IFT6162 Homework 1: Grading Scheme and Time Allocation

Overview

This homework consists of three programming assignments covering core topics in dynamic programming and optimal control. Each problem requires implementing key algorithms and demonstrating understanding through working code. **You are not required to write a formal multi-page report** — your implementations and brief comments/plots are sufficient to demonstrate understanding.

Weight Distribution

Problem	Weight	Estimated Time
Supermarket Refrigeration (Trajectory Optimization)	45%	12-15 hours
Bus Engine Replacement (Smooth Bellman & NFXP)	35%	8-10 hours
Projection Methods (Collocation & Galerkin)	20%	5-7 hours
Total	**100%**	**25-32 hours**

Problem 1: Supermarket Refrigeration (45 points)

File: `supermarket_refrigeration/trajectory_optimization.py`

What You're Implementing

Multiple shooting Model Predictive Control (MPC) for hybrid refrigeration system control with JAX autodiff and SLSQP optimization.

Four Core Sections (All Required)

1. **Objective Function (12 points)**
    - Design cost function balancing power consumption and switching penalties
    - Implement power calculation using compressor equations

- Apply proper weighting for kW-scale power and switching costs

2. **Dynamics Defects (12 points)**
   - Core of multiple shooting: enforce dynamics as equality constraints
   - Propagate forward Euler steps and compute defects for all time steps
   - Return flat constraint vector for SLSQP
3. **State Bounds (10 points)**
   - Set up box constraints on temperatures, pressure, and refrigerant mass
   - Enforce path constraints at all time steps (not just endpoints)
   - Handle state vector indexing correctly across time horizon
4. **SLSQP Optimization Setup (11 points)**
   - Assemble `NonlinearConstraint` for dynamics defects with Jacobians
   - Call `scipy.optimize.minimize` with JAX-computed gradients
   - Configure solver parameters appropriately

Grading Breakdown

- **Correctness (45%)**: Implementations produce feasible trajectories that respect constraints
- **Performance (30%)**: Energy consumption improvement over PID baseline (should beat 6.80 kW)
- **Implementation Quality (20%)**: Clean code, efficient JAX operations, proper use of autodiff
- **Analysis (5%)**: Brief discussion of results

What to Submit

- Completed `trajectory_optimization.py`
- Run output showing final metrics from `python3 trajectory_optimization.py --full`
- A **brief summary** (1-2 paragraphs or bullet points) discussing:
  - Energy improvement vs PID baseline
  - Constraint satisfaction (γ_{con} value)
  - Trade-offs observed (energy vs switching vs runtime)
  - How multiple shooting enabled path constraints

Note: No formal report needed. Present results with a plot (the code generates this) and a few sentences of commentary in a text file or simple document.

Time Recommendation

This is the most demanding problem. Start here and allocate 12-15 hours: - Understanding the problem and equations: 2-3 hours - Implementing the four sections: 6-8 hours - Debugging and tuning: 3-4 hours - Running full evaluation and writing brief summary: 1 hour

Problem 2: Bus Engine Replacement (35 points)

File: `bus_engine_replacement/bus_replacement.py`

What You're Implementing

Maximum likelihood estimation of a dynamic discrete choice model using the NFXP (Nested Fixed-Point) algorithm with smooth Bellman equations and implicit differentiation.

Three Core Functions (All Required)

1. **Smooth Bellman Operator (15 points)**
   - Implement $L(v)(s) = \log \sum_a \exp(q(s, a))$ using `logsumexp`
   - Compute Q-values for keep vs replace actions
   - Handle transition probabilities correctly
2. **Softmax Policy (10 points)**
   - Compute stochastic policy $\pi(a|s) = \frac{\exp(q(s,a))}{\sum_{a'} \exp(q(s,a'))}$
   - Work in log-space for numerical stability
   - Return probability matrix over states and actions
3. **Log-Likelihood (10 points)**
   - Solve Bellman fixed point for given parameters
   - Compute policy and index to observed state-action pairs
   - Return sum of log-probabilities

Grading Breakdown

- **Smooth Bellman Operator (15%)**: Correct implementation with proper Q-value computation
- **Softmax Policy (10%)**: Numerically stable softmax over actions
- **Log-Likelihood (10%)**: Proper integration of fixed-point solve and MLE objective

What to Submit

- Completed `bus_replacement.py`
- Generated outputs: `estimation_results.png` and `policy_evolution.mp4`
- **No separate report needed** — the plots show your results

Expected Results

- Replacement cost $RC \approx 9 - 11$ thousand dollars
- Maintenance cost $C_1 \approx 2 - 4$
- Sigmoid-shaped replacement policy increasing with mileage
- Smooth loss curve showing optimization convergence
- Runtime: 2-3 minutes

Time Recommendation

Allocate 8-10 hours: - Understanding Rust's model and smooth Bellman equations: 2-3 hours - Implementing the three functions: 3-4 hours - Debugging and verifying convergence: 2-3 hours - Generating plots and reviewing results: 1 hour

Problem 3: Projection Methods (20 points)

Files: - `projection_methods_assignment/projection_framework.py` - `projection_methods_assignment/timber_projection.py` - `projection_methods_assignment/contractio`

What You're Implementing

Two approaches to solving continuous-state dynamic programming problems: collocation and Galerkin projection methods, plus investigation of when parametric value iteration converges.

Six Strategic Implementations

Part 1: Projection Framework (6 points)

1. **Collocation Test (3 points)**: Evaluate residual at collocation nodes
2. **Galerkin Test (3 points)**: Integrate residual against basis functions using Gauss-Legendre quadrature

Part 2: Timber Application (8 points)

3. **Bellman Residual (4 points)**: Compute $R(s; a) = L\hat{v}(s) - \hat{v}(s)$
4. **Fitted Value Iteration (4 points)**: Implement parametric value iteration in coefficient space

Part 3: Contraction Analysis (6 points)

5. **Discretized Bellman Step (3 points)**: Apply one step of $\hat{T}$ mapping
6. **Lipschitz Estimation (3 points)**: Empirically test contraction property

Grading Breakdown

- **Part 1 (6%)**: Correct projection operator implementations
- **Part 2 (8%)**: Working timber harvesting solver with both Newton and iteration methods
- **Part 3 (6%)**: Correct contraction analysis showing linear splines preserve contraction

What to Submit

- Completed implementations in all three files

- Run the timber problem to verify it produces sensible harvesting policies
- **No formal report needed** — working code is sufficient

Expected Results

- Timber harvesting policy showing threshold behavior (harvest at high biomass)
- Collocation and Galerkin should give similar results
- Contraction investigation should show $L \leq \gamma$ for linear splines, potentially $L > \gamma$ for Chebyshev

Time Recommendation

Allocate 5-7 hours: - Understanding projection methods framework: 1-2 hours - Implementing Part 1 (framework): 1-2 hours - Implementing Part 2 (timber): 2-3 hours - Implementing Part 3 (contraction): 1 hour

General Submission Guidelines

What We Want

1. **Working implementations** of all required functions
2. **Successful execution** of the provided test scripts
3. **Generated outputs** (plots, animations, metrics) as specified
4. **Brief commentary** where needed (especially for Problem 1)

What We Don't Want

- Multi-page formal reports with extensive literature reviews
- Lengthy derivations (the math is already in the problem statements)
- Code comments that just restate what the code does

What Makes a Good Submission

- Clean, readable implementations following the template structure
- Efficient use of JAX/NumPy vectorization
- Results that meet or exceed baseline performance (for Problem 1)
- Brief, clear explanations of key findings (1-2 paragraphs is enough)
- Plots showing your results

Getting Help

Debugging Checklist

Problem 1 (Supermarket): - Are your dynamics defects being driven to near-zero by the optimizer? - Are temperature constraints $[2, 5]$°C being satisfied? - Is pressure staying below 1.7 bar (day) / 1.9 bar (night)? - Are you using `jnp` (JAX) instead of `np` in JIT-compiled functions?

Problem 2 (Bus): - Does your Bellman residual go to $< 10^{-6}$? - Is the replacement policy monotone increasing with mileage? - Are policy probabilities in $[0, 1]$ summing to 1 per state?

Problem 3 (Projection): - For Galerkin: Are you transforming quadrature nodes to $[0, K]$ domain? - Does parametric value iteration converge for linear splines? - Is the Lipschitz constant $\leq \gamma$ for linear splines?

Common Pitfalls

1. **Mixing NumPy and JAX**: Use `jnp` inside JAX functions
2. **Wrong indexing**: State vectors have specific orderings: follow the documentation
3. **Not warming start**: Problem 1 needs good initial guesses for SLSQP
4. **Ignoring constraints**: Check constraint violations in your final trajectories
5. **Wrong norms**: Use sup-norm for Lipschitz estimation (Problem 3)

Total Time: 25-32 hours — budget accordingly and start early!

Good luck! Focus on clean implementations and understanding the concepts. The problems are challenging but manageable if you work systematically.