Homework: Trajectory Optimization for Supermarket Refrigeration Control

This assignment adapts the benchmark from Larsen et al. (2007), "Supermarket Refrigeration System – Benchmark for Hybrid System Control" (see `paper.md`). Your job is to build a trajectory-optimization controller that coordinates valves and compressors and outperforms a well-tuned PID baseline on energy while respecting temperature and pressure limits.

Problem Setting

Unlike the single-unit heating and cooling systems in homes, commercial supermarket refrigeration involves coordinating multiple pieces of equipment. Think of a row of open display cases (the glass-fronted refrigerated shelves where milk, yogurt, or deli items sit). Each case contains a coil: a metal tube where a special fluid evaporates and pulls heat out of the air to keep food cold. The resulting vapor from all cases flows into a shared pipe called the suction manifold, which feeds a rack of compressors. These compressors are the workhorses: they compress the vapor, concentrating the heat so it can be rejected elsewhere. In short, the system moves heat around by compressing a fluid.

The control challenge is to run these compressors smartly: don't fire them all at once when you only need a little cooling, and ramp their capacity up or down dynamically to match the actual cooling load (which varies between busy daytime hours and quiet nights). Right now, each display case runs a simple rule: "too warm? open my valve and flood the coil with liquid; too cold? close the valve and let it boil off." Because all cases look identical, they have a tendency to synchronize the opening and closing of their valves together. This causes surges of vapor into the manifold, wild pressure swings, and compressors cycling on and off wastefully. Your job is to replace this chaotic, decentralized behavior with a coordinated plan computed by MPC with multiple shooting, ensuring the compressors work efficiently together.

System at a glance

We use two display cases and a two-compressor rack (each 50% capacity). The shared state is the suction pressure P_{suc}. Per case $i \in \{1, 2\}$, we track three temperatures and one refrigerant mass: goods $T_{\text{goods},i}$, wall $T_{\text{wall},i}$, air $T_{\text{air},i}$, and liquid mass in the evaporator $M_{\text{ref},i}$. Collect these into

$$x = [T_{\text{goods},1}, T_{\text{goods},2}, T_{\text{wall},1}, T_{\text{wall},2}, T_{\text{air},1}, T_{\text{air},2}, M_{\text{ref},1}, M_{\text{ref},2}, P_{\text{suc}}]^\top \in \mathbb{R}^9.$$

Controls are binary case valves $\text{valve}_i \in \{0, 1\}$ and a continuous rack capacity $\text{comp} \in [0, 100]\%$. This makes the system hybrid: the dynamics switch with valve states.

Disturbances are the thermal loads on the cases and vapor flow from other refrigerated units (e.g., cold storage rooms) sharing the same suction manifold. Use a day/night profile over a 4-hour experiment: day (0–2 h) 3000 W per case with $\dot{m}_{\mathrm{ref,const}} = 0.2$ kg/s; night (2–4 h) 1800 W, $\dot{m}_{\mathrm{ref,const}} = 0$.

Dynamics (the physics under the hood)

At the conceptual level, we model our each case via three coupled thermal masses (goods, air, coil wall). Heat flows between them proportional to temperature differences, and the evaporating refrigerant in the coil extracts heat from the wall. The temperatures evolve as:

$$\frac{dT_{\mathrm{goods},i}}{dt}, \quad \frac{dT_{\mathrm{air},i}}{dt}, \quad \frac{dT_{\mathrm{wall},i}}{dt} \quad \sim \text{(heat in - heat out) / thermal mass}$$

The refrigerant mass $M_{\mathrm{ref},i}$ in the evaporator has **switched dynamics**: - **Valve open** ($\mathrm{valve}_i = 1$): liquid floods in quickly (~40 s time constant) - **Valve closed** ($\mathrm{valve}_i = 0$): remaining liquid boils off at a rate proportional to heat extraction; when dry, stays at zero

Shared suction pressure: A mass balance on the manifold:

$$\frac{dP_{\mathrm{suc}}}{dt} \sim \frac{\text{vapor in from cases} - \text{vapor out via compressors}}{\text{manifold volume}}.$$

Higher compressor capacity pulls more vapor out, lowering pressure. More cases evaporating refrigerant pushes pressure up.

Power consumption: The compressors do work compressing vapor. Power grows nonlinearly with capacity and with suction pressure (denser vapor = more molecules to compress).

All refrigerant properties (saturation temperature, density, latent heat) are handled by polynomial fits in the code. Your job is to plan valve openings and compressor capacity over time to minimize energy while keeping temperatures and pressure in bounds.

Constraints and what we measure

Food safety fixes air temperatures between 2 and 5 °C in each case; pressure must remain below 1.7 bar during day and 1.9 bar at night. We report three metrics over 4 hours: average squared constraint violation

$$\gamma_{\mathrm{con}} = \frac{1}{T}\int_0^T \left[\varepsilon_P(t)^2 + \tfrac{1}{n}\sum_i \varepsilon_{T,i}(t)^2\right] dt,$$

with $\varepsilon_P(t) = \max(0, P_{\mathrm{suc}} - P_{\max}(t))$ and $\varepsilon_{T,i}(t) = \max(0, T_{\mathrm{air},i} - 5) + \max(0, 2 - T_{\mathrm{air},i})$; a switching count

$$\gamma_{\mathrm{switch}} = \frac{1}{T}(N_{\mathrm{comp}} + \tfrac{1}{100} N_{\mathrm{valve}})$$

that prices compressor toggles 100× higher than valve toggles; and average power

$$\gamma_{\text{pow}} = \frac{1}{T} \int_0^T \dot{W}_{\text{comp}}(t)\, dt.$$

Your primary objective is energy, without violating constraints.

The baseline you must beat

Each valve operates on its own simple control loop that checks once per second. It turns **on** when the pressure or temperature goes above a certain upper limit and **off** when it drops below a lower limit. The gap between these two limits prevents the valve from rapidly switching on and off. In the literature, this is refered to as a hysteresis controller.

$$\text{valve}_i(k) = \begin{cases} 1, & T_{\text{air},i} > 5.0 \\ 0, & T_{\text{air},i} < 2.0 \\ \text{valve}_i(k-1), & \text{otherwise.} \end{cases}$$

A simple **feedback controller** (you can think of it as a hand-tuned policy based on error signals) keeps the suction pressure close to a target value. Every 60 seconds, it measures the difference

$$e = P_{\text{ref}} - P_{\text{suc}},$$

where P_{ref} is the desired suction pressure and P_{suc} is the current one.

The controller adjusts a control signal $u(t)$ based on two terms: 1. how large the error is right now, and 2. how the error has accumulated over time.

Formally,

$$u(t) = K_p e(t) + \frac{K_p}{\tau_I} \int e(t)\, dt.$$

However, to avoid reacting to tiny fluctuations (noise), the integral term is updated **only when** the error exceeds ±0.2 bar. This "dead zone" prevents unnecessary corrections.

The output $u(t)$ is **quantized** to one of three levels (0%, 50%, or 100%) corresponding to how many compressors are active. We use the following parameters: $K_p = -75$, $\tau_I = 50$ s, and reference pressures $P_{\text{ref}} = 1.5$ bar during the day and 1.7 bar at night.

Over a 4-hour simulation, this baseline controller achieves $\gamma_{\text{con}} = 2.72\ ^\circ\text{C}^2$, $\gamma_{\text{pow}} = 6.80$ kW, $\gamma_{\text{switch}} = 0.0038$.

Your controller: multiple shooting MPC

We solve a finite-horizon NLP every 3 minutes (18 steps of 10 s). Multiple shooting treats both control and state at each step as decision variables, enforcing dynamics with equality constraints. This improves conditioning and lets us impose box constraints on path states directly.

At time t_k, solve

$$\begin{aligned}\min_{u_{0:N-1},\, x_{1:N}} \quad & \sum_{j=0}^{N-1} \left[\tfrac{1}{1000}\, \dot{W}_{\text{comp}}(x_j, u_j) + 0.01\, |\Delta u_j|_1 \right] \\ \text{s.t.} \quad & x_0 = x(t_k), \\ & x_{j+1} = f(x_j, u_j, d_j), \quad j = 0, \dots, N-1, \\ & T_{\text{air},i}(x_j) \in [2, 5], \quad P_{\text{suc}}(x_j) \in [0.8, 1.7], \\ & \text{valve}_i \in [0, 1], \quad \text{comp} \in [0, 100].\end{aligned}$$

Discretize with forward Euler,

$$x_{j+1} = x_j + \Delta t\, \dot{x}(x_j, u_j, d_j), \quad \Delta t = 10\,\text{s}.$$

Use SLSQP (`scipy.optimize.minimize`) with JAX for objective and constraint Jacobians. Warm start the first window from a short PID rollout; then shift the previous plan.

After solving, round valve_i to $\{0, 1\}$ and quantize comp to $\{0, 50, 100\}$ before applying to the simulator.

What to implement

The JAX dynamics and PID reference live in `supermarket.py`. Your job is to complete **four main sections** in `trajectory_optimization.py`. These are marked with `TODO` comments and `raise NotImplementedError(...)`. Each section tests your understanding of a core concept in trajectory optimization.

Section 1: Objective Function

File: `trajectory_optimization.py`, inside `_setup_jax_functions()` → `objective(z)`

What to implement: Design the cost function that the optimizer will minimize. You need to compute:

1. **Power consumption:** Use Equation 15 from the paper:

$$\dot{W}_{\text{comp}} = \dot{V}_{\text{comp}} \cdot \text{power\_factor}(P_{\text{suc}})$$

where the volumetric flow rate is:

$$\dot{V}_{\text{comp}} = \eta_{\text{vol}} \cdot V_{\text{sl}} \cdot \frac{\text{comp\_percentage}}{100}$$

Available: `power_factor_jax(P_suc)`, `V_sl`, `eta_vol`

2. **Switching penalty:** Count control changes to discourage chattering:

$$\text{switch\_cost} = \sum_{j=1}^{N-1} |\text{comp}_j - \text{comp}_{j-1}| + \frac{1}{100} \sum_{i,j} |\text{valve}_{i,j} - \text{valve}_{i,j-1}|$$

   Use `jnp.diff()` and `jnp.abs()` to compute differences.

3. **Return weighted sum:**

$$J(z) = \frac{1}{1000} \cdot \text{power\_cost} + 0.01 \cdot \text{switch\_cost}$$

   Power is scaled to kW, switching weighted at 0.01.

The objective encodes your control priorities. Too much weight on switching → sluggish response. Too little → rapid on/off cycling.

Available variables: - `u_traj`: control trajectory `[horizon, n_u]` - `u_traj[:, :n_cases]` are valve commands `[0,1]` - `u_traj[:, n_cases]` is compressor percentage `[0,100]` - `x_full`: state trajectory `[horizon+1, n_x]` - `x_full[:, -1]` is `P_suc` (suction pressure)

Section 2: Dynamics Defects

File: `trajectory_optimization.py`, inside `_setup_jax_functions()` → `dynamics_defects(z)`

What to implement:

In multiple shooting: - States x_j are **decision variables** (not just outputs) - Dynamics become **equality constraints**: $g_j = x_{j+1} - f(x_j, u_j, d_j) = 0$

For each time step $j = 0, \ldots, N-1$: 1. Propagate dynamics: `x_predicted = dynamics_step(x[j], u[j], d[j])` 2. Compute defect: `defect[j] = x_predicted - x[j+1]` 3. Concatenate all defects into a flat vector

The optimizer will drive these defects to zero, ensuring the trajectory satisfies physics.

This setting allows you to enforce path constraints directly on states! Single shooting only optimizes controls; states follow from forward simulation and you can't constrain them directly. Multiple shooting optimizes **both** states and controls, letting you impose $T_{\text{air}} \in [2, 5]$ and $P_{\text{suc}} \leq 1.7$ as hard bounds.

Available: - `x_full`: state trajectory `[horizon+1, n_x]` - `u_traj`: control trajectory `[horizon, n_u]` - `dynamics_step(x, u, d)`: one-step Euler integrator - `self.d_traj_jax`: disturbance trajectory `[horizon, n+1]`

Return: Flat vector of shape `[horizon * n_x]` containing all defects.

Section 3: State Bounds

File: `trajectory_optimization.py`, inside `optimize()` → set up state bounds

What to implement: Set up box constraints on state variables at **all time steps** $j = 0, \ldots, N-1$.

State vector structure (per time step): $[T_{\text{goods}}, T_{\text{wall}}, T_{\text{air}}, M_{\text{ref}}, P_{\text{suc}}]$

Required bounds: 1. **Air temperature** (food safety): $T_{\text{air},i} \in [2, 5]$ °C for all cases i 2. **Suction pressure** (hardware limit): $P_{\text{suc}} \in [0.8, 1.7]$ bar 3. **Refrigerant mass** (physical): $M_{\text{ref},i} \in [0, 1]$ kg for all cases i 4. **Goods and wall temperatures:** No explicit bounds (use $-\infty, \infty$)

Indexing: For state component s at time step k: `idx = k * n_x + s`

For our 2-case system:

```
# State order within each time step:
# [T_goods_0, T_goods_1, T_wall_0, T_wall_1, T_air_0, T_air_1, M_ref_0, M_ref_1, P_suc]
#      0          1          2          3          4          5          6          7          8
```

Why this matters: These are **path constraints** – they must hold at every point in the trajectory, not just the final state. Multiple shooting enables this by making states decision variables that SLSQP can directly constrain.

Section 4: SLSQP Optimization Setup

File: `trajectory_optimization.py`, inside `optimize()` → create constraints and call `minimize()`

What to implement: Assemble and solve the constrained nonlinear program using SLSQP.

Steps:

1. **Create dynamics constraint:**

   ```
   from scipy.optimize import NonlinearConstraint

   dynamics_constraint = NonlinearConstraint(
       fun=dynamics_constraints_np,          # Function that returns defects
       lb=-tol * np.ones(...),               # Lower bound (0)
       ub=tol * np.ones(...),                # Upper bound (0)
       jac=dynamics_jacobian_np              # Jacobian of defects
   )
   ```

   Use `tol = 1e-3` for numerical stability.

2. **Call SLSQP:**

   ```
   from scipy.optimize import minimize
   ```

```
result = minimize(
    fun=objective_np,                    # Objective function
    x0=z0,                               # Initial guess
    method='SLSQP',                      # Sequential Quadratic Programming
    jac=objective_grad_np,               # Objective gradient (from JAX!)
    bounds=bounds,                       # Box constraints on variables
    constraints=[dynamics_constraint],    # List of constraints
    options={'maxiter': max_iter, 'ftol': 1e-6}
)
```

We pass JAX-computed gradients (`objective_grad_np`, `dynamics_jacobian_np`) to SciPy's SLSQP. This gives us the best of both worlds: JAX's fast autodiff + SciPy's mature constrained optimizer.

Summary of Your Implementation

When you complete these four sections, you'll have:

1. **Designed a cost function** that balances power and switching
2. **Implemented dynamics defects** – the core of multiple shooting
3. **Set up path constraints** on states using box bounds
4. **Assembled a constrained NLP** and solved it with SLSQP + JAX gradients

The provided driver `optimize_full_trajectory()` handles the receding horizon loop, warm-starting, and comparison with PID. Run:

```
python3 trajectory_optimization.py          # Quick 30-min test
python3 trajectory_optimization.py --full   # Full 4-hour evaluation
```

Deliverables and how we'll grade

Submit your completed `trajectory_optimization.py`, plus results from a full 4-hour run (`python3 trajectory_optimization.py --full`). Include a short PDF report with: a plot comparing trajectories to PID, a table of $\gamma_{\text{con}}, \gamma_{\text{pow}}, \gamma_{\text{switch}}$, and a brief discussion of trade-offs (energy vs switching vs constraint tightness), how multiple shooting helps with path constraints, and runtime per window. Rubric: correctness 40% (feasible solves, constraints met), performance 30% (energy improvement), implementation quality 20% (clean code, efficient JAX), analysis 10%.

Getting started

```
pip install numpy scipy matplotlib jax jaxlib
# Quick 30-minute check
python3 trajectory_optimization.py
# Full 4-hour evaluation
python3 trajectory_optimization.py --full
```

Goal: beat the PID baseline on energy without breaking safety too much.

Reference Larsen, L. F. S., Izadi-Zamanabadi, R., & Wisniewski, R. (2007). Supermarket Refrigeration System – Benchmark for Hybrid System Control. *European Control Conference (ECC).*