Finite State Markov Chains

David Evans

2026-11-13

Discrete Random Variables

Definition

Discrete Random Variables

- A discrete random variable is defined by two objects:

1. A vector of possible **values**

$$\bar{x} = \begin{pmatrix} \bar{x}_1 \\ \bar{x}_2 \\ \vdots \\ \bar{x}_n \end{pmatrix}$$

2. A vector of **probabilities**

$$p = \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix}$$

Probability Constraints

- p_i represents the probability that x takes value $\bar{x}_i$
- Probabilities must sum to 1:

$$\sum_i p_i = 1$$

Example: Coin Toss

- Bet $1 on a coin toss. Your winnings w are described by:

$$\bar{w} = \begin{pmatrix} -1 \\ 1 \end{pmatrix} \quad \text{and} \quad p = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$$

- With a **loaded coin** (80% heads):

$$p = \begin{pmatrix} 0.8 \\ 0.2 \end{pmatrix}$$

Julia Implementation

Setup

```
using Distributions
using LinearAlgebra
```

```
using Plots
```

Simulating a Coin Toss

- The `Categorical` distribution represents a discrete random variable

```
cointoss = Categorical([0.5, 0.5])
```

```
Distributions.Categorical{Float64, Vector{Float64}}(support=Base.OneTo(2), p=[0.5, 0.5])
```

```
rand(cointoss)       # returns either 1 or 2
```

```
1
```

```
rand(cointoss, 5)    # simulates 5 random coin tosses
```

```
5-element Vector{Int64}:
 1
 1
 1
 1
 1
```

Translating Draws to Values

- Use a vector of values to map draws to outcomes

```
 1 wbar = [-1., 1.]
```

```
2-element Vector{Float64}:
 -1.0
  1.0
```

```
 1 w = wbar[rand(cointoss)]    # single toss
```

```
-1.0
```

- Can plug vector of indices into `wbar` to get the vector of winnings

```
 1 tosses   = rand(cointoss, 10)    # 10 state indices (1s and 2s)
 2 winnings = wbar[tosses]          # Map each index to its value
 3 hcat(tosses, winnings)           # Display side by side
```

```
10×2 Matrix{Float64}:
 1.0  -1.0
 1.0  -1.0
 1.0  -1.0
 1.0  -1.0
 2.0   1.0
 2.0   1.0
 2.0   1.0
 1.0  -1.0
 2.0   1.0
 2.0   1.0
```

Loaded Coin

```
1  loadedcoin = Categorical([0.8, 0.2])
```

```
Distributions.Categorical{Float64, Vector{Float64}}(support=Base.OneTo(2), p=[0.8, 0.2])
```

```
1  w = wbar[rand(loadedcoin, 10)]
```

```
10-element Vector{Float64}:
  1.0
 -1.0
 -1.0
 -1.0
 -1.0
 -1.0
 -1.0
  1.0
 -1.0
 -1.0
```

> **Notice**
>
> With the loaded coin, you see many more -1s (heads/losses) than 1s (tails/wins).

Finite State Markov Chains

Definition

What is a Markov Chain?

- A Markov chain describes a discrete random variable whose **conditional distribution varies over time**
- Let x_t be a Markov chain. It is characterized by:

1. A vector of possible **values** $\bar{x}$
2. A **transition matrix** of probabilities

$$P = \begin{pmatrix} P_{1,1} & P_{1,2} & \cdots & P_{1,n} \\ P_{2,1} & P_{2,2} & \cdots & P_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ P_{n,1} & P_{n,2} & \cdots & P_{n,n} \end{pmatrix}$$

Interpreting the Transition Matrix

- $P_{i,j}$ = probability that $x_t = \bar{x}_j$ **given** that $x_{t-1} = \bar{x}_i$

- All elements of P must be non-negative
- Each **row** must sum to 1

Key Property

Row i of P gives the full conditional distribution of x_t given $x_{t-1} = \bar{x}_i$.

Why Markov Chains?

- We often want to model random variables with **persistence**
- Example: business cycles
  - In a boom $\rightarrow$ more likely to stay in a boom
  - In a recession $\rightarrow$ more likely to stay in a recession

Business Cycle Example

Modeling Output Growth

- Two states: expansion (3% growth) and recession (-1% growth)

```
# ḡ is typed as g\bar then tab
ḡ = [0.03, -0.01]
```

```
2-element Vector{Float64}:
  0.03
 -0.01
```

- Transition matrix:

```
P = [0.9 0.1;
     0.3 0.7]
```

```
2×2 Matrix{Float64}:
 0.9  0.1
 0.3  0.7
```

- 90% chance of staying in a boom; 70% chance of staying in a recession

Reading the Transition Matrix

- Probability of transitioning from boom to recession:

```
1 P[1, 2]
```

```
0.1
```

- Full conditional distribution given we're in a boom:

```
1 P[1, :]
```

```
2-element Vector{Float64}:
 0.9
 0.1
```

- Draw a value of output next period (given boom today):

```
1 pboom = Categorical(P[1, :])
2 ḡ[rand(pboom)]
```

```
0.03
```

Simulating Markov Chains

The `drawDiscrete` Function

- Let's wrap this logic in a reusable function

```
"""
    drawDiscrete(p, X̄)

Draws a discrete random variable with probability vector p
over values X̄. Returns (state, value).
"""
function drawDiscrete(p, X̄)      # Declare the function
    dist = Categorical(p)         # Create the distribution from p
    s = rand(dist)                # Draw a random state index
    X = X̄[s]                     # Look up the value for that state
    return s, X                   # Return both the index and value
end
```

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

Using `drawDiscrete`

- Fair coin toss:

```
1 drawDiscrete([0.5, 0.5], [-1, 1])
```

```
(2, 1)
```

- Loaded coin toss:

```
1 drawDiscrete([0.8, 0.2], [-1, 1])
```

```
(2, 1)
```

The `simulateMarkov` Function

- Let's wrap the simulation logic in a reusable function

```
function simulateMarkov(P, X̄, T; s₁)
    # Set up storage for values and states
    # Initialize first period
    # For each period t = 2, ..., T
        # Draw next state from row s[t-1] of P
    # Return states and values
end
```

The `simulateMarkov` Function

- Let's wrap the simulation logic in a reusable function

```
function simulateMarkov(P, X̄, T; s₁=1)
    X = zeros(T)                          # Set up storage for values
    s = zeros(Int, T)                     # Set up storage for states
    s[1] = s₁                             # Initialize first period
    X[1] = X̄[s₁]                         # Initialize first period
    for t in 2:T                          # For each period t = 2, ..., T
        s[t], X[t] = drawDiscrete(P[s[t-1], :], X̄)  # Draw next state from row s[t-1] of P
    end
    return s, X                           # Return states and values
end
```

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

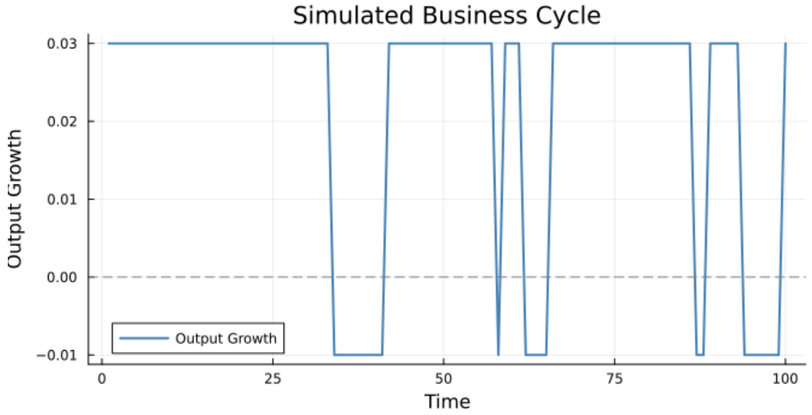
Simulating a Markov Chain

```
1 s, g = simulateMarkov(P, ḡ, 100)
```

```
([1, 1, 1, 1, 1, 1, 1, 1, 1, 1  …  1, 1, 1, 2, 2, 2, 2, 2, 2, 1], [0.03, 0.03, 0.03, 0.03, 0.03, 0.03, 0.03, 0.03, 0.03, 0.03  …  0.03, 0.03,
0.03, -0.01, -0.01, -0.01, -0.01, -0.01, -0.01, 0.03])
```

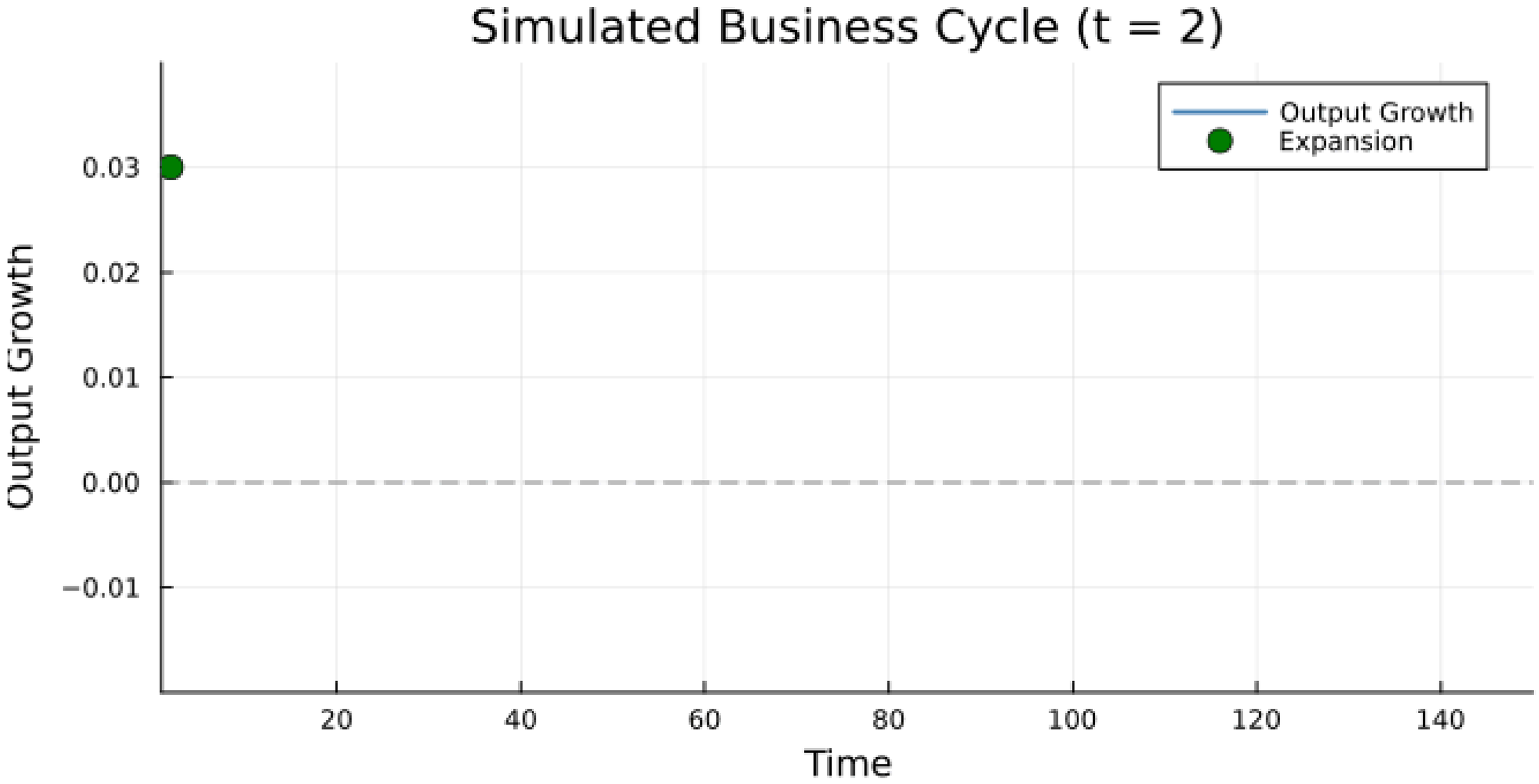
Plotting the Simulation

```
plot(g, linewidth=2, color=:steelblue, label="Output Growth",
     title="Simulated Business Cycle", size=(700, 350))
hline!([0.0], color=:gray, linestyle=:dash, label="")
xlabel!("Time")
ylabel!("Output Growth")
```



Animating the Markov Chain

```
[ Info: Saved animation to /tmp/jl_ZODQBaNU7O.gif
```



Expectations

Expected Values

Expected Value of a Discrete Random Variable

- The expected value is the probability-weighted average of all possible values:

$$\mathbb{E}[X] = p_1\bar{X}_1 + p_2\bar{X}_2 + \ldots + p_n\bar{X}_n = \sum_{j=1}^{n} p_j\bar{X}_j$$

- In Julia, this is a **dot product**!

Computing Expected Values

```
# Fair coin toss
p = [0.5, 0.5]
w̄ = [-1., 1.]
Ew = dot(p, w̄)   # expected winnings
```

```
0.0
```

```
# Loaded coin toss
p = [0.8, 0.2]
Ew = dot(p, w̄)   # expected winnings
```

```
-0.6000000000000001
```

- This is the same as computing it manually:

```
p[1]*w̄[1] + p[2]*w̄[2]
```

```
-0.6000000000000001
```

Monte Carlo Verification

- We can verify using simulation (law of large numbers):

```
N = 100000                          # Number of draws
w_simulations = zeros(N)            # Pre-allocate results

for i in 1:N                        # Simulate N coin tosses
    s, w = drawDiscrete(p, w̄)      # Toss coin, get winnings
    w_simulations[i] = w            # Store the result
end
mean(w_simulations)                 # Sample average ≈ true expectation
```

```
-0.60112
```

Monte Carlo Method

As $N \to \infty$, the sample average converges to the true expected value.

Conditional Expectations

Conditional Expectations for Markov Chains

- For a Markov chain we can compute **conditional expectations**:

$$\mathbb{E}[X_{t+1}|X_t = \bar{X}_i]$$

- Since the conditional distribution is given by row i of P:

$$\mathbb{E}[X_{t+1}|X_t = \bar{X}_i] = \sum_j P_{i,j}\bar{X}_j$$

Conditional Expectations in Julia

```julia
# Expected growth rate conditional on boom
dot(P[1, :], ḡ)
```

```
0.026
```

```julia
# Expected growth rate conditional on recession
dot(P[2, :], ḡ)
```

```
0.0019999999999999996
```

Conditional Expectations as Matrix Product

- Each conditional expectation is a **dot product** of $\bar{X}$ with a row of P:

$$\mathbb{E}[X_{t+1}|X_t = \bar{X}_i] = P_{i,.} \cdot \bar{X}$$

- Stacking all rows together, this is just matrix multiplication:

$$\begin{pmatrix} \mathbb{E}[X_{t+1}|X_t = \bar{X}_1] \\ \mathbb{E}[X_{t+1}|X_t = \bar{X}_2] \\ \vdots \\ \mathbb{E}[X_{t+1}|X_t = \bar{X}_n] \end{pmatrix} = \begin{pmatrix} P_{1,.} \cdot \bar{X} \\ P_{2,.} \cdot \bar{X} \\ \vdots \\ P_{n,.} \cdot \bar{X} \end{pmatrix} = P\bar{X}$$

Key Result

Each row of P dotted with $\bar{X}$ gives one conditional expectation. Stacking them all is just $P\bar{X}$!

Verifying with Simulation

Setting Up the Simulation

```
1 s, g = simulateMarkov(P, ḡ, 5000)   # 5000 periods for convergence
2 T = length(g)
```

```
5000
```

Computing Empirical Conditional Expectations

- Collect output growth values preceded by expansions vs. recessions:

```
gboom = []                          # Will hold growth after booms
grec = []                           # Will hold growth after recessions

for t in 1:T-1                     # Loop over all periods
    if s[t] == 1                    # Was period t a boom?
        push!(gboom, g[t+1])       #   → save next period's growth
    else                            # Was period t a recession?
        push!(grec, g[t+1])        #   → save next period's growth
    end
end
```

Comparing Empirical to Theoretical

- Empirical conditional expectations:

```
1 [mean(gboom), mean(grec)]
```

```
2-element Vector{Float64}:
 0.026010582010582158
 0.002370795734208368
```

- Theoretical values ($P\bar{X}$):

```
1 P * ḡ
```

```
2-element Vector{Float64}:
 0.026
 0.0019999999999999996
```

List Comprehension Alternative

- More concise way using **list comprehensions**:

```
gboom = [g[t+1] for t in 1:T-1 if s[t] == 1]
grec  = [g[t+1] for t in 1:T-1 if s[t] == 2]

[mean(gboom), mean(grec)]
```

```
2-element Vector{Float64}:
 0.026010582010582033
 0.002370795734208367
```

> **Julia Tip**
>
> List comprehensions with `if` conditions are a powerful and concise way to filter and collect values!

Estimating the Transition Matrix

Maximum Likelihood Estimation

The Estimation Problem

- So far we've **assumed** the transition matrix P is known
- In practice, we observe a sequence of states and want to **estimate** P

- Suppose we observe a time series of states $s_1, s_2, \ldots, s_T$
- **Question:** What transition matrix P most likely generated this data?

> **Maximum Likelihood**
>
> Pick the parameters that make the observed data **most probable**.

The Likelihood Function

- The probability of observing the full sequence $s_1, s_2, \ldots, s_T$ is:

$$L(P) = \prod_{t=2}^{T} P_{s_{t-1}, s_t}$$

- Taking logs (easier to work with):

$$\log L(P) = \sum_{t=2}^{T} \log P_{s_{t-1}, s_t}$$

Counting Transitions

- Define n_{ij} = number of times we observe a transition from state i to state j:

$$n_{ij} = \sum_{t=2}^{T} \mathbf{1}(s_{t-1} = i,\ s_t = j)$$

- Then the log-likelihood simplifies to:

$$\log L(P) = \sum_{i=1}^{n} \sum_{j=1}^{n} n_{ij} \log P_{ij}$$

The MLE Solution

- Maximizing $\log L$ subject to $\sum_j P_{ij} = 1$ for each row i gives:

$$\hat{P}_{ij} = \frac{n_{ij}}{\sum_{k=1}^{n} n_{ik}} = \frac{n_{ij}}{n_{i.}}$$

MLE of Transition Probabilities

The MLE estimate of P_{ij} is simply the **fraction of transitions from state i that go to state j**.

Julia Implementation

`countTransitions`: Pseudocode

- First, let's write a function to count transitions:

```
function countTransitions(s, n_states)
    # Initialize an n_states × n_states matrix of zeros
    # Loop through the sequence: for each consecutive pair (s[t-1], s[t])
    #   increment the count at position (i, j)
    # Return the count matrix
end
```

`countTransitions`: Full Function

```julia
"""
    countTransitions(s, n_states)

Count transitions in state sequence `s` and return
an n_states × n_states matrix of counts.
"""
function countTransitions(s, n_states)
    N = zeros(Int, n_states, n_states)   # Initialize count matrix
    for t in 2:length(s)
        N[s[t-1], s[t]] += 1            # Increment the (i,j) count
    end
    return N
end
```

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

- `N[i, j]` counts how many times the chain moved from state i to state j
- We start at $t = 2$ because each transition requires two consecutive observations

`estimateTransitionMatrix`: Pseudocode

- Normalize each row to get the MLE estimate:

```
function estimateTransitionMatrix(s, n_states)
    # Count transitions using countTransitions
    # Normalize: divide each row by its row sum
    # Return the estimated transition matrix
end
```

`estimateTransitionMatrix`: Full Function

```julia
"""
    estimateTransitionMatrix(s, n_states)

Estimate the transition matrix from a state sequence
using maximum likelihood (row-normalized transition counts).
"""
function estimateTransitionMatrix(s, n_states)
    N = countTransitions(s, n_states)
    P̂ = N ./ sum(N, dims=2)              # Divide each row by its sum
    return P̂
end
```

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

Testing on Our Simulated Data

- Recall we simulated a Markov chain with:

```
1 P
```

```
2×2 Matrix{Float64}:
 0.9  0.1
 0.3  0.7
```

- Let's estimate from the simulated state sequence `s` ($T = 100$):

```
1 s = s[1:100]
2 P̂ = estimateTransitionMatrix(s, 2)
```

```
2×2 Matrix{Float64}:
 0.894737  0.105263
 0.347826  0.652174
```

How Close Are We?

- Compare the true and estimated matrices:

```
println("True P: $(round.(P, digits=3))")
println("\nEstimated P̂: $(round.(P̂, digits=3))")
println("\nDifference (P̂ - P): $(round.(P̂ .- P, digits=3))")
```

```
True P: [0.9 0.1; 0.3 0.7]

Estimated P̂: [0.895 0.105; 0.348 0.652]

Difference (P̂ - P): [-0.005 0.005; 0.048 -0.048]
```

Consistency

As $T \to \infty$, the MLE $\hat{P}$ converges to the true P. With $T = 5{,}000$ we already get very close!

How Sample Size Affects Accuracy

- Let's see how the estimate improves with more data:

```julia
sample_sizes = [50, 200, 1000, 5000, 20000]

errors = map(sample_sizes) do T_sim    #Fancy way to loop over sample sizes :)
    s_sim, _ = simulateMarkov(P, g̃, T_sim)   # Simulate a Markov chain
    P̂_sim = estimateTransitionMatrix(s_sim, 2)
    return maximum(abs.(P̂_sim .- P))   # Max absolute error
end

hcat(sample_sizes, round.(errors, digits=4))
```

How Sample Size Affects Accuracy

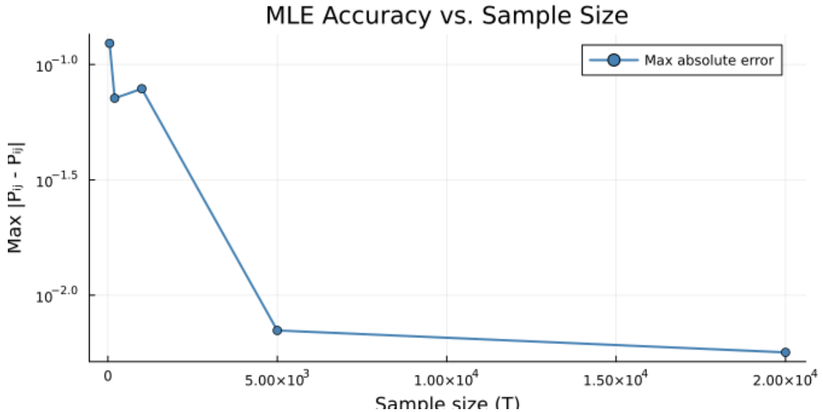
- Let's see how the estimate improves with more data:

```
sample_sizes = [50, 200, 1000, 5000, 20000]

errors = map(sample_sizes) do T_sim    #Fancy way to loop over sample sizes :)
    s_sim, _ = simulateMarkov(P, g̃, T_sim)   # Simulate a Markov chain
    P̂_sim = estimateTransitionMatrix(s_sim, 2)
    return maximum(abs.(P̂_sim .- P))   # Max absolute error
end

hcat(sample_sizes, round.(errors, digits=4))
```

```
5×2 Matrix{Float64}:
    50.0  0.1235
   200.0  0.0714
  1000.0  0.0785
  5000.0  0.007
 20000.0  0.0057
```

Visualizing Convergence

```julia
plot(sample_sizes, errors, marker=:circle, linewidth=2,
     color=:steelblue, label="Max absolute error", yscale=:log10,
     title="MLE Accuracy vs. Sample Size", size=(700, 350))
xlabel!("Sample size (T)")
ylabel!("Max |P̂ᵢⱼ - Pᵢⱼ|")
```



Log-Likelihood and Numerical Optimization

Computing the Log-Likelihood Directly

- We can also compute the log-likelihood for any candidate P:

```
"""
    loglikelihood_mc(P, s)

Compute the log-likelihood of state sequence `s`
given transition matrix `P`.
"""
function loglikelihood_mc(P, s)
    ll = 0.0
    for t in 2:length(s)
        ll += log(P[s[t-1], s[t]])
    end
    return ll
end
```

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

Comparing Log-Likelihoods

- The MLE should have a **higher** log-likelihood than any other valid P:

```julia
P_wrong = [0.5 0.5;
           0.5 0.5]

println("Log-likelihood at P̂ (MLE):    ", round(loglikelihood_mc(P̂, s), digits=2))
println("Log-likelihood at true P:      ", round(loglikelihood_mc(P, s), digits=2))
println("Log-likelihood at P = [.5 .5; .5 .5]: ", round(loglikelihood_mc(P_wrong, s), digits=2))
```

```
Log-likelihood at P̂ (MLE):    -40.43
Log-likelihood at true P:      -40.57
Log-likelihood at P = [.5 .5; .5 .5]: -68.62
```

Key Insight

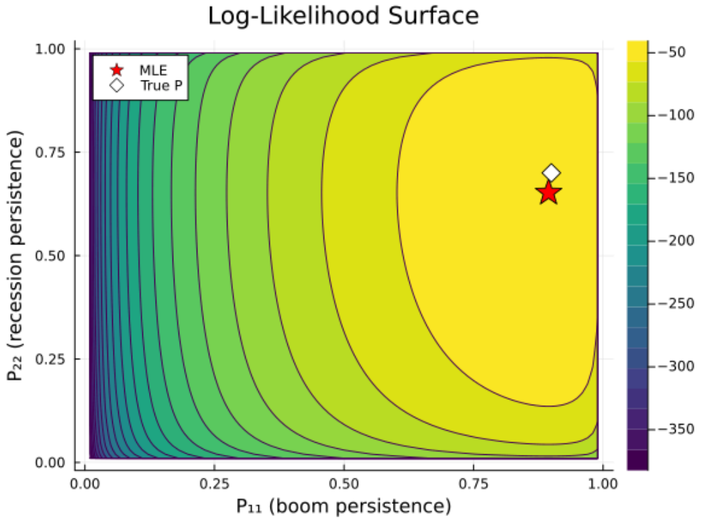
The MLE has the highest log-likelihood — it fits the observed data best. The true P is close behind, while a naive equal-probability matrix does much worse.

Visualizing the Likelihood Surface

- With a 2×2 matrix, the free parameters are P_{11} and P_{22} (the persistence parameters)

```
p11_grid = 0.01:0.01:0.99
p22_grid = 0.01:0.01:0.99

#Compute the log-likelihood for each combination of p11 and p22
ll_surface = [loglikelihood_mc([p11 1-p11; 1-p22 p22], s)
              for p11 in p11_grid, p22 in p22_grid]

#Plot the log-likelihood surface
contourf(p11_grid, p22_grid, ll_surface',
         xlabel="P₁₁ (boom persistence)",
         ylabel="P₂₂ (recession persistence)",
         title="Log-Likelihood Surface",
         color=:viridis, size=(600, 450), levels=20)
#Add the MLE and true P to the plot
scatter!([P̂[1,1]], [P̂[2,2]], marker=:star5, markersize=12,
         color=:red, label="MLE")
#Add the true P to the plot
scatter!([P[1,1]], [P[2,2]], marker=:diamond, markersize=8,
```

Visualizing the Likelihood Surface



Numerical MLE with `Optim.jl`

- We can also let the computer **find** the peak of the likelihood surface
- Use `Optim.jl` to minimize the **negative** log-likelihood

```
1 using Optim
```

Setting Up the Optimization

- Pack the free parameters into a vector $\theta = (P_{11}, P_{22})$
- Reconstruct the full 2×2 matrix inside the objective:

```
function neg_loglik(θ, s)
    p11, p22 = θ                              # Unpack the parameters
    P_candidate = [p11 1-p11;                 # Construct the candidate matrix
                   1-p22  p22]
    return -loglikelihood_mc(P_candidate, s)   # negative log-likelihood to minimize
end
```

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

Running the Optimizer

- Use box constraints to keep probabilities in $(0, 1)$:

```
θ₀ = [0.5, 0.5]                          # Initial guess (uniform)
result = optimize(θ -> neg_loglik(θ, s),  # Objective
                  [0.01, 0.01],           # Lower bounds
                  [0.99, 0.99],           # Upper bounds
                  θ₀,                     # Starting point
                  Fminbox(LBFGS()))        # Bounded L-BFGS algorithm
```

```
* Status: success

* Candidate solution
   Final objective value:     4.043376e+01

* Found with
   Algorithm:     Fminbox with L-BFGS

* Convergence measures
   |x - x'|               = 2.27e-10 ≤ 0.0e+00
   |x - x'|/|x'|          = 2.54e-10 ≤ 0.0e+00
   |f(x) - f(x')|         = 6.57e-11 ≤ 0.0e+00
   |f(x) - f(x')|/|f(x')| = 1.62e-12 ≤ 0.0e+00
   |g(x)|                 = 4.11e-09 ≤ 1.0e-08
```

Extracting the Results

```julia
θ_mle = Optim.minimizer(result)
P̂_optim = [θ_mle[1] 1-θ_mle[1];
            1-θ_mle[2]  θ_mle[2]]

println("Optim MLE: $(round.(P̂_optim, digits=3))")
println("\nClosed-form MLE: $(round.(P̂, digits=3))")
println("\nTrue P: $(round.(P, digits=3))")
```

```
Optim MLE: [0.895 0.105; 0.348 0.652]

Closed-form MLE: [0.895 0.105; 0.348 0.652]

True P: [0.9 0.1; 0.3 0.7]
```

Verify

The numerical optimizer finds the same answer as our closed-form solution — a good sanity check! In more complex models where no closed form exists, `Optim.jl` is essential.

Likelihood Sees Transitions, Not Frequencies

Two Very Different Histories

- Suppose we observe one of two sequences of $T = 100$ states:

```
s_stay   = repeat([1,1,1,1,1,1,1,1,1,1, 2,2,2,2,2,2,2,2,2,2], 5)   # Long runs in each state
s_switch = repeat([1,2,1,2,1,2,1,2,1,1, 2,1,2,1,2,1,2,1,2,2], 5)   # (Almost) always alternates
```

- Both spend **exactly half** their time in each state:

```
println("Periods in state 1:   s_stay = $(sum(s_stay .== 1)),   s_switch = $(sum(s_switch .== 1))")
```

```
Periods in state 1:   s_stay = 50,   s_switch = 50
```

> ⚠ **Frequencies Can't Tell Them Apart**
>
> If we only counted how often each state shows up, these two histories would look identical!

Three Candidate Transition Matrices

```
P_persist = [0.9 0.1;      # Usually stay put
             0.1 0.9]
P_coin    = [0.5 0.5;      # Ignore today's state: i.i.d. coin flips
             0.5 0.5]
P_switch  = [0.1 0.9;      # Usually switch
             0.9 0.1]
```

- In each matrix $P_{12} = P_{21}$, so in the long run each chain spends **half** its time in each state
- They differ only in how likely the chain is to **stay put**

Which Matrix Fits Which Sequence?

```
for (name, sq) in [("s_stay", s_stay), ("s_switch", s_switch)]
    println(rpad(name, 10),
            "log L(P_persist) = ", round(loglikelihood_mc(P_persist, sq), digits=1),
            "    log L(P_coin) = ", round(loglikelihood_mc(P_coin, sq), digits=1),
            "    log L(P_switch) = ", round(loglikelihood_mc(P_switch, sq), digits=1))
end
```

```
s_stay    log L(P_persist) = -30.2    log L(P_coin) = -68.6    log L(P_switch) = -208.2
s_switch  log L(P_persist) = -206.0    log L(P_coin) = -68.6    log L(P_switch) = -32.4
```

- `P_coin` gives **both** sequences the same log-likelihood: every transition has probability ½
- `P_persist` wins by a mile for `s_stay`; `P_switch` wins by a mile for `s_switch`
- A gap of about 175 in $\log L$ means the data are $e^{175} \approx 10^{76}$ times more likely under the better matrix!

Counting Transitions Reveals the Difference

`s_stay`

```
countTransitions(s_stay, 2)
```

```
2×2 Matrix{Int64}:
 45   5
  4  45
```

```
estimateTransitionMatrix(s_stay, 2)
```

```
2×2 Matrix{Float64}:
 0.9        0.1
 0.0816327  0.918367
```

`s_switch`

```
countTransitions(s_switch, 2)
```

```
2×2 Matrix{Int64}:
  5  45
 44   5
```

```
estimateTransitionMatrix(s_switch, 2)
```

```
2×2 Matrix{Float64}:
 0.1       0.9
 0.897959  0.102041
```

- Same number of 1s and 2s, but **opposite** transition counts $\Rightarrow$ opposite $\hat{P}$

Why? The Likelihood Only Uses Transitions

- Restrict to symmetric matrices, where p is the probability of staying put:

$$P(p) = \begin{pmatrix} p & 1-p \\ 1-p & p \end{pmatrix}$$

- Then $\log L = \sum_{i,j} n_{ij} \log P_{ij}$ becomes

$$\log L(p) = n_{\text{stay}} \log p + n_{\text{switch}} \log(1-p), \qquad n_{\text{stay}} = n_{11} + n_{22}, \quad n_{\text{switch}} = n_{12} + n_{21}$$

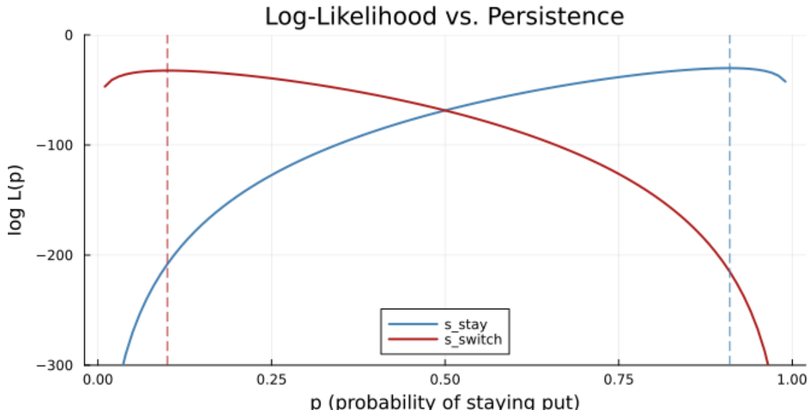
- Setting the derivative to zero gives the fraction of transitions that are stays:

$$\hat{p} = \frac{n_{\text{stay}}}{n_{\text{stay}} + n_{\text{switch}}}$$

- `s_stay`: $\hat{p} = 90/99 \approx 0.91$ `s_switch`: $\hat{p} = 10/99 \approx 0.10$

Log-Likelihood as a Function of Persistence

```julia
p_grid = 0.01:0.01:0.99
ll_stay   = [loglikelihood_mc([p 1-p; 1-p p], s_stay)   for p in p_grid]
ll_switch = [loglikelihood_mc([p 1-p; 1-p p], s_switch) for p in p_grid]

plot(p_grid, ll_stay, linewidth=2, color=:steelblue, label="s_stay",
     title="Log-Likelihood vs. Persistence", size=(700, 350),
     ylims=(-300, 0), legend=:bottom)
plot!(p_grid, ll_switch, linewidth=2, color=:firebrick, label="s_switch")
vline!([p_grid[argmax(ll_stay)]], color=:steelblue, linestyle=:dash, label="")
vline!([p_grid[argmax(ll_switch)]], color=:firebrick, linestyle=:dash, label="")
xlabel!("p (probability of staying put)")
ylabel!("log L(p)")
```



The Takeaway

- The two sequences have **identical** frequencies but **opposite** transitions
- Their likelihoods peak at very different p: ≈ 0.91 vs. ≈ 0.10

Likelihood Sees Transitions

The log-likelihood $\sum_{i,j} n_{ij} \log P_{ij}$ depends on the data **only through the transition counts** n_{ij}. That is why maximum likelihood can sharply distinguish a persistent chain from a switching one, even when both spend the same fraction of time in each state.

Application: U.S. Business Cycles

NBER Recession Dates

Estimating Business Cycles from Real Data

- So far we've **assumed** or **simulated** the transition matrix
- Can we estimate the boom/recession transition matrix from **actual U.S. data**?

- The **NBER** (National Bureau of Economic Research) maintains official U.S. recession dates
- Each recession is defined by a **peak** (start) and **trough** (end) month

NBER Post-WWII Recessions

- We'll use all post-WWII recessions (1948–2020):

```
using Dates
```

```
# NBER recession dates: (peak, trough)
nber_recessions = [
    (Date(1948,11), Date(1949,10)),
    (Date(1953,7),  Date(1954,5)),
    (Date(1957,8),  Date(1958,4)),
    (Date(1960,4),  Date(1961,2)),
    (Date(1969,12), Date(1970,11)),
    (Date(1973,11), Date(1975,3)),
    (Date(1980,1),  Date(1980,7)),
    (Date(1981,7),  Date(1982,11)),
    (Date(1990,7),  Date(1991,3)),
    (Date(2001,3),  Date(2001,11)),
    (Date(2007,12), Date(2009,6)),
    (Date(2020,2),  Date(2020,4))
]
```

```
12-element Vector{Tuple{Date, Date}}:
 (Date("1948-11-01"), Date("1949-10-01"))
 (Date("1953-07-01"), Date("1954-05-01"))
 (Date("1957-08-01"), Date("1958-04-01"))
 (Date("1960-04-01"), Date("1961-02-01"))
 (Date("1969-12-01"), Date("1970-11-01"))
 (Date("1973-11-01"), Date("1975-03-01"))
 (Date("1980-01-01"), Date("1980-07-01"))
 (Date("1981-07-01"), Date("1982-11-01"))
 (Date("1990-07-01"), Date("1991-03-01"))
 (Date("2001-03-01"), Date("2001-11-01"))
 (Date("2007-12-01"), Date("2009-06-01"))
 (Date("2020-02-01"), Date("2020-04-01"))
```

Building a Monthly State Sequence

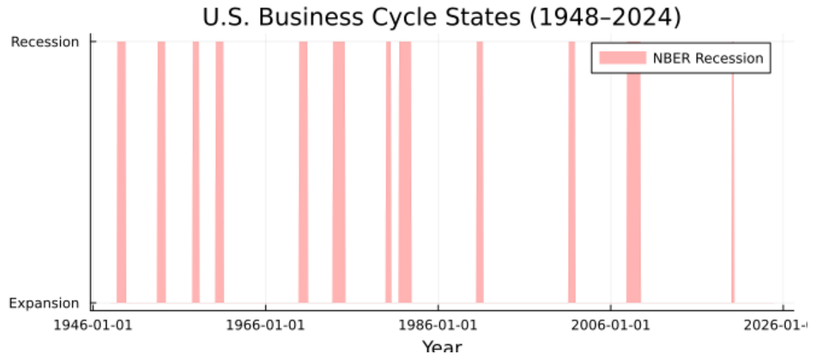
- Create a monthly indicator: **1 = expansion**, **2 = recession**

```
start_date = Date(1948, 1)
end_date   = Date(2024, 12)
months = collect(start_date:Month(1):end_date)

# Default to expansion (state 1)
states = ones(Int, length(months))

for (peak, trough) in nber_recessions
    for (t, m) in enumerate(months)
        if peak <= m <= trough
            states[t] = 2   # recession
        end
    end
end

println("Sample: $(start_date) to $(end_date) — $(length(months)) months")
println("Months in expansion: $(sum(states .== 1))")
println("Months in recession: $(sum(states .== 2))")
```

```
Sample: 1948-01-01 to 2024-12-01 — 924 months
Months in expansion: 788
Months in recession: 136
```

Visualizing the State Sequence

```
plot(months, states .== 2, fillrange=0, fillalpha=0.3,
     color=:red, label="NBER Recession", linewidth=0,
     title="U.S. Business Cycle States (1948–2024)",
     size=(700, 300), yticks=([0, 1], ["Expansion", "Recession"]),
     legend=:topright)
xlabel!("Year")
```



Estimating the Transition Matrix

Applying Our Estimator to Real Data

- We already built `estimateTransitionMatrix` — let's use it!

```
 1 P̂_nber = estimateTransitionMatrix(states, 2)
```

```
2×2 Matrix{Float64}:
 0.984752   0.0152478
 0.0882353  0.911765
```

- Interpretation:
  - $\hat{P}_{11}$: probability of **staying** in an expansion
  - $\hat{P}_{12}$: probability of **entering** a recession
  - $\hat{P}_{21}$: probability of **exiting** a recession
  - $\hat{P}_{22}$: probability of **staying** in a recession

What Does the Data Tell Us?

```
println("Estimated monthly transition matrix:")
println("  Expansion → Expansion: $(round(P̂_nber[1,1], digits=4))")
println("  Expansion → Recession: $(round(P̂_nber[1,2], digits=4))")
println("  Recession → Expansion: $(round(P̂_nber[2,1], digits=4))")
println("  Recession → Recession: $(round(P̂_nber[2,2], digits=4))")
```

```
Estimated monthly transition matrix:
  Expansion → Expansion: 0.9848
  Expansion → Recession: 0.0152
  Recession → Expansion: 0.0882
  Recession → Recession: 0.9118
```

Expected Duration

If the probability of staying in state i is P_{ii}, the **expected duration** is $\frac{1}{1-P_{ii}}$ periods.

Expected Durations

```
exp_duration = 1 ./ (1 .- diag(P̂_nber))
println("Expected duration of an expansion: $(round(exp_duration[1], digits=1)) months")
println("Expected duration of a recession:  $(round(exp_duration[2], digits=1)) months")
```

```
Expected duration of an expansion: 65.6 months
Expected duration of a recession:  11.3 months
```

- Compare to the historical averages:

```
# Average length of the 12 post-WWII recessions
avg_recession = mean([Dates.value(tr - pk) / 30.44 for (pk, tr) in nber_recessions])
println("Average post-WWII recession length: $(round(avg_recession, digits=1)) months")
```

```
Average post-WWII recession length: 10.3 months
```

Transition Counts

- Let's peek at the raw transition counts:

```
N_nber = countTransitions(states, 2)
println("Transition counts:")
println("  Expansion → Expansion: $(N_nber[1,1])")
println("  Expansion → Recession: $(N_nber[1,2])")
println("  Recession → Expansion: $(N_nber[2,1])")
println("  Recession → Recession: $(N_nber[2,2])")
```

```
Transition counts:
  Expansion → Expansion: 775
  Expansion → Recession: 12
  Recession → Expansion: 12
  Recession → Recession: 124
```

Rare Events

Transitions into recession are **rare**: only about 12 in 77 years of monthly data. This is why recessions are hard to predict!

Log-Likelihood of the NBER Data

- How does our MLE compare to a naive model?

```
P_naive = [0.5 0.5; 0.5 0.5]

println("Log-likelihood at P̂ (MLE):   $(round(loglikelihood_mc(P̂_nber, states), digits=2))")
println("Log-likelihood at naive P:   $(round(loglikelihood_mc(P_naive, states), digits=2))")
```

```
Log-likelihood at P̂ (MLE):   -102.7
Log-likelihood at naive P:   -639.77
```

Summary

Key Takeaways

Concepts

- Discrete random variables
- Transition matrices
- Markov chain simulation
- Expected values as dot products
- Conditional expectations as $P\bar{X}$
- MLE: $\hat{P}_{ij} = n_{ij}/n_{i\cdot}$
- Likelihood depends on **transitions**, not frequencies
- NBER recession dating & estimation

Julia Tools

- `Distributions`: `Categorical`, `rand`
- `LinearAlgebra`: `dot`
- `Optim`: numerical MLE
- List comprehensions with filters
- Transition counting & estimation

Remember

The transition matrix P encodes everything about a Markov chain's dynamics. Row i gives the distribution of the next state given current state i. When P is unknown, the MLE estimator — just counting and normalizing transitions — is simple, intuitive, and consistent.