIG

CEL

December 1, 2025

1 Introduction

Electricity markets create profit opportunities for storage operators who can buy power when prices are low and sell when prices spike. The core difficulty is uncertainty: real-time prices deviate unpredictably from day-ahead forecasts due to weather, demand shocks, and transmission congestion. This challenge asks solvers to find operational policies that maximize expected profit under these stochastic conditions while respecting physical constraints.

We structure the problem in two levels. Level 1 considers a single battery facing temporal price volatility. Level 2 extends to a portfolio of batteries distributed across a transmission-constrained network, introducing spatial price differences and coupling constraints. Both levels admit a clean mathematical formulation as stochastic optimal control problems, with difficulty parameters that scale computational hardness while preserving verifiability.

2 Level 1: Single-Asset Temporal Arbitrage

2.1 Problem Statement

A battery operator observes a day-ahead price forecast and must commit to charge/discharge decisions as real-time prices unfold. The real-time price at each step depends on the operator's previous action—a mechanism we introduce to prevent lookahead cheating. The goal is to maximize expected profit over a finite horizon.

2.2 State Space and Dynamics

Let $T = \{0, 1, \ldots, H-1\}$ denote discrete time steps of duration Δt. The battery state is characterized by its state of charge $E_t \in [\underline{E}, \overline{E}]$, measured in MWh. At each step, the operator chooses charging power $c_t \geq 0$ and discharging power $d_t \geq 0$, subject to:

$$c_t \leq \overline{P}^c, \quad d_t \leq \overline{P}^d, \tag{1}$$

$$c_t \cdot d_t = 0. \tag{2}$$

Constraint (2) prevents simultaneous charging and discharging. The state evolves according to:

$$E_{t+1} = E_t + \eta^c c_t \Delta t - \frac{d_t \Delta t}{\eta^d}, \tag{3}$$

where $\eta^c, \eta^d \in (0, 1]$ are charging and discharging efficiencies. The round-trip efficiency is $\eta = \eta^c \eta^d$.

2.3 Price Model with Commitment

The day-ahead price curve $\{\lambda_t^{\mathrm{DA}}\}_{t\in T}$ is known at $t = 0$. Real-time prices $\{\lambda_t^{\mathrm{RT}}\}_{t\in T}$ are revealed sequentially, with a crucial property: the distribution of $\lambda_{t+1}^{\mathrm{RT}}$ depends on the action $a_t = (c_t, d_t)$ taken at time t.

This dependency prevents solvers from pre-computing optimal responses to all future prices. We implement it via a hash-based commitment scheme.

Definition 1 (Action-Committed Price Generation)**.** *Let $\mathcal{H} : \{0,1\}^* \to \{0,1\}^{256}$ be a cryptographic hash. Define the seed sequence recursively:*

$$s_{t+1} = \mathcal{H}\big(s_t \,\|\, a_t \,\|\, t \,\|\, E_t\big), \tag{4}$$

where $\|$ denotes concatenation. The real-time price is:

$$\lambda_{t+1}^{RT} = \lambda_{t+1}^{DA} \cdot \big(1 + \mu_{t+1} + \sigma \cdot \xi_{t+1}\big) + J_{t+1}, \tag{5}$$

where $\xi_{t+1} \sim \mathcal{N}(0,1)$ is drawn from a PRNG seeded by s_{t+1}, μ_t is a deterministic bias, $\sigma > 0$ controls volatility, and J_t is a jump component for tail events.

The jump component J_t models scarcity pricing. With probability ρ, we set $J_t = \lambda_t^{\mathrm{DA}} \cdot Z$ where $Z \sim \mathrm{Pareto}(\alpha)$ for some tail index $\alpha > 2$; otherwise $J_t = 0$.

2.4 Objective Function

The instantaneous profit at time t is:

$$r_t = (d_t - c_t) \cdot \lambda_t^{\mathrm{RT}} \cdot \Delta t - \phi(c_t, d_t), \tag{6}$$

where $\phi(c, d)$ represents market frictions:

$$\phi(c, d) = \kappa_{\mathrm{tx}}(c + d)\Delta t + \kappa_{\mathrm{deg}} \cdot \mathrm{DoD}(c, d). \tag{7}$$

Here κ_{tx} is a transaction cost per MWh, κ_{deg} is a degradation cost, and DoD measures depth of discharge. A common model is $\mathrm{DoD}(c, d) = (d \cdot \Delta t/\overline{E})^{\beta}$ for some $\beta > 1$.

The operator seeks a policy $\pi : (E_t, \lambda_{t:H-1}^{\mathrm{DA}}, \lambda_t^{\mathrm{RT}}) \mapsto (c_t, d_t)$ that maximizes:

$$\max_{\pi} \; \mathbb{E}\left[\sum_{t=0}^{H-1} r_t \;\middle|\; E_0, \lambda^{\mathrm{DA}}, s_0\right]. \tag{8}$$

2.5 Bellman Formulation

Define the value function $V_t(E)$ as the expected profit-to-go from state E at time t. Terminal condition: $V_H(E) = 0$. For $t < H$:

$$V_t(E) = \max_{(c,d)\in\mathcal{A}(E)} \; \mathbb{E}\Big[r_t + V_{t+1}\big(E + \eta^c c\Delta t - d\Delta t/\eta^d\big)\Big], \tag{9}$$

where $\mathcal{A}(E)$ is the set of feasible actions from state E:

$$\mathcal{A}(E) = \Big\{(c, d) : c, d \geq 0,\; cd = 0,\; c \leq \overline{P}^c,\; d \leq \overline{P}^d,\; E + \eta^c c\Delta t - d\Delta t/\eta^d \in [\underline{E}, \overline{E}]\Big\}. \tag{10}$$

The expectation in (9) is over the conditional distribution of λ_t^{RT} given the current information. Due to the commitment mechanism (4), this distribution cannot be sampled without committing to (c, d).

2.6 Benchmark: Approximate Dynamic Programming

Stochastic Dual Dynamic Programming (SDDP) approximates V_t by a piecewise-linear lower bound constructed from Benders cuts. We adapt SDDP to our setting by sampling price scenarios in the forward pass and constructing cuts in the backward pass.

Algorithm 1 SDDP for Single-Asset Arbitrage

```
1: Input: Horizon $H$, sample count $N$, iterations $K$
2: Initialize: $\mathcal{C}_t \leftarrow \emptyset$ for all $t$ (cut sets)
3: for $k = 1, \ldots, K$ do
4:     for $n = 1, \ldots, N$ do                                              ▷ Forward pass
5:         Sample $s_0^{(n)}$; set $E_0^{(n)} \leftarrow E_0$
6:         for $t = 0, \ldots, H-1$ do
7:             Solve: $(c_t^*, d_t^*, \theta_t^*) \leftarrow \arg\max\ r_t(c,d) + \theta$
               s.t. $(c,d) \in \mathcal{A}(E_t^{(n)})$, $\theta \le \alpha + \beta E_{t+1}$ for all $(\alpha, \beta) \in \mathcal{C}_{t+1}$
8:             Commit action; generate $\lambda_{t+1}^{\mathrm{RT}}$ via (4)–(5)
9:             Update $E_{t+1}^{(n)}$ via (3)
10:        end for
11:    end for
12:    for $t = H-1, \ldots, 0$ do                                            ▷ Backward pass
13:        for $n = 1, \ldots, N$ do
14:            At state $E_t^{(n)}$, solve the stage problem; extract dual $\mu_t^{(n)}$ on state constraint
15:            Construct cut: $\mathcal{C}_t \leftarrow \mathcal{C}_t \cup \{(\alpha^{(n)}, \beta^{(n)})\}$
16:        end for
17:    end for
18: end for
19: Return: Policy defined by cut sets $\{\mathcal{C}_t\}$
```

Remark 1. *The commitment mechanism* (4) *means we cannot sample future prices during the forward pass without committing to actions. SDDP therefore operates on a* relaxation*: we sample prices from the unconditional distribution $\lambda_t^{RT} \sim \lambda_t^{DA}(1 + \mu_t + \sigma\xi_t) + J_t$ with $\xi_t \sim \mathcal{N}(0,1)$ and J_t from the jump model. The true objective* (8) *uses action-committed prices, so SDDP provides an upper bound. The gap measures the value of lookahead information that the mechanism denies.*

2.7 Difficulty Parameters

Three parameters control problem hardness:

Volatility $\sigma \in [0.1, 0.5]$: Higher values increase the variance of real-time prices around the day-ahead forecast. This widens the distribution of possible outcomes, making robust policies harder to find.

Tail index $\alpha \in (2, 5]$: Smaller values produce heavier tails in the jump distribution, with more frequent extreme spikes. The expected jump size is $\mathbb{E}[Z] = \alpha/(\alpha - 1)$ for $\alpha > 1$, diverging as $\alpha \to 1$.

Friction $\kappa = (\kappa_{\mathrm{tx}}, \kappa_{\mathrm{deg}})$: Higher costs reduce profit margins and penalize excessive cycling. This creates a tradeoff between capturing spreads and preserving battery life.

3 Level 2: Portfolio Arbitrage on a Constrained Network

3.1 Problem Statement

An operator manages m batteries at nodes $\mathcal{N} = \{1, \ldots, n\}$ of a transmission network. Each node i has locational marginal price $\lambda_{i,t}^{\mathrm{RT}}$, which can differ across nodes due to congestion. The operator must coordinate all batteries to maximize portfolio profit while respecting network flow limits.

3.2 Network Model

The transmission network is a graph $(\mathcal{N}, \mathcal{L})$ where $\mathcal{L}$ is the set of lines. Under the DC power flow approximation, power flows are linear in nodal injections. Let $p_i \in \mathbb{R}$ denote net injection at node i (positive for generation, negative for load). The flow on line $\ell = (i, j)$ is:

$$f_\ell = \sum_{k \in \mathcal{N}} \mathrm{PTDF}_{\ell k} \cdot p_k, \tag{11}$$

where $\mathrm{PTDF} \in \mathbb{R}^{|\mathcal{L}| \times n}$ is the Power Transfer Distribution Factor matrix.

Proposition 1 (PTDF Construction)**.** *Let $B \in \mathbb{R}^{n \times n}$ be the bus susceptance matrix with $B_{ij} = -b_{ij}$ for $i \neq j$ (where b_{ij} is line susceptance) and $B_{ii} = \sum_{j \neq i} b_{ij}$. Choose a slack bus s and let $\tilde{B}$ be B with row and column s deleted. Define $X = \tilde{B}^{-1}$ (extended with zeros for the slack). Then:*

$$PTDF_{\ell k} = b_\ell (X_{ik} - X_{jk}), \tag{12}$$

where $\ell = (i, j)$ and b_ℓ is the susceptance of line ℓ.

3.3 State Space and Constraints

Each battery $b \in \mathcal{B}$ is located at node $\nu(b) \in \mathcal{N}$ with parameters $(\underline{E}_b, \overline{E}_b, \overline{P}_b^c, \overline{P}_b^d, \eta_b^c, \eta_b^d)$. Let $E_{b,t}$ denote its state of charge and $(c_{b,t}, d_{b,t})$ its charge/discharge powers. The dynamics (3) apply to each battery independently.

The net injection at node i from storage is:

$$p_{i,t}^{\mathrm{stor}} = \sum_{b:\nu(b)=i} (d_{b,t} - c_{b,t}). \tag{13}$$

Total net injection includes exogenous generation and load:

$$p_{i,t} = p_{i,t}^{\mathrm{gen}} - p_{i,t}^{\mathrm{load}} + p_{i,t}^{\mathrm{stor}}. \tag{14}$$

Line flows must satisfy thermal limits:

$$|f_{\ell,t}| = \left| \sum_{k \in \mathcal{N}} \mathrm{PTDF}_{\ell k} \cdot p_{k,t} \right| \leq \overline{F}_\ell \quad \forall \ell \in \mathcal{L}. \tag{15}$$

3.4 Nodal Price Model

Locational marginal prices arise from the dual variables of the power balance and flow constraints in the economic dispatch problem. We model them as correlated stochastic processes:

$$\lambda_{i,t}^{\mathrm{RT}} = \lambda_{i,t}^{\mathrm{DA}} + \sigma_i \xi_{i,t} + \gamma \cdot \mathbf{1}\{\text{line incident to } i \text{ is congested}\} \cdot \zeta_t, \tag{16}$$

where $\xi_{i,t}$ are node-specific shocks, $\gamma > 0$ is a congestion premium, and ζ_t is a common factor. The correlation structure reflects that prices at connected nodes tend to move together unless separated by congested lines.

3.5 Objective

The portfolio profit at time t is:

$$R_t = \sum_{b \in \mathcal{B}} (d_{b,t} - c_{b,t}) \cdot \lambda^{\mathrm{RT}}_{\nu(b),t} \cdot \Delta t - \sum_{b \in \mathcal{B}} \phi_b(c_{b,t}, d_{b,t}). \tag{17}$$

The operator maximizes $\mathbb{E}[\sum_t R_t]$ subject to (15) and battery constraints.

3.6 Decomposition Approach

The coupling constraints (15) link all batteries. We apply Benders decomposition by treating battery dispatch as complicating variables.

Master problem: Determines battery schedules $(c_{b,t}, d_{b,t})$ for all b, t, using cuts from subproblems.

Subproblem (one per scenario ω): Given battery schedules, checks feasibility of network flows. If feasible, returns optimal dispatch cost; if infeasible, returns a feasibility cut.

Algorithm 2 Benders Decomposition for Portfolio Arbitrage

```
 1: Input: Network $(\mathcal{N}, \mathcal{L})$, batteries $\mathcal{B}$, scenarios $\Omega$
 2: Initialize: LB $\leftarrow -\infty$, UB $\leftarrow +\infty$, cut set $\mathcal{C} \leftarrow \emptyset$
 3: while UB $-$ LB $> \epsilon$ do
 4:     Master: Solve
            $\max \ \sum_{b,t} (d_{b,t} - c_{b,t}) \cdot \bar{\lambda}_{\nu(b),t} \cdot \Delta t + \theta$
            s.t. battery constraints $\forall b, t$
                 $\theta \le g(\mathbf{c}, \mathbf{d}) \quad \forall$ cuts in $\mathcal{C}$
 5:     Let $(\mathbf{c}^*, \mathbf{d}^*, \theta^*)$ be optimal; update LB $\leftarrow$ obj$^*$
 6:     for $\omega \in \Omega$ do
 7:         Subproblem: Fix $(\mathbf{c}^*, \mathbf{d}^*)$, solve power flow feasibility
 8:         if infeasible then
 9:             Extract dual ray $\boldsymbol{\mu}^\omega$; add feasibility cut to $\mathcal{C}$
10:         else
11:             Compute scenario profit $\Pi^\omega$; extract duals $\boldsymbol{\nu}^\omega$
12:             Add optimality cut to $\mathcal{C}$
13:         end if
14:     end for
15:     Update UB $\leftarrow$ LB $+ \frac{1}{|\Omega|} \sum_\omega \Pi^\omega - \theta^*$
16: end while
17: Return: Optimal schedules $(\mathbf{c}^*, \mathbf{d}^*)$
```

3.7 Difficulty Parameters

Network size $n = |\mathcal{N}|$: More nodes increase the state dimension and the number of coupling constraints.

Congestion factor $\gamma_{\mathrm{cong}} \in (0, 1]$: Scales line limits as $\overline{F}_\ell \leftarrow \gamma_{\mathrm{cong}} \cdot \overline{F}_\ell^{\mathrm{nom}}$. Lower values create more binding constraints and larger price separations.

Portfolio heterogeneity $h \in [0, 1]$: Controls dispersion of battery parameters. With $h = 0$, all batteries are identical; with $h = 1$, parameters vary by a factor of 3.

4 Verification Protocol

A solution to Level 1 is a transcript $\tau = \{(t, a_t, E_t, s_t, \lambda_t^{\mathrm{RT}}, r_t)\}_{t=0}^{H-1}$. Verification proceeds as:

1. Check s_0 matches the instance specification.
2. For each t: recompute s_{t+1} via (4), regenerate $\lambda_{t+1}^{\mathrm{RT}}$ via (5), verify against transcript.
3. Check all constraints (1)–(3) are satisfied.
4. Recompute r_t via (6); sum to get total profit.

This takes $O(H)$ time, making verification efficient. The hash commitment ensures that any modification to the action sequence invalidates subsequent prices.

For Level 2, verification additionally checks flow constraints (15) at each step, which requires $O(|\mathcal{L}| \cdot n)$ operations per step.

5 Instance Generation

5.1 Day-Ahead Price Curve

We model day-ahead prices as a Gaussian process with a periodic kernel to capture diurnal patterns:

$$k(t, t') = \sigma_p^2 \exp\left(-\frac{2\sin^2(\pi|t-t'|/24)}{\ell_p^2}\right) + \sigma_{\mathrm{SE}}^2 \exp\left(-\frac{(t-t')^2}{2\ell_{\mathrm{SE}}^2}\right). \tag{18}$$

The first term captures 24-hour periodicity; the second allows smooth deviations. Sampling $\lambda^{\mathrm{DA}} \sim \mathcal{GP}(\mu, k)$ with mean function $\mu(t) = \bar{\lambda}(1 + 0.3\sin(2\pi t/24))$ produces realistic price curves.

5.2 Reference Implementation

Listing 1: Level 1 Instance Generator

```python
import numpy as np
from dataclasses import dataclass
from hashlib import sha256

@dataclass
class Battery:
    capacity: float   # MWh
    power: float      # MW
    efficiency: float
    soc_min: float = 0.1
    soc_max: float = 0.9

@dataclass
class MarketParams:
    volatility: float = 0.2
    tail_prob: float = 0.05
    tail_index: float = 3.0
    tx_cost: float = 0.5
    deg_cost: float = 1.0

def gp_sample(T: int, seed: int) -> np.ndarray:
    """Sample day-ahead prices from GP with periodic kernel."""
    rng = np.random.default_rng(seed)
    t = np.arange(T)

    # Kernel parameters
    sigma_p, ell_p = 15.0, 4.0
    sigma_se, ell_se = 5.0, 6.0

    # Build covariance matrix
    dt = t[:, None] - t[None, :]
    K = (sigma_p**2 * np.exp(-2 * np.sin(np.pi * np.abs(dt) / 24)**2 / ell_p
        **2)
         + sigma_se**2 * np.exp(-dt**2 / (2 * ell_se**2)))
    K += 1e-6 * np.eye(T)

    # Mean function: base load + diurnal pattern
    mu = 50 + 20 * np.sin(2 * np.pi * t / 24 - np.pi/2)

    L = np.linalg.cholesky(K)
    return mu + L @ rng.standard_normal(T)

class Environment:
    def __init__(self, battery: Battery, params: MarketParams,
                 da_prices: np.ndarray, seed: int):
        self.bat = battery
        self.params = params
        self.da = np.maximum(da_prices, 5.0)  # floor at $5/MWh
        self.T = len(da_prices)
        self.seed = seed.to_bytes(8, 'big')
        self.reset()

    def reset(self):
        self.t = 0
        self.soc = self.bat.capacity * 0.5
        self.state = self.seed
        self.rt_price = self._generate_rt(self.da[0])
        return self._obs()

    def _generate_rt(self, da: float) -> float:
        """Generate RT price from committed state."""
        rng = np.random.default_rng(int.from_bytes(self.state[:8], 'big'))
        noise = self.params.volatility * rng.standard_normal()

        # Jump component
        jump = 0.0
        if rng.random() < self.params.tail_prob:
            jump = da * (rng.pareto(self.params.tail_index) + 1)

        return max(0.0, da * (1 + noise) + jump)

    def step(self, charge: float, discharge: float) -> tuple:
        """Execute action, return (obs, reward, done, info)."""
        # Enforce constraints
        charge = np.clip(charge, 0, self.bat.power)
        discharge = np.clip(discharge, 0, self.bat.power)
        if charge > 0 and discharge > 0:
            discharge = 0  # prioritize charging

        # Check SOC feasibility
        new_soc = (self.soc + self.bat.efficiency * charge
                   - discharge / self.bat.efficiency)
        new_soc = np.clip(new_soc,
                          self.bat.soc_min * self.bat.capacity,
                          self.bat.soc_max * self.bat.capacity)

        # Compute reward
        profit = (discharge - charge) * self.rt_price
        friction = (self.params.tx_cost * (charge + discharge)
                    + self.params.deg_cost * (discharge / self.bat.capacity)
                        **2)
        reward = profit - friction

        # Commit action to state
        action_bytes = np.array([charge, discharge]).tobytes()
        self.state = sha256(self.state + action_bytes
                             + self.t.to_bytes(4, 'big')).digest()

        # Advance
        self.soc = new_soc
        self.t += 1
        done = self.t >= self.T

        if not done:
            self.rt_price = self._generate_rt(self.da[self.t])

        return self._obs(), reward, done, {'soc': self.soc}

    def _obs(self):
        return {
            'soc': self.soc,
            'da_price': self.da[self.t] if self.t < self.T else 0,
            'rt_price': self.rt_price,
            'time': self.t
        }
```

Listing 2: Level 2 Network and Portfolio

```
import numpy as np
from scipy.optimize import linprog

def build_ring_network(n: int, susceptance: float = 10.0) -> dict:
    """Construct n-node ring network with PTDF matrix."""
    # Incidence matrix: n lines for ring
    A = np.zeros((n, n))
    for i in range(n):
        j = (i + 1) % n
        A[i, i] = 1
        A[i, j] = -1

    # Bus susceptance matrix
    B = np.zeros((n, n))
    for i in range(n):
        j = (i + 1) % n
        B[i, i] += susceptance
        B[j, j] += susceptance
        B[i, j] -= susceptance
        B[j, i] -= susceptance

    # Remove slack (node 0)
    B_red = B[1:, 1:]
    X = np.zeros((n, n))
    X[1:, 1:] = np.linalg.inv(B_red)

    # PTDF: flow on line i for injection at node k
    PTDF = np.zeros((n, n))
    for i in range(n):
        j = (i + 1) % n
        PTDF[i, :] = susceptance * (X[i, :] - X[j, :])

    return {'n_nodes': n, 'n_lines': n, 'PTDF': PTDF, 'susceptance':
        susceptance}

def solve_portfolio_lp(network: dict, batteries: list,
                       prices: np.ndarray, line_limits: np.ndarray,
                       T: int) -> dict:
    """
    Solve deterministic portfolio arbitrage via LP.

    batteries: list of dicts with keys 'node', 'capacity', 'power', 'efficiency
        '
    prices: shape (n_nodes, T)
    line_limits: shape (n_lines,)
    """
    n_bat = len(batteries)
    PTDF = network['PTDF']

    # Variables: [c_{b,t}, d_{b,t}] for b in batteries, t in T
    # Order: c_0_0, d_0_0, c_0_1, d_0_1, ..., c_B_T, d_B_T
    n_vars = 2 * n_bat * T

    # Objective: maximize sum of (d - c) * price
    c_obj = np.zeros(n_vars)
    for b_idx, bat in enumerate(batteries):
        node = bat['node']
        for t in range(T):
            idx_c = 2 * (b_idx * T + t)
            idx_d = idx_c + 1
            c_obj[idx_c] = prices[node, t]   # cost of charging
            c_obj[idx_d] = -prices[node, t]  # revenue from discharge

    # Inequality constraints: A_ub @ x <= b_ub
    A_ub_list, b_ub_list = [], []

    # Line flow limits: |sum_k PTDF_{l,k} * p_k| <= F_l
    # p_k = sum_{b at node k} (d_b - c_b)
    for t in range(T):
        for l in range(network['n_lines']):
            row_pos = np.zeros(n_vars)
            row_neg = np.zeros(n_vars)
            for b_idx, bat in enumerate(batteries):
                node = bat['node']
                idx_c = 2 * (b_idx * T + t)
                idx_d = idx_c + 1
                row_pos[idx_c] = -PTDF[l, node]
                row_pos[idx_d] = PTDF[l, node]
                row_neg[idx_c] = PTDF[l, node]
                row_neg[idx_d] = -PTDF[l, node]
            A_ub_list.extend([row_pos, row_neg])
            b_ub_list.extend([line_limits[l], line_limits[l]])

    # SOC constraints via cumulative sums
    for b_idx, bat in enumerate(batteries):
        eff = bat['efficiency']
        cap = bat['capacity']
        soc_init = cap * 0.5

        for t in range(T):
            # SOC_t = soc_init + sum_{s<=t} (eff*c_s - d_s/eff)
            # Require soc_min*cap <= SOC_t <= soc_max*cap
            row_upper = np.zeros(n_vars)
            row_lower = np.zeros(n_vars)
            for s in range(t + 1):
                idx_c = 2 * (b_idx * T + s)
                idx_d = idx_c + 1
                row_upper[idx_c] = eff
                row_upper[idx_d] = -1/eff
                row_lower[idx_c] = -eff
                row_lower[idx_d] = 1/eff

            A_ub_list.append(row_upper)
            b_ub_list.append(0.9 * cap - soc_init)
            A_ub_list.append(row_lower)
            b_ub_list.append(soc_init - 0.1 * cap)

    A_ub = np.array(A_ub_list)
    b_ub = np.array(b_ub_list)

    # Bounds: 0 <= c, d <= power
    bounds = []
    for b_idx, bat in enumerate(batteries):
        for t in range(T):
            bounds.append((0, bat['power']))  # charge
            bounds.append((0, bat['power']))  # discharge

    result = linprog(c_obj, A_ub=A_ub, b_ub=b_ub, bounds=bounds, method='highs'
       )

    if result.success:
        x = result.x
        schedule = {}
        for b_idx, bat in enumerate(batteries):
            schedule[b_idx] = {
                'charge': [x[2*(b_idx*T + t)] for t in range(T)],
                'discharge': [x[2*(b_idx*T + t) + 1] for t in range(T)]
            }
        return {'success': True, 'profit': -result.fun, 'schedule': schedule}
    else:
        return {'success': False, 'message': result.message}
```

6 Conclusion

This challenge formulates energy storage arbitrage as a stochastic optimal control problem with three distinguishing features. First, the action-committed price generation mechanism ensures computational asymmetry: solving is hard, verification is easy, and lookahead exploitation is impossible. Second, the two-level structure provides a progression from single-asset temporal arbitrage to portfolio coordination under network constraints. Third, the difficulty parameters allow precise scaling of problem hardness while maintaining the core structure.

The combination of realistic market dynamics, physical constraints, and cryptographic commitment makes this a suitable challenge for TIG's proof-of-work framework.

A Notation Reference

Symbol	**Description**
T, H	Time horizon (set and length)
Δt	Time step duration
E_t	State of charge at time t (MWh)
$\underline{E}, \overline{E}$	SOC bounds
c_t, d_t	Charge/discharge power (MW)

Symbol	**Description**
$\overline{P}^c, \overline{P}^d$	Power limits
η^c, η^d	Charging/discharging efficiency
λ_t^{DA}	Day-ahead price (\$/MWh)
λ_t^{RT}	Real-time price
s_t	Commitment seed at time t
σ	Price volatility
ρ, α	Jump probability and tail index
$\phi(\cdot)$	Friction cost function
$V_t(E)$	Value function
$\mathcal{N}, \mathcal{L}$	Nodes and lines of network
$\mathcal{B}$	Set of batteries
$\nu(b)$	Node location of battery b
PTDF	Power transfer distribution factors
f_ℓ	Flow on line ℓ
$\overline{F}_\ell$	Line flow limit