Bus Engine Replacement: Smooth Bellman Equation

Reference: Rust, J. (1987). "Optimal Replacement of GMC Bus Engines: An Empirical Model of Harold Zurcher." *Econometrica*, 55(5), 999-1033.

Problem Setup

Harold Zurcher manages the Madison Metropolitan Bus Company fleet. Each month, for each bus, he must decide:

- **Keep** (a=0): Continue operating with current engine, pay maintenance cost
- **Replace** (a=1): Install new engine, pay replacement cost

The decision depends on the **mileage since last replacement** (the state).

Historical data: 6,469 monthly observations from 217 buses (December 1974 - May 1985) with 243 documented engine replacements.

The Smooth Bellman Equation

Unlike the standard Bellman equation with a hard max operator:

$$v^\star(s) = \max_{a\in\mathcal{A}} \left\{ r(s,a) + \gamma \sum_{j\in\mathcal{S}} p(j|s,a) v^\star(j) \right\}$$

Rust's model uses the **smooth (soft) Bellman equation**, which arises from Gumbel-distributed utility shocks:

$$v^\star(s) = \log \sum_{a\in\mathcal{A}} \exp \left(r(s,a) + \gamma \sum_{j\in\mathcal{S}} p(j|s,a) v^\star(j) \right)$$

This is the $\beta = 1$ case of the general entropy-regularized formulation:

$$v^\star(s) = \frac{1}{\beta} \log \sum_{a\in\mathcal{A}} \exp \left(\beta \left(r(s,a) + \gamma \sum_{j\in\mathcal{S}} p(j|s,a) v^\star(j) \right) \right)$$

where β is the inverse temperature and $\alpha = 1/\beta$ is the entropy regularization weight.

Softmax Policy

The optimal policy is stochastic, given by the softmax over Q-values:

$$\pi^\star(a|s) = \frac{\exp(q^\star(s,a))}{\sum_{a' \in \mathcal{A}} \exp(q^\star(s,a'))}$$

where the Q-function is:

$$q^\star(s,a) = r(s,a) + \gamma \sum_{j \in \mathcal{S}} p(j|s,a) v^\star(j)$$

This is a soft-argmax, the smooth counterpart to the deterministic greedy policy.

Model Specification

State space: $s \in \{0, 1, \dots, 89\}$ representing mileage bins - State 0: 0-5k miles since last replacement - State 1: 5-10k miles
- State 89: 445-450k miles

Actions: - $a = 0$: Keep current engine - $a = 1$: Replace engine

Rewards (negative costs): - Keep: $r(s, \text{keep}) = -C_1 \cdot s \cdot 0.001$ - Replace: $r(s, \text{replace}) = -(RC + C_1 \cdot 0 \cdot 0.001) = -RC$

where RC is the replacement cost parameter and C_1 is the maintenance cost parameter.

Transitions: - Keep: $p(s'|s, \text{keep})$ follows stationary monthly mileage distribution (estimated from data) - Replace: $p(0|s, \text{replace}) = 1$ (deterministically reset to state 0)

Parameters: - $\gamma = 0.95$: discount factor (modified from Rust's 0.9999 for computational efficiency) - Cost parameters: $\theta = [RC, C_1]$ to be estimated

Estimation via Maximum Likelihood

Given observed data $\{(s_i, a_i)\}_{i=1}^N$ of states and decisions, we estimate θ by maximizing the log-likelihood:

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} \log \pi(a_i | s_i; \theta)$$

Nested structure (NFXP algorithm): - Outer loop: Optimize θ to maximize $\mathcal{L}(\theta)$ - **Inner loop:** For each θ, solve the Bellman fixed point: $v^\star = L(v^\star; \theta)$

This brings a new challenge: that of computing $\frac{\partial \mathcal{L}}{\partial \theta}$ through the Bellman solve. Rust showed that this can be tackled using the **implicit differentiation** via the implicit function theorem. If $v^\star(\theta)$ satisfies $v^\star = L(v^\star; \theta)$, then:

$$\frac{\partial v^\star}{\partial \theta} = \left[I - \frac{\partial L}{\partial v}\right]^{-1} \frac{\partial L}{\partial \theta}$$

This is computed automatically using `jaxopt.FixedPointIteration` with `implicit_diff=True`.

Implementation

The code in `bus_replacement.py` implements:

1. **Data loading:** Parses Rust's original bus data files
2. **Transition estimation:** Estimates $p(s'|s, \text{keep})$ from observed mileage increments

3. **Smooth Bellman operator:** $L : v \mapsto \log \sum_a \exp(q(s, a))$
4. **Fixed-point solver:** Uses jaxopt with implicit differentiation
5. **MLE:** Optimizes θ using Adam with gradients from implicit differentiation
6. **Softmax policy:** Computes $\pi(a|s)$ from converged $v^\star$

What to Expect

Replacement policy should be: - **Sigmoid-shaped:** Low probability at low mileage, increasing smoothly to higher probability at high mileage - **Monotone increasing:** $\pi(\text{replace}|s)$ increases with state s - **Small probabilities:** Typically 0.01% - 1% range (replacement is rare)

Parameter estimates: - RC (replacement cost): Around $9 - 11$ thousand dollars - C_1 (maintenance cost): Around $2 - 4$ (scaled by 0.001)

Validation: - Bellman residual $\|v - L(v)\| < 10^{-6}$ (converged) - implicit_diff and backprop solutions match - Smooth optimization curve (steady improvement)

Implementation Tasks

You will implement **three core functions** marked with `# TODO` in `bus_replacement.py`:

Task 1: Smooth Bellman Operator (15 points)

Function: `smooth_bellman_operator(v, theta)`

Implement the smooth Bellman operator L:

$$(Lv)(s) = \log \sum_{a \in \mathcal{A}} \exp(q(s, a))$$

What to do:

1. **Compute Q-values** for both actions:

```
q(s, keep) = r(s, keep) +   * E[v(s') | s, keep]
           = -cost_keep(s) +   * (T @ v)[s]

q(s, replace) = r(s, replace) +   * E[v(s') | s, replace]
              = -cost_replace +   * v[0]
```

2. **Apply log-sum-exp:**

```
v_new = logsumexp(q_both, axis=1)
```

Task 2: Softmax Policy (10 points)

Function: `compute_policy(v, theta)`

Compute the stochastic policy:

$$\pi(a|s) = \frac{\exp(q(s, a))}{\sum_{a'} \exp(q(s, a'))}$$

What to do:

1. Compute Q-values (same as Task 1)
2. Apply softmax in log-space:

```
log_pi = q_both - logsumexp(q_both, axis=1, keepdims=True)
pi = exp(log_pi)
```

Task 3: Log-Likelihood (10 points)

Function: `log_likelihood(theta)`

Implement the maximum likelihood objective:

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} \log \pi(a_i | s_i; \theta)$$

What to do:

1. Solve Bellman: `v = solve_smooth_bellman(theta)`
2. Compute policy: `pi = compute_policy(v, theta)`

3. Index to observed actions: `probs = pi[states_i, actions_i]`
4. Sum logs: `return sum(log(probs))`

What's Provided

You don't need to implement: - Data loading and parsing (complete in `data_processing.py`) - Transition probability estimation (computed automatically) - Fixed-point solver with implicit differentiation (jaxopt handles this) - Optimization loop (Adam with gradient clipping) - Plotting and animation generation

You focus on: The three mathematical core concepts above.

Expected Output

Running `python bus_replacement.py` generates:

1. **`estimation_results.png`** - Two-panel figure:
   - Left: Sigmoid replacement policy with both differentiation methods overlaid
   - Right: Loss evolution showing optimization progress
2. **`policy_evolution.mp4`** - Animation showing:
   - How the sigmoid policy evolves during optimization
   - Parameter values at each step
   - Loss trajectory

Runtime: ~2-3 minutes on typical laptop

References

- **Original paper:** https://editorialexpress.com/jrust/crest_lectures/zurcher.pdf
- **Data and code:** https://editorialexpress.com/jrust/nfxp.html
- **Data repository:** https://github.com/OpenSourceEconomics/zurcher-data
- **Python implementation:** https://github.com/OpenSourceEconomics/ruspy