Rust (1987): Optimal Engine Replacement

David Evans

2026-11-13

Introduction

From McCall to Rust

Recap: The McCall Search Model

- An unemployed worker receives random wage offers each period
- Two choices: **accept** (become employed) or **reject** (keep searching)
- Solved via **dynamic programming** and the Bellman equation

- The key insight: there exists a **reservation wage** — accept anything above it, reject anything below

From Optimal Search to Optimal Replacement

- McCall (1970): when should a worker **stop searching** and accept a job?
- Rust (1987): when should a bus superintendent **stop maintaining** and replace an engine?

	McCall	Rust
Agent	Unemployed worker	Bus superintendent
State	Wage offer	Accumulated mileage
Choice	Accept / Reject	Replace / Keep
Tradeoff	Good wage now vs. better offer later	Replacement cost vs. rising maintenance

- Both are **dynamic discrete choice** problems solved by dynamic programming

Harold Zurcher

- Bus maintenance superintendent at the **Madison Metropolitan Bus Company**
- Manages a fleet of GMC buses
- Each period, decides: **replace** the engine or **keep** running it?

The Key Tradeoff

Replacing is expensive but resets the engine to new condition. Keeping is cheap today but maintenance costs **rise with mileage**.

Why This Paper Matters

- **Rust (1987, *Econometrica*)** — foundational paper in **structural econometrics**
- Shows how to **estimate** a dynamic programming model from observed data
- Introduced the **Nested Fixed Point (NFXP)** algorithm

- Framework used across economics:
  - Firm entry/exit decisions
  - Investment under uncertainty
  - Health decisions, migration, education

Roadmap

What We'll Cover

1. **The Environment** — states, actions, payoffs, and mileage transitions
2. **Choices with Extreme Value Shocks** — how unobserved shocks lead to logit probabilities
3. **Solving the DP** — Bellman equation and value function iteration
4. **Estimation** — recovering structural parameters from data using NFXP

The Environment

The Agent's Problem

The Setup

- Zurcher manages a fleet of buses, each characterized by **accumulated mileage** x_t
- Mileage is **discretized** into S bins (e.g., each bin = 5,000 miles)

- Each period, Zurcher observes x_t and chooses:
  - $d_t = 0$: **Keep** the current engine
  - $d_t = 1$: **Replace** the engine (mileage resets to 0)

Per-Period Payoffs

- **Keep** ($d_t = 0$): pay maintenance cost $c(x_t, \theta_1)$

$$u(x_t, 0) = -c(x_t, \theta_1)$$

- **Replace** ($d_t = 1$): pay replacement cost RC, engine resets to new

$$u(x_t, 1) = -RC$$

- These are the payoffs that depend on variables that the **econometrician can observe**, e.g. mileage

Unobserved Components

- But the agent also considers factors we **can't see**: weather conditions, driver complaints, parts availability, gut feeling…

- We model these as **random shocks** $\varepsilon_t(d)$ added to each option:

$$\tilde{u}(x_t, d) = u(x_t, d) + \varepsilon_t(d)$$

- Key assumptions:
  - $\varepsilon_t(0)$ and $\varepsilon_t(1)$ are **iid** across options and time
  - Independent of the observed state x_t (**Conditional Independence**)

- What distribution should we assume for ε? This is a **modeling choice**
  - and it turns out to matter a lot
  - we'll tackle this in the next section

The Cost Function

- We use a **linear specification**: $c(x, \theta_1) = \theta_1 \cdot x$

- Higher mileage $\rightarrow$ higher maintenance costs
- At $x = 0$ (new engine): maintenance cost is zero

- Two structural parameters to estimate:
  - θ_1: how fast maintenance costs rise with mileage
  - RC: the one-time cost of engine replacement

Mileage Transitions

- Each period, mileage increases by a **random increment** $\Delta x \in \{0, 1, 2\}$ bins

$$\Pr(\Delta x = k) = \theta_{3,k+1} \quad \text{for } k = 0, 1, 2$$

- If **keep** ($d_t = 0$): $x_{t+1} = x_t + \Delta x$
- If **replace** ($d_t = 1$): $x_{t+1} = 0 + \Delta x$ (mileage resets, then accumulates)

- At the top state ($x = S - 1$), mileage is capped
- Only **three** possible next states — the probabilities θ_3 are all we need to take expectations

Setup in Julia

Loading Packages

```
using Plots
using LinearAlgebra
using Statistics
using Distributions
using Random
```

The `RustModel` Struct

- We bundle all model parameters into a struct (as we did with `McCallModel`):

```
@kwdef struct RustModel
    β::Float64 = 0.95                                # discount factor
    S::Int = 50                                       # number of mileage bins
    RC::Float64 = 20.0                                # replacement cost
    θ₁::Float64 = 0.04                               # maintenance cost parameter
    θ₃::Vector{Float64} = [0.36, 0.48, 0.16]         # transition probabilities
    x̄::Vector{Float64} = Float64.(0:S-1)              # mileage states
    c::Vector{Float64} = θ₁ * x̄                      # cost at each mileage level
end
```

```
RustModel
```

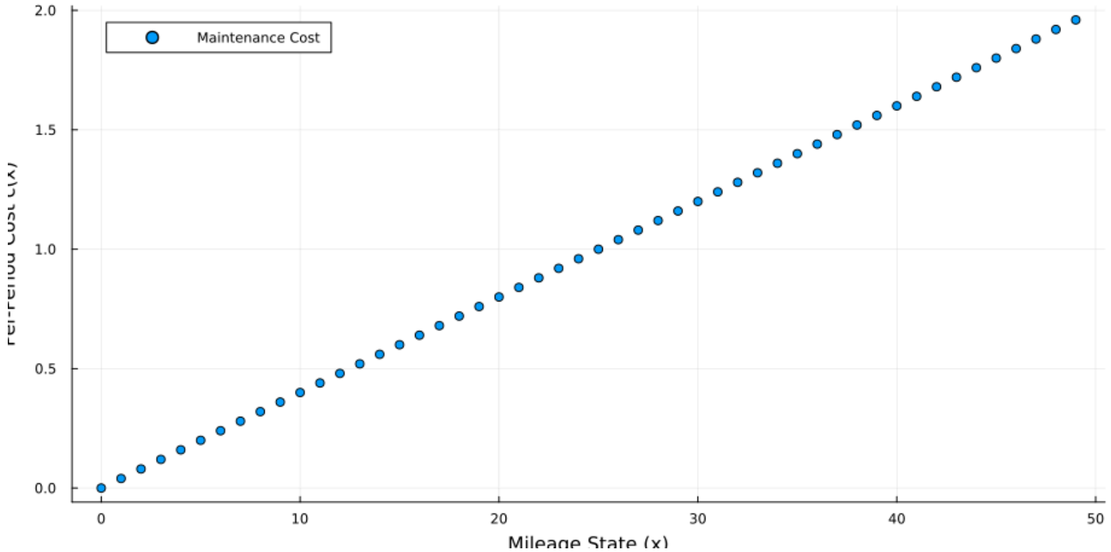
Creating an Instance

```
1  m = RustModel()                                  # all defaults
2  (; x̄, c, S, β, RC, θ₃) = m                      # destructure for easy access
```

```
RustModel(0.95, 50, 20.0, 0.04, [0.36, 0.48, 0.16], [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0  …  40.0, 41.0, 42.0, 43.0, 44.0, 45.0,
46.0, 47.0, 48.0, 49.0], [0.0, 0.04, 0.08, 0.12, 0.16, 0.2, 0.24, 0.28, 0.32, 0.36  …  1.6, 1.6400000000000001, 1.68, 1.72, 1.76, 1.8, 1.84,
1.8800000000000001, 1.92, 1.96])
```

Plotting the Cost Function

```
1 scatter(x̄, c, label="Maintenance Cost", legend=:topleft)
2 xlabel!("Mileage State (x)")
3 ylabel!("Per-Period Cost c(x)")
```



- Maintenance costs rise linearly — the engine becomes more expensive to maintain as mileage accumulates

Choices with Extreme Value Shocks

The Static Choice Problem

Setting Up the Problem

- Before solving the full DP, let's understand a **simpler** problem
- An agent chooses between two options with payoffs:

$$\text{Option 0: } v_0 + \varepsilon_0 \qquad \text{Option 1: } v_1 + \varepsilon_1$$

- v_0, v_1: the **observable** (deterministic) payoff components
- $\varepsilon_0, \varepsilon_1$: **unobservable** (random) shocks known to the agent but not to us
- The agent picks the option with the **highest total payoff**:

$$d^* = \arg\max_d \{v_d + \varepsilon_d\}$$

Why Model Unobserved Shocks?

- Without shocks: the model predicts **deterministic** choices
  - everyone at the same mileage makes the **same** decision
- But in the data, we see buses at **similar mileage** making **different** choices
- With shocks: the model generates **probabilistic** choices
  - consistent with the data

A Modeling Choice

We need to choose a distribution for ε. This is a **modeling assumption** that affects both the math and the economics. Rust chose the Type I Extreme Value distribution.

The Expected Maximum

- The **value** of facing this choice (before seeing the shocks) is:

$$\mathbb{E}\left[\max_{d}\left\{v_d + \varepsilon_d\right\}\right]$$

- This is the expected payoff from the **best option** — it matters for the Bellman equation because today's value depends on what you expect to do **optimally tomorrow**

- Can we compute this in closed form? It depends on the distribution of ε

The Type I Extreme Value Distribution

The Distribution

- The **Type I Extreme Value** (Gumbel) distribution has CDF:

$$F(\varepsilon) = e^{-e^{-\varepsilon}}$$

- Mean: $\gamma \approx 0.5772$ (Euler–Mascheroni constant)
- Variance: $\pi^2/6 \approx 1.645$
- Right-skewed: extreme positive shocks are possible but rare

Generating Extreme Value Draws

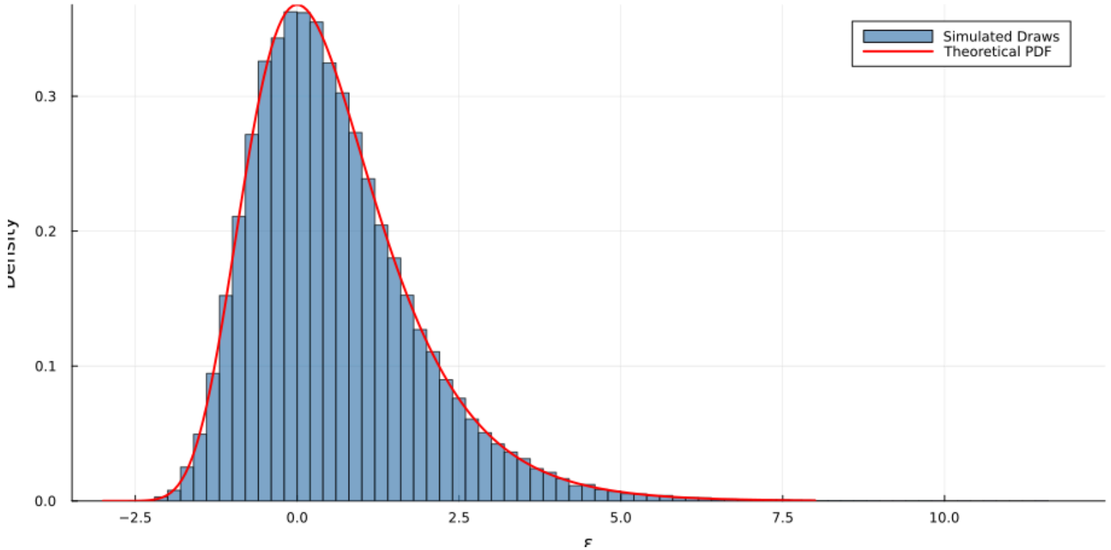
- If $U \sim \text{Uniform}(0, 1)$, then $\varepsilon = -\log(-\log(U))$ has the Type I EV distribution

```
N = 100_000
ε_draws = -log.(-log.(rand(N)))
println("Sample mean: ", round(mean(ε_draws), digits=3), "  (theory: 0.577)")
println("Sample var:  ", round(var(ε_draws), digits=3), "  (theory: 1.645)")
```

```
Sample mean: 0.574  (theory: 0.577)
Sample var:  1.65  (theory: 1.645)
```

Plotting the Distribution

```
histogram(ε_draws, bins=100, normalize=true, label="Simulated Draws",
          alpha=0.7, color=:steelblue)
ε_grid = LinRange(-3, 8, 200)
pdf_ev = exp.(-ε_grid .- exp.(-ε_grid))            # f(ε) = exp(-ε - exp(-ε))
plot!(ε_grid, pdf_ev, linewidth=2, color=:red, label="Theoretical PDF")
xlabel!("ε")
ylabel!("Density")
```



- Right-skewed: the agent occasionally gets a **very favorable** unobserved shock

Two Key Results

Result 1: The Expected Maximum

Theorem (Log-Sum-Exp)

If $\varepsilon_0, \varepsilon_1$ are iid Type I Extreme Value, then:

$$\mathbb{E}\left[\max\{v_0 + \varepsilon_0,\ v_1 + \varepsilon_1\}\right] = \log\left(e^{v_0} + e^{v_1}\right) + \gamma$$

where $\gamma \approx 0.5772$ is Euler's constant.

- This extends to J options: $\mathbb{E}\left[\max_j\{v_j + \varepsilon_j\}\right] = \log\left(\sum_j e^{v_j}\right) + \gamma$

Monte Carlo: Verifying the Expected Maximum

- Let's check with simulation: set $v_0 = 2$ and $v_1 = 3$

```
v0_val = 2.0
v1_val = 3.0

ε0 = -log.(-log.(rand(N)))                    # N extreme value draws
ε1 = -log.(-log.(rand(N)))                    # N independent draws

max_payoffs = max.(v0_val .+ ε0, v1_val .+ ε1)      # simulated max for each draw
const γ_euler = Base.MathConstants.eulergamma       # Euler's constant ≈ 0.5772

theoretical = log(exp(v0_val) + exp(v1_val)) + γ_euler
println("Simulated E[max]:   ", round(mean(max_payoffs), digits=4))
println("Theoretical:        ", round(theoretical, digits=4))
```

```
Simulated E[max]:   3.8934
Theoretical:        3.8905
```

Function: Verifying the Expected Maximum

```
function verifyExpectedMaximum(v0_val, v1_val, N)
    #N extreme value draws for ε0 and ε1
    #simulate max payoff for each draw
    #compute theoretical expected maximum
    #return simulated and theoretical expected maximum
end
```

```
verifyExpectedMaximum (generic function with 1 method)
```

Function: Verifying the Expected Maximum

```
function verifyExpectedMaximum(v0_val, v1_val, N)
    #N extreme value draws for ε0 and ε1
    ε0 = -log.(-log.(rand(N)))
    ε1 = -log.(-log.(rand(N)))
    #simulate max payoff for each draw
    max_payoffs = max.(v0_val .+ ε0, v1_val .+ ε1)
    #compute theoretical expected maximum
    theoretical = log(exp(v0_val) + exp(v1_val)) + γ_euler
    #return simulated and theoretical expected maximum
    return mean(max_payoffs), theoretical
end
```

```
verifyExpectedMaximum (generic function with 1 method)
```

Monte Carlo: Varying the Gap

- How does $\mathbb{E}[\max]$ change as $v_1 - v_0$ varies?

```
#Preallocate arrays

#Loop through each value of Δv

#Plot the results
```

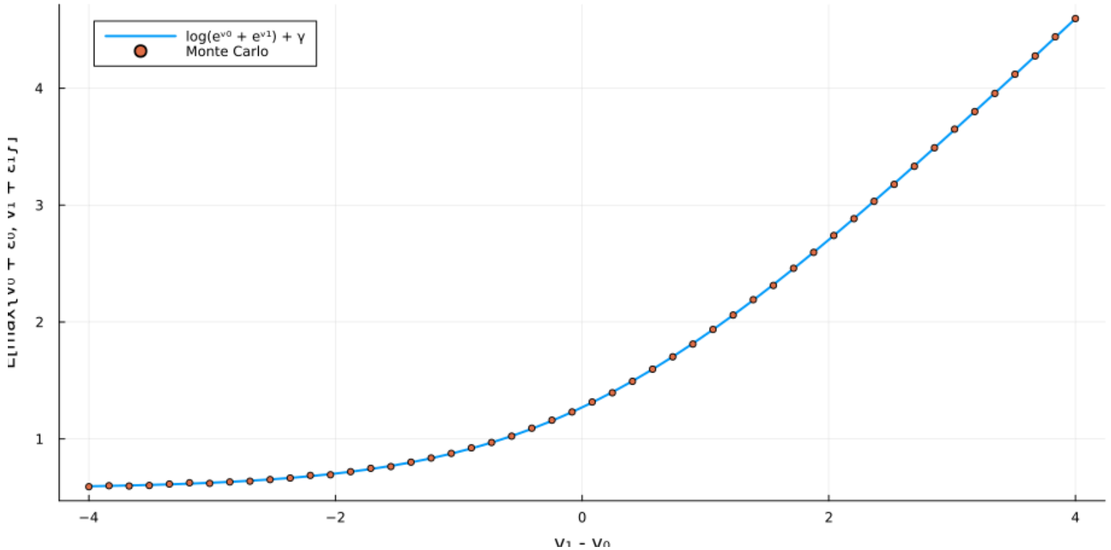
Monte Carlo: Varying the Gap

- How does $\mathbb{E}[\max]$ change as $v_1 - v_0$ varies?

```julia
#Preallocate arrays
Δv_grid = LinRange(-4, 4, 50)
Emax_sim = zeros(length(Δv_grid))
Emax_theory = zeros(length(Δv_grid))

#Loop through each value of Δv
for (i, Δv) in enumerate(Δv_grid)
    Emax_sim[i], Emax_theory[i] = verifyExpectedMaximum(0.0, Δv, N)
end

#Plot the results
plot(Δv_grid, Emax_theory, linewidth=2, label="log(e^v0 + e^v1) + γ", legend=:topleft)
scatter!(Δv_grid, Emax_sim, label="Monte Carlo", markersize=3)
xlabel!("v₁ - v₀")
ylabel!("E[max{v₀ + ε₀, v₁ + ε₁}]")
```



Result 2: Choice Probabilities

Theorem (Logit Formula)

If $\varepsilon_0, \varepsilon_1$ are iid Type I Extreme Value, then:

$$P(d=1) = \Pr(v_1 + \varepsilon_1 > v_0 + \varepsilon_0) = \frac{e^{v_1}}{e^{v_0} + e^{v_1}} = \frac{1}{1 + e^{v_0 - v_1}}$$

- This is the **logit formula**
  - the same one from discrete choice econometrics!
- The probability of choosing option 1 depends only on the **difference** $v_1 - v_0$

Monte Carlo: Verifying Choice Probabilities

```julia
choices = (v1_val .+ ε1) .> (v0_val .+ ε0)          # 1 if option 1 chosen
simulated_prob = mean(choices)

theoretical_prob = exp(v1_val) / (exp(v0_val) + exp(v1_val))
println("Simulated P(d=1):   ", round(simulated_prob, digits=4))
println("Theoretical P(d=1): ", round(theoretical_prob, digits=4))
```

```
Simulated P(d=1):   0.7315
Theoretical P(d=1): 0.7311
```

- With $v_0 = 2, v_1 = 3$: option 1 has higher deterministic value, so $P(d = 1) > 0.5$
- But it's not 1.0 — the shocks sometimes make option 0 the better choice

Function: Verifying Choice Probabilities

```
function verifyChoiceProbabilities(v0_val, v1_val, N)
    #N extreme value draws for ε0 and ε1
    ε0 = -log.(-log.(rand(N)))
    ε1 = -log.(-log.(rand(N)))
    #simulate choices for each draw
    choices = (v1_val .+ ε1) .> (v0_val .+ ε0)
    #compute theoretical choice probability
    theoretical = exp(v1_val) / (exp(v0_val) + exp(v1_val))
    return mean(choices), theoretical
end
```

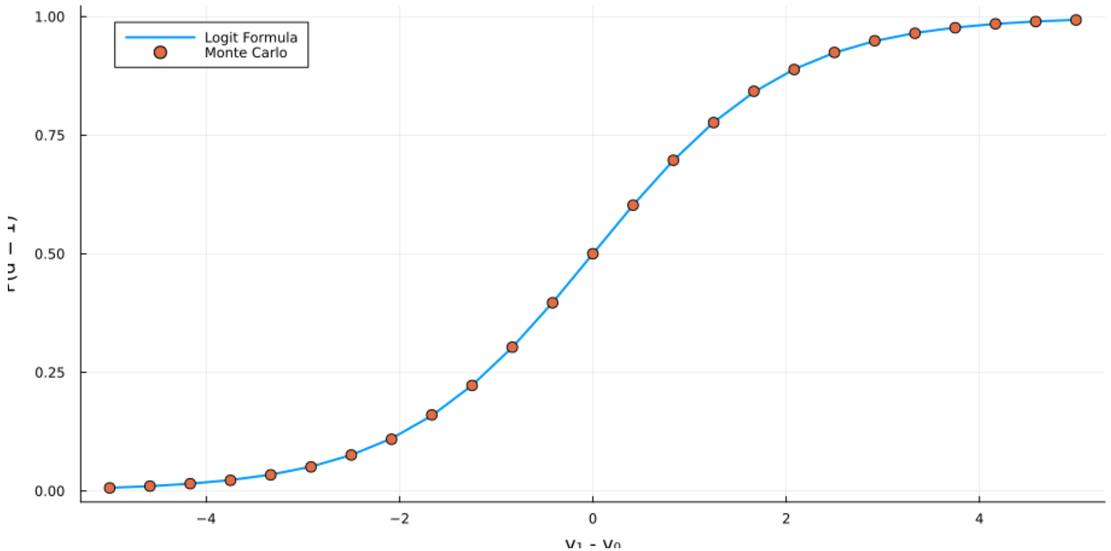
```
verifyChoiceProbabilities (generic function with 1 method)
```

Monte Carlo: The Logit Curve

```julia
# Preallocate arrays
Δv_points = LinRange(-5, 5, 25)
P_mc = zeros(length(Δv_points))
P_theory = zeros(length(Δv_points))

#Loop through each value of Δv
for (i, Δv) in enumerate(Δv_points)
    P_mc[i], P_theory[i] = verifyChoiceProbabilities(0.0, Δv, N)
end

#Plot the results
plot(Δv_points, P_theory, linewidth=2, label="Logit Formula", legend=:topleft)
scatter!(Δv_points, P_mc, label="Monte Carlo", markersize=5)
xlabel!("v₁ - v₀")
ylabel!("P(d = 1)")
```



- S-shaped curve: smooth transition from “never choose 1” to “always choose 1”

What's Next?

From Static Choices to Dynamic Programming

- We now have two powerful tools from the extreme value assumption:
  1. **Log-sum-exp**: $\mathbb{E}[\max\{v_0 + \varepsilon_0, v_1 + \varepsilon_1\}] = \log(e^{v_0} + e^{v_1}) + \gamma$
  2. **Logit**: $P(d = 1) = e^{v_1} / (e^{v_0} + e^{v_1})$

- These hold for **any** values v_0, v_1
- Next: we define v^{keep} and v^{replace} for the Rust model
  - then apply these results to build the Bellman equation

Solving the Dynamic Problem

The Bellman Equation

Zurcher's Optimization Problem

- Zurcher maximizes **expected discounted lifetime utility**:

$$\max_{d_t} \; \mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t \left(u(x_t, d_t, \theta_1) + \varepsilon_t(d_t)\right)\right]$$

- Same structure as McCall, but the state evolves stochastically regardless of the action (mileage always goes up)
- As always, the problem can be tackled with a let statement
  - Let $V(x)$ be the value of having a bus in mileage state x before the "taste" shocks ε are realized

Choice-Specific Value Functions

- Just like McCall, define the value of each option. The value of **keeping** at mileage x:

$$v^{\text{keep}}(x) = -c(x, \theta_1) + \beta \sum_{k=0}^{2} \theta_{3,k+1} \cdot V\big(\min(x + k,\ S - 1)\big)$$

- The value of **replacing** at mileage x:

$$v^{\text{replace}}(x) = -RC + \beta \sum_{k=0}^{2} \theta_{3,k+1} \cdot V(k)$$

- Flow payoff today + discounted continuation
  - the same structure as McCall's V_{accept} and V_{reject}
- These are the v_0 and v_1 from our extreme value analysis!

Defining $V(x)$

- With $v^{\text{keep}}(x)$ and $v^{\text{replace}}(x)$ defined, the agent at state x faces exactly the choice we studied:

$$\max\left\{v^{\text{keep}}(x) + \varepsilon_0,\quad v^{\text{replace}}(x) + \varepsilon_1\right\}$$

- We define $V(x)$ as the value of having a bus in state x **before** the shocks are realized:

$$\boxed{V(x) \equiv \mathbb{E}_{\varepsilon}\left[\max\left\{v^{\text{keep}}(x) + \varepsilon_0,\quad v^{\text{replace}}(x) + \varepsilon_1\right\}\right]}$$

- Compare to McCall: $V_s = \max\{V_{\text{accept},s},\ V_{\text{reject},s}\}$. Here we take an **expected** max because of the unobserved shocks.

The Bellman Equation

- Applying **Result 1** (log-sum-exp):

$$V(x) = \log\left(e^{v^{\text{keep}}(x)} + e^{v^{\text{replace}}(x)}\right) + \gamma$$

- The Euler constant γ is just a level shift
  - dropping it doesn't affect choices or convergence

$$\boxed{V(x) = \log\left(\exp(v^{\text{keep}}(x)) + \exp(v^{\text{replace}}(x))\right)}$$

Fixed Point Problem

V appears on **both sides**: on the left directly, and on the right inside v^{keep} and v^{replace}. Just like in McCall, we solve by iterating until convergence.

Choice Probabilities

- Applying **Result 2** (logit), the probability of replacing at mileage x:

$$P(d = 1 \mid x) = \frac{\exp(v^{\text{replace}}(x))}{\exp(v^{\text{keep}}(x)) + \exp(v^{\text{replace}}(x))} = \frac{1}{1 + \exp(v^{\text{keep}}(x) - v^{\text{replace}}(x))}$$

Static Logit vs. Dynamic Logit

This is the same **logit formula** we verified with Monte Carlo. The difference: v^{keep} and v^{replace} incorporate the entire **future path** of costs and decisions — the agent is **forward-looking**, not myopic.

The Expected Continuation Value

- Both continuation values are the **same expectation**, just started from different mileage:

$$EV(x) = \sum_{k=0}^{2} \theta_{3,k+1} \cdot V\big(\min(x+k,\ S-1)\big)$$

- The choice-specific values become:

$$v^{\text{keep}}(x) = -c(x, \theta_1) + \beta\, EV(x) \qquad\qquad v^{\text{replace}}(x) = -RC + \beta\, EV(0)$$

- Replacing puts you at a **new engine** ($x = 0$) no matter what x is
  - v^{replace} is a **single number** — it does not depend on current mileage
- From any state only **three** next states are possible, so we just weight them by θ_3 — no transition matrix needed

Value Function Iteration

Computing $EV(x)$

- In Julia, mileage x lives at index `i = x + 1`, and `θ₃[k]` is the probability of $\Delta x = k - 1$

```
function expectedValue(V, θ₃, S)
    #Pre-allocate EV
    for i in 1:S                                # loop over current states
        for k in eachindex(θ₃)                  # loop over possible increments
            #Find the next state (capped at S)
            #Add probability × value of next state
        end
    end
    #Return EV
end
```

Computing $EV(x)$

- In Julia, mileage x lives at index `i = x + 1`, and `θ₃[k]` is the probability of $\Delta x = k - 1$

```julia
function expectedValue(V, θ₃, S)
    EV = zeros(S)                          # EV[i] = E[V(x') | keep at state i]
    for i in 1:S                           # loop over current states
        for k in eachindex(θ₃)             # loop over possible increments
            j = min(i + k - 1, S)          # next state: i + (k-1), capped at S
            EV[i] += θ₃[k] * V[j]          # probability × value of next state
        end
    end
    return EV
end
```

```
expectedValue (generic function with 1 method)
```

Checking `expectedValue`

- Sanity check: if $V(x) = x$, then $EV(x) = x + \mathbb{E}[\Delta x] = x + 0.48 + 2(0.16) = x + 0.8$

```
 1 EV_test = expectedValue(x̄, θ₃, S)                # test with V(x) = x
 2 round.(EV_test[1:5], digits=3)                     # x = 0,...,4: next mileage is x + 0.8 on average
```

```
5-element Vector{Float64}:
 0.8
 1.8
 2.8
 3.8
 4.8
```

```
 1 round.(EV_test[S-2:S], digits=3)                   # near the top, the cap binds
```

```
3-element Vector{Float64}:
 47.8
 48.64
 49.0
```

- `EV[1]` is the expected continuation value from a **new engine** — exactly what we need for v^{replace}

The Bellman Operator: Pseudocode

```
function iterateBellman(m::RustModel, V)
    # Compute expected continuation values EV(x)
    # Compute choice-specific values: v_keep and v_replace
    # Combine using log-sum-exp: V(x) = log(exp(v_keep) + exp(v_replace))
    # Return updated V
end
```

The Bellman Operator: Choice-Specific Values

- The two components of the Bellman equation:

```
function iterateBellman(m::RustModel, V)
    (; β, RC, c, θ₃, S) = m
    EV = expectedValue(V, θ₃, S)                    # E[V(x')] from each state
    v_keep = -c + β * EV                            # flow cost + β × E[V | keep]
    v_replace = -RC + β * EV[1]                     # replacement cost + β × E[V | new engine]
    # ...
end
```

- `EV[1]`: a new engine starts at index 1 ($x = 0$)
- `v_keep` is a vector that falls with mileage (higher maintenance costs)
- `v_replace` is a single number (resetting makes current mileage irrelevant)

The Bellman Operator: Log-Sum-Exp

```julia
function iterateBellman(m::RustModel, V)
    (; β, RC, c, θ₃, S) = m
    EV = expectedValue(V, θ₃, S)                    # expected continuation values
    v_keep = -c + β * EV                            # value of keeping
    v_replace = -RC + β * EV[1]                     # value of replacing
    vmax = max.(v_keep, v_replace)                  # for numerical stability
    V_new = vmax .+ log.(exp.(v_keep .- vmax)       # log-sum-exp trick:
                       .+ exp.(v_replace .- vmax))  #   prevents overflow
    return V_new
end
```

```
iterateBellman (generic function with 1 method)
```

Log-Sum-Exp Trick

We compute $\log(e^a + e^b) = m + \log(e^{a-m} + e^{b-m})$ where $m = \max(a, b)$. This prevents numerical overflow when a or b are large.

Iterating to Convergence

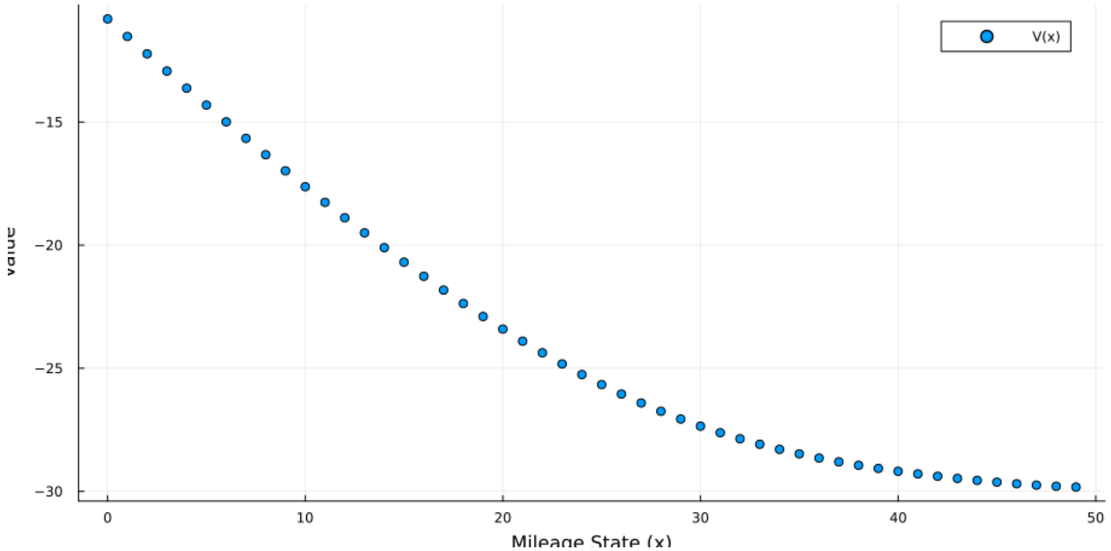
```
V = zeros(S)                                    # initial guess: all zeros
dist = 1.0                                      # initialize distance

niter = 0                                       # count iterations
while dist > 1e-10                              # loop until convergence
    V_new = iterateBellman(m, V)                # one Bellman step
    dist = norm(V - V_new, Inf)                 # max absolute change
    V = V_new                                   # update
    niter += 1
end
println("Converged in $niter iterations")
```

```
Converged in 455 iterations
```

The Value Function

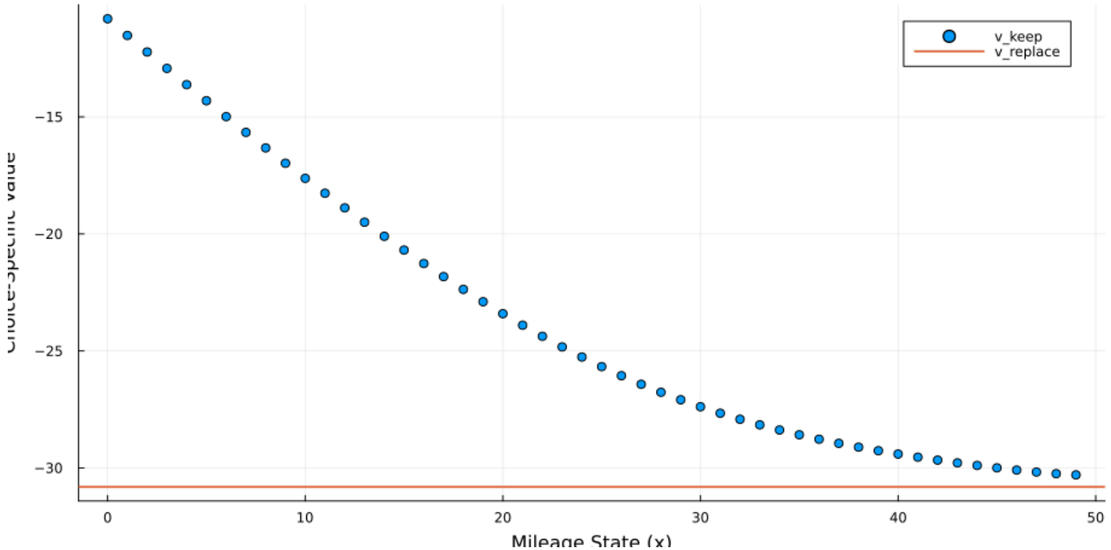
```
1 scatter(x̄, V, label="V(x)", legend=:topright)
2 xlabel!("Mileage State (x)")
3 ylabel!("Value")
```



- Value is **decreasing** in mileage — higher mileage means higher costs ahead

Choice-Specific Values

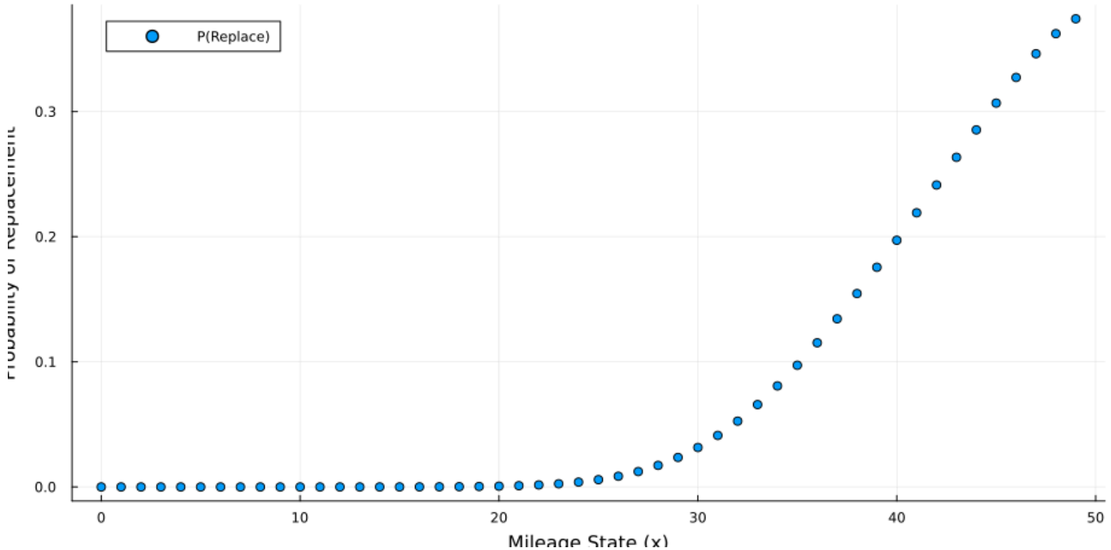
```
EV = expectedValue(V, θ₃, S)                          # expected continuation values
v_keep = -c + β * EV                                  # value of keeping
v_replace = -RC + β * EV[1]                           # value of replacing
scatter(x̄, v_keep, label="v_keep", legend=:topright)
hline!([v_replace], label="v_replace", linewidth=2)  # same value at every x
xlabel!("Mileage State (x)")
ylabel!("Choice-Specific Value")
```



- At low mileage: $v^{\text{keep}} > v^{\text{replace}}$ (keeping is better)
- At high mileage: $v^{\text{replace}} > v^{\text{keep}}$ (replacing is better)

Replacement Probability

```
P_rep = 1 ./ (1 .+ exp.(v_keep .- v_replace))       # logit formula
scatter(x̄, P_rep, label="P(Replace)", legend=:topleft)
xlabel!("Mileage State (x)")
ylabel!("Probability of Replacement")
```



- **S-shaped curve**: smooth transition from “never replace” to “always replace”
- Analogous to the **reservation wage** in McCall, but here it’s a **reservation mileage**

The `solveBellman` Function

- Wrap the VFI loop in a reusable function:

```
function solveBellman(m::RustModel; tol=1e-10)
    V = zeros(m.S)                              # initial guess
    dist = 1.0
    while dist > tol                            # iterate until convergence
        V_new = iterateBellman(m, V)
        dist = norm(V - V_new, Inf)
        V = V_new
    end
    return V
end
```

```
solveBellman (generic function with 1 method)
```

The `solveRust` Function

- Solve the Bellman equation and compute choice probabilities:

```
function solveRust(m::RustModel)
    #solve the Bellman equation
    #compute choice probabilities
    #return tuple with solution
end
```

```
solveRust (generic function with 1 method)
```

The `solveRust` Function

- Solve the Bellman equation and compute choice probabilities:

```
function solveRust(m::RustModel)
    (; β, RC, c, θ₃, S) = m
    #solve the Bellman equation
    V = solveBellman(m)                          # solve the DP
    #compute choice probabilities
    EV = expectedValue(V, θ₃, S)                 # expected continuation values
    v_keep = -c + β * EV                         # choice-specific values
    v_replace = -RC + β * EV[1]
    P_rep = 1 ./ (1 .+ exp.(v_keep .- v_replace))    # logit choice probabilities
    #return tuple with solution
    return (V=V, P=P_rep, v_keep=v_keep, v_replace=v_replace)
end
```

```
solveRust (generic function with 1 method)
```

Using `solveRust`

```
sol = solveRust(m)
sol.P[1:5]                # replacement probability at low mileage
```

```
5-element Vector{Float64}:
 2.0611536181902037e-9
 4.202144358547079e-9
 8.518414382484044e-9
 1.716378823590513e-8
 3.436078646097847e-8
```

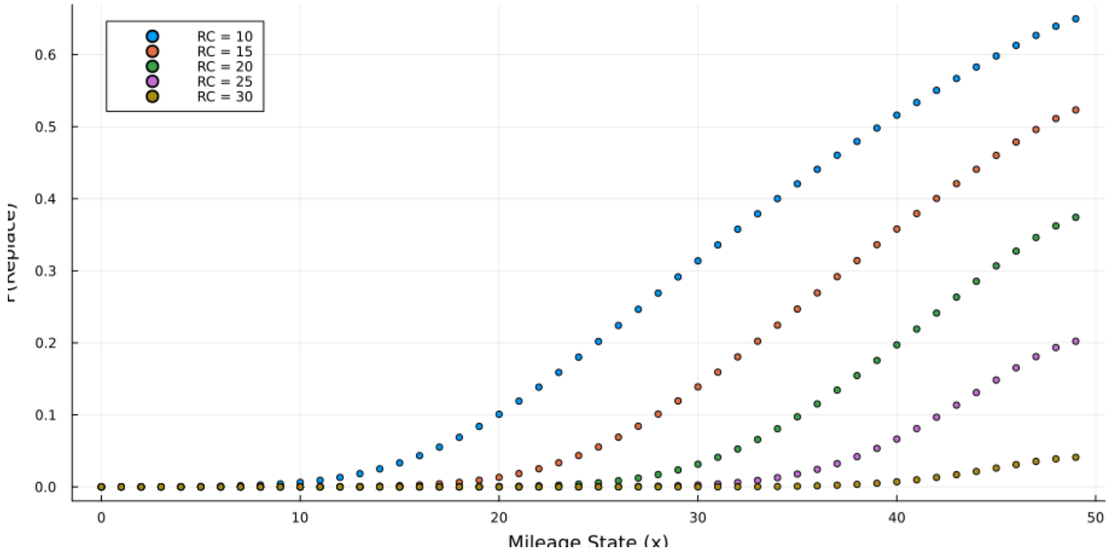
```
sol = solveRust(m)
sol.P[45:50]                # replacement probability at high mileage
```

```
6-element Vector{Float64}:
 0.28538319126651995
 0.30679678501323937
 0.327286517407775
 0.3462258786024322
 0.36228389282489654
 0.37415329641332634
```

Comparative Statics

Effect of Replacement Cost (RC)

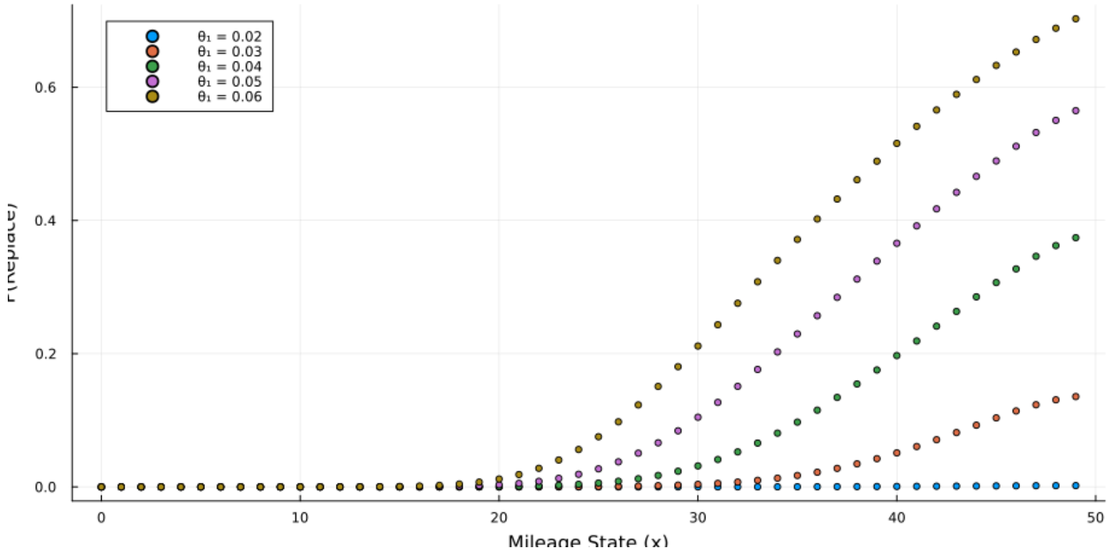
```
plot()
for RC_val in [10.0, 15.0, 20.0, 25.0, 30.0]
    sol = solveRust(RustModel(RC=RC_val))
    scatter!(x̄, sol.P, label="RC = $(Int(RC_val))", markersize=3)
end
xlabel!("Mileage State (x)")
ylabel!("P(Replace)")
```



- Higher $RC \rightarrow$ agent waits **longer** before replacing $\rightarrow$ threshold shifts right

Effect of Maintenance Cost (θ_1)

```julia
plot()
for θ₁_val in [0.02, 0.03, 0.04, 0.05, 0.06]
    sol = solveRust(RustModel(θ₁=θ₁_val))
    scatter!(x̄, sol.P, label="θ₁ = $θ₁_val", markersize=3)
end
xlabel!("Mileage State (x)")
ylabel!("P(Replace)")
```



- Higher $\theta_1 \rightarrow$ maintenance costs rise **faster** $\rightarrow$ agent replaces **sooner**

Comparing to McCall

Comparative Static	McCall	Rust
Higher benefits / lower RC	More selective (higher reservation wage)	Less replacement (higher threshold mileage)
Higher firing rate / higher θ_1	Less selective (lower reservation wage)	More replacement (lower threshold mileage)
Higher β	More selective (patient, wait for good offers)	More replacement (forward-looking, avoid future costs)

Estimation

What We Observe

The Econometrician's Problem

- We observe a **panel** of buses over time:
  - Mileage state x_t at each period
  - Replacement decision $d_t \in \{0, 1\}$
- We do **not** observe:
  - The cost parameters θ_1 and RC
  - The unobserved shocks ε_t

Goal

Recover the **structural parameters** (θ_1, RC, θ_3) from the observed data (x_t, d_t).

The Log-Likelihood Function

- The probability of the observed data given parameters:

$$\ell(\theta) = \sum_{t=1}^{T} \left[d_t \log P(d_t = 1 \mid x_t, \theta) + (1 - d_t) \log P(d_t = 0 \mid x_t, \theta) + \log \Pr(x_{t+1} \mid x_t, d_t, \theta) \right]$$

- $P(d = 1 \mid x, \theta)$ comes from the **logit formula** — but requires solving the **entire DP** to compute!
- $\Pr(x_{t+1} \mid x_t, d_t, \theta)$ is the **mileage transition** probability, governed by θ_3
  - it is just the probability of the **observed increment**: $\Pr(x_{t+1} \mid x_t, d_t, \theta) = \theta_{3,\Delta x_t + 1}$
  - $\Delta x_t = x_{t+1} - x_t$ if keep, $\Delta x_t = x_{t+1}$ if replace

Two-Step Estimation

Key Insight

The transition probabilities θ_3 can be estimated **separately** from (RC, θ_1):

1. **Step 1**: Estimate $\hat{\theta}_3$ directly from mileage transitions (just counting)
2. **Step 2**: Given $\hat{\theta}_3$, estimate (RC, θ_1) by maximizing the choice likelihood using **NFXP**

- This works because θ_3 only affects the **transition** part of the likelihood, while (RC, θ_1) only affect the **choice** part (through the DP solution)

Simulating Data

The Simulation Function

```
function simulateRust(P_rep, θ₃, S, T; seed=42)
    Random.seed!(seed)
    x_sim = zeros(Int, T)                          # mileage states
    d_sim = zeros(Int, T)                          # replacement decisions
    x_sim[1] = 1                                   # start at state 1 (0 mileage)

    for t in 1:T
        d_sim[t] = rand() < P_rep[x_sim[t]] ? 1 : 0 # Bernoulli draw

        if t < T
            Δ = rand(Categorical(θ₃)) - 1          # mileage increment: 0, 1, or 2
            if d_sim[t] == 1
                x_sim[t+1] = min(1 + Δ, S)         # replace: reset then increment
            else
                x_sim[t+1] = min(x_sim[t] + Δ, S)  # keep: current + increment
            end
        end
    end
```

```
simulateRust (generic function with 1 method)
```

Generating Simulated Data

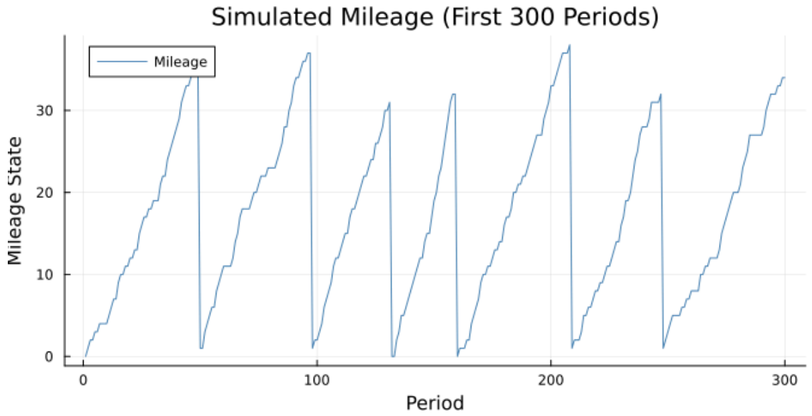
```julia
m = RustModel()                                    # true model
sol = solveRust(m)                                  # solve the DP
T = 5000                                           # number of periods

x_sim, d_sim = simulateRust(sol.P, m.θ₃, m.S, T)    # simulate data
println("Replacement rate: ", round(mean(d_sim), digits=3))
```

```
Replacement rate: 0.022
```

Simulated Mileage Path

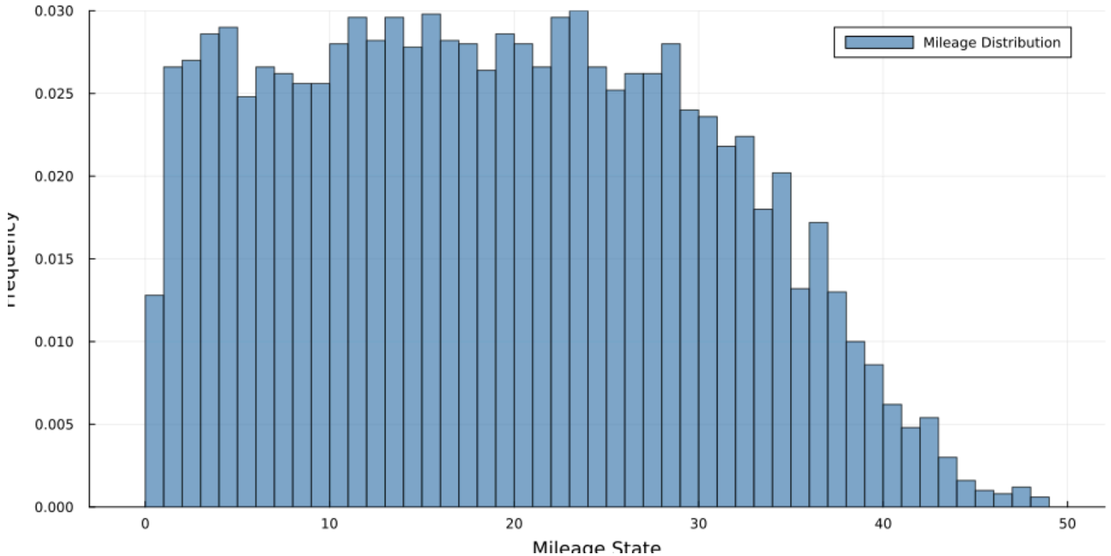
```
plot(x̄[x_sim[1:300]], linewidth=1, color=:steelblue, label="Mileage",
     title="Simulated Mileage (First 300 Periods)", size=(700, 350))
xlabel!("Period")
ylabel!("Mileage State")
```



- **Sawtooth pattern**: mileage gradually rises, then drops sharply at replacement

Mileage Distribution

```
histogram(x̄[x_sim], bins=S, normalize=true, label="Mileage Distribution",
          color=:steelblue, alpha=0.7)
xlabel!("Mileage State")
ylabel!("Frequency")
```



- More mass at **low mileage** — buses spend relatively little time at high mileage before being replaced

The NFXP Algorithm

Step 1: Estimate Transition Probabilities

- The transition likelihood $\sum_t \log \theta_{3,\Delta x_t+1}$ is maximized by the **sample frequencies** of each increment — just count them:

```
counts = zeros(3)                                  # for Δ = 0, 1, 2
for t in 1:T-1
    if d_sim[t] == 0 && x_sim[t] <= S - 2          # keep, away from boundary
        Δ = x_sim[t+1] - x_sim[t]
        counts[Δ + 1] += 1
    elseif d_sim[t] == 1                           # replace
        Δ = x_sim[t+1] - 1                         # next state = 1 + Δ
        counts[Δ + 1] += 1
    end
end

θ₃_hat = counts / sum(counts)                      # estimated transition probs
println("True θ₃:      ", round.(m.θ₃, digits=3))
println("Estimated θ₃: ", round.(θ₃_hat, digits=3))
```

```
True θ₃:      [0.36, 0.48, 0.16]
Estimated θ₃: [0.351, 0.486, 0.163]
```

Step 2: The NFXP Concept

For each candidate $\theta = (RC, \theta_1)$:

1. **Inner loop**: Solve the DP via VFI $\rightarrow$ get EV_θ
2. Compute choice probabilities $P(d \mid x, \theta)$
3. Evaluate the log-likelihood:

$$\ell(\theta) = \sum_{t=1}^{T} \left[d_t \log P(d_t = 1 \mid x_t, \theta) + (1 - d_t) \log P(d_t = 0 \mid x_t, \theta) \right]$$

Outer loop: Search over θ to maximize $\ell(\theta)$

> ⚠ **Why "Nested"?**
>
> Every single evaluation of the likelihood requires **solving the entire dynamic program**. The fixed point (VFI) is nested inside the optimization.

Step 2: The NFXP Concept

For each candidate $\theta = (RC, \theta_1)$:

1. **Inner loop**: Solve the DP via VFI $\rightarrow$ get EV_θ
2. Compute choice probabilities $P(d \mid x, \theta)$
3. Evaluate the log-likelihood:

$$\ell(\theta) = \sum_{t=1}^{T} \Big[\underbrace{d_t \log P(d_t = 1 \mid x_t, \theta)}_{\text{Replacement likelihood}} + \underbrace{(1 - d_t) \log P(d_t = 0 \mid x_t, \theta)}_{\text{Keep likelihood}} \Big]$$

Outer loop: Search over θ to maximize $\ell(\theta)$

> ⓘ **Why "Nested"?**
>
> Every single evaluation of the likelihood requires **solving the entire dynamic program**. The fixed point (VFI) is nested inside the optimization.

The NFXP Objective Function

```julia
function nfxpObjective(params, x_data, d_data, β, θ₃_hat, S)
    # unpack candidate parameters (RC, θ₁)
    # check parameter bounds
    # build trial model and solve the DP (inner loop)
    # compute log-likelihood from choice probabilities
    # return negative log-likelihood (for minimization)
end
```

```
nfxpObjective (generic function with 1 method)
```

The NFXP Objective Function

```julia
function nfxpObjective(params, x_data, d_data, β, θ₃_hat, S)
    # unpack candidate parameters (RC, θ₁)
    RC_est, θ₁_est = params
    # check parameter bounds
    if RC_est < 0 || θ₁_est < 0
        return 1e10
    end
    # build trial model and solve the DP (inner loop)
    m_trial = RustModel(β=β, S=S, RC=RC_est, θ₁=θ₁_est, θ₃=θ₃_hat)
    sol = solveRust(m_trial)
    # compute log-likelihood from choice probabilities
    ll = 0.0
    for t in eachindex(d_data)
        s = x_data[t]
        ll += d_data[t] * log(max(sol.P[s], 1e-15)) +
              (1 - d_data[t]) * log(max(1 - sol.P[s], 1e-15))
    end
    # return negative log-likelihood (for minimization)
```

```
nfxpObjective (generic function with 1 method)
```

Running the Estimation

```
using Optim
```

```
result = optimize(
    p -> nfxpObjective(p, x_sim, d_sim, β, θ₃_hat, S),
    [15.0, 0.03],                                  # initial guess: [RC, θ₁]
    NelderMead(),                                  # derivative-free optimizer
    Optim.Options(iterations=5000)
)
RC_hat, θ₁_hat = Optim.minimizer(result)
```

```
2-element Vector{Float64}:
 21.220120311603008
  0.04306776468765347
```

Comparing Estimates to Truth

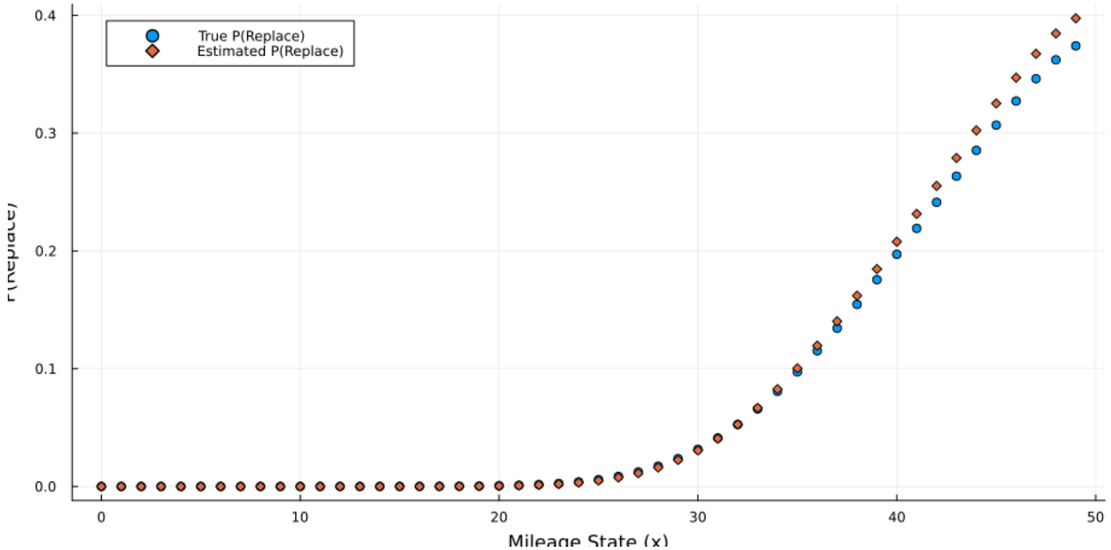
```
println("──────────────────────────────────────────")
println("Parameter |   True   | Estimated")
println("──────────────────────────────────────────")
println("    RC    |  $(lpad(m.RC, 6))  |  $(lpad(round(RC_hat, digits=3), 6))")
println("    θ₁    |  $(lpad(m.θ₁, 6))  |  $(lpad(round(θ₁_hat, digits=4), 6))")
println("──────────────────────────────────────────")
```

```
──────────────────────────────────────────
Parameter |   True   | Estimated
──────────────────────────────────────────
    RC    |    20.0  |   21.22
    θ₁    |    0.04  |  0.0431
──────────────────────────────────────────
```

Estimated vs. True Replacement Probability

```
m_hat = RustModel(RC=RC_hat, θ₁=θ₁_hat, θ₃=θ₃_hat)  # estimated model
sol_hat = solveRust(m_hat)

scatter(x̄, sol.P, label="True P(Replace)", markersize=4)
scatter!(x̄, sol_hat.P, label="Estimated P(Replace)", markersize=4,
         markershape=:diamond)
xlabel!("Mileage State (x)")
ylabel!("P(Replace)")
```



- The estimated choice probabilities should closely match the true ones

Why Structural Estimation?

The Power of Structural Models

With the estimated structural parameters, we can perform **counterfactual analysis**:

- What if the government **subsidizes replacement** (lowers RC)?
- What if a new technology **reduces maintenance costs** (lowers θ_1)?
- What if buses are driven **more intensively** (changes θ_3)?

A reduced-form model (e.g., a logit on current mileage) **cannot** answer these questions — it would break under the policy change. The structural model accounts for how agents **re-optimize** under new conditions.

Summary

Key Takeaways

Concepts

- Dynamic discrete choice models
- Bellman equation with logit shocks
- Value function iteration
- Reservation mileage (analog of reservation wage)
- NFXP: structural estimation of DPs
- Counterfactual policy analysis

Julia Tools

- `max.()`, `exp.()`, `log.()` broadcasting
- `norm()` for convergence checks
- Expectations as probability-weighted sums
- Structs with `@kwdef`, destructuring `(; ...)`
- `Categorical` for discrete simulation
- `Optim.jl` for maximum likelihood

Remember

Rust (1987) showed that **dynamic programming models can be estimated from data**. The key ingredients: (1) a structural model of forward-looking behavior, (2) distributional assumptions on unobservables, and (3) the NFXP algorithm to solve the model inside the estimator.