```julia
begin
    import Pkg
    # activate the shared project environment
    Pkg.activate(".")
    # instantiate, i.e. make sure that all packages are downloaded
    Pkg.instantiate()
end
```

```
  Activating environment at `~/Documents/teach/ECBS-6001-Advanced-Macroeconomics/class/Project.toml`
```

```julia
using LinearAlgebra
```

```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
2-element Vector{Float64}:
 1.0
 1.0
vectors:
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0
```

```julia
eigen([1 0; 0 1])
```

```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
2-element Vector{Float64}:
 -1.0
  1.0
vectors:
2×2 Matrix{Float64}:
 -0.707107  0.707107
  0.707107  0.707107
```

```julia
eigen([0 1; 1 0])
```

```
simulate (generic function with 2 methods)
```

```julia
function simulate(P::Matrix{Float64}, x0::Int64, T=30)::Vector{Int64}
    x = Vector{Int64}(undef, T)
    x[1] = x0
    for t=2:T
        π = P[x[t-1], :]
        x[t] = rand(Categorical(π))
    end
    return x
end
```

```
coin_flips =
[1, 1, 2, 1, 2, 2, 2, 1, 1, 1, 2, 1, 2, 1, 1, 2, 2, 2, 2, 2, ⋯ more ,2, 2, 1, 1, 2, 2, 1, 1
```

```julia
coin_flips = simulate([0.5 0.5; 0.5 0.5], 1, 300)
```

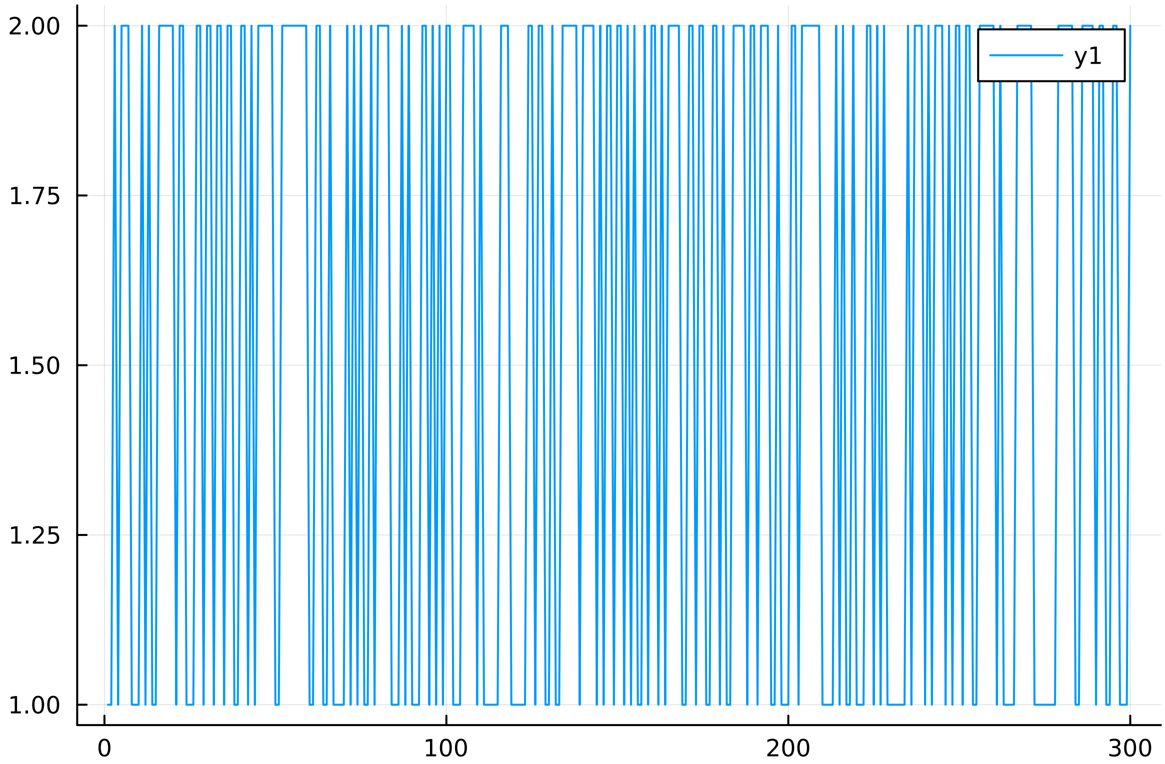
```
1.5433333333333332
```

```julia
mean(coin_flips)
```

```
1.5
```

```julia
mean(coin_flip)
```

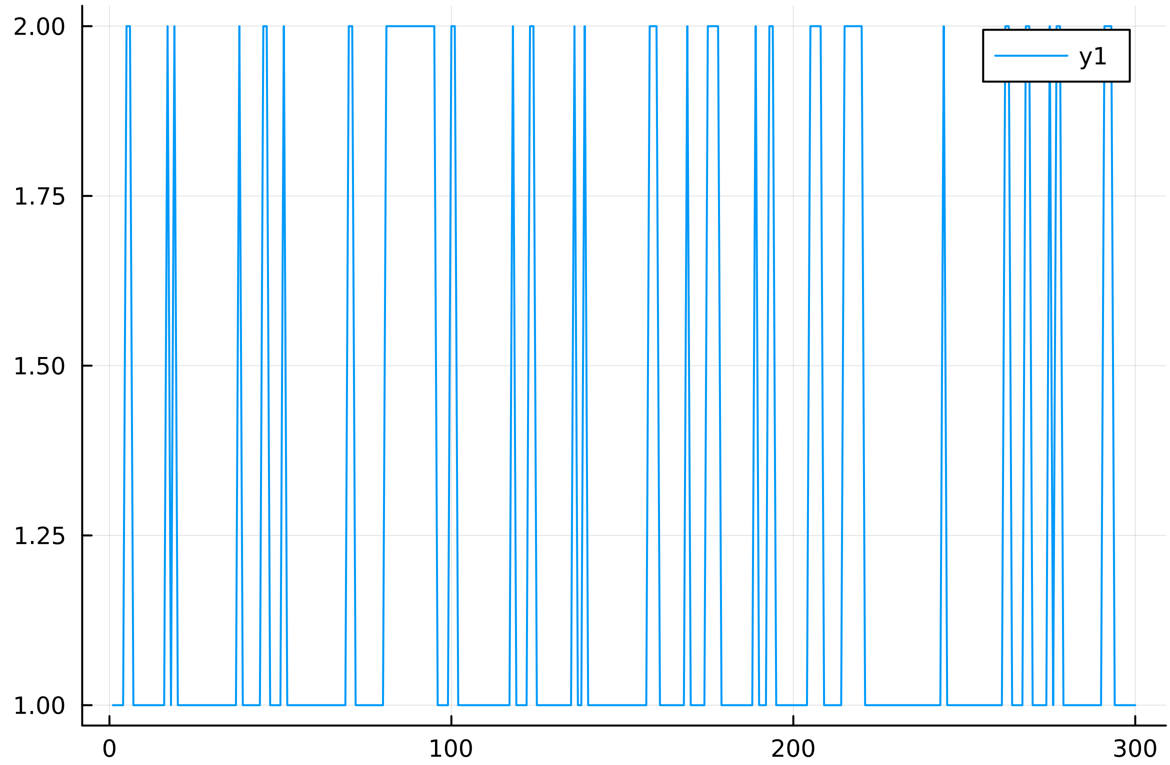
```julia
using Plots
```




```julia
plot(coin_flips)
```

```
persistent_mc =
▸[1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1, ⋯ more ,2, 2, 2, 1, 1, 1, 1, 1
```

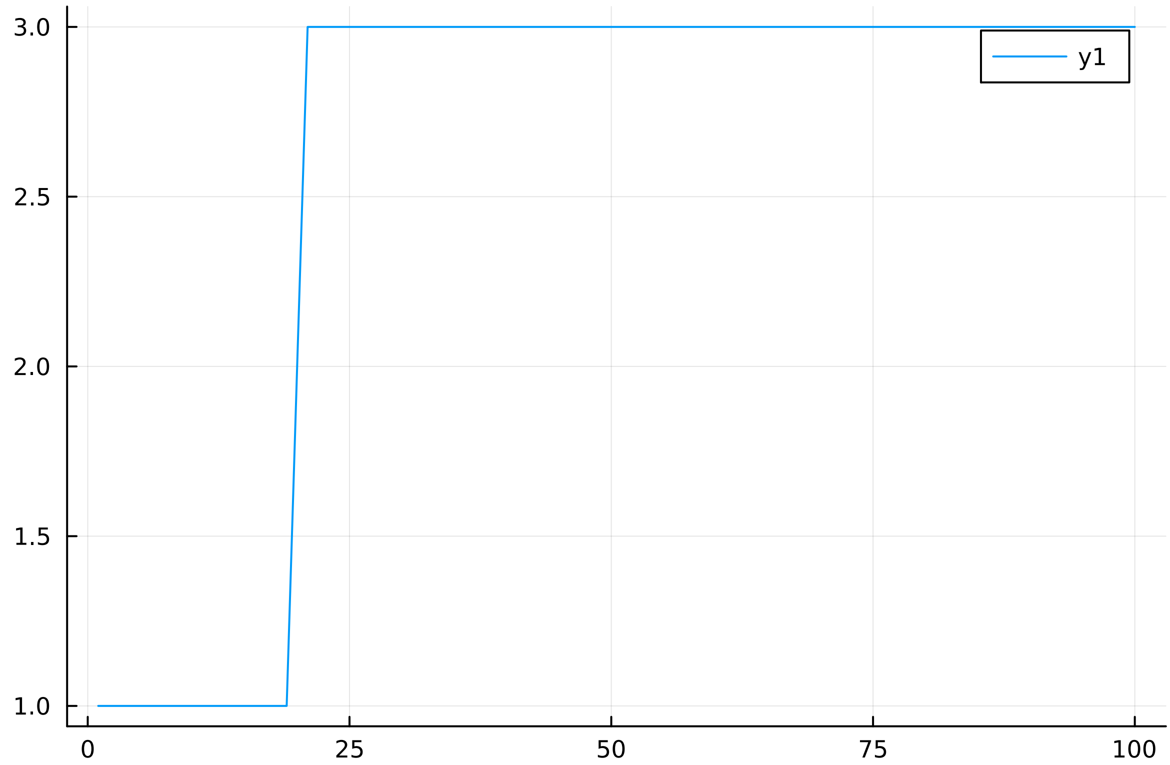
```julia
persistent_mc = simulate([0.9 0.1; 0.4 0.6], 1, 300)
```




```
plot(persistent_mc)
```

```
▸ [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, ⋯ more ,3, 3, 3, 3, 3, 3, 3, 3
```

```
begin
    P3=[0.9 0.1 0;
        0.15 0.7 0.15;
        0.0 0 1.0]
    three_state_mc = simulate(P3, 1, 100)
end
```




```
plot(three_state_mc)
```

```
▸ [0.641886, 0.958114, 1.0]
```

```julia
eigen(P3').values
```

```
3×3 Matrix{Float64}:
  0.472429  -0.570261  0.0
 -0.812936  -0.220934  0.0
  0.340507   0.791195  1.0
```

```julia
eigen(P3').vectors
```

```
3×3 adjoint(::Matrix{Float64}) with eltype Float64:
 0.9  0.15  0.0
 0.1  0.7   0.0
 0.0  0.15  1.0
```

```julia
P3'
```

```
3×3 Matrix{Float64}:
 0.825  0.24   0.0
 0.16   0.505  0.0
 0.015  0.255  1.0
```

```julia
(P3')^2
```

```
3×3 Matrix{Float64}:
 0.226113   0.131402   0.0
 0.0876015  0.0509103  0.0
 0.686285   0.817688   1.0
```

```julia
(P3')^30
```

```
Enter cell code...
```

```julia
using Distributions
```

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

```julia
coin_flip = Categorical([0.5, 0.5])
```

```
▸ [0.5, 0.5]
```

```julia
probs(coin_flip)
```

```
1.5
```

```julia
mean(coin_flip)
```

```
2
```

```julia
rand(coin_flip)
```

```
standard_normal = Distributions.Normal{Float64}(μ=0.0, σ=1.0)
```

```julia
standard_normal = Normal(0.0, 1.0)
```

```
MethodError: no method matching probs(::Distributions.Normal{Float64})
Closest candidates are:
probs(!Matched::Distributions.DiscreteNonParametric) at /Users/koren/.julia/packages/Di
probs(!Matched::Distributions.Multinomial) at /Users/koren/.julia/packages/Distribution
probs(!Matched::Distributions.MixtureModel) at /Users/koren/.julia/packages/Distributio
...

 1. top-level scope @ Local: 1  [inlined]
```

```
probs(standard_normal)
```

```
0.0
```

```
mean(standard_normal)
```

```
0.49806078206592924
```

```
rand(standard_normal)
```

Value function iteration

```
ArgumentError: Package Math not found in current path:
- Run `import Pkg; Pkg.add("Math")` to install the Math package.

 1. require(::Module, ::Symbol) @ loading.jl:893
 2. top-level scope @ Local: 1
```

```
using Math
```

```
iterate_cake_value (generic function with 2 methods)
```

```
function iterate_cake_value(V::Vector{Float64}, β=0.95)::Vector{Float64}
    K = length(V)
    V1 = Vector{Float64}(undef, K)
    for k = 1:K
        possible_consumptions = 0:k-1

        V1[k] = maximum(log.(possible_consumptions) + β * V[k .-
        possible_consumptions])

    end
    return V1
end
```

```
▸ [0.0, 0.693147, 1.09861, 1.38629, 1.60944, 1.79176, 1.94591, 2.07944, 2.19722, 2.30259]
```

```
log.(1:10)
```

```
v1 =
▸ [-Inf, 0.0, 0.693147, 1.09861, 1.38629, 1.60944, 1.79176, 1.94591, 2.07944, 2.19722, 2.3
```

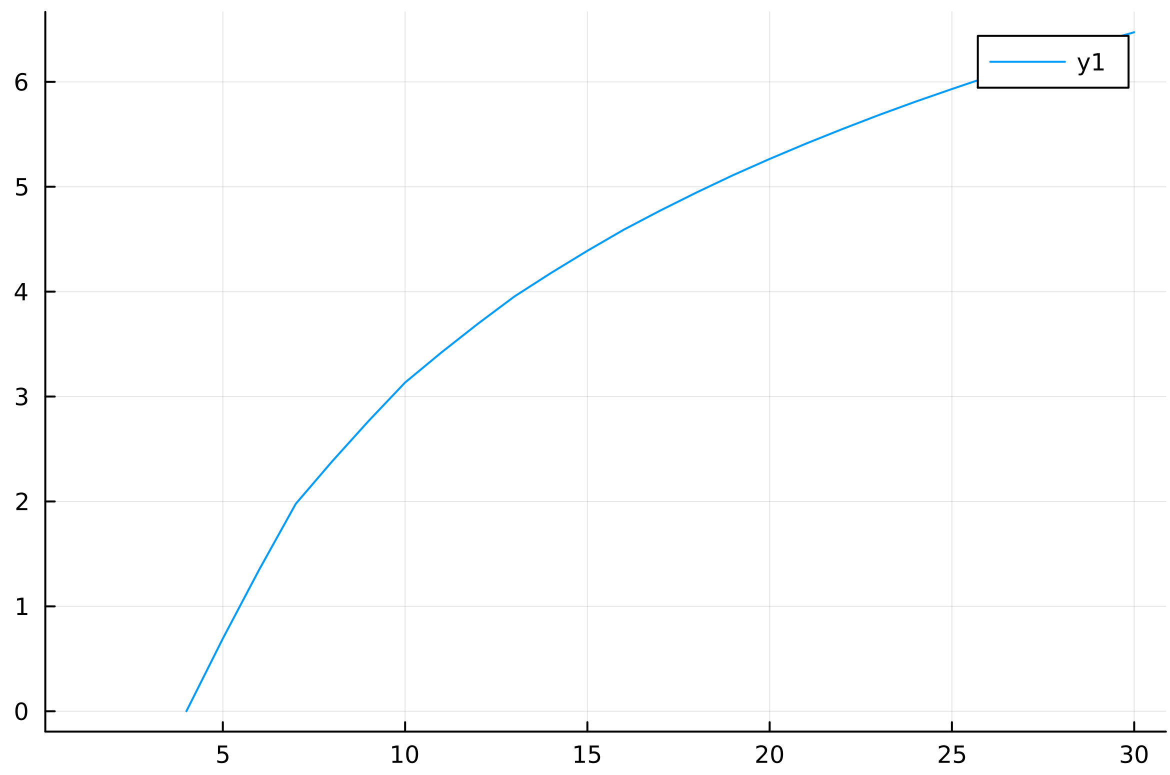
```
v1 = iterate_cake_value(zeros(30))
```

```
v2 =
▸[-Inf, -Inf, 0.0, 0.693147, 1.35164, 1.7571, 2.14229, 2.42998, 2.70327, 2.92642, 3.1384,
```

```julia
v2 = iterate_cake_value(v1)
```

```
v3 =
▸[-Inf, -Inf, -Inf, 0.0, 0.693147, 1.35164, 1.9772, 2.38267, 2.76786, 3.13379, 3.42147, 3.
```

```julia
v3 = iterate_cake_value(v2)
```




```julia
plot(v3)
```

DP with Markov chains

Composite types

I promised some intro to composite types. The next few functions work much better with this. There is no need for copy-pasting parameter values, which is very prone to errors. So I rewrote all our functions for a user-defined type.

```julia
struct MarkovChainProblem
    u::Vector{Float64}
    P::Matrix{Float64}
    β::Float64
end
```

job_search_problem = ▶ MarkovChainProblem([1.0, 0.6], 2×2 Matrix{Float64}:, 0.95)
```
                                                 0.9  0.1
                                                 0.6  0.4
```

```
job_search_problem = MarkovChainProblem(utils, P, 0.95)
```

iterate_value_mc (generic function with 1 method)

```
function iterate_value_mc(v::Vector{Float64},
mcp::MarkovChainProblem)::Vector{Float64}
    # you can refer to the components of a composite type like mcp.u
    # here we are using unpacking to assign three values at the same time
    # this is only to keep our actual formula clean
    u, P, β = mcp.u, mcp.P, mcp.β
    return u + β*P*v
end
```

```
P = 2×2 Matrix{Float64}:
     0.9  0.1
     0.6  0.4
```

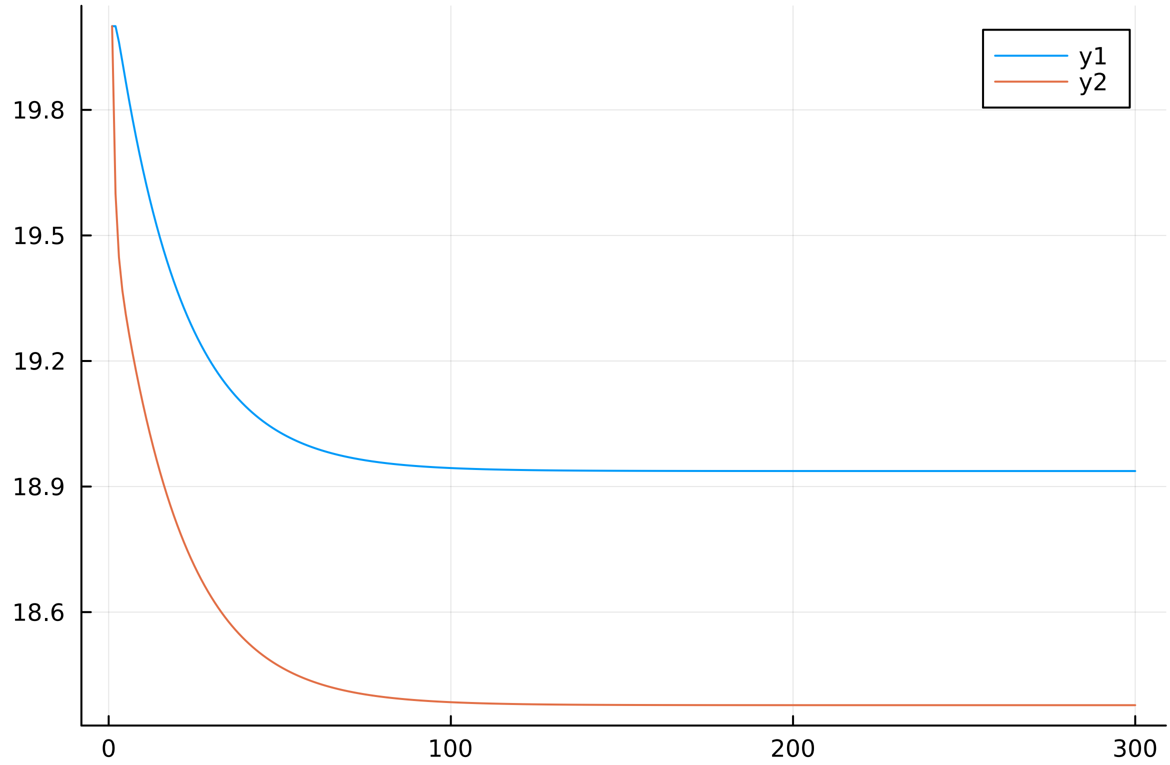
```
P = [   0.9 0.1;
        0.6 0.4]
```

utils = ▶ [1.0, 0.6]

```
utils = [1.0, 0.6]
# unemployment benefit has replacement rate of 0.6
```

▶ [19.24, 16.56]

```
iterate_value_mc(job_search_problem.u / (1-job_search_problem.β),
job_search_problem)
```




```
begin
    K = 300
    values = ones(2, K) * 20
    for k=2:K
        values[:, k] = iterate_value_mc(values[:, k-1], job_search_problem)
    end
    plot(values')
end
```

See, there is no need to ever type the invidual parameters again.

```
[18.9371, 18.3776]
```

```
inv(I - job_search_problem.β * job_search_problem.P) * job_search_problem.u
```

Reflections on the Python-R-Stata debate

```
md"# Reflections on the Python-R-Stata debate"
```

```
debate_data = 3×3 Matrix{Int64}:
               14   1   2
                6  13   2
                5   1  12
```

```
debate_data = [14 1 2; 6 13 2; 5 1 12]
```

```
transition_matrix = 3×3 Matrix{Float64}:
                     0.823529  0.0588235  0.117647
                     0.285714  0.619048   0.0952381
                     0.277778  0.0555556  0.666667
```

```
transition_matrix = debate_data ./ sum(debate_data, dims=2)
```

```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
3-element Vector{Float64}:
 0.5443936030344506
 0.564850094444541
 0.9999999999999996
vectors:
3×3 Matrix{Float64}:
  0.763307  -0.233162  0.906169
 -0.130611  -0.561082  0.194676
 -0.632696   0.794244  0.375446
```

```
eigen(transition_matrix')
```

```
steady_state = ▸[0.613815, 0.131868, 0.254317]
```

```
steady_state = eigen(transition_matrix').vectors[:, 3] ./
sum(eigen(transition_matrix').vectors[:, 3])
```

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

```
function solve_bellman_for_mc(mcp::MarkovChainProblem)::Vector{Float64}
    u, P, β = mcp.u, mcp.P, mcp.β
    return inv(I - β * P) * u
end
```

Endogenous search

```
struct EndogenousSearchProblem
    u::Vector{Float64}
    λ::Float64
    δ::Float64
    β::Float64
end
```

```
endogenous_search = ▸EndogenousSearchProblem([1.0, 0.6], 3.0, 0.1, 0.95)
```

```
endogenous_search = EndogenousSearchProblem(utils, 3.0, 0.1, 0.95)
```

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

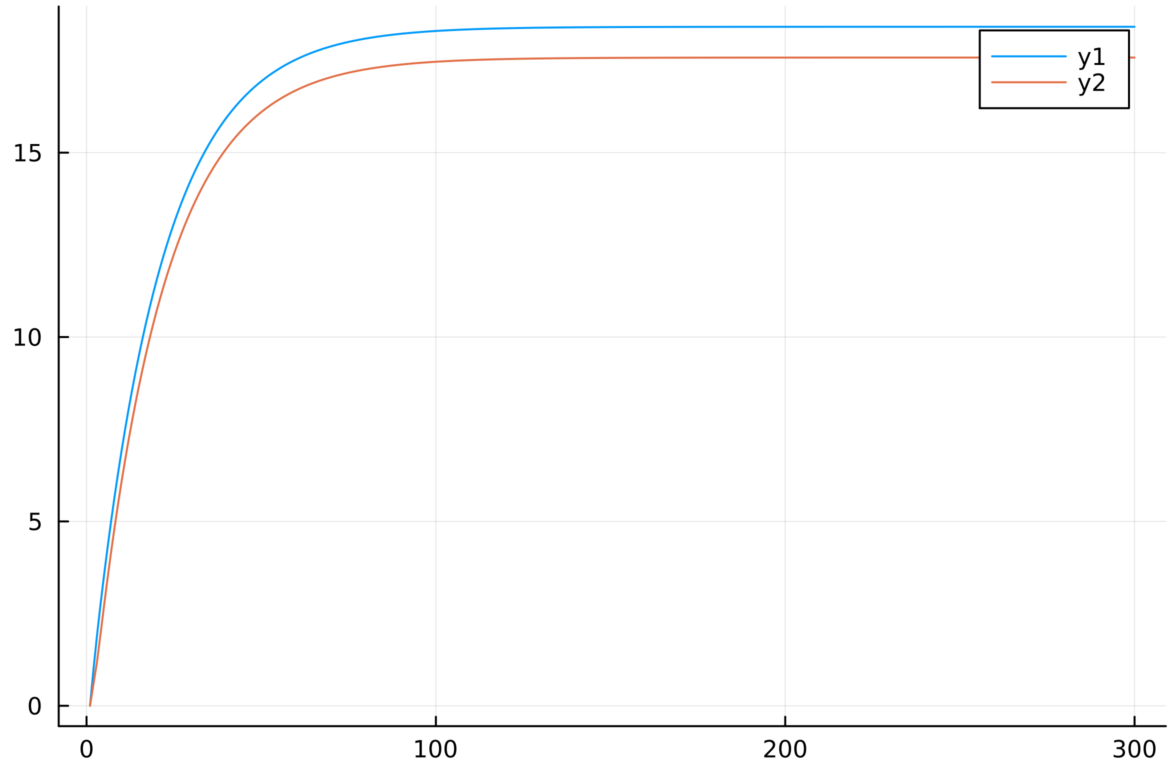
```
function solve_optimal_search(v::Vector{Float64},
es::EndogenousSearchProblem)::Float64
    β, λ = es.β, es.λ
    prob_unemployed = sqrt(1/(β*λ*(v[1] - v[2])))
    if prob_unemployed < 0.0001
        prob_unemployed = 0.0001
    elseif prob_unemployed > 1
        prob_unemployed = 1
    end
    return (1/prob_unemployed - 1)/λ
end
```

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

```julia
function iterate_value_for_search(v::Vector{Float64},
es::EndogenousSearchProblem)::Vector{Float64}
    u, β, λ, δ = es.u, es.β, es.λ, es.δ
    # given optimal solution, what is transition matrix?
    c = solve_optimal_search(v, es)
    Λ = λ * c / (1 + λ * c)
    P = [1-δ δ; Λ 1-Λ]
    return u + β * P * v
end
```

```
[1.95, 1.55]
```

```julia
iterate_value_for_search(ones(2), endogenous_search)
```




```julia
begin
    K2 = 300
    v = zeros(K2, 2)
    for k = 2:K2
        v[k, :] = iterate_value_for_search(v[k-1, :], endogenous_search)
    end
    plot(v)
end
```

```
300×2 Matrix{Float64}:
  0.0       0.0
  1.0       0.6
  1.912     1.1941
  2.7482    1.9396
  3.53397   2.70477
  4.2785    3.44485
  4.98538   4.15078
  ⋮
 18.4138   17.5789
 18.4138   17.5789
 18.4138   17.5789
 18.4138   17.5789
 18.4138   17.5789
 18.4138   17.5789
```

```julia
v
```

```
0.6482922184085216
```

```julia
sqrt(1/(0.95*3*(v[300, 1] - v[300, 2])))
```

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

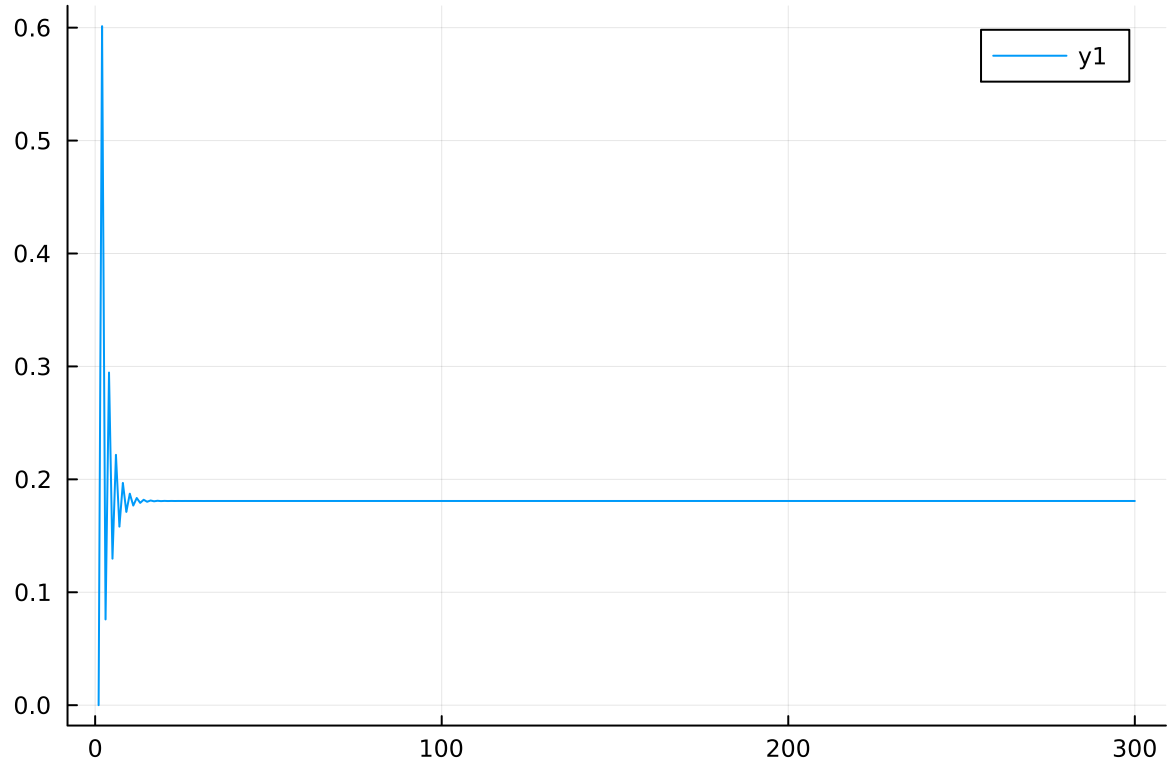
```julia
function iterate_policy_for_search(c::Float64, es::EndogenousSearchProblem)::Float64
    u, β, λ, δ = es.u, es.β, es.λ, es.δ
    Λ = λ*c / (1 + λ*c)
    P = [1-δ δ; Λ 1-Λ]
    # because solve_bellman_for_mc now takes a MarkovChainProblem, we have to create one that we can pass on
    mcp = MarkovChainProblem(u, P, β)
    v = solve_bellman_for_mc(mcp)
    return solve_optimal_search(v, es)
end
```

```
0.6013127057589025
```

```julia
iterate_policy_for_search(0.0, endogenous_search)
```

```
0.07607092440074004
```

```julia
iterate_policy_for_search(0.6, endogenous_search)
```




```
begin
    K3 = 300
    c = zeros(K3)
    for k = 2:K3
        c[k] = iterate_policy_for_search(c[k-1], endogenous_search)
    end
    plot(c)
end
```

```
Enter cell code...
```